Assignment 1 Solution

Name: Mingzhe Wang MacID: wangm235

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.

Note for modification of codes

To improve the performance of my code and test, it should be noticed that there are several modifications after the Part 1 deadline. Specifically, they are:

Changes for test_driver.py

- Add three test cases, named `test_complex_con`, `test_triangle_con_normal`, and `test_triangle_con_except` to test the constructors for `ComplexT` and `TriangleT`, respectively.
- For each `get_phi`, `sqrt`, `equal`, and `tri_type`, separate the original single test case to multiple ones to better represent the rationale that test cases should cover all branches/exceptions. This modification is also helpful for determining exactly which branch is wrong.
- Add more test cases for testing the branch of `tri_type` that returns `TriType.right`, because the single original test case is found not enough to cover all conditions.

Changes for complex_adt.py

- Add `@throws ZeroDivisionError` to the document comments of `get_phi`, `recip`, and `div`, because of the fact that they all have flexible variables as the denominators in certain calculation, which can induce `ZeroDivisionError`.

1 Assumptions and Exceptions

1.1 Assumptions of complex_adt.py

- The argument type of the constructor of `ComplexT` must be float, there is no auto float converting.
- As is the Python's philosophy, no type check performed for the constructor of `ComplexT`.
- The method `add`, `div`, `mult`, `recip`, `sqrt`, and `sub` is a getter and will not imutate the current object.
- The argument of `div` must be an non-zero complex number.
- The method `div` is an "exact" equal, which compares two complex numbers' real and imaginary parts by bit-by-bit equal.
- For `get_phi`, the real part and the imaginary part of the given complex number should not be both zero.
- For `recip`, the current complex number must be non-zero.
- For `sqrt`, the imaginary part of the current complex number should not be zero. The output is the positive square root.

1.2 Exceptions of complex_adt.py

- For `div`, a `ZeroDivisionError` will be raised if the real and imaginary parts of the argument complex number are so small that raises float division by zero error.
- For `get_phi` and `recip`, a `ZeroDivisionError` will be raised if the real and imaginary parts of the current complex number are so small that raises float division by zero error.

1.3 Assumptions of triangle_adt.py

- For the constructor of `TriangleT`, a `TriangleT` can be formed even if it is not valid (i.e.its three sides can not form a geometric triangle). However, a `TriangleT` can be formed only if all its three sides are positive integers. As is the Python's philosophy, no type check performed.

- To use methods such as `area`, `perim`, and `tri_type`, the current triangle must be a valid triangle.
- For `tri_type`, if a `TriangleT` belongs to both equilat and isosceles, it will be equilat. Because equilat is a more special type; if a `TriangleT` belongs to both scalene and right, it will be right. Because right is a more specical type.

1.4 Exceptions of triangle_adt.py

- For the constructor of `TriangleT`, a `ValueError` will be raised if some inputs are negative.

2 Test Cases and Rationale

2.1 Test Cases

There are totally 27 test cases performed, in each test cases there could be multiple assertions that can test the performance of the program to extremely small, large numbers and special numbers. However, it should be noticed that each test case is designed to test one branch/exception of a given method. Through this, it is easy to determine which statement is not reliable in a single method.

2.2 Rationale

The selection of these test cases follows certain rationales, in particular, they are:

- The test cases should cover every branches of the tested method. For example, there should be at least four test cases for `tri_type`, because there are four branches that will return `TriType.right`, `TriType.equilat`, `TriType.isosceles`, and `TriType.scalene` respectively in the definition of `tri_type`.
- If a method or constructor could raise certain exceptions, then these exceptions could be treated as branches, which means the test cases should also cover them. However, in my implementation, instead of throwing exceptions, most of them are marked as invalid input in the assumptions. So the only example for testing exceptions will be the test for the constructor of `TriangleT`.
- For float numbers comparison, normally we only need to perform approximate equality under certain tolerated difference. In this test, they are set as following:

```
def isClose(a, b, rel_tol = 1e-09, abs_tol = 0.0):
    return abs(a - b) <= max(rel_tol * max(abs(a), abs(b)),
       abs_tol)
```

- Test cases that can be verified in another way easily should be chosen. For example, test cases like `ComplexT(math.sqrt(2), -math.sqrt(2))` were chosen, because they can be calculated manually quickly, however they may not be that easy for computers to calculate.
- Test cases that use different mathematical types of numbers, such as negative numbers, rational numbers, irrational numbers, numbers expressed in scientific notation etc. should be chose. Because they can test the performance of program in different circumstances.
- Test cases that use extremely large or small numbers as input should be chosen. For example, test cases like `ComplexT(1e10,1e10)` and `ComplexT(-1e10,-1e10)` were chosen because they help test the reliability of the program in edge cases.

3 Results of Testing Partner's Code

3.1 Results of testing my code

```
python3  src/test_driver.py
-------------------------Tests for
   complex_adt.py--------------------
test_complex_con() test: PASSED.
real() test: PASSED.
imag() test: PASSED.
get_r() test: PASSED.
get_phi() not (real < 0 and imag = 0) test: PASSED.
get_phi() (real < 0 and imag = 0) test: PASSED.
equal() test: PASSED.
conj() test: PASSED.
add() test: PASSED.
sub() test: PASSED.
mult() test: PASSED.
recip() test: PASSED.
div() test: PASSED.
sqrt() (imag > 0) test: PASSED.
sqrt() (imag < 0) test: PASSED.
```

```
-------------------------Tests for
   triangle_adt.py-------------------
test_triangle_con() normal test: PASSED
test_triangle_con() except test: PASSED
get_sides() test: PASSED.
equal() true test: PASSED.
equal() false test: PASSED.
perim() test: PASSED.
area() test: PASSED.
is_valid() test: PASSED.
tri_type() right test: PASSED.
tri_type() equilat test: PASSED.
tri_type() isosceles test: PASSED.
tri_type() scalene test: PASSED.
-----------------------------------Total----------------------------
Passed test number:  27
Total test number:  27
```

When running the tests on my code, there is no exception raised. My code passes all the 27 tests.

