Assignment 1 Solution

Mehak Khan khanm294

17th January 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

Exceptions in `ComplexT`:

1. The method **recip()** throws a `ZeroDivisionError` if both the real and imaginary part of the complex number are 0. This is when the reciprocal is undefined.

2. The method **div()** throws a `ZeroDivisionError` if a complex number is being divided by 0, which is when both parts of a complex number are 0. This is when the result of the division would be undefined.

Assumptions in `ComplexT`:

1. The constructor assumes that the programmer will input the correct data type for the real and imaginary part values of the complex number, which in our case is `Float`. This is in accordance with Python's philosophy.

2. The **get_r()** method assumes that the domain of the argument of a complex number is (-pi, pi]

3. The **get_r()** method also assumes that the argument of a 0 number is 0.0, and not undefined in contrast.

4. The **sqrt()** method assumes that for calculating the sqaure root of a complex number, we can not have the imaginary part be equal to 0. If the imaginary part is 0,

the method assumes that the complex number is just a real number and calculates the square root of that number. If the real number is negative, the square root will be an imaginary number. The square root of 0 will be 0.

Exceptions in `TriangleT`:

1. The constrctor throws an `Exception` is the triangle side lengths used to create an instance of `TriangleT` do not form a valid triangle mathemtically and/or have negative or 0 lengths.

Assumption in `TriangleT`:

1. The constructor assumes that the programmer will provide the correct data type for the side lengths of the triangle, which in our case is an `Integer`. This is in accordance with Python's philosophy.

2. The **equal()** method assumes that the orientation of the triangle does not matter as long as both triangles have the same 3 side lengths.

3. The **tri_type()** method assumes the priority of triangle's types to be in the order: Equilateral, Isosceles, Right and Scalene respectively. This means that if a triangle is both right and scalene, the method will return type right.

2 Test Cases and Rationale

The test cases for both classes were approached to cover normal cases, boundary cases and cases with exceptions to ensure that all situations are covered for testing.

To ensure functionality of **complex_adt.py**, complex numbers were tested such that real and imaginary parts covered all combinations of the four quadrants. This way, the testing made sure that the class covers all possibilities of complex numbers. These tests were intended to cover all the normal cases as complex numbers can normally lie on either side of x and y axis. To cover boundary cases, which also in `ComplexT` often raised exceptions, complex numbers which are essentially the zero number were tested. With the real and imaginary part 0, methods such as **div()** and **recip()** raised an exception which was tested for. Some methods returned very specific values for when the imaginary part is 0, and these special cases were also covered for methods such as **sqrt()** and **get_phi()**.

To ensure functionality of **triangle_adt.py**, triangles were created such that any combination of integer side lengths can be tested. Invalid triangles were only tested when constructing them as this raised an exception, and triangles that do not exist could not be constructed. All other methods were then tested on normal cases as the constructor

covered the boundary and exception case. Triangles were also created such that they were either one or more of the following types: isosceles, right, scalane, equailateral. This ensured creating all types of triangles to make sure that the assumed priority is correctly met in the **tri_type()** method. Lastly, two triangles were also created so that their orientation differs but are equal to test that the assumption is met in the **equal()** method.

Therefore, the test cases in both classes covered all possible inputs for every method, and used assertions to make sure the expected result is equal to the actual result. The test driver also made use of a method that helped assert that floating point numbers are approximately equal. This is because floating points are hard to test for exact equality due to rounding off differences and limitations with Python's floating point arithmetic. There were also some extra try-except cases to test that exceptions are raised where they are expected. The above tests all passed in our **test_driver.py** and no problems surfaced through these test cases.

3 Results of Testing Partner's Code

My partner's code for ComplexT passed 9/12 tests from my test driver. The test cases failed for the following methods: **mult()**, **div()**, and **recip()**.

1. The test for **mult()** failed due to an error in my partner's code. The calculation for the imaginary part of the result of multiplication has a mathematical error.

2. The test for **div()** failed because my test_driver was expecting an Exception when a complex number is divided by 0. However, my partner's ComplexT prints a statement notifying the user of an undefined result instead of raising an Exception. This is due to the ambiguity in the design specifications that did not clarify whether the code expects an exception, a print statement, or simply as assumption. Due to this, my method and my partner's method are different in what path was chosen to handle this special case.

3. The test for the **recip()** failed because of the same reason, as my test_driver was expecting an Exception for the reciprocal of a zero number, however my partner's ComplexT prints a statement of the result being undefined instead of raisning an Exception.

