Assignment 1

Hamrish Saravanakumar, saravah

January 28, 2021

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

1 Assumptions and Exceptions

Assumptions were made based on a method's behaviour for specific cases that were not specified completely in the assignment's specifications. Specifically, it was assumed that:

- `ComplexT` and `TriangleT` can only take inputs of type "int" or "float".
- Since the assignment specifications state that TriangleT only takes inputs of type "int", inputs of type "float" would be considered an invalid triangle.
- `tri_type`: A triangle that can be classified as both a right-angled triangle and another type of triangle, would be considered as a right-angled triangle. This is because the method should return an output of `TriType`, which means only a singular element of the set of triangles types should be returned.

In addition, I decided to return None when the input should be undefined instead of assuming that these inputs would not be used, including:

- `get_phi`: The argument of a complex number is undefined if both the real and imaginary numbers are equal to zero.
- `recip`: The reciprocal of a complex number is undefined if the real and imaginary numbers are equal to zero.

- `div`: The quotient of an argument complex number divided by the initial complex number is undefined if the real and imaginary parts of the argument complex number are equal to zero.

Finally, I decided to return None when there should not be an output if the input does not match the specifications of the assignments, including:

- `get_sides`: An invalid triangle does not technically have 3 side lengths, so the method returns None
- `__eq__`: This method in `triangle_adt.py` should only compare two valid triangles in order to determine if they are equal. If either the argument triangle or current triangle is invalid, the method returns None as a result
- `perim`: An invalid triangle does not have a perimeter, so the method returns None
- `area`: An invalid triangle does not have an area, so the method returns None
- `tri_type`: An invalid triangle can not be categorized to have a specific type, so the method returns None

Exceptions were not thrown, as every reasonable case for both programs was accounted for through returning None for all undefined and invalid cases.

2 Test Cases and Rationale

Tests were written to prove that each individual method from both `complex_adt.py` and `triangle_adt.py` would handle regular, edge, and invalid cases. This was done by defining variables of type `ComplexT` and `TriangleT` that would be used for the test cases.

For each method, a variable is defined to represent if my method returns the correct value. This correct value was either calculated or determined by using online calculators that were cited in the details of both python modules, referencing Wikepedia articles that were also cited, or using `cmath` to calculate certain properties for `complex_adt.py`. For example, if `complex1.get_phi` returns the same value as `cmath.phase(x)`, where `x` represents complex1 in the form `a + bj` instead of as type `ComplexT`, then the variable `get_phiTest1` is equal to `True`. If not, the variable would have been equal to `False`.

Once all tests for a particular method have been conducted, a for loop is used to iterate through a list that contains the results of all of the test cases. For every test case in the list that has a value of True, a statement is printed to indicate that the test does in fact pass. On the other hand, a statement to indicate that the test fails is printed for every case with a value of False. Finally, to improve readability, a final statement is printed to show how many cases passed for that particular method out of the total number of cases conducted.

For every method, there would always be one test case for inputs that the method would expect, based on the assignment's specification. However, in order to further prove the functionality and usability of both programs, edge cases were included that would require special handling as a means of testing the boundaries of the program. For example, the variable `complex1` represents a complex number of type `ComplexT` with inputs 1.0 and 2.0, and is used in every method for `complex_adt.py` to represent the expected case. On the other hand, the variable `complex4` represents a complex number of type `ComplexT` with inputs 0.0 and 0.0, and is used in every method for `complex_adt.py` to represent edge cases that often return None based on the properties of many of the calculations including the `div`, `recip`, and `get_phi methods`. Instead of only including one test case to represent edge cases, I ensured to include multiple test cases that would prove the adaptability of the program to handle various boundary conditions.

3 Results of Testing Partner's Code

In order to test my partner's `complex_adt.py` and `triangle_adt.py` files, I copied my `test_driver.py` file into the partner folder temporarily and ran the test driver.

After running the driver for the first time, the code passes all of the test cases for the first 5 test methods other than 2 cases in the `get_phi` method, fails all of the cases for the next 4 methods, and then terminates with the error "Zero Division Error: float division by zero". In order to get results for all cases for the `complex_adt.py` module, I temporarily deleted `recipTest3` and `divTest2`. The code failed all remaining cases for the `ComplexT` class. Before examining the code itself, it is important to note of the assumptions made by my partner. An assumption that the methods will only ever interact with a nonzero complex number of type `ComplexT`. This explains the "Zero division error", for both `recipTest3` and `divTest2` , as my code did not make this assumption, and I instead returned None under these conditions. This is also why two of the test cases in `get_phi` failed. However, none of these assumptions explain the failed cases for the remaining methods. I went into the python file itself and decided return the value of each method to determine where the differences lied. An example of what I ran at first for the `recip` method is `print(complex1.recip())`. The value that was returned for each of these cases were a variation of the following: `<__main__.ComplexT object at 0x000001BFD27EAD00>`. This was expected, as this is the way that the computer processes the object. To get a more comprehendible number, I ran `print(complex1.recip().real())` and `print(complex1.recip().imag())`. I was surprised to see that for all cases they are not supposed to return None, the real and imaginary parts of the returned complex number were the same, suggesting that all of the test cases for the `ComplexT` class technically passed as a result of the assumptions made by the partner. It may have been more effective to compare the real and imaginary parts of the object rather than comparing the object itself for the test cases, in order to ensure the driver would be compatible with different python modules.