3.2 Results of testing partner's code: before modification

```
python3  partner/test_driver.py
-------------------------Tests for
   complex_adt.py--------------------
test_complex_con() test: PASSED.
real() test: PASSED.
imag() test: PASSED.
get_r() test: PASSED.
get_phi() not (real < 0 and imag = 0) test: PASSED.
get_phi() (real < 0 and imag = 0) test: PASSED.
equal() test: PASSED.
conj() test: PASSED.
add() test: PASSED.
sub() test: PASSED.
mult() test: PASSED.
recip() test: PASSED.
div() test: PASSED.
sqrt() (imag > 0) test: PASSED.
sqrt() (imag < 0) test: PASSED.
```

```
-------------------------Tests for
   triangle_adt.py-------------------
test_triangle_con() normal test: PASSED
test_triangle_con() except test: FAILED
get_sides() test: PASSED.
equal() true test: PASSED.
equal() false test: PASSED.
perim() test: PASSED.
area() test: PASSED.
is_valid() test: PASSED.
tri_type() right test: FAILED.
tri_type() equilat test: PASSED.
Traceback (most recent call last):
  File "partner/test_driver.py", line 477, in <module>
    test_tri_type_isosceles()
  File "partner/test_driver.py", line 418, in
     test_tri_type_isosceles
    assert t3.tri_type() == TriType.isosceles
  File "/usr/lib/python3.6/enum.py", line 326, in __getattr__
    raise AttributeError(name) from None
AttributeError: isosceles
Makefile:15: recipe for target 'test' failed
make: *** [test] Error 1
```

When running the tests on the partner's code, an `AttributeError` is raised because there is a typo in "isoceles", which should be "isosceles". To make the test continue, the typo in "isoceles" is modified, and then the partner's code is tested again.

3.3 Results of testing partner's code: after modification

```
python3  partner/test_driver.py
-------------------------Tests for
   complex_adt.py--------------------
test_complex_con() test: PASSED.
real() test: PASSED.
imag() test: PASSED.
get_r() test: PASSED.
get_phi() not (real < 0 and imag = 0) test: PASSED.
get_phi() (real < 0 and imag = 0) test: PASSED.
equal() test: PASSED.
conj() test: PASSED.
add() test: PASSED.
```

```
sub() test: PASSED.
mult() test: PASSED.
recip() test: PASSED.
div() test: PASSED.
sqrt() (imag > 0) test: PASSED.
sqrt() (imag < 0) test: PASSED.
-------------------------Tests for
   triangle_adt.py-------------------
test_triangle_con() normal test: PASSED
test_triangle_con() except test: FAILED
get_sides() test: PASSED.
equal() true test: PASSED.
equal() false test: PASSED.
perim() test: PASSED.
area() test: PASSED.
is_valid() test: PASSED.
tri_type() right test: FAILED.
tri_type() equilat test: PASSED.
tri_type() isosceles test: PASSED.
tri_type() scalene test: PASSED.
-----------------------------------Total---------------------------
Passed test number:  25
Total test number:  27
```

After running the tests on the partner's code again, the output shows the partner's code passes 25 tests of the total 27 tests.

The `test_triangle_con except test` failed because my partner's code didn't raise an exception when some of the arguments of the constructor of `TriangleT` are not positive. I then check if my partner denoted this assumption in the document comment instead of raising an exception, but the answer is no. So I should conclude there is a flaw in my partner's implementation in the constructor of `TriangleT`.

The `test_tri_type_right test` failed because there is a logic flaw in my partner's implementation. The test case is as the following:

```
def test_tri_type_right():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3,4,5)
                t2 = TriangleT(4,5,3)
```

```
            t3 = TriangleT(5,3,4)
            assert t1.tri_type() == TriType.right
            assert t2.tri_type() == TriType.right
            assert t3.tri_type() == TriType.right
            passed += 1
            print("tri_type() right test: PASSED.")
        except AssertionError:
            print("tri_type() right test: FAILED.")
```

The partner's code is as the following:

```
        if (a**2 + b**2) == c**2:
            return TriType.right
```

Obviously, my partner forget to take into account that for right triangles, each side is possible to be the hypotenuse.

To sum up, my partner's code passed 24 of 27 total test cases (if regarding the typo as a failed case). One of them is raised by typo, while two of them are raised by not considering all the conditions. It should also be noticed that my partner's code don't include any assumption in the `@details` field. Because my test cases selection is based on my assumptions on the valid input. If I assume that no assumption showing in my partner's comment means all the inputs are accepted, then there could be more test failures derived from my partner's code.

4 Critique of Given Design Specification

For `ComplexT`, the thing I like is that the whole ADT is very cohesive, where state variables and operations included are all related to mathematical complex numbers. However, the names of the methods are things that need to be improved. For example, `real` and `imag` are simply getters, so it is better to use prefix `get_` to denote them. But these naming concepts may not be extended to methods like `conj`, `recip` or `sqrt`, because also they are getters, their names should be consistent with other binary complex number operations like `add` etc. Another thing that should be improved is that the valid inputs for constructing a `ComplexT` must be the type of float. Although it may be more difficult to implement `ComplexT` that also accept integer or other types as the constructor argument, it is a possible improvement to make the whole ADT more consistent with the mathematical complex number concept.

For `TriangleT`, there are many areas I do not like. For example, by the default assumption, a `TriangleT` can be formed even it is not geometrically valid. This can induces

many weird behaviors of its method. Another area is about type of the sides, in fact, a geometric triangles can have sides type of float, it is not necessary to narrow this type to just integer. The next area is is about the `tri_type`, actually even triangles with integer sides could have multiple shape types. However, this design only want us to return one type. To achieve that, we must do some assumptions that when a triangle belongs to multiple types, which one should be return. By doing that, we are loosing some information. So my suggestion is that this method should return type of set of shapes to fully record the shape information. One thing to be improved is similar as `ComplexT`, for methods like `perim` and `area`, it is better to use prefix `get_` to denote them. Another thing is that if we make constructor checks if the triangle is geometrically valid, then `is_valid` is redundant. For the principle of detecting likely changes, we may also need to add more constructors that can form a triangle using different elements like 1 phase and 2 sides to increase the reusability of our code.

5 Answers to Questions

(a) In `ComplexT`, there is no mutators, while selectors are `real`, `imag`, `get_r`, `get_phi`, `conj`, `recip`, and `sqrt`. In `TriangleT`, there is no mutators, while selectors are `get_sides`, `perim`, `area`, `is_valid`, and `tri_type`.

