Assignment 2 Solution

Aamina Hussain

February 25, 2021

This report discusses the testing of the `CircleT`, `TriangleT`, `BodyT`, and `Scene` classes written for Assignment 2. 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 simply tested the getters for CircleT.py, TriangleT.py, and BodyT.py using one test case each. This is because any boundary cases are included in the exceptions, and were therefore tested separately. Other than the exceptions, there were no condition statements to test. For CircleT.py, a ValueError exception was to be raised if the radius or mass of the circle was not greater than zero. test_circle_value_error in my test_driver.py file tested if the exception would be raised if the radius was zero or negative. test_circle_value_error_2 tested if the exception would be raised if the mass was zero or negative. test_circle_value_error_3 tested if the exception would be raised if the radius and the mass were zero or negative. The same went for TriangleT.py. A ValueError should have been raised if the side length or the mass was not above zero. test_triangle_value_error tested if the exception would be raised if the side length was zero or negative. test_triangle_value_error_2 tested if the exception would be raised if the mass was zero or negative. test_triangle_value_error_3 tested if the exception would be raised if the radius and the mass were zero or negative. For BodyT.py, a ValueError should have been raised if the length of the three sequences given as inputs were not the same. This was tested using test_body_value_error which tried to create an object with sequences of different lengths. Another ValueError should have been raised if any value in the mass sequence was not greater than zero. This was tested using test_body_value_error_2 which tried to create an object using a sequence for mass that had zeros or negative numbers.

2 Results of Testing Partner's Code

There were no failed test cases when I tested my partners code using my test_driver.py file, and the same graphs were produced. Our code for CircleT.py and TriangleT.py were basically the same. For BodyT.py, I did not explicitly define the local functions. Instead I just implemented what the local function would do within the constructor. My partner, on the other hand, defined private local functions, and then called on those functions within the constructor method. While I used list comprehension and imported reduce from functools, my partner used for loops to create the desired sequences. For Scene.py, while I defined the local ode function inside of the sim function, my partner defined the ode function outside of sim, as a separate function. Although we implemented many functions differently, or did not implement them at all (I did not explicitly implement the local functions in BodyT.py), we still both passed the same test cases. This shows that there is still a level of abstraction even though we are following the MIS, since it tells us where to go, but not how to get there. I think one reason why there were no failed test cases is because we were all following the MIS given, unlike assignment one where there was no MIS and as a result there were more ambiguities.

3 Critique of Given Design Specification

The design specification had many strengths, including modularity and abstraction. Modularity because the methods were independent from each other, but were still related. Some of the individual methods could be used in multiple scenarios, not only in the environment they were presented in. Abstraction because the MIS only told us what the final result should be for each method, but not how we should implement it. This allowed us more degrees of freedom, and we were able to choose the most efficient way to implement the design. Another strength was formality. Because we used the MIS for this assignment, the expectations were provided to us in a mathematical form. This allowed there to be no ambiguities, unlike assignment one. This formality allowed all exceptions to be specified, so I knew where to raise exceptions and where to make assumptions if necessary. The inputs, outputs, and their types were clear, along with which functions returned a value and which functions just made a transition. The modules were also all essential; none of the modules could have been done using a combination of the other modules. However, I feel as though the specification for the Plot.py did not properly specify some things. For example, if the image of the graphs was not shown in the A2 pdf, then I would not know that the three graphs should be stacked on top of each other, or displayed in a single window. I would also not know what the title or axes titles would be. I think in this case it is easier to explain what you want the graph to look like by showing an image, however, since we are using the MIS specification for this assignment, it should be consistent and

use it for all of the modules we need to implement.