Moving on to the `TriangleT` class, the first run of `test_driver.py` returned only 1 out of 3 passed cases for the `get_sides` and `perim` tests, 3 out of 4 passed cases for the `equal` tests, and then terminates with the error "Value Error: math domain error". In order to get results for all cases for the `triangle_adt.py` module, I temporarily deleted `areaTest3`. For the remaining three methods that were tested, only one test case for each method failed. Similar to `complex_adt.py`, I examined the assumptions made before critiquing the results of the test cases. It was assumed that all functions are passed by the correct data type of parameters, which assumes that only integers are passed as the side lengths. This explains why the code fails for any tests involving `triangle2`, as this triangle of type `TriangleT` represents an edge case that uses side length values of type float. The issue with the `TriangleT class`, is that if a triangle returns False after running the `is_valid` method, it is still allowed to be constructed and run as if it was a perfectly valid triangle. This is why the rest of the test cases failed. For example, for `triangle5`, that represents an edge case where all of the side lengths of the triangle are 0, the test cases expect the value None for methods `area`, `perim`, `equal`, and `tri_type` because an invalid triangle can not have characteristics of a geometrically valid triangle. There should have been an effort to either ensure that an invalid triangle did not get constructed, or so that an error is raised when an invalid triangle is attempted to be passed in one of the other methods in the `TriangleT` class.

4 Critique of Given Design Specification

One particular strength of the design is specifying the types of inputs chosen to represent the data of either module, such as representing the real and imaginary numbers of a complex number as a float, and the different side lengths of a triangle as an integer. Specifically in the `ComplexT` class, having two separate variables to represent the real and imaginary numbers allowed for quick and easy retrieval of these values, and the ability to utilize these values in the methods that require calculations using both variables. Additionally, the specifications of the state variables that were chosen to represent the identity of the complex number and triangle were effective, and provided students with an efficient way to use these values for the various methods within each class.

Both classes themselves had a large number of methods that represented the well-versed properties of both complex numbers and triangles. The `is_valid` method in the `TriangleT` class was especially useful, especially for those making the assumption that an invalid triangle should have invalid outputs for the remaining methods in the class. Rather than having to check if the triangle was invalid in every individual method, the `is_valid` method was ran, and the output passed None if the `is_valid` method was False. This significantly improved the efficiency of the code written for the `TriangleT` class.

A major drawback to this design, as shown through the multiple assumptions made in Section 1, is the number of ambiguities that were present. This was a result of writing the specifications in natural language. Specifically, there was very little instruction on how to handle instances in the program that would have an undefined output. An example of this can be found in the `recip` method, where a zero complex number would return an undefined output due to a "division by zero" error. Students could have handled these occurrences in multiple ways. An exception could have been raised to catch a particular error that would be expected based on the passed inputs, an assumption could have been stated to indicate that inputs that would cause for an undefined output would never be passed, or these cases could be formatted to return None. The preferred method for handling such cases should have been stated, as this proved problematic during the testing of the partner's code, especially when the partner made different assumptions in comparison to the student themselves.

5 Answers to Questions

(a) There are no mutators because according to the assignment specification, there are no methods that modify the state of the variable of the object itself. A getter method is one that does not change the state variables and returns a value based on that variable. This would include methods that simply returns a value of one of the variables without making any calculations. These would include `real` and `imag` methods from the `ComplexT` class, and the `get_sides` method from the `TriangleT` class. A getter method would also include those methods that use variables to calculate a value to be returned, without modifying the state of the variables. These would include `get_phi`, `get_r`, `conj`, `recip`, and `sqrt` methods from the `ComplexT` class, and the `perim`, `area`, and `tri_type` methods from the `TriangleT` class.

(b) State variables, otherwise known as instance variables, are the variables that are passed through the constructor in a particular class. In the current implementations as defined by the assignment specifications, the state variables for `ComplexT` are two variables that represent the real and imaginary parts of a complex number, and the state variables for `TriangleT` are three variables that represent each unique side length of a triangle. The state variables for `ComplexT` could have been represented as a list, where the first number represents the real part of the complex number, and the second represents the imaginary part. The state variables for `TriangleT` could have been represented as a list as well, where each member of the list represents a side length of the triangle. Using lists as variables would not have been as effective, as many of the methods required calculations based on the state variables, and it would have been less efficient to have to call on members of a list to have to perform these calculations.