(b) **Option 1**
State variables in `ComplexT`: `x` that denotes the real part and `y` that denotes the imaginary part.
State variables in `TriangleT`: `a`, `b` and `c` that represents the three sides of the triangle respectively.
Option 2
State variables in `ComplexT`: `x` that represents the real part, `y` that represents the imaginary part, `r` that represents the absolute value, `phi` that represents the phase.
State variables in `TriangleT`: `a`, `b` and `c` that represents the three sides of the triangle respectively, `perim` that represents the perimeter, `area` that represents the area, `is_valid` that check if the given triangle is geometrically valid and `tri_type` that represents the type of current triangle.

(c) No. Because the definition of orders in complex numbers varies. For example, you can compare the real parts and ignore the imaginary parts; you can use the lexicographic order, comparing the real parts and then comparing the imaginary parts if the real parts are equal; you can also compare complex numbers by their modulus... It's hard for us to predict which one meets the client's requirement, so it will be not necessary and not meaningful to add a methods for greater than and less than.

(d) Yes, it is possible. In my opinion, the constructor of `TriangleT` should raise an exception when it is given an invalid triangle. Because all the methods after the construction, such as `get_sides`, `perim`, `area` and `tri_type`, they all need the assumption that the triangle is geometrically valid, otherwise their output will be meaningless from the geometric view. For example, for an invalid "triangle" 1, 2, 3, calculating its perimeter and area makes no sense, and you are not able to return any triangle type for it.

(e) It is a bad idea. When a state variable is set, its main function is to be reused in other methods to accelerate or simplify our computation. However, for the type of triangle, it is more like a "final" output, i.e. no method in the current implementation need them, which means the reusability of the type of triangle is low. Therefore, we do not need to sacrifice the space complexity of our code to add a state variable whose use frequency is low.

(f) Poor performance often adversely affects the usability. For example, a software system that produces answers more slowly than the user requires is not user-friendly, i.e. the usability is bad, even if the answers are displayed in beautiful color.

(g) No. According to A RATIONAL DESIGN PROCESS: HOW AND WHY TO FAKE IT, a software design "process" will always be an idealisation, which means we should always "fake" a rational design process. Some of reasons are listed below:

- In most cases the people who commission the building of a software system do not know exactly what they want and are unable to tell us all that they know.
- Even if we knew the requirements, there are many other facts that we need to know to design the software. Such details could only be released as we progress in the implementation.
- Even if we knew all of the relevant facts before we started, experience shows that human beings are unable to comprehend fully the plethora of details that must be taken into account in order to design and build a correct system.
- Even if we could master all of the detail needed, all but the most trivial projects are subject to change for external reasons.
- Human errors can only be avoided if one can avoid the use of humans.
- We are often burdened by preconceived design ideas, ideas that we invented, acquired on related projects, or heard about in a class. Such ideas may not be derived from our requirements by a rational process.

- Often we are encouraged, for economic reasons, to use software that was developed for some other project. So we should share our software with another ongoing project to save effort.

(h) Reusability may affect the reliability in both positive and negative way. The positive effect is that if we can use the codes which was proven to perform good in other modules, then we have the confidence that the code will also perform well in the current implementation, which will increase the reliability. However, one negative effect is that this kind of reuse may also propagates errors to the new implementation if its reliability has not been tested thoroughly. Another negative effect is that reuse principle can make the design more general that reduces the reliability of the codes. For example, we should not reuse the code for a clock application that was designed for people as an alarm to implement another clock used in the scientific field, which requires extremely high precision.

(i) An example of this kind of abstraction is Java. As a portable programming language, Java works by first compiling the source code into bytecode. Then, the bytecode can be compiled into machine code with the Java Virtual Machine (JVM). As we use Java to programming, we do not necessarily know how hardware performs by the machine code derived from our source code. This process is a secret, i.e. hided information for us. Another example is Python, which is a dynamic, interpreted (bytecode-compiled) language, similar to Java, the bytecode interpreted will also be run on the python virtual machine. And this process is an abstraction on top of hardware. Even for low-level assembly languages, like NASM and RISC-V, they also provide some level of abstraction, where we do not need to write the binary machine code for each instruction, instead we can easily convert each of our instruction statement to some binary machine codes.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Mingzhe Wang
#  @brief Implementation of complex number ADT.
#  @date Jan 20, 2021

import math

## @brief An ADT for reprensenting a complex number.
class ComplexT:

        ## @brief Constructor for ComplexT.
        #  @details Create a complex number by provding two floats number as the
        #                   real and the imaginary number, respectively.<br/>
        #           Assumption 1: the input must be float, there is no auto float
        #           converting.<br/>
        #           Assumption 2: as it is Python's philosophy, no type check
        #           performed.
        #  @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 Returns the real part of the complex number.
        #  @return Float representing the real part of the complex number.
        def real(self):
                return self.__x

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

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

        ## @brief Returns the phase of the complex number.
        #  @details The value of the phase is expressed in radians.<br/>
        #               Assumption 1: the real part and the imaginary part of the
        #           given complex number should not be both zero.
        #  @return Float representing the phase in radians of the complex number.
        #          This value is in the internal (-pi, pi].
        #  @throws ZeroDivisionError if the real and imaginary parts of the
        #                   current complex number are so small that raises float
        #          division by zero error.
        def get_phi(self):
                if self.__x < 0 and self.__y == 0:
                        return math.pi
                else:
                        return 2 * math.atan(self.__y
                                / (math.sqrt(self.__x ** 2 + self.__y ** 2) + self.__x))

        ## @biref Checks if the current complex numer is equal to another complex
        #                   number.
        #  @details The two complex number is equal iff both the real parts and
        #                   the imaginary parts of them equal.<br/>
        #           Assumption 1: this equal is an "exact" equal, bit-by-bit
        #                   equal.
        #  @param newCom ComplexT Another complex number to be compared with.
        #  @return True if the current complex numer is equal to another complex
        #                   number.
        #          False otherwise.
        def equal(self, newCom):
                return self.__x == newCom.__x and self.__y == newCom.__y