My partner's code for TriangleT passed 4/7 tests. The test cases failed for the Constructor, **area()**, and **tri_type()**.

1. The test for Constructor failed due to the difference in our TriangleT. I raised an Exception in my constructor to prevent an invalid triangle from being created. My

partner's code does not check for invalidity in the constructor, and therefore her file did not meet my test case.

2. The test for **area()** failed due to mathematical errors in the calculation in my partner's method. The formula used to calculate area is incorrect. I believe my partner tried to use Heron's formula, which is what I used as well. However, she did not divide the perimeter by 2 in the formula which led to an incorrect result.

3. The test for **tri_type()** failed because of the different in assumption of priority of the triangle types. My TriangleT prioritizes in the respective order: equaliteral, isosceles, right, and scalene. My partner's TriangleT prioritizes right over any other type. This difference is also due to the ambiguity in the design specification.

4 Critique of Given Design Specification

The design specifications had some strenghts which included minimal design. Methods were specified such that each one only had a single, specific and non-redundant purpose. For example, the **equal()** method in `ComplexT` could have been redundant with two seperate methods that check for equality of imaginary part and equality of real part. However, the design does so with one method, increasing efficiency. The design also implements the good properties of modules. This enables seperation of concerns which in turn increases maintability of code. The modules have low coupling as they are not interdependent on each other. `ComplexT` and `TriangleT` can be independently changed without affecting each other. The modules also have high cohesion as everything in **complex_adt.py** is related to complex numbers and everything in **triangle_adt.py** is related to triangles. Moreover, the design specifications also implement an enumerated type in `TriangleT` which helps with readability of the code and also allows encapsulation of data as the enum type is hidden from the user. Lastly, the design specifications are clear about each method's return type and arguments which makes it easier to verify the code and the design specifications are abstract (implementation free) which allow the ADTs to be reused for future programs.

The design specifications also had some areas of improvements. These include that the specifications were not formal which led to ambiguity as natural language is ambiguous and is open to interpretation. The ambiguity in the design is also evident by the assumptions that had to be made, listed in section 1. Making the design specifications formal would be a way to improve this.

5 Answers to Questions

(a) In ComplexT, the getter methods were **real()**, **imag()**, **get_phi()**, **sqrt()**, **conj()**, **recip()** and **get_r()**. All these methods return a value based on the state variables. There are however no setters in ComplexT.

As for TriangleT, the getter methods were **get_sides()**, **perim()**, **area()**, **is_valid()** and **tri_type()**. All these methods are also using the state variables of triangle class and returning a value. There are also no setters in TriangleT.

(b) As the specification did not specify state variables for either ADT, there could be more possibilities for state variables.

In ComplexT, I implemented the real and imaginary part to be the state variables. However, phase (argument) of a complex number and absolute value of a complex number could be also be two other options for state variables. These are calculated in our **get_phi()** and **get_r()** methods respectively.

In TriangleT, I implemented the three side lengths to be the state variables. However, perimeter and area of a triangle are two other options for the state variables. These are calculated in out **perim()** and **area()** methods respectively.

(c) Complex numbers can be compared for equality by checking the real and imaginary part of two complex numbers. However, complex numbers do not follow a natural linear ordering. This means that having methods for greater than and less than for `ComplexT` would not make sense, and therefore should not be implemented.

(d) It is possible that the three integer inputs to the constructor of TriangleT will not form a valid triangle geometrically. In this case, the constructor should raise an exception as all other methods would be calculations on a triangle that cannot exist, which does not make sense. If the constructor does allow an invalid triangle to be built, every method would have to check for invalidity of triangle, resulting in redundancy. A better approach would be to check for invalidity in the constructor once and not allow an invalid object to be built. Therefore, the constructor should not construct an invalid triangle and raise a `ValueError` instead. My current `TriangleT` raises an `Exception` when an invalid object is created.

(e) The TriangleT could have the type of triangle as a state variable. This, however, is not a good idea because the user can incorrectly enter the type of triangle which does not represent the triangle they created. This can lead to an invalid type attached to the triangle object. Moreover, one triangle can be more than a single type and leaving that decision up to the user is not smart as either they have no assumed priority of

the types of triangle, or their assumption does not match the one the programmer implements. This can lead to inconsistency in the software.

(f) Usability in software is the quality of the user's experience with a program in relation to effectiveness, efficiency and satisfaction. This depends on the context of use of a software. Performance, on the other hand, refers to the the amount of time and speed a software requires. These two software qualities are related as performance affects usability. For example, a software with poor performance would require more time and memory to run, which would result in the user having to wait longer periods of time to use the product. This reduces usability as the product is less satisfactory. Therefore, poor performance can in turn reduce the usability of a software.