4 Answers

a) Yes, getters and setters should be unit tested. Unit testing getters is a good way to ensure that the values have been assigned to the correct corresponding state variable. Unit testing is necessary for the setters as well, for similar reasons. It is a good way to ensure that the state variable is actually being changed. Issues with setters and getters could result in issues with the rest of the code, since setters and getters are usually used when implementing other methods. It is important for every executable part of the code to be tested.

b) I would test the getters by creating a Scene object. Then I would print(object.get_unbal_forces()) and see if they return the same functions I set the unbalanced forces as. To test the setters, I would first check what the original function the unbalanced forces are for that object by print(object.get_unbal_forces()). Then I would set the forces to different functions (any other function) using object.set_unbal_forces(fxnA, fxnB). Then I would print(object.get_unbal_forces()) again to check if the function changed. This is what I would do if I did not have to perform an automated test.

c) Since matplotlib can generate and save a file for the plot you make, in order to perform an automated test, I could compare the two files. This can be done by reading the rows of pixels for each file, and seeing if they are equal. Two pixels are equal if they are the same color. If all the pixels for the two files are the same, then the two graphs are the same, and the test passes. If they are not the same, then the test fails.

d) Routine name - close_enough
In - x_calc : seq of $\mathbb{R}$, x_true : seq of $\mathbb{R}$
Out - $\mathbb{B}$

close_enough(x_calc, x_true):

- *out* := very_close(subtract(x_calc, x_true), x_true)
- exception: $\neg(|x\_calc| = |x\_true|) \Rightarrow$ ValueError

Local Functions

subtract : seq of $\mathbb{R}\times$ seq of $\mathbb{R} \Rightarrow$ seq of $\mathbb{R}$
subtract(x, y) $\equiv [i : \mathbb{N} | i \in [0..|x| - 1] : x_i - y_i]$

abs : seq of $\mathbb{R} \Rightarrow$ seq of $\mathbb{R}$
abs(x) $\equiv [i : \mathbb{N} | i \in [0..|x| - 1] : |x_i|]$

max: seq of $\mathbb{R} \Rightarrow \mathbb{R}$
max(x) $\equiv$ the largest value in the sequence

very_close : seq of $\mathbb{R}\times$ seq of $\mathbb{R} \Rightarrow \mathbb{B}$
very_close(x, y) $\equiv max(abs(x))/max(abs(y)) < \epsilon$

e) No, there should not be exceptions for negative coordinates. There were exceptions for shape dimensions and mass because you cannot have a negative side length of a shape for example. Coordinates, however, are arbitrary positions in space, and they are used to see the position of something with reference to the positions of everything else on the coordinate grid. For example, if shape A was at (0, 0) and shape B was at (0, -5), then shape B would be 5 units below shape A. This shows that the positions are relative, and negative coordinates are not incorrect.

f) This invariant is always satisfied because in the specification there is an exception when you are creating a new TriangleT. If s and m are both not greater than zero, then the TriangleT constructor raises a ValueError exception. This means a new TriangleT will not even be created unless both s and m are greater than zero. Therefore, the invariant is always satisfied.

g)
```
from math import sqrt
question_g = [sqrt(i) for i in range(5, 20) if i%2 == 1]
```

h)
```
def remove_uppercase_letters(word):
    split_words = [letter for letter in word if letter.islower()]
    word = ""
    for i in range(len(split_words)):
        word += split_words[i]
    return word
```

i) Abstraction and generality are related by the fact that abstraction can be used to create a more general solution. A design is abstract if it focuses on what is important and disregards anything irrelevant. For example, it could tell you what the final outcome must be without telling you how you must produce that outcome. A design that is general allows you to use that design for multiple purposes. A design is generalized by using abstraction since the higher level of abstraction means more degrees of freedom, and these degrees of freedom allow the design to be generalized.

j) A module that is used by many other modules is a better scenario. This means that module is portable, and can be used in different environments. It also means it is maintainable, since if you modify that single module, it will automatically modify the other modules that use it. In the other scenario, where one module uses many modules, is not desirable because the module is highly coupled. Since the module depends on many other modules, it has to wait for all the other modules to be completed before it can be used or tested.

