Assignment 2 Solution

Safwan Hossain

February 25, 2021

This report discusses the testing phase for .... It also discusses the results of running the same tests on the partner files. The assignment specifications are then critiqued and the requested discussion questions are answered.

1 Testing of the Original Program

I passed 11/15 method cases, which is 24/38 total test cases passed. In my previous test module, it said that I passed every case, but there was a small bug in my **cm()** method where the method was adding instead of multiplying in the inner loop, resulting in a test failure for the methods **cm()**, **cm_x()**, **cm_y()** and **m_inert()** inside the class BodyT. The test rationale was to test a normal case by using random value, a boundary case (where possible) by using a number close to the least or greatest accepted number, and an exception test by trying to test the exception directly if there is an exception. Note that my **Plot.py** did not properly push, but both **Plot.py** and **test_All.py** are now updated.

2 Results of Testing Partner's Code

My partner passed all tests in this assignment as well as the previous project. As mentioned in the previous section my **cm()** method took the sum of the sum of each element in list 1 and list 2, when it shouldve been taking the sum of the multiplication of each element of the same index between list 1 and list 2.

3 Critique of Given Design Specification

The program is consistent with its design, as all naming conventions and ordering of variables are consistent. For example, **x** and **y** were used as the names for the variables that

represent coordinates of a shape and they were always called first in the respective order (**x** first, then **y**, then other variables). The design had excellent essentiality. Only necessary methods were created, and there were no methods that couldve been replaced/used better by an existing method. The design was minimal because all methods focused on a single task, and no method did multiple different tasks. The design was also made to be general. **CircleT** and **TriangleT** allow for any regular circle or equilateral triangle to be created, and **BodyT** allows for any irregular shape to be created, which means that **Scene.py** can be given practically any 2D shape with existing mass and dimensions. The design also leaves room for other shape modules to be added since every module calculates and keeps track of its own properties. The design has high cohesion as all modules within each class relate to one another since they all keep track of their class objects properties. The design also has low coupling because no modules strongly depend on another module. For example, **TriangleT**, **CircleT**, and **BodyT** only rely on **Shape.py** for structure and are entirely independent otherwise. The interface does not allow the user to check for exceptions because the exceptions in the design must be checked from outside the design. All exceptions are inside constructor methods, which means that if the exception is raised, the constructor does not create an object, so the constructors class cannot have a method to check if the object does not exist. Although there are no checks for the exceptions, there are errors raised in the console to let the user know that their numbers are invalid.

4 Answers

a) I dont think that testing getters and setters is necessary for small projects. In small projects, the implementation would typically be simple and straightforward. If the getter or setter has code that does more (e.g. modifying values before setting) than just getting/setting, they should be unit tested to make sure that the entire code works. Also, when dealing with larger projects, I think that spending a few more minutes to test the getters and setters could potentially save a lot of time because other methods can return an error if the getter/setter is implemented incorrectly, and it may take a while to trace the problem back to the getter and setter.

b) The getter method is used to access private properties of a class from outside the class, so a getter for fx and fy can simply return the function like a variable. The functions can be called by the following snippet of code: get_fx()() or get_fy()(). The setter methods can be implemented the traditional way, by overwriting the current function with the given function. For example: self.fx = fx.

c) Testing Plot.py would be a difficult task because we are trying to make the program check if a graph is right. The only way to test Plot.py would be by checking if a file

exists in the specified save location. Checking if the graph is correct would require the plot values which is difficult for a program to read off a graph image.

d) close_enough(x_{calc}, x_{true}):

- output: $out := \frac{||x_{calc} - x_{true}||}{||x_{true}||} < \epsilon$
- exception: none

e) The specification has exceptions for non-positive values of shape dimensions and mass because neither negative mass nor negative dimensions (height, width, length) exist. Also, objects with zero mass or zero-dimension values are not considered to exist either; if we were to consider shapes with zero mass or dimensional values, then they would have no forces acting on them, making them irrelevant to this interface. Coordinates are relative to a certain point, which means they can be negative because the sign of a coordinate can tell us the direction of a point relative to the coordinate system.

f) The specification $(\neg(s_s > 0 \wedge m_s > 0) \implies \text{ValueError})$ states that if the state invariant is not true, return an error. Let $a = s > 0 \wedge m > 0$, the specification would be $(\neg(a) \implies \text{ValueError})$, which means if a is false, return an error, otherwise do nothing. The variable a can only be false if the state invariant is not satisfied; thus, when a triangle is constructed, the invariant will always be satisfied.