(g) A rational design process would be going from your problem to development to requirements to design and documentation to code and finally to a V&V Report. However, according to *Parnas* and *Clements*, there's rarely ever any software projects that proceed "rationally". The reasons for these are that more often than not teams who are responsible for software projects do not know the exact requirements of their program. Even if they think they know, many details only start surfacing during the implementation process which would require the programmer to go back and change the design specifications. Therefore, most design processes are "faked" to be rational as programmers often backtrack to change their design due to errors that are comprehended later in the design process.

(h) Reusability can arguably increase the reliability of a software product. This is because reusable products often go through more testing and careful designing, and are used more extensively. This would cause the program to also be more reliable as the probability of being free of failure will be higher in a more heavily tested program.

(i) Programming languages are abstractions built on top of hardware. An example of this would be the development of programming languages. Programming languages, such as Java, C++ and Python (high level languages), are developed with abstraction from machine language to assembly language to high level language. Machine language is directly executed by a computer, as it composes of binary patterns and is only understood by the computer. Assembly language, although not directly executed, is machine specific and understood by the processor as it directly accesses registers or memory locations on your system. Assembly language is a low-level language and is easier to understand as it uses some alphabets, however it needs an assembler to be converted to machine code. Further abstractions are made as high-level languages are machine independent and are easiest to interpret for humans. These languages can run on different processors and monitors. High-level languages are converted

to assembly language through a compiler. Therefore, programming languages are layers of abstractions over hardware. These abstractions, from machine language to assembly language to high level language, make it easier for humans to create software in a language that is closer to natural language and hides the complexity of machine specific commands. The further abstractions also make a language less direct with a specific machine, and more portable. Examples of hardware abtraction layers, which help hide the differences in hardware from the operating system to allow code to run on different hardwares, include Microsoft Windows which is portable to different processors. Android also has a HAL to develop forward compatible software.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Mehak Khan
#  @brief Contains a class for working with complex numbers
#  @date 12th January 2020

import math

## @brief An abstract data type for complex numbers
#  @details A complex number is composed of a real part, and an imaginary part

class ComplexT:

## @brief Constructor for ComplexT
#  @details Creates instance of a ComplexT based on real part and imaginary part of the number.
     Assumes that user will provide the correct data type for the imaginary and real part.
#  @param x Floating point representing real part of number
#  @param y Floating point representing imaginary part of number

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

## @brief Gets the real part of the complex number
#  @return Floating point representing the real part of the complex number

  def real (self):
    return self.x

## @brief Gets the imaginary part of the complex number
#  @return Floating point representing the imaginary part of the complex number

  def imag (self):
    return self.y

# Formula from:
     https://www2.clarku.edu/faculty/djoyce/complex/abs.html#:~:text=For%20a%20complex%20number%20z,on%20the%20real%20numbe
## @brief Calculates the absolute value of the complex number
#  @return Floating point representing the absolute value of the complex number

  def get_r (self):
    return math.sqrt(self.x*self.x + self.y*self.y)

# Formula from: https://gubner.ece.wisc.edu/notes/MagnitudeAndPhaseOfComplexNumbers.pdf
## @brief Calculates the argument of the complex number
#  @details The argument of a complex number is calculated using tan inverse. Assume the domain of
     this argument is (-pi, pi]. If the real part is 0, ZeroDivisionError is handled and the argument
     is pi/2 if the imaginary part is positive, and -pi/2 if the imaginary part is negative. Assume
     that the argument of 0 is 0.0.
#  @return Floating point representing the argument of the complex number in radians

  def get_phi (self):
    try:
      if self.x > 0:
        return math.atan(self.y/self.x)
      elif (self.x < 0 and self.y >= 0):
        return math.atan(self.y/self.x) + math.pi
      else:
        return math.atan(self.y/self.x) - math.pi

    except ZeroDivisionError:
      if (self.y > 0):
        return math.radians(90)
      elif (self.y < 0):
        return math.radians(90)*-1
      else:
        return 0.0

## @brief Checks equality of two complex numbers
#  @details Two complex numbers are equal if their real and imaginary part are the same
#  @param c The complex number to compare equality with
#  @return Boolean value indicating if the complex numbers are equal

  def equal(self, c):
    return (self.x == c.real()) and (self.y == c.imag())
```

```
# Formula from:
    https://www.mathcentre.ac.uk/resources/sigma%20complex%20number%20leaflets/sigma-complex6-2009-1.pdf
## @brief Calculates complex conjugate of current complex number
#  @return A complex number that is the cojugate

  def conj(self):
    return ComplexT(self.x, -1*self.y)