E Code for Shape.py

```
## @file Shape.py
#  @author Aamina Hussain
#  @brief An interface for modules that implement shapes
#  @date 02/16/2021

from abc import ABC, abstractmethod


## @brief Shape provides an interface for shapes
## @details The abstract methods in this interface
#  are overridden by the modules that inherit it

class Shape(ABC):

    @abstractmethod
    ## @brief a generic method for getting the x-component of the
    #  centre of mass of a shape
    #  @return a float representing the x-component of the centre
    #  of mass of a shape
    def cm_x(self):
        pass

    @abstractmethod
    ## @brief a generic method for getting the y-component of the
    #  centre of mass of a shape
    #  @return a float representing the y-component of the centre
    #  of mass of a shape
    def cm_y(self):
        pass

    @abstractmethod
    ## @brief a generic method for getting the mass of a shape
    #  @return a float representing the mass of a shape
    def mass(self):
        pass

    @abstractmethod
    ## @brief a generic method for getting the moment of inertia of a shape
    #  @return a float representing the moment of inertia of a shape
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Aamina Hussain
#  @brief Contains the type CircleT to represent circles
#  @date 02/16/2021

from Shape import Shape


## @brief CircleT represents circles

class CircleT(Shape):

    ## @brief constructor for class CircleT
    #  @param x a real number that represents the x-component
    #  centre of mass of the circle
    #  @param y a real number that represents the y-component
    #  centre of mass of the circle
    #  @param r a real number that represents the position of
    #  the circle
    #  @param m a real number that represents the mass of the circle
    #  @throws ValueError if the position or mass are not greater
    #  than zero
    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 getter for circle's centre of mass x-component
    #  @return a float representing the circle's centre of
    #  mass x-component
    def cm_x(self):
        return self.x

    ## @brief getter for circle's centre of mass y-component
    #  @return a float representing the circle's centre of
    #  mass y-component
    def cm_y(self):
        return self.y

    ## @brief getter for circle's mass
    #  @return a float representing the circle's mass
    def mass(self):
        return self.m

    ## @brief getter for circle's moment of inertia
    #  @return a float representing the circle's moment of
    #  inertia
    def m_inert(self):
        return (self.m * (self.r**2)) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Aamina Hussain
#  @brief Contains the type TriangleT to represent triangles
#  @date 02/16/2021

from Shape import Shape


## @brief TriangleT represents equilateral triangles

class TriangleT(Shape):

    ## @brief constructor for class TriangleT
    #  @param x a real number that represents the x-component
    #  centre of mass of the triangle
    #  @param y a real number that represents the y-component
    #  centre of mass of the triangle
    #  @param s a real number that represents the side length of
    #  the triangle
    #  @param m a real number that represents the mass of the circle
    #  @throws ValueError if the side length or mass are not greater
    #  than zero
    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 getter for triangles's centre of mass x-component
    #  @return a float representing the triangle's centre of
    #  mass x-component
    def cm_x(self):
        return self.x

    ## @brief getter for triangle's centre of mass y-component
    #  @return a float representing the triangle's centre of
    #  mass y-component
    def cm_y(self):
        return self.y

    ## @brief getter for triangle's mass
    #  @return a float representing the triangle's mass
    def mass(self):
        return self.m

    ## @brief getter for triangle's moment of inertia
    #  @return a float representing the triangle's moment of
    #  inertia
    def m_inert(self):
        return (self.m * (self.s**2)) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Aamina Hussain
#  @brief Contains the type BodyT to represent bodies,
#  or a sequence of masses in space
#  @date 02/16/2021

from Shape import Shape
from functools import reduce


## @brief BodyT represents bodies, which are a sequence
#  of masses in space

class BodyT(Shape):

    ## @brief constructor for class BodyT
    #  @param x a sequence of real numbers
    #  @param y a sequence of real numbers
    #  @param m a sequence of real numbers
    #  @throws ValueError if the length of the sequences x, y, and m are
    #  not equal
    #  @throw ValueError if all the values in sequence m are not greater
    #  than zero
    def __init__(self, x, y, m):
        if not(len(x) == len(y) == len(m)):
            raise ValueError
        if not(reduce(lambda x, y: x > 0 and y > 0, m, True)):
            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))])

    ## @brief getter for body's centre of mass x-component
    #  @return a float representing the body's centre of
    #  mass x-component
    def cm_x(self):
        return self.cmx

    ## @brief getter for body's centre of mass y-component
    #  @return a float representing the body's centre of
    #  mass y-component
    def cm_y(self):
        return self.cmy

    ## @brief getter for body's mass
    #  @return a float representing the body's mass
    def mass(self):
        return self.m

    ## @brief getter for body's moment of inertia
    #  @return a float representing the body's moment of
    #  inertia
    def m_inert(self):
        return self.moment
```

