2AA4 Assignment 1

Alan Scott

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

ComplexT

An assumption that was made in `ComplexT` involves the angle in `get_phi()`. It was assumed that the angle would fall within $[0, 2\pi]$ radians. The other possible assumptions were instead covered by exceptions. The first exception was for `get_phi()`, which in the case of real = imaginary = 0, would throw a ZeroDivisionError. In the event that this error occurs, the exception is caught, and the function will instead return a None value. The error catch applies to `recip()` as it will also throw a ZeroDivisionError in the case of real = imaginary = 0. This function will also return a None value.

TriangleT

For the triangle data type, it was assumed that the order of the sides would not matter. It was not assumed however that the independent side lengths would have to be valid (that is to say length ≥ 0), but rather the `is_valid()` would handle it. Another assumption made was for the `tri_type()` function. Since the function only returns a single type, it was assumed that the right triangle type should be prioritized over isosceles and scalene triangles in the case of overlaps, since right triangles are more specific.

2 Test Cases and Rationale

ComplexT

For the functions `real(), imag(), get_r(), equal(), conj(), add(), sub()` and `recip()`, there was only one test case tried (a generic answer) as there is very little complexity in the calculations. `get_phi()` was tested against a generic function, but also against a boundary case of $0 + 0i$ since this input could throw a ZeroDivisionError. This case checked for a return of **None**, which was the expected type in the case of an exception. The `mult()` function was tested against a generic answer, as well as a boundary case of multiplying by $0+0i$, which is expected to return 0 as a result of a zero multiplication, regardless of the current values of the real and imaginary values. `recip()` was tested on a generic answer, as well as on a `ComplexT` of $0 + 0i$, as this should cause a ZeroDivisionError. The test driver checked for a **None** return value, as this was the result of an error catch in the function. `div()` was put through a generic case as well as two special cases. The first case was a self-cancellation test, which, regardless of the values entered, should always return a value of 1 $(1 + 0i)$. The second was a case for a division by zero $(0 + 0i)$, which should cause a ZeroDivisionError. The test driver checks for a **None** return value, as the function returns this when an exception is thrown. The `sqrt()` function was tested against a generic tests, as well as two boundary cases. The first edge case tests the square root of a zero value, which should always return a zero value, as well as a test on just a regular number $(a + 0i)$, which should just return the square root of the component.

TriangleT

The functions `get_sides(), triangle_equals(), perim()` and `area()` were checked against generic values as the computations they performed were relatively non-complex. The `is_valid()` function was tested against a generic triangle, as well as two edge cases. The first case was for a triangle with a side length greater than the sum of the other two sides, and the other was against a triangle with negative sides. Since both of these triangles are invalid for different reasons, they should both return false. The `tri_type()` function was tested against all types of triangles for their corresponding outputs, as well as the two invalid triangles tested in `is_valid()`; since an invalid triangle cannot be considered to be any of the four types, it was expected that the function would return a **None** value.

3 Results of Testing Partner's Code

The first thing that happened when I ran the partner code was an AttributeError. This was a result of a misspelling of "isosceles" as "isoceles" in their `TriType` class, which had to be corrected.

ComplexT

Their `get_phi()` method failed my test on a technicality, as they chose to interpret an invalid angle as 0, whereas I chose to return a **None** value; the method still functions properly otherwise. The `mult()` function failed the test case for multiplying be zero, which should have returned a zero value $(0 + 0i)$, but instead returned a nonzero value. The `recip()` and `div()` tests failed for the same reason as `get_phi()`, as they both returned a different value to indicate an error than they test driver was expecting. `sqrt()` failed the regular test case due to a floating point inaccuracy, but otherwise functioned correctly.

TriangleT

The `get_sides()` test failed as the partner code returned the three side lengths as a list, rather than as a tuple, as was specified. The `get_area()` method failed as well, since the program did not halve the perimeter when using Heron's Formula. The `tri_type()` method failed to identify isosceles and equilateral triangles, instead assigning them as right triangles, despite them not having any overlap. This is likely due to the use of rounding in their test for right triangles. The `tri_type()` function also failed to check for invalid triangles before computing.

