Assignment 1 Solution

Daniel Guoussev-Donskoi, guoussed

January 28, 2021

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

1 Assumptions and Exceptions

The following Assumptions were made:

- Inputs x and y were always assumed to be floating point numbers or integers when passed to the ComplexT constructor.(Essentially assuming the input was the correct type described in the Assignment outline).
- Methods such as equal were assumed to receive proper input of a ComplexT object.
- It was assumed for the method get_phi that the ComplexT object it was called upon represented a non-zero complex number, and was not equal to 0 + 0i as that would result in an undefined phase.
- It was assumed for the method recip that the ComplexT object it was called upon represented a non-zero complex number, and was not equal to 0 + 0i as that would result in an undefined reciprocal.
- It was assumed for the method div that the ComplexT object being used as the divisor represented a non-zero complex number, and was not equal to 0 + 0i as that would result in an undefined quotient.
- The methods for get_phi, get_r and sqrt all rounded the float returned by them to two decimal places, so the test cases for all three of said modules expected floats rounded to two decimal places.

- `TriangleT` assumes that the parameters passed to it's constructor are three positive integer values
- Method `equal` assumes that the `TriangleT` used for the comparison is a valid object of class `TriangleT`

2 Test Cases and Rationale

Due to assumptions for edge cases already being documented, test cases for edge cases regarding `ComplexT` object's were documented instead of implemented as exceptions, and thus test cases were not designed for inputs of zero for `ComplexT` objects or zero/negative inputs for `TriangleT`. Simple getter methods only used a couple tests to ensure they functioned correctly, as all they did was obtain certain values from an Object. However, more complicated methods that produced new Objects from a combination of objects, or methods that can produced many outputs(Such as `TriType`) had more test cases because of the large variation in potentially valid objects as seen in `ComplexT`. For this reason, I used both positive and wholly negative complex numbers, as well as ones that cancelled one another out entirely. For `TriangleT`, I carried out more tests when it came to `TriType` because one can easily test for all of the results that can be created by an enumerated type, as well as because there can be overlap in the results. This is because a triangle can be both right and isoceles simultaneously, which is why I tested for this specifically(So that it would return right over isoceles, as I documented in my assumption).

3 Results of Testing Partner's Code

All test cases were passed, save for those implemented for the modules `get_phi`, `get_r` and `sqrt`.The reason for these failures is the same in all cases, when I implemented those three modules, I implemented rounding, and rounded the floats returned by all three to two decimal places. My partner however, used the built in cmath module to return the whole float, and did not round, causing the values to not be equal. In retrospect, I could have implemented a ”close enough” case in my test case, where if the float returned by `get_phi` was reasonably close to my rounded answer, it would have passed.

4 Critique of Given Design Specification

The main and most obvious drawback of the Given Design Specification was it's vagueness and ambiguity, particularly regarding it's handling of edge cases. Due to the mathematical nature of the modules assigned to be written, many of the modules had edge cases that would return an undefined value, even if the input provided was ostensibly "valid". Another issue with the specification was that it was vague on how to handle improper input, and followups were required to determine whether it should be documented as an assumption, or thrown as an exception depending on the case. For both of these, I would recommend additional documentation of both these things in the Design Specification, as well as complete clarity on which approach to use in which case, to better suit the requirements. In regards to the Design's strengths however, I liked the modular and compartmentalised nature of the methods within the classes defined. It made it extremely easy to debug issues with the code, and made the code both readable and easy to understand as well as document. Each method has a clearly defined purpose and while they CAN be used in conjunction with one another, it remains clear which components belong to which method. This also actually made it quite easy to fake a rational design process, as a process from Design, to development plan to implementation is very easy to follow for each individual method. I also liked that all the methods implemented served a clear purpose, with each being extremely effective at producing reliability. I never once felt confused about what a particular method was meant to do generally, as a requirement. However, the requirements weren't exactly helpful for producing robustness or correctness, due to their lack of description for edge cases, how to deal with errors or incorrect input. I also liked the formality of the way the Design Specification laid out the Classes should be defined, as they directly stated what sort of input should be passed, and what should be returned for each and every method, making the methods both very uniform in how they were structured, but also very easy to utilise and effective at interacting with one another when used.