        ## @brief Returns a complex number that is the complex conjugate of the
        #          current complex number.
        #  @return ComplexT representing the complex conjugate of the current
        #          complex number.
        def conj(self):
                c = ComplexT(self.__x, -self.__y)
                return c

        ## @brief Calculates a complex number representing the result of adding
```

```
#          another complex number to the current one.
#   @details Assumption 1: the behaviour of this method is a getter and
#                          will not imutate the current object.
#   @param newCom ComplexT Another complex number to be added to the
#                          current one.
#   @return ComplexT representing the result of the addition.
def add(self, newCom):
        x = self.__x + newCom.__x
        y = self.__y + newCom.__y
        c = ComplexT(x, y)
        return c

## @brief Calculates a complex number representing the result of
#            subtracting another complex number from the current one.
#   @details Assumption 1: the behaviour of this method is a getter and
#                          will not imutate the current object.
#   @param newCom ComplexT representing the subtrahend of the subtraction.
#   @return ComplexT representing the result of the subraction, which is
#           the current complex number - another complex number.
def sub(self, newCom):
        x = self.__x - newCom.__x
        y = self.__y - newCom.__y
        c = ComplexT(x, y)
        return c

## @brief Calculates a complex number representing the result of
#          multiplying the current complex number with another complex
#                 number.
#   @details Assumption 1: the behaviour of this method is a getter and
#                          will not imutate the current object.
#   @param newCom ComplexT representing the multiplier of the
#                 multiplication.
#   @return ComplexT representing the result of the multiplication, which
#           is the current complex number * another complex number.
def mult(self, newCom):
        x = self.__x * newCom.__x - self.__y * newCom.__y
        y = self.__x * newCom.__y + self.__y * newCom.__x
        c = ComplexT(x, y)
        return c

## @brief Calculates the reciprocal of the current complex number.
#   @details Assumption 1: the behaviour of this method is a getter and
#                          will not imutate the current object.<br/>
#            Assumption 2: the current complex number must be non-zero.
#   @return ComplexT representing the reciprocal of the current complex
#           number.
#   @throws ZeroDivisionError if the real and imaginary parts of the
#                   current complex number are so small that raises float
#           division by zero error.
def recip(self):
        x = self.__x / (self.__x ** 2 + self.__y ** 2)
        y = - (self.__y / (self.__x ** 2 + self.__y ** 2))
        c = ComplexT(x, y)
        return c


## @brief Calculates a complex number representing the result of dividing
#         the current complex number by another complex number.
#   @details Assumption 1: the behaviour of this method is a getter and
#                          will not imutate the current object.<br/>
#            Assumption 2: the argument must be an non-zero complex number.
#   @param newCom ComplexT representing the divisor of the divison.
#   @return ComplexT representing the result of the division, which is
#                   current Complex number divides by another complex number.
#   @throws ZeroDivisionError if the real and imaginary parts of the
#                   argument complex number are so small that raises float
#           division by zero error.
def div(self, newCom):
        x = ((self.__x * newCom.__x + self.__y * newCom.__y)
                / (newCom.__x ** 2 + newCom.__y ** 2))
        y = ((self.__y * newCom.__x - self.__x * newCom.__y)
                / (newCom.__x ** 2 + newCom.__y ** 2))
        c = ComplexT(x, y)
        return c

## @brief Calculates the positive square root of the current object.
#   @details Assumption 1: the imaginary part of the current complex number
#            is not equal to 0.<br/>
#                          Assumption 2: the result is the positive square root.<br/>
#            Assumption 3: the behaviour of this method is a getter and
```

```
#                       will not imutate the current object.
#   @return ComplexT represnting the positive square root of the current
#           complex number.
def sqrt(self):
        absValue = self.get_r()
        x = math.sqrt((self.__x + absValue) / 2)
        if self.__y > 0:
                y = 1 * math.sqrt(((- self.__x) + absValue) / 2)
        else:
                y = (- 1) * math.sqrt(((- self.__x) + absValue) / 2)
        c = ComplexT(x, y)
        return c
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Mingzhe Wang
#  @brief Implementation of triangle ADT.
#  @date Jan 20, 2021

import math
from enum import Enum, auto

## @brief An Enum for representing different types of triangles.
#  @details Types are elements of the set {equilat, isosceles, scalene,
#           right}.
class TriType(Enum):
        equilat = auto()
        isosceles = auto()
        scalene = auto()
        right = auto()

## @brief An ADT for representing triangles with integer sides.
#  @details Triangles are defined by its three sides.
class TriangleT:

        ## @brief Constructor for TriangleT.
        #  @details Assumption 1: a triangle CAN be formed even if it is not valid
        #                   (i.e.its three sides can not form a geometric triangle).<br/>
        #                           Assumption 2: however, a triangle can be formed only if all its
        #                           three sides are positive integers.<br/>
        #                           Assumption 3: to be consistent with Python's philospphy, no
        #           type check.
        #  @param a Integer representing the first side of this triangle.
        #  @param b Integer representing the second side of this triangle.
        #  @param c Integer representing the third side of this triangle.
        #  @throws ValueError If some side(s) <= 0.
        def __init__(self, a, b ,c):
                if a > 0 and b > 0 and c > 0:
                        self.__a = a
                        self.__b = b
                        self.__c = c
                else:
                        raise ValueError("sides should > 0")

        ## @brief Returns a tuple of three sides of the given triangle.
        #  @details The order of sides in the output tuple is as the order with
        #           which they are constructed.
        #  @return Tuple representing the three sides of the triangle.
        def get_sides(self):
                return (self.__a, self.__b, self.__c)

        ## @brief Checks if the current triangle is equal to another one.
        #  @details "Equal" here means these two triangles are geometric equal,
        #                           i.e. their shapes are the same.
        #  @return True if the current triangle is equal to another one.
        #          False otherwise.
        def equal(self, newTri):
                if sorted(self.get_sides()) == sorted(newTri.get_sides()):
                        return True
                else:
                        return False

        ## @brief Returns the perimeter of the current triangle.
        #  @details Assumption 1: the current triangle is a valid triangle.
        #  @return Integer representing the the perimeter of the current triangle.
        def perim(self):
                return self.__a + self.__b + self.__c