g) roots = [math.sqrt(i) for i in range(5, 20, 2)]

h)
```
def remove_upper(string):
    string2 = ""
    for i in string:
        if not i.isupper():
            string2 += str(i)
    return string2
```

i) Abstraction is the process of focusing on what is important while ignoring what is irrelevant. Generality is the process of solving a more general problem than the one at hand; generalization involves reusing a concept for many problems. These two principles are related because you can become more general by using abstraction. Abstraction will only deal with the more important concepts and dismiss less important concepts; this makes an idea easier to reuse.

j) Generally, it is better for one module to be used by many other modules than many other modules using one module because that way, only one module would need to be

maintained. If one module (lets call it module **a**) depends on many others, then when we make a change to module **a**, we would need to change all the other modules within module **a**.

E Code for Shape.py

```
## @file Shape.py
#  @author Safwan Hossain
#  @brief An interface for modules that implement shape entities
#  @date 2/16/2021

from abc import ABC, abstractmethod


##@brief Shape provides an interface for shape entities
##@details The methods used in this interface are abstract
# and must be overriden by modules that inherit this interface.

class Shape(ABC):

    @abstractmethod
    ## @brief a generic method for getting the x value of the center of
    # mass of a shape
    #  @details should be inherited and overridden for specific shape
    #  entity ADTs
    # @return Float representing the x position of the center of mass

    def cm_x(self):
        pass

    ## @brief a generic method for getting the y value of the center of
    # mass of a shape
    #  @details should be inherited and overridden for specific shape
    #  entity ADTs
    # @return Float representing the y position of the center of mass
    @abstractmethod
    def cm_y(self):
        pass

    ## @brief a generic method for getting the mass of a shape
    #  @details should be inherited and overridden for specific shape
    #  entity ADTs
    # @return Float representing the mass of a shape
    @abstractmethod
    def mass(self):
        pass

    ## @brief a generic method for getting the moment of inertia
    #  @details should be inherited and overridden for specific shape
    #  entity ADTs
    # @return Float representing the moment of inertia of a shape
    @abstractmethod
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Safwan Hossain, Hossam18
#  @brief contains a class for working with circles
#  @date  2/16/2021
from Shape import Shape

## @brief An ADT for handling circles that inherits Shape.py
# @details A circle has an x and y coordinate, a non-zero radius,
# and a non-zero mass
class CircleT(Shape):
    ## @brief constructor for CircleT
    #  @param x Float representing the x coordinate of the circle
    #  @param y Float representing the y coordinate of the circle
    #  @param r Float representing the radius of the circle
    #  @param m Float representing the mass of the circle
    #  @throws ValueError when either mass or radius is 0 or lower

    def __init__(self, x, y, r, m):
        if r <= 0 or m <= 0:
            raise ValueError("mass and radius should be greater than zero")

        self.x = x
        self.y = y
        self.r = r
        self.m = m

    ## @brief gets the x coordinate of the circle
    # @return Float representing the x coordinate of the circle
    def cm_x(self):
        return self.x

    ## @brief gets the y coordinate of the circle
    # @return Float representing the y coordinate of the circle
    def cm_y(self):
        return self.y

    ## @brief gets the mass of the circle
    # @return Float representing the mass of the circle
    def mass(self):
        return self.m

    ## @brief gets the inertia of the circle
    # @details calculates the moment of inertia of the circle using the formula
    # that can be found in
        https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
    # @return Float representing the moment of inertia of the circle
    def m_inert(self):
        return (self.r ** 2 * self.m) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Safwan Hossain
#  @brief ADT for representing equilateral triangles
#  @date 2/16/2021

from Shape import Shape


## @brief An ADT for handling equliateral triangle that inherits Shape.py
# @details An equilateral triangle has an x and y coordinate,
# non-zero (equal) side lengths, and a non-zero mass
class TriangleT(Shape):
    ## @brief constructor for TriangleT
    #  @param x Float representing the x coordinate of the triangle
    #  @param y Float representing the y coordinate of the triangle
    #  @param s Float representing the side lengths of the triangle
    #  @param m Float representing the mass of the triangle
    #  @throws ValueError when either mass or side lengths are 0 or lower

    def __init__(self, x, y, s, m):
        if s <= 0 or m <= 0:
            raise ValueError("mass and side lengths must be greater than zero")

        self.x = x
        self.y = y
        self.s = s
        self.m = m

    ## @brief gets the x coordinate of the triangle
    # @return Float representing the x coordinate of the triangle
    def cm_x(self):
        return self.x

    ## @brief gets the y coordinate of the triangle
    # @return Float representing the y coordinate of the triangle
    def cm_y(self):
        return self.y

    ## @brief gets the mass of the triangle
    # @return Float representing the mass of the triangle
    def mass(self):
        return self.m

    ## @brief gets the inertia of the triangle
    # @details calculates the moment of inertia of the triangle using the formula
    # that can be found in
        https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
    # @return Float representing the moment of inertia of the triangle
    def m_inert(self):
        return (self.m * (self.s ** 2)) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Safwan Hossain
#  @brief ADT for handling multiple shapes
#  @date 2/16/2021

from Shape import Shape


## @brief helper method that adds all the items in a list and returns the total
#  @details adds all the elements of the list and divides by the number of elements
#  in the list and returns the Float value
#  @param m List containing Floats
#  @return total Float representing the addition of all elements of the list
def sum(m):
    total = 0
    for i in m:
        total += i
    return total

## @brief helper method that calculates the moment of inertia of multiple
#  for a collective of objects
# @details calculates the moment of inertia using the formula
# that can be found in
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
#  @param x List containing x coordinates of BodyT
#  @param y List containing y coordinates of BodyT
#  @param m List containing masses of BodyT
#  @return inertia Float representing the moment of inertia of BodyT
def mnom(x, y, m):
    inertia = 0
    for i in range(len(m)):
        inertia += m[i] * (x[i] ** 2 + y[i] ** 2)
    return inertia

## @brief helper method that calculates the center of mass
#  for a collective of objects
#  @details calculates the center of mass using the formula
#  that can be found in
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
#  @param z List containing x coordinates of BodyT
#  @param m List containing masses of BodyT
#  @return Float representing the center of mass of a BodyT
def cm(z, m):
    top = 0
    for i in range(len(m)):
        top += z[i] * m[i]
    return top / sum(m)

## @brief An ADT for handling a collective of multiple shapes that inherits Shape.py
# @details BodyT contains a Float list of x coordinates, a Float list of y coordinates
# and a Float list of masses
class BodyT(Shape):
    ## @brief constructor for BodyT
    #  @param x List containing Floats representing the x coordinate of the BodyT
    #  @param y List containing Floats representing the y coordinate of the BodyT
    #  @param m List containing Floats representing the masses of the BodyT
    #  @throws ValueError when the lengths of the parameters are not of equal
    # length

    def __init__(self, x, y, m):
        if not (len(x) == len(y) == len(m)):
            raise ValueError("all lists must be the same size")
        self.cmx = cm(x, m)
        self.cmy = cm(y, m)
        self.m = sum(m)
        self.moment = mnom(x, y, m) - sum(m) * (cm(x, m) ** 2 + cm(y, m) ** 2)

    ## @brief gets the x portion of the center of mass of the BodyT
    # @return Float representing the x poriton of the
    # center of mass of the BodyT
    def cm_x(self):
        return self.cmx

    ## @brief gets the y portion of the center of mass of the BodyT
    # @return Float representing the y poriton of the
    # center of mass of the BodyT
```

```
def cm_y(self):
    return self.cmy

## @brief gets the overall mass of the BodyT
# @return Float representing the overall mass of the BodyT
def mass(self):
    return self.m

## @brief gets the inertia of the BodyT
# @return Float representing the moment of inertia of the BodyT
def m_inert(self):
    return self.moment
```