5 Answers to Questions

(a) For `ComplexT`, the clear getters were `real`, `imag`, `get_r`, `get_phi` and `sqrt`. For `TriangleT`, the clear getter was `get_sides`. All of these methods "got" or returned properties of a class instance and are thus "getters". However, I wouldn't really categorise any of the other methods as "setters" as setters are mutators, and the other methods return entirely new objects, rather than mutating existing ones.

(b) We could assign state variables for the type of triangle and it's area for TriangleT and could assign variables for the modulus and phase of ComplexT.

(c) I would say it does not make sense to implement a greater or less than method for ComplexT, both from a mathematical and programming perspective. From a programming perspective, the `equal` method does not compare mathematical size or order, it compares two objects to see if they have the same properties assigned to them. This is because from a mathematical perspective, complex numbers do not have a set order like natural numbers, they are a plane, not uni dimensional. The only way I could see referring to complex numbers as "greater" or "less than" one another is using the modulus and their magnitude from the point of origin, but I still wouldn't consider this worthy of implementing as it's own method due to this declaring a complex number with two negative components but a larger modulus to be "greater" than one with two positive components but a smaller modulus.

(d) It is perfectly possible to provide invalid integer values to TriangleT when forming a triangle, as according to the triangle inequality theorem, if the sum of two sides of a triangle are smaller than the remaining side, they cannot form a triangle(They will be too short). Thus, this is easy to test for, by checking using conditionals if any side is bigger than the sum of the other two. In this case, an external module could check to see if the object being initialised can in fact make a correct triangle, and raise an error if not.

(e) If TriangleT were to have a state variable for the type of triangle, said state variable could be modified, which is something you would absolutely not want to do accidentally as this could result in the software deciding that a triangle with three side lengths of length 6 each per say, was isoceles all of a sudden. Something we definitely do not want.

(f) Performance refers to how well software fulfills the external requirements set for it, in terms of speed and storage, specifically how "efficient" it is. Usability on the other hand, refers to the ability of others to use or easily employ software. The relationship here is clear, poor performance will lead to others being unable to use your software and decreased usability as a whole. Thus, good performance improves usability of a piece of software, while poor performance will do the opposite.

(g) In the overwhelming majority of software projects you will absolutely want to fake a rational design process, in order to have a better method of measuring progress, to improve the rational decisions you yourself will make as the project progresses and to provide better guidance as to what has to be done next. These components are so vital on larger projects that there is no larger project where you wouldn't want to fake a rational design process. The only case I can think of where you wouldn't bother is in an extremely small bit of code, or module, simply because it wouldn't be worth the time or effort. However even then, I would often recommend faking a rational design process, just to maintain good practices.

(h) To reuse software, is to use software that has already been made and re purpose it for another task, increasing it's reliability because it's individual components have already been shown to function correctly before, and the challenge becomes less building from scratch and more building on top of or redesigning that which has already been built.

(i) Most conventionally known programming languages(Python, C, Java) are ”higher level languages”, that is to say they’re not actually carrying out tasks themselves so much as they issue instructions to lower level, assembly languages who interpret those higher level programs as a series of smaller instructions, stored in registers. This assembly code is in turn, read by the OS and converted into machine code actually readable by the hardware which actually executes the tasks we need it to. This perfectly shows how higher level programming languages are best described as abstractions of what we want a machine to ”do”, and are passed down and converted to a form that a machine can actually understand. The comparison I would use, is an architect sketching out a building, and passing it down to a site supervisor who passes it down to the actual construction workers. Essentially every step is an ”abstract” of what the next stage is to do.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Daniel Guoussev-Donskoi
#  @title ComplexT
#  @date January 21st, 2020