4 Critique of Given Design Specification

One critique of the design specification is the vagueness of what to do in the event of exceptions. The specifications never clearly stated what to do in the event of invalid computations, such as with calling `div()` with a zero value. I chose to return a **None** value, whereas my partner return a string error message instead. This discrepancy in approach lead to the test driver failing in many cases despite the fact the functions reached the same logical conclusion. It was also not made clear what assumptions could be made as well, and this difference in assumptions could lead to significant differences between peoples' codes.

5 Answers to Questions

(a) There are no setter functions in both `ComplexT` and `TriangleT`, as there exist no functions which mutate the values of any state variables, as all of the functions return values rather than modifying internal variables. `ComplexT` has two getter functions, `real()` and `imag()`, both of which return the value of their respective state variables. `TriangleT` has a single getter, `get_sides()`, which returns the three values of the side lengths.

(b) Two possible state variables for `ComplexT` are the radius `r` or the angle `phi`. Currently both of the possible state variables are represented by respective return methods. For the `TriangleT` class, a possible state variable could be the type of triangle (ie isosceles, right, etc) or a variable representing whether or not the triangle is valid which can be used in other functions.

(c) It would not make sense to add a greater/less than function for `ComplexT` since complex numbers operate on 2 coordinate axes (real and imaginary), rather than just a single one, which the greater/less operator works on.

(d) It is possible for the constructor to take in invalid values, such as having side lengths of 0 or less, or by having one side be longer than the sum of the other two. If the constructor detects invalid data, possibly by running the `is_valid()` function, it can return a message informing the user that it they have constructed an invalid triangle. The class can then refused to execute any functions, either by using a state variable for triangle validity, or by calling the `is_valid()` function before performin computations.

(e) A state variable for the type of triangle could be either good or bad depending on your implementation of the `TriangleT` class. On the plus side, it would save you from having to call the `tri_type()` function, as you would just have to access the variable through a getter method. One the flip side, if your implementation of `TriangleT` has mutators for the side lengths, the state variable would need to be updated every time a side length is modified, making it easier just to compute the type whenever it's needed by `tri_type()`.

(f) The usability of a product and its performance are directly linked; as performance increases, so does usability, as the user will generally prefer higher performance.

(g) The process of rational development is an inherently ideal, and will therefore almost always needed to be faked. The client rarely knows exactly what they want, and many details only become clear as development progresses. Combined with human error and miscommunication, the establishment of a rigid design process straight from the start *is* technically possible, yet extremely unlikely.

(h) Reusability is typically positively correlated with reliability, since in order for code to be reused, it must already be proven to be reliable.

(i) Programming levels perform abstractions on various levels. For example, with Object Oriented Programming Languages, an entire program can be abstracted into a data type (some kind of object), which contains various operations and sets of data. Programming languages also use functions/methods, which abstract a set of instructions into a single function/method call. The simplest syntactic operations (such as "x = 1 + 2") represent abstractions of many different mathematical operations on the hardware level.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author scotta30
#  @brief Contains a class for representing a complex number
#  @date 2021-01-13

import math

## @brief An ADT for complex numbers
# @details This class represents a complex number composed of real and
#          imaginary components.
class ComplexT:
    x, y = 0, 0

    ## @brief Constructor for ComplexT class
    # @details This constructor creates an object which represents a complex
    #          number in the form a + bi.
    # @param x The real value of the complex number
    # @param y The imaginary value of the complex number
    def __init__(self, x, y):
        self.x = x
        self.y = y

    ## @brief Returns the real value of the complex number
    # @details This function returns the stored real value of the complex number.
    # @return The real value of the number
    def real(self):
        return self.x

    ## @brief Returns the real value of the complex number
    # @details This function returns the stored imaginary value of the complex number.
    # @return The imaginary value of the number
    def imag(self):
        return self.y

    ## @brief Returns radius of the complex number
    # @details This function returns the polar length of the complex number
    # @return The radius of the complex number
    def get_r(self):
        return math.sqrt(self.x**2 + self.y**2)

    ## @brief Returns angle of the complex number
    # @details This function returns the angle that the number makes with the
    #          positive real axis. It assumes the angle is within 2 pi
    #          radians. The function assumes the complex number is greater
    #          than zero, and will return a None value in that case, otherwise
    #          it would cause a ZeroDivisionError.
    # @return The angle of the complex number
    # @throws ZeroDivisionError if the denominator comes out to zero
    #         (such as in the case of 0 + 0i)
    def get_phi(self):
        try:
            out = 2*math.atan(self.y/(math.sqrt(self.x**2 + self.y**2)+self.x))
            return out
        except ZeroDivisionError:
            print("Cannot divide by zero")
            return None