I Code for Scene.py

```
## @file Scene.py
#  @author Aamina Hussain
#  @brief contains the class Scene which integrates
#  to find the motion of the body about the centre of mass
#  @date 02/16/2021

from scipy.integrate import odeint


## @brief Scene provides the forces and initial velocities
#  of a shape, and integrate in order to find the modtion of
#  the body.

class Scene:

    ## @brief constructor for class BodyT
    #  @param s is a Shape
    #  @param fx is a real number representing the unbalanced
    #  force function in the x-direction
    #  @param fy is a real number representing the unbalanced
    #  force function in the y-direction
    #  @param vx is a real number representing the initial
    #  velocity in the x-direction
    #  @param vy is a real number representing the initial
    #  velocity in the y-direction
    def __init__(self, s, fx, fy, vx, vy):
        self.s = s
        self.Fx = fx
        self.Fy = fy
        self.vx = vx
        self.vy = vy

    ## @brief getter for the Shape
    #  @return the Shape
    def get_shape(self):
        return self.s

    ## @brief getter for unbalanced forces
    #  @return floats representing the values of the x-
    #  and y-components of the unbalanced forces
    def get_unbal_forces(self):
        return self.Fx, self.Fy

    ## @brief getter for initial velocities
    #  @return floats representing the values of the x-
    #  and y-components of the initial velocities
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief setter for the Shape
    #  @param new is a Shape that the original Shape will be
    #  replaced with
    def set_shape(self, new):
        self.s = new

    ## @brief setter for the unbalanced forces
    #  @param new_x is a float that the original x-component
    #  of the unbalanced force will be replaced with
    #  @param new_y is a float that the original y-component
    #  of the unbalanced force will be replaced with
    def set_unbal_forces(self, new_x, new_y):
        self.Fx = new_x
        self.Fy = new_y

    ## @brief setter for the initial velocities
    #  @param new_x is a float that the original x-component
    #  of the initial velocity will be replaced with
    #  @param new_y is a float that the original y-component
    #  of the initial velocity will be replaced with
    def set_init_velo(self, new_x, new_y):
        self.vx = new_x
        self.vy = new_y

    ## @brief method that performs integration
    #  @param t_final is a real number
    #  @param nsteps is a natural number
```

```
#  @return a sequence of real numbers and a 2-dimensional
#  sequence of real numbers
def sim(self, t_final, nsteps):
    t = [i * t_final / (nsteps - 1) for i in range(nsteps)]

    def __ode(w, t):
        return [w[2], w[3], self.Fx(t) / self.s.mass(), self.Fy(t) / self.s.mass()]
    return t, odeint(__ode, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
```

J Code for Plot.py

```
## @file Plot.py
#  @author Aamina Hussain
#  @brief contains a method that plots graphs
#  @date 02/16/2021

import matplotlib.pyplot as plt


## @brief plots three graphs that describe the motion
#  of a body
#  @param w is a 2-dimensional sequence of real numbers
#  @param t is a sequence of real numbers
def plot(w, t):
    if not(len(w) == len(t)):
        return ValueError
    x = [w[i][0] for i in range(len(w))]
    y = [w[i][1] for i in range(len(w))]
    fig, axs = plt.subplots(3)
    fig.suptitle('Motion Simulation')
    axs[0].plot(t, x)
    axs[0].set(ylabel='x (m)')
    axs[1].plot(t, y)
    axs[1].set(ylabel='y (m)')
    axs[2].plot(x, y)
    axs[2].set(xlabel='x (m)', ylabel='y (m)')
    plt.show()
```