## @brief This class represents a complex number
# @details This class represents a complex number as a an (x,y) pair of
# real and imaginary components respectively
# @details All modules which take an input outside the object itself, assume the
# input is another ComplexT object composed of two integers
import math
import cmath
class ComplexT:
  #@brief Constructor for ComplexT
  #@details Constructor accepts two parameters for real and imaginary parts
  #@param x, for real number part of a complex number
  #@param y, for imaginary part of a complex number
  def __init__(self, x, y):
    self.__x = x
    self.__y = y
  ##@brief Returns the real part of a complex number
  #@return x integer, for real number
  def real(self):
      return self.__x
  ##@brief Returns the imaginary part of a complex number
  #@return y integer, for imaginary number
  def imag(self):
      return self.__y
  ##@brief Returns the magnitude of the complex number(as int)
  #@return final integer, for modulus/magnitude of complex
  def get_r(self):
      first = (self.__x)**2
      second = (self.__y)**2
      modulus = (first + second)**0.5
      final = round(modulus,2)
      return final
  ##@brief Returns the phase of the complex number(In radians)
  #@return final integer, for phase of complex number in rad
  #@details Assumed x and y were not both zero, which would result in an undefined argument
  def get_phi(self):
      first = (self.__x)
      second = (self.__y)
      if (second == 0 ):
        return math.pi
      else:
        denominator = (((first**2)+(second**2))**0.5) + first
        semifinal = second / denominator
        final = round(semifinal,2)
        return final
  ##@brief Compares two ComplexT objects and returns True if their value is same
  #@param compare: A second ComplexT object being compared to the first
  #@return Returns a True or False boolean depending on if the objects match
  def equal(self,compare):
      if (self.__x == compare.__x) & (self.__y == compare.__y):
        return True
      else:
        return False
  #@brief Takes a ComplexT object and returns it's conjugate in the form of a ComplexT object
  #@return Returns a ComplexT object with values that are the conjugate of the original object
  def conj(self):
    final = ComplexT(self.__x,-self.__y)
    return final
  #@brief Takes a ComplexT object and adds it's values to those of another ComplexT object
  #@param other: A second ComplexT object, with an associated real and imaginary part
  #@return Returns a ComplexT object that is the sum of self and the passed parameter object
  def add(self,other):
    newx = self.__x + other.__x
    newy = self.__y + other.__y
    final = ComplexT(newx,newy)
    return final
  #@brief Takes a ComplexT object and subtracts it's values to those of another ComplexT object
  #@param other: A second ComplexT object, with an associated real and imaginary part
  #@return Returns a ComplexT object that is the difference of self and the passed parameter object
  def sub(self,other):
    newx = self.__x - other.__x
    newy = self.__y - other.__y
  final = ComplexT(newx,newy)
  return final
#@brief Takes a ComplexT object and another ComplexT object and returns a multiple of both
#complex numbers represented as a real and imaginary part
#@param other: A second ComplexT object, representing a complex number
#@return Returns a ComplexT object that represents the result of multiplying the two
def mult(self,other):
  a = self.__x* other.__x
  b = (self.__x*other.__y) + (self.__y*other.__x)
  c = (self.__y * other.__y * (-1))
  real = a + c
  imag = b
  final = ComplexT(real,imag)
  return final
#@brief Takes a complex number and returns it's reciprocal
#@return Returns a ComplexT object that represents the real and imag parts of the reciprocal
#@details Assumed that x and y were not both 0, As that would cause a zero division error
def recip(self):
  x = self.__x
  y = self.__y
  real = (x / ((x**2)+(y**2)))
  imag = -(y / ((x**2)+(y**2)))
  final = ComplexT(real,imag)
  return final
#@brief Takes a complex number and divides it by another complex number
#@param other: A second complex number, acting as the divisor
#@details Assumed the other variable was not 0 + 0i as that would cause a ZeroDivisionError
#@return Returns a ComplexT object that represents the real and imaginary parts
def div(self,other):
  a = self.__x
  b = self.__y
  c = other.__x
  d = other.__y
  real = ( ((a * c)+(b * d)) / ((c**2)+(d**2)) )
  imag = ( ((b * c)-(a * d)) / ((c**2)+(d**2)) )
  final = ComplexT(real,imag)
  return final
#@brief Takes a complex number and returns it's positive square root
#@return: Returns a ComplexT object that is the square root of the original object
def sqrt(self):
  x = self.__x
  y = self.__y
  root = cmath.sqrt(complex(x, y))
  real1 = root.real
  imag1 = root.imag
  real = round(real1,2)
  imag = round(imag1,2)
  return ComplexT(real,imag)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Daniel Guoussev-Donskoi
#  @brief A class that simulates a triangle, and several related methods
#  @details A class simulating a triangle is initialised, with 3 variables
#  @details The class assumes a, b and c are valid positive integer inputs
#  representing it's sides
#  @date January 21st, 2020
import math
from enum import Enum
#@brief A class that enumerates the 4 types of triangle in the requirements
#@details The 4 types are, equilateral, isoceles, scalene and right
class TriType(Enum):
  equilat = "equilat"
  isoceles = "isoceles"
  scalene = "scalene"
  right = "right"
class TriangleT:
  #@brief Constructor for TriangleT
  #@param a,b,c: 3 integer sides of triangle
  def __init__(self, a, b ,c):
    self.__a = a
    self.__b = b
    self.__c = c
  #@brief Function to return sides of a TriangleT object
  #@return final: Returns as tuple of integer sides of the TriangleT object
  def get_sides(self):
    a = self.__a
    b = self.__b
    c = self.__c
    final = (a,b,c)
    return final
  #@brief Function to check if two triangles are equal
  #@return Returns a boolean based on whether the two triangle objects are the same or not
  def equal(self,other):
    a = self.__a
    b = self.__b
    c = self.__c
    d = other.__a
    e = other.__b
    f = other.__c
    if ((a == d) & (b == e) & (c == f)):
        return True
    else:
        return False
  #@brief Function to return the perimeter of a triangle, as the sum of it's sides
  #@return Returns an integer in final, that is the sum of the sides of a TriangleT object
  def perim(self):
    a = self.__a
    b = self.__b
    c = self.__c
    final = a + b + c
    return final
  #@brief Function to calculate the area of a triangle using Heron's formula
  #@return Returns an integer in area, that is the calculated area of the TriangleT object
  def area(self):
      a = self.__a
      b = self.__b
      c = self.__c
      s= (a + b + c ) / 2
      areabeforesqrt = (s * (s - a) * (s - b) * (s - c))
      area = (areabeforesqrt ** 0.5)
      return area
  #@brief Function to determine if a triangle forms a triangle, using the inequality theorem
  #@return Returns True if the sides form a valid triangle, False if they do not
  def is_valid(self):
      a = self.__a
      b = self.__b
      c = self.__c
      if (a + b > c ) & ( a + c > b ) & ( b + c > a ):
          return True
      else:
          return False
  #@brief Function to determine which of the enumerated types a TriangleT type falls under
  #@details The triangle can be a TriType of types equilateral, right, isoceles or scalene
  #@details In the event a triangle is both right and isoceles, it is treated as right
  #@return Returns an enumerated TriType object
```