(c) Given two complex numbers, it is impossible to determine which of the two is either greater or less than the other, based on their real and imaginary parts alone. This is because complex numbers are not simply an integer or a float, but instead, a set of complex numbers form a field, but not necessarily an ordered field. This is why an equal method is the only appropriate method that can be included in class `ComplexT` when comparing the complex numbers alone. Instead, the magnitude of 2 complex numbers can be compared to determine the greater/smaller of the two, as the magnitude is represented by an integer, as shown in the `get_r` method.

(d) There are certain conditions that determine whether three side lengths of particular lengths are able to form a geometrically valid triangle. The approach is that a triangle is valid if the sum of two of the sides is greater than the third sides. These conditions were used to create the `is_valid` method. There were no specific instructions regarding invalid triangles and their behaviour when passed through the other methods in `TriangleT`. However, I decided to return None for all remaining methods in `TriangleT` if an invalid triangle is given. This is because, if a triangle is not geometrically valid, it is not technically a triangle in the first place. Therefore, a triangle with such properties should not have an area, a perimeter, a triangle type, nor should it be comparable to another valid triangle.

(e) The type of triangle is based on the behaviour of the three side lengths of the triangle. Therefore, the user will most likely be forced to determine the triangle type on their own. The point of the ADT class of `TriangleT` is to return various different characteristics based purely on the state variables, which are the different side lengths. It would be far more useful to implement a method to determine the type of triangle, as done through the `tri_type` method. The only situation where a state variable for the triangle type would be appropriate, is if there were methods that would actually require the type of a triangle in order to return a particular value. A potential example of this would be a `get_hypotenuse` method that would return the hypotenuse of a triangle. The method could take advantage of a state variable for the triangle type, and will only return the largest side length if the passed triangle is a "right-angled" triangle.

(f) Performance of a product is heavily reliant on the external quality requirements for both speed and storage, whereas the usability of a software product refers to the ease with which a user is able to use that particular product. If a product has poor performance, this often adversely affects the usability and the scalability of the product. For example, if a product is poor as a result of slow speed and low storage, a user will have difficulties being able to use that particular product for their needs.

(g) A rational development process was introduced by Parnas and Clements to provide meaning to the ideal process in which programs are derived from its requirements. Despite claiming that a rational design process is needed in order to produce quality software, it is argued that it is impossible to find a process in which the software is designed in a perfectly rational way. Instead, it is suggested to fake this process, by presenting the system and its associated documents to others as if the user had followed this particular idealized design process. Although upholding this software best practice is in the interest of the user more often than not, I can propose one situation where this design process may not have to be faked. For example, let us say an individual were to document their design process as it happens, without correcting for human errors and including everything that happens from the start of the design process all the way until creation. This would be impactful as a student when submitting an assignment, to show the learning process that the student went through during the process of the assignment. "Faking" this process could be seen as an example of academic dishonesty, and the student should instead be authentic with all the steps they took, regardless of whether they were correct or not. However, in the production world where documentation is crucial for others to interpret a particular software practice, it is important to present this in the most rational way possible.

(h) A software product is re-usable if it can be used to create a new product and is reliable if it is able to carry out the function that it is intended to perform on a regular basis. By designing software products that prioritize reusability ensure that they are more durable and easier to maintain. These directly impact the overall quality of the product, and thus the reliability that users experience when using the product. A reason why this is the case, is because software reusability minimizes the need to use newly developed products, eliminating the risk of issues that may arise with a new product.

(i) Hardware abstractions are defined as specific sets of routines in software, that provide programs with access to hardware resources. This is usually done through a programming interface, which allows devices in a particular class of hardware devices to be accessed through identical interfaces. Specifically, high level programming languages include many built-in abstractions that make it easier for the user to focus on the problem that is aimed to be solve by using that programming language, rather than worrying about how the computer hardware works in the background. High-level languages, such as Scheme and Snap! are often used to produce programs that are less likely to have bugs in memory use and can improve conveniency to allow for shorter and cleaner code.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Hamrish Saravanakumar
#  @brief ComplexT ADT Class
#  @date 21/01/2020

import math

## @brief An ADT class that represents a complex number.
#  @details Formulas are all based on Complex Numbers Wikepedia Page
#           https://en.wikipedia.org/wiki/Complex_number
class ComplexT:

    ## @brief ComplexT constructor
    #  @details Initializes a ComplexT object that takes two floats that represents the
    #           real and imaginary part of the complex number.
    #  @param x The real part of the complex number.
    #  @param y The imaginary part of the complex number.
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

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

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

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

    ## @brief Calculates the argument of the complex number.
    #  @return The argument of the complex number, and returns None if the argument is undefined.
    def get_phi(self):
        absolute = math.sqrt(self.__x**2 + self.__y**2)
        if(self.__y != 0):
            return 2*math.atan(self.__y / (absolute + self.__x))
        elif(self.__x < 0 and self.__y == 0):
            return pi
        else:
            return None

    ## @brief Determines if the current complex number is equal to the argument complex number.
    #  @details A private method "__eq__" is used to support the "equal" method.
    #  @param other The second complex number that is compared to the current complex number.
    #  @return Returns the output of the __eq__ method.
    def equal(self, other):
        return ComplexT.__eq__(self, other)