# Formula from: https://byjus.com/complex-number-formula/
## @brief Performs addition of two complex numbers
#  @param c The complex number to add to the current complex number
#  @return A complex number that is the result of the addition of the two complex numbers

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

# Formula from: https://byjus.com/complex-number-formula/
## @brief Subtracts a complex number from the current complex number
#  @param c The complex number that is subtracted from the current complex number
#  @return A complex number that is the result of the subtraction of the two complex numbers

  def sub(self, c):
    return ComplexT(self.x - c.real(), self.y - c.imag())

# Formula from: https://byjus.com/complex-number-formula/
## @brief Multiplies two complex numbers
#  @param c The complex number that is multiplied with the current complex number
#  @return A complex number that is the result of the multiplication of the 2 complex numbers

  def mult(self, c):
    return ComplexT(self.x*c.real() - self.y*c.imag(), self.x*c.imag() + self.y*c.real())

# Formula from: https://tinyurl.com/yxf3zmfr
## @brief Calculates reciprocal of complex number
#  @details The calculation uses division by summation of imaginary part squared and real part
    squared. If both values are 0, the reciprocal is undefined.
#  @throws ZeroDivisionError If both real and imaginary parts are 0, the reciprocal is undefined.
#  @return A complex number that is the resulting reciprocal

  def recip(self):
    if (self.x == 0 and self.y == 0):
      raise ZeroDivisionError("Undefined")
    else:
      return ComplexT(self.x/(self.x*self.x + self.y*self.y), -1*self.y/(self.x*self.x +
          self.y*self.y))

# Formula from: https://byjus.com/complex-number-formula/
## @brief Divides current complex number by given complex number
#  @throws ZeroDivisionError If complex number is being divided by a complex number with both real and
    imaginary parts equal to 0
#  @param c The complex number that is the divisor
#  @return A complex number that is the result of the division of the 2 complex numbers

  def div(self, c):
    if (c.real() == 0 and c.imag() == 0):
      raise ZeroDivisionError("Undefined")
    else:
      return ComplexT((self.x*c.real() + self.y*c.imag())/(c.real()*c.real() + c.imag()*c.imag()),
          (self.y*c.real() - self.x*c.imag())/(c.real()*c.real() + c.imag()*c.imag()))

# Formula from: https://en.wikipedia.org/wiki/Complex_number#Square_root
## @brief Calculates the positive square root of current complex number
#  @Details The calculation of the positive square root of a complex number requires that the
    imaginary part cannot be 0. If the imaginary part is 0, assume the number is a normal number and
    take the square root of it. If the number is negative, the square root will be an imaginary
    number.
#  @return A complex number that is the resulting square root

  def sqrt(self):
      if (self.y == 0):
        if (self.x > 0):
          return ComplexT(math.sqrt(self.x), 0.0)
        elif (self.x < 0):
          return ComplexT(0.0, math.sqrt(-self.x))
        else:
          return ComplexT(0.0, 0.0)
      else:
        real = math.sqrt((self.x + self.get_r())/2)
        imaginary = self.y/abs(self.y) * math.sqrt((-1*self.x + self.get_r())/2)
        return ComplexT(real, imaginary)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Mehak Khan
#  @brief Contains a class for working with triangles
#  @date 13 January 2021

import math
from enum import Enum, auto

## @brief An enumerated type for type of triangle
#  @details A triangle is either one of the following types: isosceles, right, scalene or equilateral.

class TriType(Enum):
  equilat = auto()
  isosceles = auto()
  right = auto()
  scalene = auto()

## @brief An abstract data type for triangles
#  @details A triangle is composed of three side lengths

class TriangleT:

## @brief Constructor for TriangleT
#  @details Creates instance of a ComplexT based on three side lengths. Assume that the type of side
    lengths provided will be Integer.
#  @throws Exception If the triangle lengths do not form a valid triangle or have negative or 0
    lengths.
#  @param x Integer representing side 1 of a triangle
#  @param y Integer representing side 2 of a triangle
#  @param z Integer representing side 3 of a triangle

  def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z
    if (self.is_valid() == False or self.x < 1 or self.y < 1 or self.z < 1 ):
      raise Exception("Invalid Triangle!")