```python
def tri_type(self):
    a = self.__a
    b = self.__b
    c = self.__c
    if (a == b == c):
        return TriType.equilat
    elif ( a ** 2 + b **2 == c ** 2):
        return TriType.right
    elif ( a == b != c ) or (b == c != a) or (a == c != b):
        return TriType.isoceles
    else:
        return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Daniel Guoussev-Donskoi
#  @brief A driver file made to test modules belonging to the ComplexT and TriangleT classes
#  @date January 21st, 2020
from complex_adt import ComplexT
from triangle_adt import TriangleT, TriType
import math
##TESTS FOR COMPLEX BEGIN HERE
#Example ComplexT objects for tests
a = ComplexT(1.0, 2.0)
b = ComplexT(0.5, -0.5)
c = ComplexT(1.0, 2.0)
d = ComplexT(1.0,0.0)
#Test cases for ComplexT modules
#Only four test cases for real and imag, due to simplicity of modules
if (a.real() == 1.0):
    print("first real test passes")
else:
    print("first real test FAILS")
if (b.real() == 0.5):
    print("second real test passes")
else:
    print("second real test FAILS")
if (a.imag() == 2.0):
    print("first imag test passes")
else:
    print("first imag test FAILS")
if (b.imag() == -0.5):
    print("second imag test passes")
else:
    print("second imag test FAILS")
#Two test cases for get_r, one including negatives, one without
if (a.get_r() == 2.24):
    print("first get_r test passes")
else:
    print("first get_r test FAILS")
if (b.get_r() == 0.71):
    print("second get_r test passes")
else:
    print("second get_r test FAILS")