I Code for Scene.py

```
## @file Scene.py
#  @author Safwan Hossain
#  @brief ADT for handling shape objects for establishing motion
#  @date 2/16/2021
#  @details
from scipy.integrate import odeint
from Shape import Shape
from SciPy import SciPy


class Scene(Shape, SciPy):

    ## @brief constructor for Scene
    #  @param s shape obj,
    #  @param Float representing the sum of all x forces
    #  @param Float representing sum of all y forces
    #  @param Float representing x velocity
    #  @param Float representing y velocity

    def __init__(self, s, fx, fy, vx, vy):
        self.s = s
        self.fx = fx
        self.fy = fy
        self.vx = vx
        self.vy = vy

    ## @brief gets the shape of the scene
    # @returns Shape s representing the shape of the scene
    def get_shape(self):
        return self.s

    ## @brief gets the unbalanced forces from the scene
    # @returns Float fx representing all the x forces
    # @returns Float fy representing all the y forces
    def get_unbal_forces(self):
        return self.fx, self.fy

    ## @brief gets the initial velocities from the scene
    # @returns Float vx representing the initial x velocity of the shape
    # @returns Float vy representing the initial y velocity of the shape
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief sets the shape of the scene
    # @param s the new shape of the scene
    def set_shape(self, s):
        self.s = s

    ## @brief sets the forces of the scene
    # @param fx the new x force of the scene
    # @param fy the new y force of the scene
    def set_unbal_forces(self, fx, fy):
        self.fx = fx
        self.fy = fy

    ## @brief sets the velocities of the scene
    # @param vx the new x velocity of the shape
    # @param vy the new y velocity of the shape
    def set_init_velo(self, vx, vy):
        self.vx = vx
        self.vy = vy

    ## @brief
    # @param t_final a Float value representing final time
    # @param nsteps representing number of steps
    # @return List containing Floats that represent time
    # @reutrn List containing Floats that represent the solution.
    def sim(self, t_final, nsteps):
        t = []
        for i in range * nsteps:
            t.append((i * t_final) / (nsteps - 1))
        return t, odeint(self.ode, [self.s.cm_x(), self.s.cm_y(), self.Vx, self.Vy], t)

    ## @brief a method for finding the ode
    # @details This method is used to find the ode more details about ode can
    # be found in https://en.wikipedia.org/wiki/Ordinary_differential_equation
```

```
# @param w List containing Floats that represent rx, ry, vx and vy
# @param t List containing Floats that represent time
# @return List of Floats that represent the solution
def ode(self, w, t):
    return [w[2], w[3], self.Fx(t) / self.s.mass(), self.Fy(t) / self.s.mass()]
```