## @brief Gets the side lengths of a triangle
#  @return Tuple of the three side lengths of the triangle

  def get_sides(self):
    return (self.x, self.y, self.z)

## @brief Checks equality of two triangles
#  @details Two trianges are equal if they have the same side lengths. Assume the orientation of
    triangle does not matter.
#  @param t TriangleT The triangle to compare equality with
# @return Boolean indicating if the triangles are equal

  def equal(self, t):
    sorted_self = sorted(self.get_sides())
    sorted_t = sorted(t.get_sides())
    return (sorted_self == sorted_t)

## @brief Calculates the perimeter of a triangle
#  @return Integer representing the perimeter value of the triangle

  def perim(self):
    return self.x + self.y + self.z

#From:
    https://en.wikipedia.org/wiki/Heron%27s_formula#:~:text=In%20geometry%2C%20Heron's%20formula%20(sometimes,distances%20i
## @brief Calculates the area of a triangle
#  @details The area is calculated with the three sides of the triangle using Heron's formula
#  @return Float representing the area of the triangle

  def area(self):
    p = self.perim()/2
    return math.sqrt(p*(p-self.x)*(p-self.y)*(p-self.z))

# From:
    https://www.geeksforgeeks.org/check-whether-triangle-valid-not-sides-given/#:~:text=Approach%3A%20A%20triangle%20is%20v
## @brief Calculates if the sides of the current triangle form a valid triangle
#  @details A valid triangle is formed when the sum of any two sides is greater than the third side.
#  @return Boolean indicating the validity of the triangle
```

```
  def is_valid(self):
    return (self.x + self.y > self.z and self.x + self.z > self.y and self.y + self.z > self.x)

## @brief Calculates the type of the current triangle
#  @details A triangle is equilateral if all the sides have the same side length, isosceles if two
     sides have the same length, right is the sum of the square of two side lengths equals the square
     of the third side length, and scalene if it meets none of these and has 3 different side lengths.
     Assume the priority of triangle types is in the order Equilateral, Isosceles, Right and Scalane.
#  @return TriType The type of triangle - isosceles, scalene, right or equilateral

  def tri_type(self):
    if self.x == self.y and self.x == self.z and self.y == self.z:
      return TriType.equilat
    elif (self.x == self.y or self.x == self.z or self.y == self.z):
      return TriType.isosceles
    elif self.x*self.x + self.y*self.y == self.z*self.z or self.y*self.y + self.z*self.z ==
        self.x*self.x or self.z*self.z + self.x*self.x == self.y*self.y:
      return TriType.right
    else:
      return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Mehak Khan
#  @brief Tests for complex_adt.py and triangle_adt.py
#  @date 14 January 2021

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

#ComplexT used for testing
testComplex = ComplexT(2.56,5.34)
negativeImag = ComplexT(5.33, -5.05)
negativeReal = ComplexT(-4, 7.0)
negativeComplex = ComplexT (-7.33, -33.56)
zeroComplex = ComplexT(0.0,0.0)
zeroReal = ComplexT(0.0, 44.0)
zeroRealNeg = ComplexT(0.0, -100.3)
zeroImag = ComplexT(8.5, 0.0)
zeroImagNeg = ComplexT(-4.7, 0.0)

# Keeping track of total tests and tests passed
totTest = 0
passed = 0

#Testing approximate equality due to Python floating point arithmetic
#From: https://www.python.org/dev/peps/pep-0485/#proposed-implementation
def isClose(a, b, rel_tol = 1e-05, abs_tol = 0.0):
  return abs(a - b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)

#Test the real number getter
def test_real():
  global totTest, passed
  totTest += 1

  try:
    expected_real = 2.56
    #TEST CASE 1: get real
    assert (testComplex.real() == expected_real)
    passed += 1
    print ("real() test passed")
  except AssertionError:
    print ("real() test failed")



#Test the imaginary number getter
def test_imag():
  global totTest, passed
  totTest += 1

  try:
    expected_imag = 5.34
    #TEST CASE 1: get imaginary
    assert(testComplex.imag() == expected_imag)
    passed += 1
    print("imag() test passed")
  except AssertionError:
   print("imag() test failed")

#Test the absolute value calculation. Use isClose.
def test_get_r():
  global totTest, passed
  totTest += 1

  try:
    #TEST CASE 1: get absolute value with zero real part
    expected_answer = 44.0
    assert(zeroReal.get_r() == expected_answer)