        ## @brief Returns the area of the current triangle.
        #  @details Assumption 1: the current triangle is a valid triangle.
        #  @return Float representing the area of the current triangle.
        def area(self):
                # s is semi-perimeter of the given triangle
                # use 2.0 to set the float divide
                s = (self.__a + self.__b + self.__c) / 2.0
                # a is area of the given triangle
                a = math.sqrt(s * (s - self.__a) * (s - self.__b) * (s - self.__c))
                return a

        ## @brief Checks if the current triangle is geometrically valid.
```

```
#   @details A triangle is valid iff its three sides can form a triangle in
#            geometry.
#   @return True if the current triangle can form a valid triangle.
#                   False otherwise.
def is_valid(self):
        return (self.__a + self.__b > self.__c
                        and self.__a + self.__c > self.__b
                        and self.__b + self.__c > self.__a)


## @brief Returns the type of a valid triangle. The type is an element of
#                  the set {equilat, isosceles, scalene, right}.
#   @details Assumption 1: the current triangle must be a valid triangle.<br/>
#                         Assumption 2: if a triangle belongs to both equilat and isosceles,
#                         it will be equilat. Because equilat is a more special type.<br/>
#                         Assumption 3: if a triangle belongs to both scalene and right,
#                         it will be right. Because right is a more specical type.
#   @return TriType representing the shape of the triangle. It should be one
#                   of equilat, isosceles, scalene and right.
def tri_type(self):
        if (self.__a ** 2 + self.__b ** 2 == self.__c ** 2
                or self.__a ** 2 + self.__c ** 2 == self.__b ** 2
                or self.__b ** 2 + self.__c ** 2 == self.__a ** 2):
                return TriType.right
        elif self.__a == self.__b == self.__c:
                return TriType.equilat
        elif (self.__a == self.__b
                or self.__b == self.__c
                or self.__a == self.__c):
                return TriType.isosceles
        else:
                return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Mingzhe Wang
#  @brief test cases for complex and triangle ADT
#  @date Jan 20, 2021

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

# Note: this is cited from the previous year 2017 Assignment test.
def isClose(a, b, rel_tol = 1e-09, abs_tol = 0.0):
    return abs(a - b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)

# approximate equal of complex number for test ONLY.
# private method.
def __appEqualCom__(comA, comB):
        if isClose(comA.real(), comB.real()) and isClose(comA.imag(), comB.imag()):
                return True
        else:
                return False

def test_complex_con():
        global testTot, passed
        testTot += 1
        try:
                c1 = ComplexT(5.0, 1.921)
                passed += 1
                print("test_complex_con() test: PASSED.")
        except:
                print("test_complex_con() test: FAILED.")

def test_real():
        global testTot, passed
        testTot += 1
        try:
                assert isClose(c1.real(), 1.0)
                assert isClose(c2.real(), 1e10)
                assert isClose(c3.real(), -1e10)
                assert isClose(c4.real(), 1.4142135623730951)
                assert isClose(c5.real(), -2.0)
                passed += 1
                print("real() test: PASSED.")
        except AssertionError:
                print("real() test: FAILED.")

def test_imag():
        global testTot, passed
        testTot += 1
        try:
                assert isClose(c1.imag(), 1.0)
                assert isClose(c2.imag(), 1e10)
                assert isClose(c3.imag(), -1e10)
                assert isClose(c4.imag(), -1.4142135623730951)
                assert isClose(c5.imag(), 3.4641016151377544)
                passed += 1
                print("imag() test: PASSED.")
        except AssertionError:
                print("imag() test: FAILED.")

def test_get_r():
        global testTot, passed
        testTot += 1
        try:
                assert isClose(c1.get_r(), math.sqrt(2))
                assert isClose(c2.get_r(), math.sqrt(2) * 10 ** 10)
                assert isClose(c3.get_r(), math.sqrt(2) * 10 ** 10)
                assert isClose(c4.get_r(), 2.0)
                assert isClose(c5.get_r(), 4.0)
                passed += 1
                print("get_r() test: PASSED.")
        except AssertionError:
                print("get_r() test: FAILED.")

def test_get_phi_branch1():
        global testTot, passed
        testTot += 1
```

```
        try:
                assert isClose(c1.get_phi(), math.pi / 4)
                assert isClose(c2.get_phi(), math.pi / 4)
                assert isClose(c3.get_phi(), (-3.0 / 4.0) * math.pi)
                assert isClose(c4.get_phi(), -math.pi / 4)
                assert isClose(c5.get_phi(), (2.0 / 3.0) * math.pi)
                passed += 1
                print("get_phi() not (real < 0 and imag = 0) test: PASSED.")
        except AssertionError:
                print("get_phi() not (real < 0 and imag = 0) test: FAILED.")

def test_get_phi_branch2():
        global testTot, passed
        testTot += 1
        try:
                c1 = ComplexT(-1e10, 0.0)
                c2 = ComplexT(-math.sqrt(2), 0.0)
                assert isClose(c1.get_phi(), math.pi)
                assert isClose(c2.get_phi(), math.pi)
                passed += 1
                print("get_phi() (real < 0 and imag = 0) test: PASSED.")
        except AssertionError:
                print("get_phi() (real < 0 and imag = 0) test: FAILED.")

# test for exact equal.
def test_com_equal():
        global testTot, passed
        testTot += 1
        try:
                ct1 = ComplexT(1.4142135623730951, -1.4142135623730951)
                ct2 = ComplexT(math.sqrt(2), -math.sqrt(2))
                assert c1.equal(ct1) == False
                assert c2.equal(ct1) == False
                assert c3.equal(ct1) == False
                assert c4.equal(ct1) == True
                assert c5.equal(ct1) == False
                assert c1.equal(ct2) == False
                assert c2.equal(ct2) == False
                assert c3.equal(ct2) == False
                assert c4.equal(ct2) == True
                assert c5.equal(ct2) == False
                passed += 1
                print("equal() test: PASSED.")
        except AssertionError:
                print("equal() test: FAILED.")