J Code for Plot.py

```
## @file Plot.py
#  @author Safwan Hossain, hossam18
#  @brief Plots the x and y positions of a shape in Scene.py
#  @date 2/16/2021
#  @details Takes the x, y and time values from lists and compares first
#  the x values to time, then y values to time, then x and y values without
#  time.


import matplotlib.pyplot as plt

## @brief method for plotting moving shapes in Scene.py
#  @param w List containing another list of Floats, where the first
#  index is the x value and the second index is the y value of a shape
#  @param t List containing Floats of time
#  @throws ValueError if lists are not all the same size

def plot(w, t):
    list_x = []
    list_y = []

    for i in w:
        list_x.append(i[0])
        list_y.append(i[1])

    if len(list_x) != len(t):
        raise ValueError("Lists are not of same length")

    plt.figure(1)

    plt.subplot(3, 1, 1)
    plt.plot(t, list_x, "k")
    plt.xlabel("time(s)")
    plt.ylabel("x(m)")

    plt.subplot(3, 1, 2)
    plt.plot(t, list_y, "k")
    plt.xlabel("time(s)")
    plt.ylabel("y(m)")

    plt.subplot(3, 1, 3)
    plt.plot(list_x, list_y, "k")
    plt.xlabel("x(m)")
    plt.ylabel("y(m)")

    # plt.show()
```