K Code for test_driver.py

```
## @file test_All.py
#  @author Aamina Hussain
#  @brief tests implementation of python files for graphing
#  the motion of a body
#  @date 02/16/2021

from pytest import *
from CircleT import CircleT
from TriangleT import TriangleT
from BodyT import BodyT


class TestCircleT:

    def setup_method(self, method):
        self.c1 = CircleT(1.0, 10.0, 2.0, 1.0)

    def teardown_method(self, method):
        self.c1 = None

    def test_circle_cm_x(self):
        assert self.c1.cm_x() == 1.0

    def test_circle_cm_y(self):
        assert self.c1.cm_y() == 10.0

    def test_circle_mass(self):
        assert self.c1.mass() == 1.0

    def test_circle_m_inert(self):
        assert self.c1.m_inert() == 2.0

    def test_circle_value_error(self):
        with raises(ValueError):
            CircleT(1, 10, 0, 1)

    def test_circle_value_error_2(self):
        with raises(ValueError):
            CircleT(1, 10, 0.5, -1)

    def test_circle_value_error_3(self):
        with raises(ValueError):
            CircleT(1, 10, -1, 0)


class TestTriangleT:

    def setup_method(self, method):
        self.t1 = TriangleT(1.0, 10.0, 2.0, 3.0)

    def teardown_method(self, method):
        self.t1 = None

    def test_triangle_cm_x(self):
        assert self.t1.cm_x() == 1.0

    def test_triangle_cm_y(self):
        assert self.t1.cm_y() == 10.0

    def test_triangle_mass(self):
        assert self.t1.mass() == 3.0

    def test_triangle_m_inert(self):
        assert self.t1.m_inert() == 1.0

    def test_triangle_value_error(self):
        with raises(ValueError):
            CircleT(1, 10, 0, 3)

    def test_triangle_value_error_2(self):
        with raises(ValueError):
            CircleT(1, 10, 2, -1)

    def test_triangle_value_error_3(self):
        with raises(ValueError):
            CircleT(1, 10, -1, 0)
```

```
class TestBodyT:

    def setup_method(self, method):
        self.b1 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])

    def teardown_method(self, method):
        self.b1 = None

    def test_body_cmx(self):
        assert self.b1.cmx == 0

    def test_body_cmy(self):
        assert self.b1.cmy == 0

    def test_body_mass(self):
        assert self.b1.mass() == 40

    def test_body_m_inert(self):
        assert self.b1.m_inert() == 80

    def test_body_value_error(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10])

    def test_body_value_error_2(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 0, -4])
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Shrill Patel
#  @brief Creates a module representing circle as a shape
#  @date 03/02/21

from Shape import Shape

## @brief Implements an ADT class representing a circle as an object
#  @details Creates a circular object that will abide by the laws of physics.
# It is assumed that the arguments provided to the access programs will be of
# correct type.


class CircleT(Shape):

    ## @brief Constructor for the CircleT class
    #  @details A circle object is made up of the x and y components of its
    # center of mass, a radius, and the mass of the circular object
    #  @param xs A real number representing x-component of the center of mass
    #  @param ys A real number representing y-component of the center of mass
    #  @param rs A real number representing the radius of the circular object
    #  @param ms A real number representing the mass of the circular object
    #  @throws ValueError if the radius and/or mass are less than or equal
    # to zero
    def __init__(self, xs, ys, rs, ms):
        if (not (rs > 0 and ms > 0)):
            raise ValueError("Radius and/or mass cannot be equal to or less",
                             "than zero.")
        self.x = xs
        self.y = ys
        self.r = rs
        self.m = ms

    ## @brief Gets the x-component of the center of mass for the circular
    # object
    #  @return A real number representing the x-component of the center of mass
    def cm_x(self):
        return self.x

    ## @brief Gets the y-component of the center of mass for the circular
    # object
    #  @return A real number representing the y-component of the center of mass
    def cm_y(self):
        return self.y

    ## @brief Gets the mass of a circular object
    #  @return A real number representing the mass of the circular object
    def mass(self):
        return self.m

    ## @brief Calculates the moment of inertia of a circular object
    #  @details This function calculates the moment of inertia by multiplying
    # the mass by the radius squared and then dividing by two
    #  @return A real number representing the moment of inertia of a circular
    # object
    def m_inert(self):
        return (self.mass() * (self.r ** 2)) / 2
```

M Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Shrill Patel
#  @brief Creates a module representing a triangle as a shape
#  @date 03/02/21

from Shape import Shape

## @brief Implements an ADT class representing a triangle as an object
#  @details Creates a triangular object that will abide by the laws of
# physics. It is assumed that the arguments provided to the access programs
# will be of the correct type.


class TriangleT(Shape):