    #TEST CASE 2: get absolute value normal
    expected_ans = 5.9219254
    assert isClose(testComplex.get_r(), expected_ans)

    passed += 1
    print("get_r() test passed")
  except AssertionError:
```

```
    print("get_r() test failed")

#Test the argument calculation. Use isClose to test.
def test_get_phi():
  global totTest, passed
  totTest += 1

  try:

    #TEST CASE 1: Test negative real part and 0 imaginary part
    expected_and_zeroImagNeg = 3.14159265359
    assert isClose(zeroImagNeg.get_phi(), expected_and_zeroImagNeg)

    #TEST CASE 2: Test negative real part and negative imaginary part
    expected_ans_negativeComplex = -1.78583409192
    assert isClose(negativeComplex.get_phi(), expected_ans_negativeComplex)

    #TEST CASE 3: Test positive real part and 0 imaginary part
    expected_ans_zeroImag = 0.0
    assert isClose(zeroImag.get_phi(), expected_ans_zeroImag)

    #TEST CASE 4: Test zero complex number
    expected_ans_zeroComplex = 0.0
    assert isClose(zeroComplex.get_phi(), expected_ans_zeroComplex)

    #TEST CASE 5: Test zero real part with positive imaginary part
    expected_ans_zeroReal = 1.57079632679
    assert isClose(zeroReal.get_phi(), expected_ans_zeroReal)

    #TEST CASE 6: Test zero real part with negative imaginary part
    expected_ans_zeroRealNeg = -1.57079632679
    assert isClose(zeroRealNeg.get_phi(), expected_ans_zeroRealNeg)

    #TEST CASE 7: Test non zero real part complex number
    expected_ans_negativeImag = -0.758429752
    assert isClose(negativeImag.get_phi(), expected_ans_negativeImag)

    passed+= 1
    print("get_phi() test passed")

  except AssertionError:
    print("get_phi() test failed")



#Test equality of two complex numbers
def test_equal():

  global totTest, passed
  totTest += 1

  try:
    #TEST CASE 1: Equal numbers
    assert(zeroImag.equal(zeroImag))

    #TEST CASE 2: Unequal numbers
    assert(not zeroImag.equal(zeroReal))

    passed+=1
    print("equal() test passed")
  except AssertionError:
    print("equal() test failed")

#Test the conjugate of a complex number calculation
def test_conj():
  global totTest, passed
  totTest += 1

  try:
    #TEST CASE 1: Test positive imaginary number
    expected_ans_real = 2.56
    expected_ans_imag = -5.34
    assert(testComplex.conj().real() == expected_ans_real )
    assert(testComplex.conj().imag() == expected_ans_imag )

    #TEST CASE 2: Test negative imaginary number
    expected_ans_real = 5.33
    expected_ans_imag = 5.05
    assert(negativeImag.conj().real() == expected_ans_real )
    assert(negativeImag.conj().imag() == expected_ans_imag )
```

```
        passed +=1
        print("conj() test passed")

    except AssertionError:
        print("conj() test failed")

#Test the adding complex numbers calculation
def test_add():
    global totTest, passed
    totTest += 1

    try:
        expected_ans_real = -1.44
        expected_ans_imag = 12.34

        #TEST CASE 1: Addition
        assert(testComplex.add(negativeReal).real() == expected_ans_real)
        assert(testComplex.add(negativeReal).imag() == expected_ans_imag)

        #TEST CASE 2: Addition with zero
        assert(testComplex.add(zeroComplex).real() == testComplex.real())
        assert(testComplex.add(zeroComplex).imag() == testComplex.imag())

        passed+=1
        print("add() test passed")

    except AssertionError:
     print("add() test failed")

#Test the subtraction. Use isClose to test.
def test_sub():
    global totTest, passed
    totTest += 1

    try:
        expected_ans_real = 6.56
        expected_ans_imag = -1.66

        #TEST CASE 1: Subtraction

        assert isClose (testComplex.sub(negativeReal).real(), expected_ans_real)
        assert isClose(testComplex.sub(negativeReal).imag(), expected_ans_imag)

        #TEST CASE 2: Subtracting zero
        assert(testComplex.sub(zeroComplex).real() == testComplex.real())
        assert(testComplex.sub(zeroComplex).imag() == testComplex.imag())

        passed+=1
        print("sub() test passed")

    except AssertionError:
        print("sub() test failed")

#Test the multiplication calculation. Use isClose to test.
def test_mult():
    global totTest, passed
    totTest += 1

    try:
        #TEST CASE 1: Normal test case
        expected_ans_real = -47.62
        expected_ans_imag = -3.44
        assert isClose(testComplex.mult(negativeReal).real(), expected_ans_real)
        assert isClose(testComplex.mult(negativeReal).imag(), expected_ans_imag)