K Code for test_All.py

```
from CircleT import CircleT
from TriangleT import TriangleT
from BodyT import BodyT


def compareFloats(f1, f2):
    return abs(f1 - f2) < 10e-9


## @brief helper method that adds all the items in a list and returns the total
#  @details adds all the elements of the list and divides by the number of elements
#  in the list and returns the Float value
#  @param m List containing Floats
#  @return total Float representing the addition of all elements of the list
def sum(m):
    total = 0
    for i in m:
        total += i
    return total


## @brief helper method that calculates the moment of inertia of multiple
#  for a collective of objects
# @details calculates the moment of inertia using the formula
# that can be found in
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
#  @param x List containing x coordinates of BodyT
#  @param y List containing y coordinates of BodyT
#  @param m List containing masses of BodyT
#  @return inertia Float representing the moment of inertia of BodyT
def mnom(x, y, m):
    inertia = 0
    for i in range(len(m)):
        inertia += m[i] * (x[i] ** 2 + y[i] ** 2)
    return inertia


## @brief helper method that calculates the center of mass
#  for a collective of objects
#  @details calculates the center of mass using the formula
#  that can be found in
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/A2/A2.pdf
#  @param z List containing x coordinates of BodyT
#  @param m List containing masses of BodyT
#  @return Float representing the center of mass of a BodyT
def cm(z, m):
    top = 0
    for i in range(len(m)):
        top += z[i] * m[i]
    return top / sum(m)


def circle_cm_x_test():
    global counter
    c1 = CircleT(12, 13, 15, 10)
    c2 = CircleT(1, 2, 0.0001, 0.0000001)

    try:
        assert compareFloats(c1.cm_x(), 12)
        assert compareFloats(c2.cm_x(), 1)
        counter += 1
        print("CircleT cm_x() test passed")
    except:
        print("CircleT cm_x() test FAILED")


def circle_cm_y_test():
    global counter
    c1 = CircleT(12, 13, 15, 10)
    c2 = CircleT(1, 2, 0.0001, 0.0000001)

    try:
        assert compareFloats(c1.cm_y(), 13)
        assert compareFloats(c2.cm_y(), 2)
        counter += 1
        print("CircleT cm_y() test passed")
```

```
    except:
        print("CircleT cm_y() test FAILED")


def circle_mass_test():
    global counter
    c1 = CircleT(23, 43, 1, 10000000)
    c2 = CircleT(3, 4, 29, 0.00001)
    c3 = CircleT(80, 1.5, 71, 132)

    try:
        assert compareFloats(c1.mass(), 10000000)
        assert compareFloats(c2.mass(), 0.00001)
        assert compareFloats(c3.mass(), 132)
        counter += 1
        print("CircleT mass() test passed")
    except:
        print("CircleT mass() test FAILED")


def circle_m_inert_test():
    global counter
    c1 = CircleT(3, 4, 1355555, 12)
    c2 = CircleT(1, 2, 0.00000001, 4)
    c3 = CircleT(25, 25, 1, 1)

    try:
        assert compareFloats(c1.m_inert(), (1355555 ** 2 * 12) / 2)
        assert compareFloats(c2.m_inert(), 0.00000001 ** 2 * 2)
        assert compareFloats(c3.m_inert(), 0.5)
        counter += 1
        print("CircleT m_inert() test passed")
    except:
        print("CircleT m_inert() test FAILED")


def triangle_cm_x_test():
    global counter
    c1 = TriangleT(13, 42, 15, 104)
    c2 = TriangleT(1000000, 23, 12, 0.000001)

    try:
        assert compareFloats(c1.cm_x(), 13)
        assert compareFloats(c2.cm_x(), 1000000)
        counter += 1
        print("TriangleT cm_x() test passed")
    except:
        print("TriangleT cm_x() test FAILED")


def triangle_cm_y_test():
    global counter
    c1 = TriangleT(13, 42, 15, 104)
    c2 = TriangleT(1000000, 23, 12, 0.000001)

    try:
        assert compareFloats(c1.cm_y(), 42)