def test_conj():
        global testTot, passed
        testTot += 1
        try:
                assert __appEqualCom__(c1.conj(), ComplexT(1.0,-1.0))
                assert __appEqualCom__(c2.conj(), ComplexT(1e10,-1e10))
                assert __appEqualCom__(c3.conj(), ComplexT(-1e10,1e10))
                assert __appEqualCom__(c4.conj(), ComplexT(math.sqrt(2),math.sqrt(2)))
                assert __appEqualCom__(c5.conj(), ComplexT(-2.0,-2 * math.sqrt(3)))
                passed += 1
                print("conj() test: PASSED.")
        except AssertionError:
                print("conj() test: FAILED.")

def test_add():
        global testTot, passed
        testTot += 1
        try:
                ct = ComplexT(math.sqrt(5), 10.982)
                ans1 = ComplexT(1.0 + math.sqrt(5),11.982)
                ans2 = ComplexT(1e10 + math.sqrt(5),1e10 + 10.982)
                ans3 = ComplexT(-1e10 + math.sqrt(5),-1e10 + 10.982)
                ans4 = ComplexT(math.sqrt(2) + math.sqrt(5),-math.sqrt(2) + 10.982)
                ans5 = ComplexT(-2.0 + math.sqrt(5),2 * math.sqrt(3) + 10.982)
                assert __appEqualCom__(c1.add(ct), ans1)
                assert __appEqualCom__(c2.add(ct), ans2)
                assert __appEqualCom__(c3.add(ct), ans3)
                assert __appEqualCom__(c4.add(ct), ans4)
                assert __appEqualCom__(c5.add(ct), ans5)
                passed += 1
                print("add() test: PASSED.")
        except AssertionError:
                print("add() test: FAILED.")
```

```
def test_sub():
        global testTot, passed
        testTot += 1
        try:
                ct = ComplexT(1e10, 2e10)
                ans1 = ComplexT(-9999999999.0,-19999999999.0)
                ans2 = ComplexT(0.0,-1e10)
                ans3 = ComplexT(-2e10,-3e10)
                ans4 = ComplexT(-9999999998.585787,-20000000001.414215)
                ans5 = ComplexT(-10000000002.0,-19999999996.5359)
                assert __appEqualCom__(c1.sub(ct), ans1)
                assert __appEqualCom__(c2.sub(ct), ans2)
                assert __appEqualCom__(c3.sub(ct), ans3)
                assert __appEqualCom__(c4.sub(ct), ans4)
                assert __appEqualCom__(c5.sub(ct), ans5)
                passed += 1
                print("sub() test: PASSED.")
        except AssertionError:
                print("sub() test: FAILED.")

def test_mult():
        global testTot, passed
        testTot += 1
        try:
                ct1 = ComplexT(2.6,3.8)
                ct2 = ComplexT(2e10,2e10)
                ct3 = ComplexT(1e9,2e8)
                ct4 = ComplexT(math.sqrt(6),2 * math.sqrt(2))
                ans1 = ComplexT(-1.2,6.4)
                ans2 = ComplexT(0.0,4.0 * 10 ** 20)
                ans3 = ComplexT(-8e18,-12e18)
                ans4 = ComplexT(2*math.sqrt(3) + 4,4 - 2*math.sqrt(3))
                assert __appEqualCom__(c1.mult(ct1), ans1)
                assert __appEqualCom__(c2.mult(ct2), ans2)
                assert __appEqualCom__(c3.mult(ct3), ans3)
                assert __appEqualCom__(c4.mult(ct4), ans4)
                passed += 1
                print("mult() test: PASSED.")
        except AssertionError:
                print("mult() test: FAILED.")

def test_recip():
        global testTot, passed
        testTot += 1
        try:
                ans1 = ComplexT(0.5,-0.5)
                ans2 = ComplexT(1.0 / (2 * 10 ** 10),-1.0 / (2 * 10 ** 10))
                ans3 = ComplexT(-1.0 / (2 * 10 ** 10),1.0 / (2 * 10 ** 10))
                ans4 = ComplexT(math.sqrt(2) / 4,math.sqrt(2) / 4)
                ans5 = ComplexT(-1 / 8.0, -math.sqrt(3) / 8)
                assert __appEqualCom__(c1.recip(), ans1)
                assert __appEqualCom__(c2.recip(), ans2)
                assert __appEqualCom__(c3.recip(), ans3)
                assert __appEqualCom__(c4.recip(), ans4)
                assert __appEqualCom__(c5.recip(), ans5)
                passed += 1
                print("recip() test: PASSED.")
        except AssertionError:
                print("recip() test: FAILED.")

def test_div():
        global testTot, passed
        testTot += 1
        try:
                ct1 = ComplexT(2.0, 3.0)
                ct2 = ComplexT(-3e10, 4e10)
                ct3 = ComplexT(-3e10, 4e10)
                ct4 = ComplexT(math.sqrt(3), 2 * math.sqrt(3))
                ct5 = ComplexT(6.0, 10.0)
                ans1 = ComplexT(5/13.0, -1/13.0)
                ans2 = ComplexT(1/25.0, -7/25.0)
                ans3 = ComplexT(-1/25.0, 7/25.0)
                ans4 = ComplexT(-math.sqrt(6)/15, -math.sqrt(6)/5)
                ans5 = ComplexT((5 * math.sqrt(3) - 3)/34, (5 + 3 * math.sqrt(3))/34)
                assert __appEqualCom__(c1.div(ct1), ans1)
                assert __appEqualCom__(c2.div(ct2), ans2)
                assert __appEqualCom__(c3.div(ct3), ans3)
                assert __appEqualCom__(c4.div(ct4), ans4)
                assert __appEqualCom__(c5.div(ct5), ans5)
                passed += 1
```

```
                print("div() test: PASSED.")
        except AssertionError:
                print("div() test: FAILED.")

def test_sqrt_branch1():
        global testTot, passed
        testTot += 1
        try:
                ca1 = ComplexT(math.sqrt((1 + math.sqrt(2))/2),
                        math.sqrt((-1 + math.sqrt(2))/2))
                ca2 = ComplexT(10**5 * math.sqrt((1 + math.sqrt(2))/2),
                        10**5 * math.sqrt((-1 + math.sqrt(2))/2))
                ca5 = ComplexT(1.0, math.sqrt(3))
                assert __appEqualCom__(c1.sqrt(), ca1)
                assert __appEqualCom__(c2.sqrt(), ca2)
                assert __appEqualCom__(c5.sqrt(), ca5)
                passed += 1
                print("sqrt() (imag > 0) test: PASSED.")
        except AssertionError:
                print("sqrt() (imag > 0) test: FAILED.")