    ## @brief Checks if two numbers are equal
    # @details This function determines if two complex numbers are equal by
    #          checking their respective real and imaginary values against
    #          eachother
    # @returns True if the two numbers are equivalent
    # @param obj Complex number being checked against
    def equal(self, obj):
        return (obj.real() == self.x) and (obj.imag() == self.y)

    ## @brief Calculates the conjugate of the complex number
    # @details This function returns the reciprocal of the function by
    #          duplicating the current ComplexT, but with a negative imaginary
    # @return The reciprocal of the complex number
    def conj(self):
        return ComplexT(self.x, -self.y)

    ## @brief Adds two complex numbers
    # @details This function returns the sum of two complex numbers by
    #          by creating a new complex number with the real and imaginary
    #           components respectively summed
    # @return The sum of the complex numbers
    def add(self, obj):
        return ComplexT(self.x + obj.real(), self.y + obj.imag())

    ## @brief Subtracts two complex numbers
    # @details This function returns the difference of two complex numbers by
    #           by creating a new complex number with the real and imaginary
    #           components respectively subtracted
    # @return The difference of the complex numbers
    def sub(self, obj):
        return ComplexT(self.x - obj.real(), self.y - obj.imag())

    ## @brief Multiplies two complex numbers
    # @details This function returns the product of two complex numbers by
    #           by creating a new complex number with the respective components
    #           multiplied by expansion.
    # @return The product of the complex numbers
    # @param obj The second factor of the multiplication
    def mult(self, obj):
        r = self.x * obj.real() - self.y * obj.imag()
        i = self.x * obj.imag() + self.y * obj.real()
        return ComplexT(r, i)

    ## @brief Reciprocal function
    # @details This function returns the reciprocal of the current complex
    #           number. This method assumes that the length of the number
    #           is greater than zero, and will return a None value if it is,
    #           as a 0 length would cause a ZeroDivisionError.
    # @throws ZeroDivisionError when x = y = 0
    # @return The reciprocal of the complex number
    def recip(self):
        denom = self.x ** 2 + self.y ** 2
        try:
            out = ComplexT(self.x/denom, -self.y/denom)
            return out
        except ZeroDivisionError:
            print("Cannot divide by zero")
            return None

    ## @brief Divides two complex numbers
    # @details This function returns the quotient of two complex numbers by
    #           by creating a new complex number that is the result of the
    #           complex number multiplied by the reciprocal of the input.
    # @return The quotient of the complex numbers
    # @param obj The divisor
    def div(self, obj):
        rec = obj.recip()
        if (rec == None):
            return None
        return self.mult(rec)

    ## @brief Square root of the complex number
    # @details This function returns the square root of the complex number
    #           by computing the value of each respective component. The
    #           function returns the positive square root of the complex
    #           number, and omits the negative square root. Formula taken from
    #           http://stanleyrabinowitz.com/bibliography/complexSquareRoot.pdf
    # @return The square root of the complex number
    def sqrt(self):
        if (self.y == 0):
            return ComplexT(math.sqrt(self.x),0)
        r = self.get_r()
        real = math.sqrt((r + self.x)/2)
        imag = (abs(self.y)/self.y)*math.sqrt((r - self.x)/2)
        return ComplexT(real, imag)
```

G Code for triangle_adt.py

```
##  @file triangle_adt.py
#  @author Alan Scott
#  @brief Contains a class which represents a given triangle
#  @date 01/18/2020

import math
from enum import Enum


##  @brief Triangle defined by 3 side lengths
#  @details An ADT for a triangle represented by 3 side lengths
class TriangleT:
    a, b, c = 0, 0, 0

    ##  @brief Constructor for TriangleT
    #  @details This constructor creates a triangle from 3 given side lengths
    #  @param a An integer representing the length of the first side
    #  @param b An integer representing the length of the second side
    #  @param c An integer representing the length of the third side
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    ## @brief Returns the side lengths of the triangle
    #  @details This function returns the three side lengths as a tuple
    #  @return The three side lengths in a tuple
    def get_sides(self):
        sides = self.a, self.b, self.c
        return sides