        #TEST CASE 2: Test multiplication with zero complex number
        expected_ans_real = 0.0
        expected_ans_imag = 0.0
        assert isClose(testComplex.mult(zeroComplex).real(), expected_ans_real)
        assert isClose(testComplex.mult(zeroComplex).imag(), expected_ans_imag)

        passed+=1
        print("mult() test passed")

    except AssertionError:
        print("mult() test failed")

#Test the reciprocal of complex number. Use isClose to test.
def test_recip():
```

```
  global totTest, passed
  totTest += 1
  try:
    #TEST CASE 1: Normal Test Case
    expected_ans_real = 0.07299852862
    expected_ans_imag = -0.1522703683
    assert isClose(testComplex.recip().real(), expected_ans_real)
    assert isClose(testComplex.recip().imag(), expected_ans_imag)

    #TEST CASE 2: Test zero imaginary part
    expected_ans_real = -0.21276595744
    expected_ans_imag = 0.0
    assert isClose(zeroImagNeg.recip().real(), expected_ans_real)
    assert isClose(zeroImagNeg.recip().imag(), expected_ans_imag)

    #TEST CASE 3: Test zero real part
    expected_ans_real = 0.0
    expected_ans_imag = -0.02272727272
    assert isClose(zeroReal.recip().real(), expected_ans_real)
    assert isClose(zeroReal.recip().imag(), expected_ans_imag)

    #TEST CASE 4: Test zero complex number
    #Expected result: Undefined
    try:
      zeroComplex.recip()
      print("recip() test failed")

    except ZeroDivisionError:
      passed+=1
      print("recip() test passed")

  except AssertionError:
    print("recip() test failed")

#Test division. Use isClose to test.
def test_div():
  global totTest, passed
  totTest += 1
  try:
    #TEST CASE 1: Normal test case
    expected_ans_real = 0.41753846153
    expected_ans_imag = -0.6043076923
    assert isClose(testComplex.div(negativeReal).real(), expected_ans_real)
    assert isClose(testComplex.div(negativeReal).imag(), expected_ans_imag)

    #TEST CASE 2: Test division by zero imaginary part
    expected_ans_real = 0.30117647058
    expected_ans_imag = 0.62823529411
    assert isClose(testComplex.div(zeroImag).real(), expected_ans_real)
    assert isClose(testComplex.div(zeroImag).imag(), expected_ans_imag)

    #TEST CASE 3: Test division by zero real part
    expected_ans_real = -0.05324027916
    expected_ans_imag = 0.02552342971
    assert isClose(testComplex.div(zeroRealNeg).real(), expected_ans_real)
    assert isClose(testComplex.div(zeroRealNeg).imag(), expected_ans_imag)

    try:
      #TEST CASE 4: Test divison by zero complex number
      #Raise ZeroDivisonError: Undefined
      testComplex.div(zeroComplex)
      print("div() test failed")

    except ZeroDivisionError:
      passed+=1
      print("div() test passed")

  except AssertionError:
    print("div() test failed")

#Test square root of complex number. Use isClose to test.
def test_sqrt():
  global totTest, passed
  totTest += 1
  try:
    #TEST CASE 1: Normal Test Case
    expected_ans_real = 2.0593598
    expected_ans_imag = 1.2965195
    assert isClose(testComplex.sqrt().real(), expected_ans_real)
    assert isClose(testComplex.sqrt().imag(), expected_ans_imag)
```

```
    #TEST CASE 2: Test zero imaginary part with negative real part
    expected_ans_real = 0.0
    expected_ans_imag = 2.1679483
    assert isClose(zeroImagNeg.sqrt().real(), expected_ans_real)
    assert isClose(zeroImagNeg.sqrt().imag(), expected_ans_imag)

    #TEST CASE 3: Test zero imaginary part with positive real part
    expected_ans_real = 2.9154759
    expected_ans_imag = 0.0
    assert isClose(zeroImag.sqrt().real(), expected_ans_real)
    assert isClose(zeroImag.sqrt().imag(), expected_ans_imag)

    #TEST CASE 4: Test zero real part
    expected_ans_real = 4.6904158
    expected_ans_imag = 4.6904158
    assert isClose(zeroReal.sqrt().real(), expected_ans_real)
    assert isClose(zeroReal.sqrt().imag(), expected_ans_imag)