    ## @brief A private method "__eq__" that is used to support the "equal" method.
    #  @details Magic method used to aid in determining if the current complex number is equal to the
     argument complex number.
    #  @param other The second complex number that is compared to the current complex number.
    #  @return Returns true if the two complex numbers are equal; false if the two complex numbers are
     not equal
    def __eq__(self, other):
        return self.__x == other.__x and self.__y == other.__y

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

    ## @brief Calculates the sum of the current and argument complex numbers.
    #  @param other second complex number to add to current complex number.
    #  @return The sum of the two complex numbers.
    def add(self, other):
        return ComplexT(self.__x + other.__x, self.__y + other.__y)

    ## @brief Calculates the difference of the argument complex number from the current complex number.
    #  @param other second complex number to subtract from first complex number
```

```
    #  @return The difference between the two complex numbers
    def sub(self, other):
        return ComplexT(self.__x - other.__x, self.__y - other.__y)

    ## @brief Calculates the product of the argument and current complex numbers.
    #  @param other second complex number to multiply to the first complex number.
    #  @return The product of the two complex numbers.
    def mult(self, other):
        return ComplexT(self.__x*other.__x - self.__y*other.__y, self.__y*other.__x +
            self.__x*other.__y)

    ## @brief Calculates the reciprocal of the complex number.
    #  @details Complex number must be non-zero
    #  @return The reciprocal of the complex number, and returns None if the complex number is zero.
    def recip(self):
        absolute = self.__x**2 + self.__y**2
        if absolute == 0:
            return None
        return ComplexT(self.__x / absolute, -(self.__y / absolute))

    ## @brief Calculates the quotient of the argument complex number divided by the current complex
        number.
    #  @details Complex number must be non-zero.
    #  @param other second complex number that divides the first complex number.
    #  @return The quotient of two complex numbers, and returns None if the complex number is zero.
    def div(self, other):
        magnitude = float(other.__x**2 + other.__y**2)
        if magnitude == 0:
            return None
        return ComplexT((self.__x*other.__x + self.__y*other.__y) / magnitude, (self.__y*other.__x -
            self.__x*other.__y) / magnitude)

    ## @brief Calculates the square root of the complex number
    #  @return The square root of the complex number
    def sqrt(self):
        modulus = math.sqrt(self.__y**2 + self.__x**2)
        a = math.sqrt((self.__x + modulus)/2)
        if self.__y == 0:
            b = 0.0
        elif self.__y > 0:
            b = math.sqrt((-self.__x + modulus)/2)
        else:
            b = -(math.sqrt((-self.__x + modulus)/2))
        return ComplexT(a, b)

c1 = ComplexT("t", "b")
c2 = ComplexT(2, 3)
c3 = (c2.div(c1))
print(c3.real())
print(c3.imag())
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Hamrish Saravanakumar
#  @brief TriangleT ADT class
#  @date 1/21/2020

import math
from enum import Enum

## @brief An enumerated classused to sort through a set of potential types of triangles.
class TriType(Enum):
    ## List of types of triangles as provided in assignment documentation.
    types = ["equilat", "isosceles", "scalene", "right"]
    ## enum value representing a triangle with 3 equivalent side lengths.
    equilat = types[0]
    ## enum value representing a triangle with 2 equivalent side lengths.
    isosceles = types[1]
    ## enum value representing a triangle with no equivalent side lengths.
    scalene = types[2]
    ## enum value representing a triangle that has a 90 degree angle.
    right = types[3]

## @brief An ADT class that represents a Triangle.
class TriangleT:

    ## @brief TriangleT constructor.
    #  @details Initializes a TriangleT object with three defined side lengths.
    #  @param side1 The length of the first side of the triangle.
    #  @param side2 The length of the second side of the triangle.
    #  @param side3 The length of the third side of the triangle.
    def __init__(self, side1, side2, side3):
        self.__side1 = side1
        self.__side2 = side2
        self.__side3 = side3

    ## @brief Gets the the lengths of all three sides
    #  @return A tuple of the 3 lengths of the sides of the triangle
    def get_sides(self):
        if (TriangleT.is_valid(self)):
            return self.__side1, self.__side2, self.__side3
        else:
            return None
    ## @brief Determines if the current triangle is equal to the argument triangle.
    #  @details A private method "__eq__" is used to support the "equal" method.
    #  @param other The second triangle that is compared to the current triangle.
    #  @return Returns the output of the __eq__ method.
    def equal(self, other):
        return TriangleT.__eq__(self, other)