    ## @brief Compares the current triangle and a given triangle
    #  @details This function checks if two side triangles are equivalent by
    #           checking if the sorted sets of their side lengths are equal
    #  @return True if the triangles are equal
    #  @param obj The triangle being compared to
    def equal(self, obj):
        objsides = obj.get_sides()
        sides = self.a, self.b, self.c
        return sorted(sides) == sorted(objsides)

    ## @brief Sums the side lengths of all 3 sides
    #  @details This function returns the sum of all of the side lengths
    #  @return The perimeter of the triangle
    def perim(self):
        return self.a + self.b + self.c

    ## @brief Computes the area of the triangle
    #  @details Computes the area of the triangle using Heron's theorem. Taken
    #           from: https://www.mathsisfun.com/geometry/herons-formula.html
    #  @return The area of the triangle
    def area(self):
        p = self.perim()/2
        return math.sqrt(p*(p - self.a)*(p - self.b)*(p - self.c))

    ## @brief Determines whether the given triangle is possible in Euclidian space
    #  @details This fuction tests if all the side lengths are greater than 0,
    #           and that no side length is greater than the sum of the other
    #           two.
    #  @return True if the triangle is physically possible, False if otherwise
    def is_valid(self):
        if (self.a + self.b < self.c):
            return False
        if (self.a + self.c < self.b):
            return False
        if (self.b + self.c < self.a):
            return False
        if (self.a > 0 and self.b > 0 and self.c > 0):
            return True
        return False

    ## @brief Determines the type of the triangle
    #  @details This function tests what type of triangle the current object
    #           is. If the triangle is not a possible triangle, it returns
    #           a None type. Due to only being able to return a single value,
    #           right triangles are prioritized over isosclese and scalene
    #           triangles in the case that it happens to be both.
    #  @return A TriType value representing the type of triangle
    def tri_type(self):
        if (self.is_valid() == False):
            return None
        if (self.a == self.b and self.a == self.c):
            return TriType.equilat
        sort_list = [self.a, self.b, self.c]
        sort_list.sort()
        if (sort_list[0]**2 + sort_list[1]**2 == sort_list[2]**2):
            return TriType.right
        if (self.a == self.b or self.b == self.c or self.a == self.c):
            return TriType.isosceles
        return TriType.scalene


## @brief TriType enumerate object type
#  @details This class is an enumerated list which represents one of four
#           different types of triangles.
class TriType(Enum):
    equilat = 1
    isosceles = 2
    scalene = 3
    right = 4
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Alan Scott
#  @brief Test driver for classes
#  @date

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

c1 = ComplexT(3,4)
c2 = ComplexT(3,4)
c3 = ComplexT(2,4)
c4 = ComplexT(0,0)
c5 = ComplexT(4,0)
c6 = ComplexT(8,-6)

## @brief Test for the real() function
#  @details This function checks the output of the real() function. Tests it
#           against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def real_test():
    if (c1.real() == 3):
        print("Passed real() test")
        return 1
    else:
        print("Failed real() test")
        return 0

## @brief Test for the imag() function
#  @details This function checks the output of the imag() function. Tests it
#           against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def imag_test():
    if (c1.imag() == 4):
        print("Passed imag() test")
        return 1
    else:
        print("Failed imag() test")
        return 0

## @brief Test for the get_r() function
#  @details This function checks the output of the get_r() function. Tests it
#           against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def get_r_test():
    if (c1.get_r() == 5):
        print("Passed get_r() test")
        return 1
    else:
        print("Failed get_r() test")
        return 0

## @brief Test for the get_phi() function
#  @details This function checks the output of the get_phi() function. Tests
#           the function against a correct answer, and checks the case for
#           which the method throws an error and returns None
#  @returns 1 if test is passed, 0 otherwise
def get_phi_test():
    did_pass = 1
    if (c1.get_phi() != 2*math.atan(4/(math.sqrt(3**2 + 4**2)+3))):
        print("Failed get_phi():regular test")
        did_pass = 0
    if (c4.get_phi() != None):
        print("Failed get_phi():divide by zero test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed get_phi() test")
        return 1
    return 0

## @brief Test for the equal() function
#  @details This function checks the output of the equal() function. Tests
#           the function against a correct answer and an incorrect answer.
#  @returns 1 if test is passed, 0 otherwise
def equal_test():
    did_pass = 1
    if (not c1.equal(c2)):
        print("Failed equal():equal test")
        did_pass = 0
    if (c1.equal(c6)):
        print("Failed equal():not equal test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed equal() test")
        return 1
    return 0