def test_sqrt_branch2():
        global testTot, passed
        testTot += 1
        try:
                ca3 = ComplexT(10**5 * math.sqrt((-1 + math.sqrt(2))/2),
                        -10**5 * math.sqrt((1 + math.sqrt(2))/2))
                ca4 = ComplexT(math.sqrt((math.sqrt(2)+2)/2),
                        -math.sqrt((-math.sqrt(2)+2)/2))
                assert __appEqualCom__(c3.sqrt(), ca3)
                assert __appEqualCom__(c4.sqrt(), ca4)
                passed += 1
                print("sqrt() (imag < 0) test: PASSED.")
        except AssertionError:
                print("sqrt() (imag < 0) test: FAILED.")

def test_triangle_con_normal():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(10,1,2)
                passed += 1
                print("test_triangle_con() normal test: PASSED")
        except:
                print("test_triangle_con() normal test: FAILED")

def test_triangle_con_except():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(-1,10,9)
                print("test_triangle_con() except test: FAILED")
        except ValueError:
                passed += 1
                print("test_triangle_con() except test: PASSED")
        except:
                print("test_triangle_con() except test: FAILED")

# need to test is_Valid first.
def test_get_sides():
        global testTot, passed
        testTot += 1
        t1 = TriangleT(3,4,5)
        t2 = TriangleT(4,5,6)
        t3 = TriangleT(10,1,2)
        try:
                assert t1.get_sides() == (3,4,5)
                assert t2.get_sides() == (4,5,6)
                assert t3.get_sides() == (10,1,2)
                passed += 1
                print("get_sides() test: PASSED.")
        except AssertionError:
                print("get_sides() test: FAILED.")

def test_tri_equal_true():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3, 4, 5)
                t2 = TriangleT(4, 5, 6)
```

```
                t3 = TriangleT(5, 3, 4)
                assert t1.equal(t1) == True
                assert t1.equal(t3) == True
                assert t3.equal(t1) == True
                passed += 1
                print("equal() true test: PASSED.")
        except AssertionError:
                print("equal() true test: FAILED.")

def test_tri_equal_false():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3, 4, 5)
                t2 = TriangleT(4, 5, 6)
                t3 = TriangleT(5, 3, 4)
                assert t1.equal(t2) == False
                assert t2.equal(t3) == False
                assert t2.equal(t1) == False
                assert t3.equal(t2) == False
                passed += 1
                print("equal() false test: PASSED.")
        except AssertionError:
                print("equal() false test: FAILED.")

def test_perim():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3, 4, 5)
                t2 = TriangleT(10, 7, 8)
                t3 = TriangleT(11111,22222,33000)
                assert t1.perim() == 12
                assert t2.perim() == 25
                assert t3.perim() == 66333
                passed += 1
                print("perim() test: PASSED.")
        except AssertionError:
                print("perim() test: FAILED.")

def test_area():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3,4,5)
                t2 = TriangleT(5,6,7)
                t3 = TriangleT(1111111111,2222222222,3333333330)
                assert isClose(t1.area(), 6.0)
                assert isClose(t2.area(), 6 * math.sqrt(6))
                assert isClose(t3.area(), 111111110956944.45)
                passed += 1
                print("area() test: PASSED.")
        except AssertionError:
                print("area() test: FAILED.")

def test_is_valid():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3,4,5)
                t2 = TriangleT(10,2,1)
                t3 = TriangleT(2,4,6)
                assert t1.is_valid()
                assert not t2.is_valid()
                assert not t3.is_valid()
                passed += 1
                print("is_valid() test: PASSED.")
        except AssertionError:
                print("is_valid() test: FAILED.")

def test_tri_type_right():
        global testTot, passed
        testTot += 1
        try:
                t1 = TriangleT(3,4,5)
                t2 = TriangleT(4,5,3)
                t3 = TriangleT(5,3,4)
                assert t1.tri_type() == TriType.right
                assert t2.tri_type() == TriType.right
                assert t3.tri_type() == TriType.right
```

```
                passed += 1
                print("tri_type() right test: PASSED.")
        except AssertionError:
                print("tri_type() right test: FAILED.")

def test_tri_type_equilat():
        global testTot, passed
        testTot += 1
        try:
                t2 = TriangleT(1,1,1)
                assert t2.tri_type() == TriType.equilat
                passed += 1
                print("tri_type() equilat test: PASSED.")
        except AssertionError:
                print("tri_type() equilat test: FAILED.")

def test_tri_type_isosceles():
        global testTot, passed
        testTot += 1
        try:
                t3 = TriangleT(2,2,3)
                assert t3.tri_type() == TriType.isosceles
                passed += 1
                print("tri_type() isosceles test: PASSED.")
        except AssertionError:
                print("tri_type() isosceles test: FAILED.")

def test_tri_type_scalene():
        global testTot, passed
        testTot += 1
        try:
                t4 = TriangleT(10,11,12)
                assert t4.tri_type() == TriType.scalene
                passed += 1
                print("tri_type() scalene test: PASSED.")
        except AssertionError:
                print("tri_type() scalene test: FAILED.")