        assert compareFloats(c2.cm_y(), 23)
        counter += 1
        print("TriangleT cm_y() test passed")
    except:
        print("TriangleT cm_y() test FAILED")


def triangle_mass_test():
    global counter
    c1 = TriangleT(13, 42, 15, 1000004)
    c2 = TriangleT(1000000, 23, 12, 0.00000001)

    try:
        assert compareFloats(c1.mass(), 1000004)
        assert compareFloats(c2.mass(), 0.00000001)
        counter += 1
        print("TriangleT mass() test passed")
    except:
        print("TriangleT mass() test FAILED")


def triangle_m_inert_test():
    global counter
```

```
    c1 = TriangleT(13, 42, 15, 1000004)
    c2 = TriangleT(1000000, 23, 12, 0.00000001)

    try:
        assert compareFloats(c1.m_inert(), (15 ** 2 * 1000004) / 12)
        assert compareFloats(c2.m_inert(), (12 ** 2 * 0.00000001) / 12)
        counter += 1
        print("TriangleT m_inert() test passed")
    except:
        print("TriangleT m_inert() test FAILED")


def body_cm_x_test():
    global counter
    x1 = [12, 21, 3, 1]
    y1 = [3, 1, 5, 9]
    m1 = [41, 3, 31, 91]

    body1 = BodyT(x1, y1, m1)

    x2 = [2, 25, 35, 11]
    y2 = [13, 21, 4, 29]
    m2 = [41, 31, 31, 191]

    body2 = BodyT(x2, y2, m2)

    x3 = [12]
    y3 = [3]
    m3 = [41]

    body3 = BodyT(x3, y3, m3)

    try:
        print(body1.cm_x())
        print('expected')
        print(cm(x1, m1))
        assert compareFloats(body1.cm_x(), cm(x1, m1))
        assert compareFloats(body2.cm_x(), cm(x2, m2))
        assert compareFloats(body3.cm_x(), cm(x3, m3))
        counter += 1
        print("BodyT cm_x() test passed")
    except:
        print("BodyT cm_x() test FAILED")


def body_cm_y_test():
    global counter
    x1 = [12, 21, 3, 1]
    y1 = [3, 1, 5, 9]
    m1 = [41, 3, 31, 91]

    body1 = BodyT(x1, y1, m1)

    x2 = [2, 25, 35, 11]
    y2 = [13, 21, 4, 29]
    m2 = [41, 31, 31, 191]

    body2 = BodyT(x2, y2, m2)

    x3 = [12]
    y3 = [3]
    m3 = [41]

    body3 = BodyT(x3, y3, m3)

    try:
        assert compareFloats(body1.cm_y(), cm(y1, m1))
        assert compareFloats(body2.cm_y(), cm(y2, m2))
        assert compareFloats(body3.cm_y(), cm(y3, m3))
        counter += 1
        print("BodyT cm_y() test passed")
    except:
        print("BodyT cm_y() test FAILED")


def body_mass_test():
    global counter
    x1 = [12, 21, 3, 1]
    y1 = [3, 1, 5, 9]
    m1 = [41, 3, 31, 91]
```

```
    body1 = BodyT(x1, y1, m1)

    x2 = [2, 25, 35, 11]
    y2 = [13, 21, 4, 29]
    m2 = [41, 31, 31, 191]

    body2 = BodyT(x2, y2, m2)

    x3 = [12]
    y3 = [3]
    m3 = [41]

    body3 = BodyT(x3, y3, m3)

    try:
        assert compareFloats(body1.mass(), sum(m1))
        assert compareFloats(body2.mass(), sum(m2))
        assert compareFloats(body3.mass(), sum(m3))
        counter += 1
        print("BodyT mass() test passed")
    except:
        print("BodyT mass() test FAILED")


def body_m_inert_test():
    global counter
    x1 = [12, 21, 3, 1]
    y1 = [3, 1, 5, 9]
    m1 = [41, 3, 31, 91]

    body1 = BodyT(x1, y1, m1)

    x2 = [2, 25, 35, 11]
    y2 = [13, 21, 4, 29]
    m2 = [41, 31, 31, 191]

    body2 = BodyT(x2, y2, m2)

    x3 = [12]
    y3 = [3]
    m3 = [41]

    body3 = BodyT(x3, y3, m3)

    inert1 = mnom(x1, y1, m1) - sum(m1) * (cm(x1, m1) ** 2 + cm(y1, m1) ** 2)
    inert2 = mnom(x2, y2, m2) - sum(m2) * (cm(x2, m2) ** 2 + cm(y2, m2) ** 2)
    inert3 = mnom(x3, y3, m3) - sum(m3) * (cm(x3, m3) ** 2 + cm(y3, m3) ** 2)

    try:
        assert compareFloats(body1.m_inert(), inert1)
        assert compareFloats(body2.m_inert(), inert2)
        assert compareFloats(body3.m_inert(), inert3)
        counter += 1
        print("BodyT m_inert() test passed")
    except:
        print("BodyT m_inert() test FAILED")


def sum_test():