## @brief Test for the conj() function
#  @details This function checks the output of the conj() function. Tests
#           the function against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def conj_test():
    if (c1.conj().equal(ComplexT(3,-4))):
        print("Passed conj() test")
        return 1
    else:
        print("Failed conj() test")
        return 0

## @brief Test for the add() function
#  @details This function checks the output of the add() function. Tests
#           the function against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def add_test():
    if (c1.add(ComplexT(3,4)).equal(ComplexT(6,8))):
        print("Passed add() test")
        return 1
    else:
        print("Failed add() test")
        return 0

## @brief Test for the sub() function
#  @details This function checks the output of the sub() function. Tests
#           the function against the correct answer.
#  @returns 1 if test is passed, 0 otherwise
def sub_test():
    if (c1.sub(ComplexT(1,2)).equal(ComplexT(2,2))):
        print("Passed sub() test")
        return 1
    else:
        print("Failed sub() test")
        return 0

## @brief Test for the mult() function
#  @details This function checks the output of the mult() function. Tests the
#           function by multiplying be 0 and by multiplying a regular number.
#  @returns 1 if test is passed, 0 otherwise
def mult_test():
    did_pass = 1
    if (not c1.mult(ComplexT(0,0)).equal(ComplexT(0,0))):
        print("Failed mult():by zero test")
        did_pass = 0
    if (not c1.mult(c1).equal(ComplexT(-7,24))):
        print("Failed mult():normal test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed mult() test")
        return 1
    return 0

## @brief Test for the recip() function
#  @details This function checks the output of the recip() function. Tests
#           the function by finding the reciprocal of a regular number and
#           by finding the reciprocal of the zero value to test the function
#           error catch.
#  @returns 1 if test is passed, 0 otherwise
def recip_test():
    did_pass = 1
    if (not c3.recip().equal(ComplexT(1/10,-2/10))):
        print("Failed recip():normal test")
        did_pass = 0
    if (c4.recip() != None):
        print("Failed recip():divide by 0 test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed recip() test")
        return 1
    return 0

## @brief Test for the div() function
#  @details This function checks the output of the div() function. Tests the
#           function against a regular number, a self cancellation and a zero
#           division. Due to the presence of a floating point inaccuracy, the
#           components need only be withing a certain range of accuracy.
#  @returns 1 if test is passed, 0 otherwise
def div_test():
    did_pass = 1
    ct = ComplexT(1,-3)
    ct = ct.div(ComplexT(1,2))
    if (abs(-1 - ct.real()) > 0.00001 or abs (-1 - ct.imag()) > 0.00001):
        print("Failed div():normal test")
        did_pass = 0
    if (not c1.div(c1).equal(ComplexT(1,0))):
        print("Failed div():self-cancellation test")
        did_pass = 0
    if (c1.div(c4) != None):
        print("Failed div():divide by zero test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed div() test")
        return 1
    return 0

## @brief Test for the div() function
#  @details This function checks the output of the sqrt() function. Tests the
#           function against a regular number, against a 0 + 0i case and
#           against a regular non complex number (4+0i)
#  @returns 1 if test is passed, 0 otherwise
def sqrt_test():
    did_pass = 1
    if (not c6.sqrt().equal(ComplexT(3,-1))):
        print("Failed sqrt():regular test")
        did_pass = 0
    if (not c4.sqrt().equal(ComplexT(0,0))):
        print("Failed sqrt():empty complex number test")
        did_pass = 0
    if (not c5.sqrt().equal(ComplexT(2,0))):
        print("Failed sqrt():empty imaginary component test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed sqrt() test")
        return 1
    return 0

print("complex_adt: ---------------------")
passed = real_test() + imag_test() + get_r_test() + get_phi_test() + equal_test() + conj_test() +
    add_test() + sub_test() + mult_test() + recip_test() + div_test() + sqrt_test()
print("complex_adt: Passed " + str(passed) + " tests out of 12")

tr1 = TriangleT(3,4,5)
tr2 = TriangleT(1,1,1)
tr3 = TriangleT(1,1,10)
tr4 = TriangleT(4,6,9)
tr5 = TriangleT(0,1,1)
tr6 = TriangleT(2,2,1)