    #TEST CASE 5: Test zero complex number
    expected_ans_real = 0.0
    expected_ans_imag = 0.0
    assert isClose(zeroComplex.sqrt().real(), expected_ans_real)
    assert isClose(zeroComplex.sqrt().imag(), expected_ans_imag)

    passed+=1
    print("sqrt() test passed")


  except AssertionError:
    print("sqrt() test failed")


#TriangleT objects for testing
validTriangle = TriangleT(7,10,5)
validTriangleEqual = TriangleT(5,7,10)
scaleneTriangle = TriangleT(2,3,4)
isoscelesTriangle = TriangleT(5,5,2)
equilateralTriangle = TriangleT(10,10,10)
rightTriangle = TriangleT(8,10,6)

#Keep track of passed and total tests
passedT = 0
totTestT = 0

#Test for constructor creating an invalid triangle - should raise Exception
def test_constructor():
  global totTestT, passedT
  totTestT+=1
  #TEST CASE 1: Create invalid triangle
  try:
    invalidTriangle = TriangleT(5,2,2)
    print("Constructor test failed")
  except Exception:
    passedT+=1
    print("Constructor test passed")

#Test for getting side lengths of triangle
def test_get_sides():
  global passedT, totTestT
  totTestT += 1
  try:
    #TEST CASE 1: Normal test case
    side1 = 7
    side2 = 10
    side3 = 5
    assert(validTriangle.get_sides()[0] == side1)
    assert(validTriangle.get_sides()[1] == side2)
    assert(validTriangle.get_sides()[2] == side3)
    passedT+= 1
    print("get_sides() test passed")

  except AssertionError:
    print("get_sides() test failed")

#Test for equality of triangles
def test_equal_T():
  global passedT, totTestT
  totTestT+= 1
```

```
    try:
      #TEST CASE 1: Test with same ordering of side lengths
      assert(validTriangle.equal(validTriangle))

      #TEST CASE 2: Test with different orientation of triangle
      assert(validTriangle.equal(validTriangleEqual))

      #TEST CASE 3: Test unequal triangles
      assert (not validTriangle.equal(isoscelesTriangle))

      passedT += 1
      print("test_equal() test passed")

    except AssertionError:
      print("equal() test failed")

#Test for perimeter of triangle
def test_perim():
    global passedT, totTestT
    totTestT +=1

    try:
      #TEST CASE 1: Calculate perimeter
      validTriangleP = 22
      assert(validTriangle.perim() == validTriangleP)
      passedT+=1
      print("perim() test passed")

    except AssertionError:
      print("perim() test failed")

#Test for area of triangle. Use isClose to test.
def test_area():
    global passedT, totTestT
    totTestT += 1

    try:
      #TEST CASE 1: Normal test case for area
      validTriangleA = 16.24808
      assert isClose(validTriangle.area(), validTriangleA)

      passedT += 1
      print ("area() test passed")

    except AssertionError:
      print("area() test failed")

#Test for validity of triangle
def test_is_valid():
    global passedT, totTestT
    totTestT+= 1
    try:
      #TEST CASE 1: Test valid triangle. Invalid triangles cannot be constructed.
      assert(validTriangle.is_valid() == True)
      passedT+= 1
      print("is_valid() test passed")

    except AssertionError:
      print("is_valid() test failed")

#Test the types of triangle
def test_tri_type():
    global passedT, totTestT
    totTestT+= 1
    try:
      #TEST CASE 1: Scalene - Priority 4
      assert(scaleneTriangle.tri_type() == TriType.scalene)
      #TEST CASE 2: Right - Priority 3
      assert(rightTriangle.tri_type() == TriType.right)
      #TEST CASE 3: Isosceles - Priority 2
      assert(isoscelesTriangle.tri_type() == TriType.isosceles)
      #TEST CASE 4: Equilateral - Priority 1
      assert(equilateralTriangle.tri_type() == TriType.equilat)
      passedT+=1
      print("tri_type() test passed")
    except AssertionError:
      print("tri_type() test failed")
```

```
#Call the tests for complex_adt.py
print("Tests for complex.adt:\n")
test_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()
print('\n',passed,"of ",totTest,"passed for complex_adt")

#Call the tests for triangle_adt.py
print("\nTests for triangle.adt:\n")
test_constructor()
test_get_sides()
test_equal_T()
test_perim()
test_area()
test_is_valid()
test_tri_type()
print('\n',passedT,"of ",totTestT,"passed for triangle_adt")
```

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
```