    ## @brief A private method "__eq__" that is used to support the "equal" method.
    #  @details Magic method used to aid in determining if the triangle is equal to the argument
     triangle.
    #  @param other The second triangle that is compared to the current triangle.
    #  @return Returns true if the two triangles are equal; false if the two triangles are not equal;
    #          and returns None if either of the triangles are not valid.
    def __eq__(self, other):
        if TriangleT.get_sides(self) == None or TriangleT.get_sides(other) == None:
            return None
        count = 0
        for i in TriangleT.get_sides(self):
            for j in TriangleT.get_sides(other):
                if(i == j):
                    count = count + 1
                    break
        if(count == 3):
            return True
        return False

    ## @brief Calculates the perimeter of the triangle.
    #  @return The perimeter of the triangle, and returns None if the triangle is invalid.
    def perim(self):
        if (TriangleT.is_valid(self)):
            return self.__side1 + self.__side2 + self.__side3
        else:
            return None

    ## @brief Calculates the area of the triangle.
```

```
#   @details Heron's Formula was used to determine the area of the triangle.
#            https://www.mathopenref.com/heronsformula.html
#   @return The area of the triangle, and returns None if the triangle is not valid.
def area(self):
    p = (self.__side1 + self.__side2 + self.__side3) / 2
    a = p - self.__side1
    b = p - self.__side2
    c = p - self.__side3
    if (TriangleT.is_valid(self)):
        return math.sqrt(p * a * b * c)
    else:
        return None

## @brief Determines if the side lengths are able to create a valid triangle.
#   @details Method of determining the validity of a triangle
#            https://www.wikihow.com/Determine-if-Three-Side-Lengths-Are-a-Triangle
#   @return Returns true if a triangle is valid, and returns false if the triangle is invalid.
def is_valid(self):
    a = self.__side1
    b = self.__side2
    c = self.__side3
    if (a + b <= c) or (a + c <= b) or (b + c <= a) :
        return False
    elif (isinstance(a,int) == False or isinstance(b,int) == False or isinstance(c,int) == False):
        return False
    else:
        return True

## @brief Determines the type of triangle
#   @details A triangle that fits the requirements of a right triangle and a second
#            type is assumed to only have the type "right".
#   @return Returns the triangle type and returns None if the triangle is not valid.
def tri_type(self):
    a = self.__side1
    b = self.__side2
    c = self.__side3
    if(TriangleT.is_valid(self) == False):
        return None
    elif max(a**2, b**2, c**2) == (a**2 + b**2 + c**2 - max(a**2, b**2, c**2)):
        return TriType.right
    elif a==b and b==c:
        return TriType.equilat
    elif a==b or b==c or a==c:
        return TriType.isosceles
    else:
        return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Hamrish Saravanakumar
#  @brief Test driver for python modules triangle_adt.py and complex_adt.py
#  @date 1/21/2021

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

print("Test cases for ComplexT Module:".center(50, '-'))
print(" ")

## Complex numbers of type ComplexT used in test cases for every method in
#  complex_adt.py module
complex1 = ComplexT(1.0, 2.0)
complex2 = ComplexT(-4.5, -5.5) #4
complex3 = ComplexT(1.33, 0.0)
complex4 = ComplexT(0.0, 0.0) #7
complex5 = ComplexT(1.0, 2.0)

# Test cases for real method
realTest1 = complex1.real() == 1.0
realTest2 = complex4.real() == 0.0
realTest3 = complex2.real() == -4.5

realTests = [realTest1, realTest2, realTest3]
passed, numTests = 0, 0
for i in realTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes real Test", numTests)
    else:
        print("Fails real Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "real tests out of", numTests, "\n")

#Test cases for imag method
imagTest1 = complex1.imag() == 2.0
imagTest2 = complex3.imag() == 0.0
imagTest3 = complex2.imag() == -5.5

imagTests = [imagTest1, imagTest2, imagTest3]
passed, numTests = 0, 0
for i in imagTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes imag Test", numTests)
    else:
        print("Fails imag Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "imag tests out of", numTests, "\n")

#Test cases for get_r method
get_rTest1 = complex1.get_r() == math.sqrt(5)
get_rTest2 = complex2.get_r() == math.sqrt(4.5**2+5.5**2)
get_rTest3 = complex3.get_r() == math.sqrt(1.33**2)
get_rTest4 = complex4.get_r() == math.sqrt(0)

get_rTests = [get_rTest1, get_rTest2, get_rTest3, get_rTest4]
passed, numTests = 0, 0
for i in get_rTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes get_r Test", numTests)
    else:
        print("Fails get_r Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "get_r tests out of", numTests, "\n")

#Test cases for get_phi method
#The value of the phase is rounded to 5 decimal places
get_phiTest1 = round(complex1.get_phi(),5) == round(cmath.phase(1 + 2j),5)
```

```
get_phiTest2 = round(complex2.get_phi(),5) == round(cmath.phase(-4.5 - 5.5j),5)
get_phiTest3 = complex3.get_phi() == None
get_phiTest4 = complex4.get_phi() == None

get_phiTests = [get_phiTest1, get_phiTest2, get_phiTest3, get_phiTest4]
passed, numTests = 0, 0
for i in get_phiTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes get_phi Test", numTests)
    else:
        print("Fails get_phi Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "get_phi tests out of", numTests, "\n")