    global counter
    list1 = [1, 2, 3, 12, 23, 2]
    list2 = [5, 5]
    list3 = [1, 2, 10, 1, 2, 3, 4, 5]

    sum1 = 0
    sum2 = 0
    sum3 = 0

    for i in list1:
        sum1 += i

    for i in list2:
        sum2 += i

    for i in list3:
        sum3 += i

    try:
        assert compareFloats(sum(list1), sum1)
```

```
        assert compareFloats(sum(list2), sum2)
        assert compareFloats(sum(list3), sum3)
        counter += 1
        print("sum() test passed")
    except:
        print("sum() test FAILED")


def cm_test():
    global counter
    list1 = [1, 2, 3, 12, 23, 2]
    list2 = [5, 5, 2, 3, 4, 5]
    list3 = [1, 2]
    list4 = [12222, 304955]

    total = 0
    for i in range(len(list1)):
        total += list1[i] * list2[i]
    cm1 = total / sum(list2)
    cm2 = total / sum(list1)

    total = 0
    for i in range(len(list4)):
        total += list3[i] * list4[i]
    cm3 = total / sum(list4)

    try:
        assert compareFloats(cm(list1, list2), cm1)
        assert compareFloats(cm(list2, list1), cm2)
        assert compareFloats(cm(list3, list4), cm3)
        counter += 1
        print("cm() test passed")
    except:
        print("cm() test FAILED")


def mnom_test():
    global counter
    list1 = [13, 29, 54, 33]
    list2 = [53, 55, 33, 22]
    list3 = [11, 13, 14, 123]
    sum1 = 0
    for i in range(len(list3)):
        sum1 += list3[i] * (list1[i] ** 2 + list2[i] ** 2)

    list4 = [312, 22, 231213, 232, 123, 4324]
    list5 = [321, 321, 213, 132, 213, 3441]
    list6 = [688999, 674, 98, 754, 123, 34]
    sum2 = 0
    for i in range(len(list4)):
        sum2 += list6[i] * (list4[i] ** 2 + list5[i] ** 2)
    try:
        assert compareFloats(mnom(list1, list2, list3), sum1)
        assert compareFloats(mnom(list4, list5, list6), sum2)
        counter += 1
        print("mnom() test passed")
    except:
        print("mnom() test FAILED")


counter = 0
circle_cm_x_test()
circle_cm_y_test()
circle_m_inert_test()
circle_mass_test()
triangle_cm_x_test()
triangle_cm_y_test()
triangle_m_inert_test()
triangle_mass_test()
body_cm_x_test()
body_cm_y_test()
body_mass_test()
body_m_inert_test()
cm_test()
sum_test()
mnom_test()
print("")
print("Passed " + str(counter) + " / 15 tests")
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Samarth Kumar
#  @brief Contains a class for circle objects
#  @date Feb. 12th, 2021

from Shape import Shape


## @brief An ADT for handling circles
#  @details Every circle has a mass, radius,
#  moment of inertia, and x and y coordinates
#  representing its center of mass
class CircleT(Shape):

    ## @brief Constructor for CircleT objects
    #  @details It is assumed that the arguments provided to
    #  the access programs will be of the correct type
    #  @param x Float representing the x coordinate
    #  @param y Float representing the y coordinate
    #  @param r Float representing the radius
    #  @param m Float representing the mass
    #  @throws ValueError if the radius or mass is 0 or negative
    def __init__(self, x, y, r, m):
        if not(r > 0 and m > 0):
            raise ValueError
        self.x = x
        self.y = y
        self.r = r
        self.m = m

    ## @brief Gets the x coordinate center of mass of the circle
    #  @return Float representing the x coordinate
    def cm_x(self):
        return self.x

    ## @brief Gets the y coordinate center of mass of the circle
    #  @return Float representing the y coordinate
    def cm_y(self):
        return self.y

    ## @brief Gets the mass of the circle
    #  @return Float representing the mass
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the circle
    #  @return Float representing the moment of inertia
    def m_inert(self):
        return self.m * (self.r * self.r) / 2
```

M Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Samarth Kumar
#  @brief Contains a class for triangle objects
#  @date Feb. 12th, 2021

from Shape import Shape


## @brief An ADT for handling equilateral triangles
#  @details Every triangle has 3 equal sidelengths,
#  a mass, moment of inertia, and x and y
#  coordinates representing its center of mass
class TriangleT(Shape):

    ## @brief Constructor for TriangleT objects
    #  @details It is assumed that the arguments provided to
    #  the access programs will be of the correct type
    #  @param x Float representing the x coordinate
    #  @param y Float representing the y coordinate
    #  @param s Float representing one sidelength of the equilateral triangle
    #  @param m Float representing the mass
    #  @throws ValueError if a sidelength or mass is 0 or negative
    def __init__(self, x, y, s, m):
        if not(s > 0 and m > 0):
            raise ValueError
        self.x = x
        self.y = y
        self.s = s
        self.m = m

    ## @brief Gets the x coordinate center of mass of the triangle
    #  @return Float representing the x coordinate
    def cm_x(self):
        return self.x

    ## @brief Gets the y coordinate center of mass of the triangle
    #  @return Float representing the y coordinate
    def cm_y(self):
        return self.y

    ## @brief Gets the mass of the triangle
    #  @return Float representing the mass
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the triangle
    #  @return Float representing the moment of inertia
    def m_inert(self):
        return self.m * (self.s * self.s) / 12
```

N Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Samarth Kumar
#  @brief Contains a class for BodyT objects
#  @date Feb. 12th, 2021

from Shape import Shape
from functools import reduce


## @brief An ADT for handling BodyT objects
#  @details Every BodyT object has a sequence of masses,
#  each with x and y center of mass coordinates. It also has a
#  total mass and moment of inertia.
class BodyT(Shape):

    ## @brief Constructor for BodyT objects
    #  @details It is assumed that the arguments provided to
    #  the access programs will be of the correct type
    #  @param x Sequence of Floats representing the x coordinates
    #  for each center of mass
    #  @param y Sequence of Floats representing the y coordinates
    #  for each center of mass
    #  @param m Sequence of Floats representing the masses
    #  @throws ValueError if sequences are not all the same size,
    #  a sequence is empty, or at least one of the masses is 0 or negative
    def __init__(self, x, y, m):
        if not(len(x) == len(y) == len(m)):
            raise ValueError
        if not reduce(lambda u, v: u and v, [u > 0 for u in m], True):
            raise ValueError
        # Empty lists
        if (len(x) == 0):
            raise ValueError

        self.cmx = sum([x[i] * m[i] for i in range(len(m))]) / sum(m)
        self.cmy = sum([y[i] * m[i] for i in range(len(m))]) / sum(m)
        self.m = sum(m)
        self.moment = sum([m[i] * (x[i]**2 + y[i]**2) for i in range(len(m))]) \
            - sum(m) * ((self.cmx**2) + self.cmy**2)

    ## @brief Gets the x coordinate center of mass of the BodyT object
    #  @return Float representing the x coordinate
    def cm_x(self):
        return self.cmx

    ## @brief Gets the y coordinate center of mass of the BodyT object
    #  @return Float representing the y coordinate
    def cm_y(self):
        return self.cmy

    ## @brief Gets the total mass of the BodyT object
    #  @return Float representing the mass
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the BodyT object
    #  @return Float representing the moment of inertia
    def m_inert(self):
        return self.moment
```

O Code for Partner's Scene.py

```
## @file Scene.py
#  @author Samarth Kumar
#  @brief Contains a class for a Scene
#  @date Feb. 12th, 2021

import scipy.integrate as sp


## @brief A class for creating a scene
#  @details Used to simulate movement of a shape in a scene, based on its
#  shape properties, initial velocity, and unbalanced external forces
class Scene():

    ## @brief Constructor for Scene objects
    #  @param s Shape object
    #  @param F_x Function: Float -> Float,
    #  representing the unbalanced force in the x direction at a given time
    #  @param F_y Function: Float -> Float,
    #  representing the unbalanced force in the y direction at a given time
    #  @param v_x Float representing the initial velocity in the x direction
    #  @param v_y Float representing the initial velocity in the y direction
    def __init__(self, s, F_x, F_y, v_x, v_y):
        self.s = s
        self.F_x = F_x
        self.F_y = F_y
        self.v_x = v_x
        self.v_y = v_y

    ## @brief Gets the shape in the scene
    #  @return Shape object representing the shape in the scene
    def get_shape(self):
        return self.s

    ## @brief Gets the unbalanced forces in the scene
    #  @return Tuple of Functions: Float -> Float, representing the
    #  unbalanced forces in the x and y directions
    def get_unbal_forces(self):
        return self.F_x, self.F_y

    ## @brief Gets the initial velocities in the scene
    #  @return Tuple of Floats representing the initial velocities
    #  in the x and y directions
    def get_init_velo(self):
        return self.v_x, self.v_y

    ## @brief Sets the shape of the scene
    #  @param s Shape object
    def set_shape(self, s):
        self.s = s

    ## @brief Sets the unbalanced forces of the scene
    #  @param F_x Function: Float -> Float,
    #  representing the unbalanced force in the x direction at a given time
    #  @param F_y Function: Float -> Float,
    #  representing the unbalanced force in the y direction at a given time
    def set_unbal_forces(self, F_x, F_y):
        self.F_x = F_x
        self.F_y = F_y

    ## @brief Sets the initial velocities of the scene
    #  @param v_x Float representing the initial velocity in the x direction
    #  @param v_y Float representing the initial velocity in the y direction
    def set_init_velo(self, v_x, v_y):
        self.v_x = v_x
        self.v_y = v_y

    ## @brief Simulation based on the properties of the scene
    #  @param t_final Float representing the time to simulate up to (in seconds)
    #  @param nsteps Integer representing the number of steps to divide
    #  the time interval into
    #  @returns Tuple of Sequences, where the first sequence represents the
    #  time values, and the second sequence is the result of the solved ODE
    def sim(self, t_final, nsteps):
        t = [i * t_final / (nsteps - 1) for i in range(nsteps)]
        return t, sp.odeint(lambda w, t: [w[2], w[3], self.F_x(t) / self.s.mass(),
                            self.F_y(t) / self.s.mass()], [self.s.cm_x(), self.s.cm_y(),
```

```
        self.v_x, self.v_y], t)
```