## @brief Test for the get_sides() function
#  @details This function checks the output of the get_sides() function.
#           Checks against a correct answer.
#  @returns 1 if test is passed, 0 otherwise
def get_sides_test():
    correct = 3, 4, 5
    if (tr1.get_sides() == correct):
        print("Passed get_sides() test")
        return 1
    else:
        print("Failed get_sides() test")
        return 0

## @brief Test for the triangle_adt equal() function
#  @details This function checks the output of the equal() function. Checks
#           against an identical triangle, an identical but rotated triangle
#           and a non-identitical triangle.
#  @returns 1 if test is passed, 0 otherwise
def triangle_equal_test():
    did_pass = 1
    if (not tr1.equal(TriangleT(3,4,5))):
        print("Failed equal() test")
        did_pass = 0
    if (not tr1.equal(TriangleT(5,4,3))):
        print("Failed equal() test")
        did_pass = 0
    if (tr1.equal(TriangleT(1,1,1))):
        print("Failed equal() test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed equal() test")
        return 1
    return 0

## @brief Test for the perim() function
#  @details This function checks the output of the perim() function. Tests
#           against correct answer.
#  @returns 1 if test is passed, 0 otherwise
def perim_test():
    if (tr1.perim() == 12):
        print("Passed perim() test")
        return 1
    else:
        print("Failed perim() test")
        return 0

## @brief Test for the area() function
#  @details This function checks the output of the area() function. Tests
#           against correct answer.
#  @returns 1 if test is passed, 0 otherwise
def area_test():
    if (tr1.area() == 6):
        print("Passed area() test")
        return 1
    else:
        print("Failed area() test")
        return 0

## @brief Test for the is_valid() function
#  @details This function checks the output of the is_valid() function. Tests
#           against a triangle with a too large side, one with a zero length
#           side, and a valid triangle.
#  @returns 1 if test is passed, 0 otherwise
def is_valid_test():
    did_pass = 1
    if (tr1.is_valid() == False):
        print("Failed is_valid():regular test")
        did_pass = 0
    if (tr3.is_valid() == True):
        print("Failed is_valid():large length test")
        did_pass = 0
    if (tr5.is_valid() == True):
        print("Failed is_valid():0 or less test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed is_valid() test")
        return 1
    return 0

## @brief Test for the is_valid() function
#  @details This function checks the output of the is_valid() function. Tests
#           against all types of triangles, and 2 triangles of invalid sides.
#  @returns 1 if test is passed, 0 otherwise
def tri_type_test():
    did_pass = 1
    if (tr1.tri_type() != TriType.right):
        print("Failed tri_type():right test")
        did_pass = 0
    if (tr2.tri_type() != TriType.equilat):
        print("Failed tri_type():equilateral test")
        did_pass = 0
    if (tr3.tri_type() != None):
        print("Failed tri_type():invalid side length test")
        did_pass = 0
    if (tr4.tri_type() != TriType.scalene):
        print("Failed tri_type():scalene test")
        did_pass = 0
    if (tr5.tri_type() != None):
        print("Failed tri_type():zero length test")
        did_pass = 0
    if (tr6.tri_type() != TriType.isosceles):
        print("Failed tri_type():isosceles test")
        did_pass = 0
    if (did_pass == 1):
        print("Passed tri_type() test")
        return 1
    return 0

print("triangle_adt: ----------------------------------")
passed = get_sides_test() + triangle_equal_test() + perim_test() + area_test() + is_valid_test() +
     tri_type_test()
print("triangle_adt: Passed " + str(passed) + " tests out of 6")
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Samia Anwar
#  @brief Contains a class to manipulate complex numbers
#  @Date January 21st 2021

import math
import numpy

## @brief An ADT for representing complex numbers
#  @details The complex numbers are represented in the form x + y*i
class ComplexT:
        ## @brief Constructor for ComplexT
        #  @details Creates a complext number representation based on given x and
        #                         y assuming they are always passed as real numbers. Real numbers
        #                         are in the set of complex numbers, therefore, y can be 0.
        #  @param x is a real number constant
        #  @param y is a real number coefficient of the square root of  -1.

        def __init__(self, x, y):
                self.x = x
                self.y = y

        ## @brief Gets the constant x from a ComplexT
        #  @return A real number representing the constant of the instance
        def real(self):
                return self.x