#Two test cases for get_phi, in two separate quadrants
if (a.get_phi() == 0.62):
    print("First get_phi test case passes")
else:
    print("First get_phi test case FAILS")
if (d.get_phi() == math.pi):
    print("Second get_phi test case passes")
#Test case for equal

if (a.equal(c)):
   print("Equal test passes")
else:
   print("Equal test fails")
if (a.equal(b)):
   print("Equal test FAILS")
else:
   print("Equal test passes")
#Test case for conj
conjtest = a.conj()
if (conjtest.equal(ComplexT(1.0,-2.0))):
   print("Conj test passes")
else:
   print("Conj test FAILS")

#Test cases for add, one producing a negative value one without
addtest = a.add(b)
if (addtest.equal(ComplexT(1.5,1.5))):
   print("First add test passes")
else:
   print("First add test FAILS")
addtest2 = a.add(ComplexT(- 2.0, -2.5))
if (addtest2.equal(ComplexT(-1.0,-0.5))):
   print("Second Add test passes")
else:
   print("Second Add test FAILS")
```

```
#Test cases for sub, one subtracting a negative

subtest = a.sub(b)
if (subtest.equal(ComplexT(0.5,2.5))):
  print("First sub test passes")
else:
  print("First sub test FAILS")
subtest2 = a.sub(ComplexT(- 2.0, -2.5))
if (subtest2.equal(ComplexT(3.0,4.5))):
  print("Second sub test passes")
else:
  print("Second sub test FAILS")

#Test case for mult

multtest=a.mult(b)
if (multtest.equal(ComplexT(1.5,0.5 ))):
  print("mult test passes")
else:
  print("mult test FAILS")

#Test case for recip
reciptest=a.recip()
if (reciptest.equal(ComplexT(0.2,-0.4))):
  print("recip test passes")
else:
  print("recip test FAILS")

#Test case for div
divtest=a.div(b)
if (divtest.equal(ComplexT(-1.0,3.0))):
  print("div test passes")
else:
  print("div test FAILS")

#Test case for sqrt
sqrttest=a.sqrt()
if (sqrttest.equal(ComplexT(1.27,0.79))):
  print("sqrt test passes")
else:
  print("sqrt test FAILS")

#TESTS FOR TRIANGLE BEGIN HERE
#Example objects for Triangle
triangle = TriangleT(3,4,5)
triangle2 = TriangleT(3,4,8)
triangle3 = TriangleT(3,3,3)
triangle4 = TriangleT(5,12,13)
#Test cases for TriangleT in triangle_adt.py

#Test cases for get_sides
if (triangle.get_sides()==(3,4,5)):
  print("First get_sides test passed")
else:
  print("First get_sides test FAILED")
if (triangle3.get_sides()==(3,3,3)):
  print("Second get_sides test passed")
else:
  print("Second get_sides test FAILED")

#Test cases for equal, one where the two are equal, one where the two are not
if (triangle.equal(triangle3) == False):
  print("First equal test passed")
else:
  print("First equal test FAILED")
if (triangle.equal(TriangleT(3,4,5)) == True):
  print("Second equal test passed")
else:
  print("Second equal test FAILED")