#Test cases for equal method
equalTest1 = complex1.equal(complex2) == False
equalTest2 = complex1.equal(complex5) == True
equalTest3 = complex1.equal(complex1) == True

equalTests = [equalTest1, equalTest2, equalTest3]
passed, numTests = 0, 0
for i in equalTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes equal Test", numTests)
    else:
        print("Fails equal Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "equal tests out of", numTests, "\n")

#Test cases for conj method
conjTest1 = complex1.conj() == ComplexT(1.0, -2.0)
conjTest2 = complex3.conj() == ComplexT(1.33, 0.0)
conjTest3 = complex2.conj() == ComplexT(-4.5, 5.5)

conjTests = [conjTest1, conjTest2, conjTest3]
passed, numTests = 0, 0
for i in conjTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes conj Test", numTests)
    else:
        print("Fails conj Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "conj tests out of", numTests, "\n")

#Test cases for add method
addTest1 = (complex1.add(complex2)) == ComplexT(-3.5, -3.5)
addTest2 = (complex3.add(complex4)) == ComplexT(1.33, 0.0)

addTests = [addTest1, addTest2]
passed, numTests = 0, 0
for i in addTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes add Test", numTests)
    else:
        print("Fails add Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "add tests out of", numTests, "\n")

#Test cases for sub method
subTest1 = (complex1.sub(complex2)) == ComplexT(5.5, 7.5)
subTest2 = (complex2.sub(complex1)) == ComplexT(-5.5, -7.5)
subTest3 = (complex3.sub(complex4)) == ComplexT(1.33, 0.0)

subTests = [subTest1, subTest2, subTest3]
passed, numTests = 0, 0
for i in subTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes sub Test", numTests)
    else:
        print("Fails sub Test", numTests)
```

```
        numTests = numTests + 1
print("Passed", passed, "sub tests out of", numTests, "\n")

#Test cases for mult method
multTest1 = (complex1.mult(complex2)) == ComplexT(6.5, -14.5)
multTest2 = (complex1.mult(complex4)) == ComplexT(0, 0)

multTests = [multTest1, multTest2]
passed, numTests = 0, 0
for i in multTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes mult Test", numTests)
    else:
        print("Fails mult Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "mult tests out of", numTests, "\n")

# Test cases for recip method
# Verification of test cases calculated using symbolab math calculator:
# https://www.symbolab.com/solver/complex-numbers-calculator
recipTest1 = complex1.recip() == ComplexT(0.2, -0.4)
recipTest2 = complex3.recip() == ComplexT(float(100/133), 0.0)
recipTest3 = complex4.recip() == None

recipTests = [recipTest1, recipTest2, recipTest3]
passed, numTests = 0, 0
for i in recipTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes recip Test", numTests)
    else:
        print("Fails recip Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "recip tests out of", numTests, "\n")

#Test cases for div method
# Verification of test cases calculated using symbolab math calculator:
# https://www.symbolab.com/solver/complex-numbers-calculator
divTest1 = (complex1.div(complex2)) == ComplexT(-0.3069306930693069, -0.06930693069306931)
divTest2 = (complex1.div(complex4)) == None

divTests = [divTest1, divTest2]
passed, numTests = 0, 0
for i in divTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes div Test", numTests)
    else:
        print("Fails div Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "div tests out of", numTests, "\n")

#Test cases for sqrt method
# Verification of test cases calculated using symbolab math calculator:
# https://www.symbolab.com/solver/complex-numbers-calculator
sqrtTest1 = complex1.sqrt() == ComplexT(1.2720196495140691, 0.7861513777574233)
sqrtTest2 = complex2.sqrt() == ComplexT(1.1415636648422083, -2.408976463332088)
sqrtTest3 = complex3.sqrt() == ComplexT(1.1532562594670797, 0.0)
sqrtTest4 = complex4.sqrt() == ComplexT(0.0, 0.0)

sqrtTests = [sqrtTest1, sqrtTest2, sqrtTest3, sqrtTest4]
passed, numTests = 0, 0
for i in sqrtTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes sqrt Test", numTests)
    else:
        print("Fails sqrt Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "sqrt tests out of", numTests, "\n")


print("Test cases for TriangleT Module:".center(50, '-'))
print(" ")
```

```
## Triangles of type TriangleT used in test cases for every method in
#  triangle_adt.py module

triangle1 = TriangleT(4, 5, 3)
triangle2 = TriangleT(4.5, 5.5, 3.5)
triangle3 = TriangleT(3, 4, 5)
triangle4 = TriangleT(1, 2, 9)
triangle5 = TriangleT(0, 0, 0)
triangle6 = TriangleT(-4, -5, -3)
triangle7 = TriangleT(4, 3, 6)
triangle8 = TriangleT(3, 6, 6)
triangle9 = TriangleT(6, 6, 6)