        ## @brief Gets the constant x from a ComplexT
        #  @return A real number representing the coefficient of the instance
        def imag(self):
                return self.y

        ## @brief Calculates the absolute value of the complex number
        #  @return The absolute value of the complex number as a float
        def get_r(self):
                self.abs_value = math.sqrt(self.x*self.x + self.y*self.y)
                return self.abs_value

        ## @brief Calculates the phase value of the complex number
        #  @details Checks for the location of imaginary number on the real-imaginary
        #                plane, and performs the corresponding quadrant calculation
        #  @return The phase of the complex number as a float in radians
        def get_phi(self):
                if self.x > 0:
                        self.phase = numpy.arctan(self.y/self.x)
                elif self.x < 0 and self.y >= 0 :
                        self.phase = numpy.arctan(self.y/self.x) + math.pi
                elif  self.x < 0 and self.y < 0:
                        self.phase = numpy.arctan(self.y/self.x) - math.pi
                elif self.x == 0 and self.y > 0:
                        self.phase = math.pi/2
                elif self.x == 0 and self.y < 0:
                        self.phase = -math.pi/2
                else:
                        self.phase = 0
                return self.phase

        ## @brief Checks if a different ComplexT object is equal to the current one
        #  @details Compares the real and imaginary compoenets of the two instances
        #  @param Accepts a ComplexT object, arg
        #  @return A boolean corresponding to whether or not the two specified
        #          objects are equal to one another, True for they are equal and False otherwise
        def equal(self, arg):
                self._argx = arg.real()
                self._argy = arg.imag()
                return self._argx == self.x and self._argy == self.y

        ## @brief Calculates the conjunct of the imaginary number
        #  @return A ComplexT Object corresponding to the conjunct of the specific instance

        def conj(self):
                return ComplexT (self.x, - self.y)

        ## @brief Adds a different ComplexT object to the current object
        #  @details Adds the real and imaginary components of the two instances
        #  @param Accepts a ComplexT object, num_add
```

```
    #  @return A ComplexT object corresponding to the sum of the real and imaginary
    #          and imaginary components
    def add(self, num_add):
            self.__newx = num_add.real() + self.x
            self.__newy = num_add.imag() + self.y
            return ComplexT (self.__newx, self.__newy)

    ## @brief Subtracts a different ComplexT object from the current object
    #  @details Individually subtracts the real and imaginary components of the two instances
    #  @param Accepts a ComplexT object, num_sub
    #  @return A ComplexT object corresponding to the difference of the real and imaginary
    #          and imaginary components
    def sub(self, num_sub):
            self.__lessx = self.x - num_sub.real()
            self.__lessy = self.y - num_sub.imag()
            return ComplexT (self.__lessx, self.__lessy)

    ## @brief Multiplies a different ComplexT object with the current object
    #  @details Arithmetically solved formula for (a + b*i) * (x + y*i) and seperated
    #                        the constant (a*x - y*b) and the coefficient (b*x + a*y)
    #  @param Accepts a ComplexT object, num_mult which acts as a multiplier (a + bi)
    #  @return A ComplexT object corresponding to the product of two multipliers
    def mult(self, num_mult):
            self.__multx = num_mult.real() * self.x - self.y * num_mult.imag()
            self.__multy = num_mult.imag() * self.x + self.real() * self.y
            return ComplexT (self.__multx, self.__multy)

    ## @brief Calculates the reciprocal or inverse of the complex number
    #  @details The formula was retrieved from www.suitcaseofdreams.net/Reciprocals.html
    #  @return A ComplexT object corresponding to the reciprocal of the current number
    def recip(self):
            if self.x == 0 and self.y == 0:
                    return "The reciprocal of zero is undefined"
            else:
                    self.__recipx = self.x / (self.x * self.x + self.y * self.y)
                    self.__recipy = - self.y / (self.x * self.x + self.y * self.y)
                    return ComplexT(self.__recipx, self.__recipy)