#Test cases for perim, using two separate triangles
if (triangle.perim()==12):
  print("First perim test case passed")
else:
  print("First perim test case FAILED")
if (triangle3.perim()==9):
  print("First perim test case passed")
else:
  print("First perim test case FAILED")
```

```
#Test cases for area using heron's formula.
if (triangle.area()==6):
  print("First area test case passed")
else:
  print("First area test case FAILED")
if (triangle4.area()==30):
  print("Second area test case passed")
else:
  print("Second area test case FAILED")

#Test cases for whether a triangle is possible using the inequality theorem
if (triangle.is_valid() == True):
  print("First is_valid test case passed")
else:
  print("First is_valid test case FAILED")
if (triangle2.is_valid() == False):
  print("Second is_valid test case passed")
else:
  print("Second is_valid test case FAILED")

#Test cases for tri_type, using multiple types of triangles
if (triangle.tri_type() == TriType.right):
  print("First tri_type test case passed")
else:
  print("First tri_type test case FAILED")
if (triangle3.tri_type() == TriType.equilat):
  print("Second tri_type test case passed")
else:
  print("Second tri_type test case passed")
if (TriangleT(2,2,5).tri_type() ==TriType.isoceles):
  print("Third tri_type test case passed")
else:
  print("Third tri_type test case failed")
if (TriangleT(2,7,8).tri_type() == TriType.scalene):
  print("Fourth tri_type test case passed")
else:
  print("Fourth tri_type test case FAILED")
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Prakarsh Kamal
#  @brief Complex numbers module (class)
#  @date 13 Jan 2021

import math
import cmath

## @brief This class represents complex numbers
#  @details This class represents complex numbers as (x, y) coordinates
#  where x is the real part and y is the imaginary part
class ComplexT:


        ## @brief init constructor for class
        #  @details initializing attributes (x, y) of ComplexT class
        #  @param x is real part of complex number
        #  @param y is imaginary part of complex number
        def __init__(self, x, y):
                self.x = x
                self.y = y



        ## @brief getter method real
        #  @return returns real part 'x' of complex number
        def real(self):
                return self.x



        ## @brief getter method imag
        #  @return returns imaginary part 'y' of complex number
        def imag(self):
                return self.y



        ## @brief getter method get_r
        #  @return returns magnitude of complex number utilizing math.sqrt()
        def get_r(self):
                return math.sqrt(self.x**2 + self.y**2)



        ## @brief getter method get_phi
        #  @details assuming that self.x > 0
        #  @return returns argument(angle/phase) of complex number utilizing cmath.phase)
        def get_phi(self):
                i = complex(self.x, self.y)
                return cmath.phase(i)



        ## @brief method checks current object with argument
        #  @param i is a new complex number with real and imaginary parts
        #  @return returns boolean value after comparing corresponding parts of current object and
            argument
        def equal(self, i):
                if (self.x == i.real()) and (self.y == i.imag()):
                        return True
                return False



        ## @brief method returning ComplexT object
        #  @return returns conjugate of complex number
        def conj(self):
                return ComplexT(self.x, -self.y)