# Rationale:
# 1. Choose test cases that test more complex methods.
# 2. Choose test cases that can be verified in another way easily.
# 3. Choose different types of numbers: negative numbers, rational number,
#    irrational number etc.
# 4. Choose the numbers in the edge like large numbers or small numbers.

testTot = 0
passed = 0

print("------------------------------Tests for complex_adt.py------------------------")
test_complex_con()
c1 = ComplexT(1.0,1.0)
c2 = ComplexT(1e10,1e10)
c3 = ComplexT(-1e10,-1e10)
c4 = ComplexT(math.sqrt(2), -math.sqrt(2))
c5 = ComplexT(-2.0, 2 * math.sqrt(3))
test_real()
test_imag()
test_get_r()
test_get_phi_branch1()
test_get_phi_branch2()
test_com_equal()
test_conj()
test_add()
test_sub()
test_mult()
test_recip()
test_div()
test_sqrt_branch1()
test_sqrt_branch2()
print("------------------------------Tests for triangle_adt.py-----------------------")
test_triangle_con_normal()
test_triangle_con_except()
test_get_sides()
test_tri_equal_true()
test_tri_equal_false()
test_perim()
test_area()
test_is_valid()
test_tri_type_right()
test_tri_type_equilat()
```

```
test_tri_type_isosceles()
test_tri_type_scalene()
print("----------------------------------------Total----------------------------------")
print("Passed test number: ", passed)
print("Total test number: ", testTot)
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Steven K.
#  @brief Python module for Complex Numbers
#  @date 2021-01-16

import math


## @brief Class for Complex Numbers
class ComplexT:

        ## @brief Constructor for Complex Numbers
        #  @param x The real part of the complex number
        #  @param y The imaginary part of the complex number
    def __init__(self,x,y):
        self.x = x
        self.y = y

    ## @brief Returns the real part of a complex number
    #  @return Real part of the complex number
    def real(self):
        return self.x

    ## @brief Returns the imaginary part of a complex number
    #  @return Imaginary part of the complex number
    def imag(self):
        return self.y

    ## @brief Calculates the absolute value of a complex number
    #  @return Absolute value of a complex number
    def get_r(self):
        return math.sqrt(self.x**2 + self.y**2)

    ## @brief Calculates the phase of a complex number
    #  @throws Exception thrown if the phase of the complex number is not defined
    #  @return Phase of a complex number
    def get_phi(self):
        if (self.x < 0) and (self.y == 0):
            return math.pi
        elif (self.real() == 0) and (self.imag() == 0):
            raise Exception("Phase of this number is not defined")
        else:
            return 2 * math.atan(self.y / (self.get_r()+self.x))

    ## @brief Compares two complex numbers and checks if they are equal
    #  @param c Second complex number
    #  @return True if both complex numbers are equal, or False if they are not
    def equal(self,c):
        if (self.real() == c.real()) and (self.imag() == c.imag()):
            return True
        else:
            return False


    ## @brief Calculates the conjugate of a complex number
    #  @return Conjugate of the complex number
    def conj(self):
        return ComplexT(self.x, self.y*(-1))

    ## @brief Adds two complex numbers
    #  @param c Second complex number
    #  @return Result of the complex number addition
    def add(self,c):
        return ComplexT((self.x+c.real()),(self.y+c.imag()))

    ## @brief Subtracts two complex numbers
    #  @param c Second complex number
    #  @return Result of the complex number subtraction
    def sub(self,c):
        return ComplexT((self.x-c.real()),(self.y-c.imag()))

    ## @brief Multiplies two complex numbers
    #  @param c Second complex number
    #  @return Result of the complex number multiplication
    def mult(self,c):
        return ComplexT((self.x*c.real()-self.y*c.imag()),(self.x*c.imag()+self.y*c.real()))
```

```
    ## @brief Calculates the reciprocal of a complex number
    #  @throws Exception thrown if the complex number is zero
    #  @return Reciprocal of the complex number
    def recip(self):
        if (self.x == 0) and (self.y == 0):
            raise Exception("Division by zero is not possible")
        else:
            return ComplexT((self.x/(self.x**2+self.y**2)),(self.y/(self.x**2+self.y**2))*(-1))

    ## @brief Divides a complex number by another complex number
    #  @param c Denominator
    #  @throws Exception thrown if the denominator is zero
    #  @return Result of the complex number division
    def div(self,c):
        if (c.real() == 0) and (c.imag() == 0):
            raise Exception("Division by zero is not possible")
        else:
            return self.mult(c.recip())

    ## @brief Calculates the square root of a complex number
    #  @return Square root of the complex number
    def sqrt(self):
        if (self.y < 0):
            return
                ComplexT(math.sqrt(((self.x+self.get_r())/2)),(-1)*math.sqrt(((self.x*(-1)+self.get_r())/2)))
        else:
            return
                ComplexT(math.sqrt(((self.x+self.get_r())/2)),math.sqrt(((self.x*(-1)+self.get_r())/2)))
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Steven K.
#  @brief Python module for Triangles
#  @date 2020-01-20

import math
from enum import Enum

## @brief Enumeration class for all triangles types
class TriType(Enum):
    equilat = "Equilateral triangle"
    isoceles = "Isoceles triangle"
    scalene = "Scalene triangle"
    right = "Right-angle triangle"

## @brief Class for Triangles
class TriangleT:

    ## @brief Constructor for Complex Numbers
    #  @param a First side of the triangle
    #  @param b Second side of the triangle
    #  @param c Third side of the triangle
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    ## @brief Getter method for all the sides of a triangle
    #  @return Tuple containing 3 sides of the triangle
    def get_sides(self):
        return (self.a,self.b,self.c)

    ## @brief Constructor for Complex Numbers
    #  @param t Second triangle
    #  @return True if both Triangles are equal and False if not
    def equal(self,t):
        setA = set(self.get_sides())
        setB = set(t.get_sides())
        if (setA == setB):
            return True
        else:
            return False
```

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

## @brief Checks if the triangle is valid (mathematically possible)
#  @return True if it's a valid triangle and False if not
def is_valid(self):
    a = self.a
    b = self.b
    c = self.c
    if (a + b <= c) or (a + c <= b) or (b + c <= a):
        return False
    else:
        return True

## @brief Calculates the area of a triangle
#  @throws Exception thrown if the triangle is not valid
#  @return Area of the triangle
def area(self):
    if self.is_valid():
        semiP = self.perim()/2
        return math.sqrt(semiP*(semiP-self.a)*(semiP-self.b)*(semiP-self.c))
    else:
        raise Exception("Please enter a valid triangle")

## @brief Returns the type of the triangle
#  @throws Exception thrown if the triangle is not valid
#  @return TriType object corresponding to the type of triangle
def tri_type(self):
    a = self.a
    b = self.b
    c = self.c
    if self.is_valid():
        if (a**2 + b**2) == c**2:
            return TriType.right
        elif a == b == c:
            return TriType.equilat
        elif (a == b) or (b == c) or (a == c):
            return TriType.isoceles
        else:
            return TriType.scalene
    else:
        raise Exception("Please enter a valid triangle")
```