    ## @brief Divides a given complex number from the current number
    #  @details The formula was retrieved from
         www.math-only-math.com/divisio-of-complex-numbers.html
    #  @param An object of ComplexT which acts as the divisor to the current dividend
    #  @return A ComplexT Object corresponding to the quotient of the current number over the input
    def div(self, divisor):
            self.__divx = divisor.real()
            self.__divy = divisor.imag()
            if self.__divx == 0 and self.__divy == 0:
                    return "Cannot divide by zero"
            else:
                    return ComplexT ( (self.x*self.__divx + self.y*self.__divy)
                                                 / (self.__divx * self.__divx +
                                                     self.__divy*self.__divy),
                                                 (self.y * self.__divx - self.__divy * self.x)
                                                 / (self.__divx * self.__divx +
                                                     self.__divy*self.__divy))
    ## @brief Calculates the square root of the current ComplexT object
    #  @details The formula was retrieved from Stanley Rabinowitz's paper "How to find
    #           the Square Root of a Complex Number" published online, found via google search
    #  @return A ComplexT object corresponding to the square root of the current number
    def sqrt(self):
            self.__sqrtx = math.sqrt((self.x) + math.sqrt(self.x*self.x + self.y*self.y)) /
                 math.sqrt(2)
            self.__sqrty = math.sqrt(math.sqrt(self.x*self.x + self.y*self.y) - self.x) /
                 math.sqrt(2)
            return ComplexT (self.__sqrtx, self.__sqrty)
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Samia Anwar anwars10
#  @brief
#  @date January 21st, 2021
from enum import Enum
import math
```

```
## @brief An ADT for representing individual triangles
#  @details The triangle are represented by the lengths of their sides
class TriangleT:
        ## @brief Constructor for Triangle T
        #  @details Creates a representation of triangle based on the length of its sides,
        #           I have assumed the inputs to be the set of real numbers not including zero.
        #  @param The constructor takes 3 parameters corresponding to the three sides of a triangle
        def __init__(self, s1, s2, s3):
                self.s1 = s1
                self.s2 = s2
                self.s3 = s3
        ## @brief Tells the user the side dimensions of the triangle
        #  @return An array of consisting of the length of each side
        def get_sides(self):
                return [self.s1, self.s2, self.s3]

        ## @brief Tells the user if two TriangleT objects are equal to one another
        #  @param Accepts a TriangleT type to compare with the current values
        #  @return A boolean type true for the two are the same and false otherwise
        def equal(self, compTri):
                return set(self.get_sides()) == set(compTri.get_sides())

        ## @brief Tells the user the sum of all the sides of the triangle
        #  @return An num type representing the perimetre of the triangle
        def perim(self):
                return (self.s1 + self.s2 + self.s3)

        ## @brief Tells the user the area of the TriangleT referenced
        #  @return A float representing the are of the TriangleT referenced
        def area(self):
                if self.is_valid() :
                        return math.sqrt(self.perim() * (self.perim() - self.s1) * (self.perim() -
                             self.s2) * (self.perim() - self.s3) )
                else:
                        return 0

        ## @brief Tells the user if the triangle referenced is valid
        #  @details Determines the validity of the triangle based on the sides
        #  @return A boolean value which is true if the triangle is valid, false otherwise
        def is_valid(self):
                if (((self.s1 + self.s2) > self.s3) and ((self.s1 + self.s3) > self.s2) and ((self.s2
                     + self.s3) > self.s1)):
                        return True
                else:
                        return False
        ## @brief Tells the user one name for the type of triangle TriangleT referenced
        #  @details This program prioritises right angle triangle over the others, so
        #           if the triangle is right, it will give only a right angle result and
        #                      not isoceles or scalene.
        #  @return An instance of the TriType class corresponding to right/equilat/isoceles/or scalene
        def tri_type(self):
                if (round(math.sqrt(self.s1 * self.s1  + self.s2 * self.s2)) == round(self.s3)
                        or round(math.sqrt(self.s1 * self.s1 + self.s3 * self.s3)) == round(self.s2)
                        or round(math.sqrt(self.s3 * self.s3 + self.s2 * self.s2)) == round(self.s1)):
                        return TriType.right
                elif (self.s1 == self.s2 and self.s2 == self.s3):
                        return TriType.equilat
                elif(self.s1 == self.s2 or self.s1 == self.s3 or self.s2 == self.s3):
                        return TriType.isoceles
                else:
                        return TriType.scalene

## @brief Creates an enumeration class to store the type of triangle to be referenced by
#         tri_type method within TriangleT
class TriType(Enum):
        equilat = 1
        isoceles = 2
        scalene = 3
        right = 4
```