        ## @brief method to calculate sum of current object and argument
        #  @param i is a new complex number with real and imaginary parts
        #  @return returns ComplexT object after adding corresponding parts of current object and
            argument
```

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


    ## @brief method to calculate difference of current object and argument
    #  @param i is a new complex number with real and imaginary parts
    #  @return returns ComplexT object after subtracting corresponding parts of current object and
        argument
    def sub(self, i):
        return ComplexT(self.x - i.real(), self.y - i.imag())


    ## @brief method to calculate multiplication of current object and argument
    #  @param i is a new complex number with real and imaginary parts
    #  @return returns ComplexT object after properly calculating corresponding parts of current
        object and argument
    def mult(self, i):
        return ComplexT((self.x * i.real() - self.y * i.imag()), (self.y * i.real() + self.x *
            i.imag()))


    ## @brief method to calculate reciprocal of current object
    #  @details assuming current object > 0 (denominator is not 0)
    #  @return returns ComplexT object after correctly computing reciprocal
    def recip(self):
        denom = (self.x) ** 2 + (self.y) ** 2
        f = self.x / denom
        g = -self.y / denom
        return ComplexT(f, g)


    ## @brief method to calculate division of current object and argument
    #  @details assuming that i > 0 (denominator is not 0)
    #  @param i is a new complex number with real and imaginary parts
    #  @return returns ComplexT object after properly calculating the division of current object
        and argument
    def div(self, i):
        mag = (i.real() ** 2 + i.imag() ** 2)
        r_1 = (self.x * i.real() + self.y * i.imag())
        r_2 = (self.y * i.real() - self.x * i.imag())
        ans_1 = r_1 / mag
        ans_2 = r_2 / mag
        return ComplexT (ans_1, ans_2)


    ## @brief method to calculate positive square root of current object
    #  @details current object is > 0 (not negative)
    #  @return returns ComplexT object utilizing cmath.sqrt()
    def sqrt(self):
        i = complex(self.x, self.y)
        j = cmath.sqrt(i)
        return ComplexT(j.real, j.imag)
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Prakarsh Kamal
#  @brief Triangle module (class)
#  @date 18 Jan 2021

import math
import enum

## @brief This class represents types of triangles
#  @details initializing 4 types of triangles using enum
class TriType(enum.Enum):
    equilat = 1
    isoceles = 2
    scalene = 3
```

```
        right = 4


## @brief This class represents triangles
#  @details This class represents triangles as integer arguments
#  where each argument is the length of the side of the triangle
class TriangleT:



        ## @brief init constructor for class
        #  @details initializing attributes (a, b, c) for TriangleT class
        #  For input to be valid triangle, all side lengths must be > 0
        #  @param a is length of side of triangle
        #  @param b is length of side of triangle
        #  @param c is length of side of triangle
        def __init__(self, a, b, c):
                self.a = a
                self.b = b
                self.c = c



        ## @brief getter method get_sides
        #  @details assuming input is a valid triangle
        #  @return returns tuple of integers where each element
        #  is the length of side of triangle
        def get_sides(self):
                return (self.a, self.b, self.c)



        ## @brief method checks current object with argument
        #  @details assuming that input is a valid triangle
        #  @param z is a new triangle with three sides
        #  @return returns Boolean value after comparing side lengths of current object and argument
        def equal(self, z):
                if ((self.a == z.a or self.a == z.b or self.a == z.c)
                and (self.b == z.b or self.b == z.a or self.b == z.c)
                and (self.c == z.c or self.c == z.a or self.c == z.b)):
                        return True
                return False



        ## @brief method calculating perimeter
        #  @details assuming that input is a valid triangle
        #  @return returning perimeter of triangle which is sum of all 3 sides as an integer
        def perim(self):
                return self.a + self.b + self.c



        ## @brief method calculating perimeter
        #  @details assuming that input is a valid triangle
        #  @return returning area of triangle implementing Heron's formula and utilizing math.sqrt()
        def area(self):
                semi = self.perim() / 2
                return math.sqrt( semi * (semi - self.a) * (semi - self.b) * (semi - self.c) )



        ## @brief method to validate if object is triangle
        #  @details assuming that input is a valid triangle
        #  @return returns Boolean value if sum of two sides is greater than third
        def is_valid(self):
                if ( ((self.a + self.b) > self.c) and ((self.b + self.c) > self.a) and ((self.a +
                     self.c) > self.b) ):
                        return True
                return False



        ## @brief method to check type of triangle
        #  @details assuming that input is a valid triangle
        #  @return returns type of triangle based on relative properties
        def tri_type(self):
                if (self.a == self.b and self.a == self.c and self.b == self.c):
                        return TriType.equilat
                elif ((self.a == self.b and self.a != self.c)
                or (self.b == self.c and self.b != self.a)
                or (self.c == self.a and self.c != self.b)):
                        return TriType.isoceles
                elif ((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
                else:
                        return TriType.scalene
```