# Test cases for get_sides method
# Any triangle of type TriangleT that is not valid will return None
get_sidesTest1 = triangle1.get_sides() == (4, 5, 3)
get_sidesTest2 = triangle2.get_sides() == None
get_sidesTest3 = triangle5.get_sides() == None

get_sidesTests = [get_sidesTest1, get_sidesTest2, get_sidesTest3]
passed, numTests = 0, 0
for i in get_sidesTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes get_sides Test", numTests)
    else:
        print("Fails get_sides Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "get_sides tests out of", numTests, "\n")

# Test cases for equal method
# If either triangle of type TriangleT that is not valid, None is returned
equalTest1 = triangle1.equal(triangle7) == False
equalTest2 = triangle1.equal(triangle1) == True
equalTest3 = triangle1.equal(triangle3) == True
equalTest4 = triangle1.equal(triangle6) == None

equalTests = [equalTest1, equalTest2, equalTest3, equalTest4]
passed, numTests = 0, 0
for i in equalTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes equal Test", numTests)
    else:
        print("Fails equal Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "equal tests out of", numTests, "\n")


# Test cases for perim method
# Any triangle of type TriangleT that is not valid will return None
perimTest1 = triangle1.perim() == 12
perimTest2 = triangle4.perim() == None
perimTest3 = triangle6.perim() == None

perimTests = [perimTest1, perimTest2, perimTest3]
passed, numTests = 0, 0
for i in perimTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes perim Test", numTests)
    else:
        print("Fails perim Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "perim tests out of", numTests, "\n")

#Test cases for area method
# Any triangle of type TriangleT that is not valid will return None
areaTest1 = triangle1.area() == 6
areaTest2 = triangle2.area() == None
areaTest3 = triangle4.area() == None

areaTests = [areaTest1, areaTest2, areaTest3]
passed, numTests = 0, 0
for i in areaTests:
    if i == True:
        passed = passed + 1
```

```
        numTests = numTests + 1
        print("Passes area Test", numTests)
    else:
        print("Fails area Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "area tests out of", numTests, "\n")

#Test cases for is_valid method
is_validTest1 = triangle1.is_valid() == True
is_validTest2 = triangle2.is_valid() == False
is_validTest3 = triangle4.is_valid() == False
is_validTest4 = triangle5.is_valid() == False
is_validTest5 = triangle6.is_valid() == False

is_validTests = [is_validTest1, is_validTest2, is_validTest3, is_validTest4, is_validTest5]
passed, numTests = 0, 0
for i in is_validTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes is_valid Test", numTests)
    else:
        print("Fails is_valid Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "is_valid tests out of", numTests, "\n")

#Test cases for tri_type method
# Any triangle of type TriangleT that is not valid will return None
tri_typeTest1 = triangle1.tri_type() == TriType.right
tri_typeTest2 = triangle2.tri_type() == None
tri_typeTest3 = triangle7.tri_type() == TriType.scalene
tri_typeTest4 = triangle8.tri_type() == TriType.isosceles
tri_typeTest5 = triangle9.tri_type() == TriType.equilat

tri_typeTests = [tri_typeTest1, tri_typeTest2, tri_typeTest3, tri_typeTest4, tri_typeTest5]
passed, numTests = 0, 0
for i in tri_typeTests:
    if i == True:
        passed = passed + 1
        numTests = numTests + 1
        print("Passes tri_type Test", numTests)
    else:
        print("Fails tri_type Test", numTests)
        numTests = numTests + 1
print("Passed", passed, "tri_type tests out of", numTests)
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Laura Southwood (southwol)
#  @brief implements an Abstract Data Type for complex numbers
#  @date January 21, 2021

#Assume that all functions are passed the correct number of parameters
#Assume that all function are passed the correct data type of parameters
#Assume that .get_phi() is only ever executed on a nonzero ComplexT (since atan2(0,0) undefined)
#Assume that .recip() is only ever executed on a nonzero ComplexT (can't divide by 0)
#Assume that .div() is only ever used to divide by a nonzero ComplexT (can't divide by 0)

import math

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

        ## @brief Constructor for ComplexT
        # @details Creates a complex number with real and imaginary parts
        # @param x float representing the real part of the complex number
        # @param y float representing the imaginary part of the complex number
        def __init__(self, x, y):
                self.__x = x
                self.__y = y

        ## @brief Getter for the real part (x) of the complex number
        # @return float representing the real part of the complex number
        def real(self):
                return self.__x

        ## @brief Getter for the imaginary part (y) of the complex number
        # @return float representing the imaginary part of the complex number
        def imag(self):
                return self.__y

        ## @brief Calculates the absolute value (also called modulus or magnitude)
        # @return float representing the absolute value of the complex number
        def get_r(self):
                return math.sqrt(self.__x**2 + self.__y**2)

        ## @brief Calculates the phase (also called the argument)
        # @return float representing the phase of the complex number (in radians)
        def get_phi(self):
                return math.atan2(self.__y, self.__x)