    ## @brief Constructor for the TriangleT class
    #  @details A triangle object is made up of the x and y components of its
    # center of mass, its side length, and the mass of the triangle
    #  @param xs A real number representing x-component of the center of mass
    #  @param ys A real number representing y-component of the center of mass
    #  @param ss A real number representing the side lengths of the triangle
    #  @param ms A real number representing the mass of the triangle
    #  @throws ValueError if side length and/or mass are less than or equal
    # to zero
    def __init__(self, xs, ys, ss, ms):
        if (not (ss > 0 and ms > 0)):
            raise ValueError("Side length and/or mass cannot be equal to or",
                             "less than zero.")
        self.x = xs
        self.y = ys
        self.s = ss
        self.m = ms

    ## @brief Gets the x-component of the center of mass for the triangle
    #  @return A real number representing the x-component of the center of mass
    def cm_x(self):
        return self.x

    ## @brief Gets the y-component of the center of mass for the triangle
    #  @return A real number representing the y-component of the center of mass
    def cm_y(self):
        return self.y

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

    ## @brief Calculates the moment of inertia of a triangle
    #  @details This functions calculates the moment of inertia by multiplying
    # the mass by the side length squared then dividing that by twelve
    #  @return A real number representing the moment of inertia of a triangle
    def m_inert(self):
        return (self.mass() * (self.s ** 2)) / 12
```

N Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Shrill Patel
#  @brief Creates a module representing an irregular object
#  @date 03/02/21

from Shape import Shape

## @brief Implements an ADT class representing an irregular shape
#  @details Creates an irregular shape that follows the laws of physics. It is
# assumed that the arguments provided to the access programs will be of the
# correct type.


class BodyT(Shape):

    ## @brief Constructor for the BodyT class
    #  @details A body is made up of point masses and the x and y coordinates
    # of its center of mass
    #  @param xs A sequence of real numbers representing the x-coordinates of
    # the center of mass
    #  @param ys A sequence of real numbers representing the y-coordinates of
    # the center of mass
    #  @param ms A sequence of real numbers representing the point masses of
    # the body
    #  @throws ValueError if length of xs, ys, and ms are not equal
    #  @throws ValueError if any one of the point masses are less than or equal
    # to zero
    def __init__(self, xs, ys, ms):
        if (not (len(xs) == len(ys) == len(ms))):
            raise ValueError("Sequences of x and y components of the center of",
                             "mass and the mass must be of equal length.")

        for mass in ms:
            if mass <= 0.0:
                raise ValueError("No point masses can have a value of zero.")

        moment = self.__mmom(xs, ys, ms) - self.__sum(ms) * \
            (self.__cm(xs, ms) ** 2 + self.__cm(ys, ms) ** 2)

        if moment < 0:
            raise ValueError("Moment of inertia cannot be less than zero.")
        else:
            self.cmx = self.__cm(xs, ms)
            self.cmy = self.__cm(ys, ms)
            self.m = self.__sum(ms)
            self.moment = moment

    ## @brief Gets the x-component of the center of mass of the irregular
    # object
    #  @return A real number representing the x-component of the center of mass
    def cm_x(self):
        return self.cmx

    ## @brief Gets the y-component of the center of mass of the irregular
    # object
    #  @return A real number representing the y-component of the center of mass
    def cm_y(self):
        return self.cmy

    ## @brief Gets the mass of an irregular object
    #  @return A real number representing the mass of the object
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the irregular object
    #  @return A real number representing the moment of inertia for the object
    def m_inert(self):
        return self.moment

    ## @brief Calculates the total mass of an irregular object
    #  @details This function calculates the mass of an irregular object by
    # summing all of the object's point masses
    #  @param ms A sequence of real numbers
    #  @return A real number representing the total mass of the object
    def __sum(self, ms):
        total_mass = 0.0
```

```
        for mass in ms:
            total_mass += mass
        return total_mass

    ## @brief Calculates the center of mass of an irregular object
    #  @param z A sequence of real numbers
    #  @param m A sequence of real numbers representing point masses
    #  @return A real number representing the center of mass of the object
    def __cm(self, z, m):
        total = 0.0
        for i in range(len(m)):
            total += z[i] * m[i]
        return total / self.__sum(m)

    ## @brief Calculates the irregular object's mass moment
    #  @param x A sequence of real numbers
    #  @param y A sequence of real numbers
    #  @param m A sequence of real numbers representing point masses
    #  @return A real number representing the mass moment of an object
    def __mmom(self, x, y, m):
        total = 0.0
        for i in range(len(m)):
            total += m[i] * ((x[i] ** 2) + (y[i] ** 2))
        return total
```