        ## @brief Checks if the two complex numbers (z and self) are equal
        # @param z ComplexT
        # @return boolean representing if z is equal to the current ComplexT object
        def equal(self, z):
                return self.__x == z.real()  and  self.__y == z.imag()

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

        ## @brief Adds a complex number to the current ComplexT
        # @param z ComplexT
        # @return ComplexT representing the result of z added to the current object
        def add(self, z):
                a = self.__x + z.real()
                b = self.__y + z.imag()
                return ComplexT(a, b)

        ## @brief Subtracts a complex number from the current ComplexT
        # @param z ComplexT
        # @return ComplexT representing the result of z subtracted from the current object
        def sub(self, z):
                a = self.__x - z.real()
                b = self.__y - z.imag()
                return ComplexT(a, b)

        ## @brief Multiplies a complex number with the current ComplexT
        # @param z ComplexT
        # @return ComplexT representing the result of z multiplied with the current object
        def mult(self, z):
```

```
        u = z.real()
        v = z.imag()
        a = self.__x*u - self.__y*v
        b = self.__x*v + self.__y*u
        return ComplexT(a, b)

    ## @brief Calculates the reciprocal of the current ComplexT object
    # @return ComplexT representing the reciprocal of the complex number
    def recip(self):
        denom = self.__x**2 + self.__y**2
        return ComplexT(self.__x/denom, -self.__y/denom)

    ## @brief divides the current ComplexT by a complex number
    # @param z ComplexT
    # @return ComplexT representing the result of the current object divided by z
    def div(self, z):
        return self.mult(z.recip())

    ## @brief Calculates the positive square root of the current ComplexT object
    # @return ComplexT representing the positive square root of the complex number
    def sqrt(self):
        if(self.__y == 0):
            return ComplexT(math.sqrt(self.__x), 0)

        mod = self.get_r()
        gamma = math.sqrt((self.__x + mod)/2)
        delta = math.sqrt((-self.__x + mod)/2)

        #Multiply delta by the sgn function.
        if(self.__y < 0):
            delta = -delta
        elif(self.__y == 0):
            delta = 0

        return ComplexT(gamma, delta)
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Laura Southwood (southwol)
#  @brief implements an ADT for triangles
#  @date January 21, 2021

#Assume that all functions are passed the correct number of parameters
#Assume that all function are passed the correct data type of parameters

import math
from enum import Enum

## @brief An ADT for representing triangles
# @details A triangle is composed of 3 side lengths
class TriangleT:

    ## @brief Constructor for TriangleT
    # @details Creates a triangle with 3 side lengths
    # @param x Integer first side length
    # @param y Integer second side length
    # @param z Integer third side length
    def __init__(self, x, y, z):
        self.__x = x
        self.__y = y
        self.__z = z

    ## @brief Gets the side lengths of the current triangle object
    # @return tuple of 3 integers containing the side lengths of the triangle
    def get_sides(self):
        return (self.__x, self.__y, self.__z)

    ## @brief Checks if tri is equal to the current TriangleT
    # @details The triangles are equal if they have the same 3 side lengths (in any order)
    # @param tri TriangleT
    # @return True if the current obj is equal to tri, False otherwise
    def equal(self, tri):
        t1 = tri.get_sides()
```

```
                if(t1 == (self.__x, self.__y, self.__z) or t1 == (self.__x, self.__z, self.__y)):
                        return True
                elif(t1 == (self.__y, self.__x, self.__z) or t1 == (self.__z, self.__x, self.__y)):
                        return True
                elif(t1 == (self.__z, self.__y, self.__x) or t1 == (self.__y, self.__z, self.__x)):
                        return True
                return False

        ## @brief Calculates the perimeter of the triangle
        # @return Integer representing the perimeter of the current TriangleT
        def perim(self):
                return self.__x + self.__y + self.__z

        ## @brief Calculates the area of the triangle using Heron's Formula
        # @return float representing the area of the current TriangleT
        def area(self):
                p = self.perim()/2
                return math.sqrt(p*(p-self.__x)*(p-self.__y)*(p-self.__z))

        ## @brief Checks that the 3 sides of the current object form a valid triangle
        # @return True if the triangle is valid, False otherwise
        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 Classifies the type of the current TriangleT object
        # @return A TriType representing the type of the current TriangleT object
        def tri_type(self):
                if(self.__x == self.__y and self.__y == self.__z):
                        return TriType(1)#equilat
                elif(self.__x == self.__y  or  self.__x == self.__z  or  self.__y == self.__z):
                        return TriType(2)#isosceles
                elif(self.__x**2+self.__y**2==self.__z**2  or  self.__x**2+self.__z**2==self.__y**2
                    or  self.__y**2+self.__z**2==self.__x**2):
                        return TriType(4)#right
                else:
                        return TriType(3)#scalene


## @brief An enumerated set {equilat, isosceles, scalene, right}
# @details Used to classify the type of a TriangleT object
class TriType(Enum):
        equilat = 1
        isosceles = 2
        scalene = 3
        right = 4
```