O Code for Partner's Scene.py

```
## @file Scene.py
#  @author Shrill Patel
#  @brief Creates a module to calculate all of the needed information to plot
# the simulation of a Shape.
#  @date 07/02/21

from scipy import integrate

## @brief This template module will help calculate all necessary data to create
# a simulation of an object in 2D space.
#  @details This module will be in charge of determining the information/data
# needed to plot position vs. time, velocity vs. time, and acceleration vs.
# time graphs which are used to simulate how an object will react to forces in
# 2D space. These simulations are based off of the equation for Newton's
# second law of motion.


class Scene():

    ## @brief Constructor for the Scene module
    #  @details The simulation of a shape in 2D space is created with the data
    # of the shape, unbalanced force function in the x direction, the
    # unbalanced force function in the y direction, the initial velocity in the
    # x direction, and the initial velocity in the y direction.
    #  @param shape A Shape object that the simulation will be based on
    #  @param Fx_prime An unbalanced force function in the x direction
    #  @param Fy_prime An unbalanced force function in the y direction
    #  @param vx_prime The initial velocity of the object in the x direction
    #  @param vy_prime The initial velocity of the object in the y direction
    def __init__(self, shape, Fx_prime, Fy_prime, vx_prime, vy_prime):
        self.s = shape
        self.Fx = Fx_prime
        self.Fy = Fy_prime
        self.vx = vx_prime
        self.vy = vy_prime

    ## @brief Gets the type of shape that is being simulated
    #  @return A shape object depending on the type of shape
    def get_shape(self):
        return self.s

    ## @brief Gets the x and y direction forces acting on the shape
    #  @return A tuple of the x and y components of the forces
    def get_unbal_forces(self):
        return self.Fx, self.Fy

    ## @brief Gets the x and y components of the initial velocity of the object
    #  @return A tuple of the x and y components of the initial velocity
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief Mutates the current shape into another kind of shape
    #  @param s_prime A new type of Shape
    def set_shape(self, s_prime):
        self.s = s_prime

    ## @brief Mutates the unbalanced force functions in both the x and y
    # directions
    #  @param Fx_prime A new function for the forces acting on the object in
    # the x direction
    #  @param Fy_prime A new function for the forces acting on the object in
    # the y direction
    def set_unbal_forces(self, Fx_prime, Fy_prime):
        self.Fx, self.Fy = Fx_prime, Fy_prime

    ## @brief Mutates the x and y components of the initial velocity of the
    # object
    #  @param A new x component of the initial velocity
    #  @param A new y component of the initial velocity
    def set_init_velo(self, vx_prime, vy_prime):
        self.vx, self.vy = vx_prime, vy_prime

    ## @brief Computes all of the ODE's requires to plot the simulations
    #  @details Using th scipy library provided by Python, this function
    # will calculate the ODE equations through the process of integration.
    #  @param t_final A real number representing the end time of the simulation
```

```
#  @param nsteps A natural number presenting the number of time intervals
# to calculate
#  @return A tuple of sequences that will be used to graph its positions in
# 2D space
def sim(self, t_final, nsteps):
    t = [(i * t_final) / (nsteps - 1) for i in range(nsteps)]
    return t, integrate.odeint(self.__ode, [self.s.cm_x(), self.s.cm_y(),
                               self.vx, self.vy], t)

## @brief This is a used to help calculate the ODE's that represent the
# simulation of the object
#  @param w A sequence of real numbers containing the center of mass and
# initial velocity components
#  @param t A real number representing a moment in time
#  @return A sequence of real numbers used to compute ODE's required for
# the simulation
def __ode(self, w, t):
    return [w[2], w[3], self.Fx(t) / self.s.mass(),
            self.Fy(t) / self.s.mass()]
```