Assignment 2 Solution

Mehak Khan
khanm294

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

The test cases for all classes were approached to cover boundary, normal and exception cases. Unit testing was done using `pytest`.

To ensure the functionality of `CircleT` and `TriangleT`, all the getters were tested. Only one circle and triangle was constructed to test these getters as their only purpose was to return a value based on the state variables, and did not cover any complicated cases. The constructors of both classes were tested to raise an exception in the case of a negative or 0 radius/side length or mass.

To ensure the functionality of `BodyT`, again all the getters were tested. Only one body instance was constructed to test these getters as their only purpose was to return a value based on the state variables, and did not cover any complicated cases. The testing for this class used pytest approximation due to floating point values. Lastly, the constructor was tested to raise an exception when expected.

To ensure the functionality of `Scene`, the getters and setters for the shape and velocity were tested normally on three different Scene instances. The simulation testing was done with the help of error calculation using the norm of a sequence. The error was picked to be a small value of 2e-3. The simulation was tested on projectile motions and shape falling under force of gravity in the negative y direction. There were no exceptions in this class.

Lastly, `Plot` was tested manually with comparing expected plot to calculated plot.

There were 48 tests which all passed and no problems surface through these test cases.

2 Results of Testing Partner's Code

My partner's code was tested using my `test_driver`. The code, however, only passed 22 tests and had 26 errors. After some debugging, I figured that the errors surfaced due to my partner's `Scene` class inheriting `Shape` but not actually implementing all the methods in `Shape` class in `Scene`. This resulted in errors stating *"Can't instantiate abstract class Scene with abstract methods cm_x, cm_y, m_inert, mass"*. As the MIS did not state that `Scene.py` is inheriting `Shape.py`, I ran my partner's code again after removing the inheritance. Doing that resulted in all 48 test cases passing. Therefore, all the errors were only a result of the incorrect inheritance. The result of this exercise vs the one done in A1 is obvious as in A1, many test cases failed mainly due to the decision differences in exceptions and assumptions made. However, A2 had a formal specification which led to the same assumptions and exceptions made in my code and my partner's code and resulting in a much smoother test cases result. Debugging why test cases failed was also easier in A2 than in A1 due to use the of `pytest`. Pytest is more informative about test cases failing and errors that caused the failure.

3 Critique of Given Design Specification

The specification of this design was provided formally via an MIS. I specifically liked this aspect of the specification because it clearly communicated what is required and what invariants should be met in a module. The input and output type were clearly communicated, along with any exceptions that the program should raise. This avoided confusion when interpreting the specification. This also allowed for consistency in naming conventions and exception types which made it easier to test the program. However, reading the formal specification took slightly longer than it would to read a natural language specification and gave less room for personal design decisions or creativity as there is no ambiguity in formal language. The design specification did not meet the criteria of low coupling. `Shape.py` was inherited by 3 different modules, which promoted high coupling and does not allow the modules to be treated as individuals. However, the design did have high cohesion as the elements in the modules were closely related as they all corresponded to the movement of shapes in 2D space. As for minimality, the design was minimal for the most part except in `Scene.py`, the getters and setters for initial velocities and unbalanced forces both change two state variable in one method. This is not minimal design, however, considering the application, this works over here and we do not necessarily need minimality for these two methods. The specification did provide checks to avoid generating exceptions. Using the specification, we were required to raise an exception in several cases, such as when the mass is not greater than 0 in `TriangleT` and `CircleT`. These exceptions made use of conditional checks which avoid generated exceptions. Overall, the design specifications were adequate to me. However, due to difficulty in understanding the formal language and purpose behind each class, I would add notes in the design specification that explain the purpose of each module.

4 Answers

a) I believe getters and setters should not be unit tested if they do not contain any logic, and all they do is get or set a state variable. An example of such would be `cm_x` and `cm_y` in **CircleT**. However, if the getter or setter contains some sort of logic or calculation, it should be tested. An example of this would be `cm_x` and `cm_y` in **BodyT**. Therefore, it is important to test for getters and setters that contain logic and/or calculations but simply returning or setting a value does not require a unit test.

b) The setter and getter for the (F_x and F_y) in `Scene.py` would be tested in a very similar way that the getter and setter for the initial velocities were tested. The return value of the method for (get_unbal_forces()) would be set equal to a tuple of the functions it is supposed to return, which would assert to True. Likewise, after calling the setter to set a new F_x or F_y, the getter can be used again to assert that the functions are equal to the expected functions. An example of the testing included below:

```
assert(s1.get_unbal_forces() == (Fx, Fy))
s1.set_unbal_forces(Fx_2, Fy_2)
assert(s1.get_unbal_forces() == (Fx_2, Fy_2))
```

c) If automated tests were required for `Plot.py`, I would import `compare_images` from `matplotlib.testing.decorators`. We are able to save any plots using `plt.savefig("file_name.png"`
Therefore, having two saved plots, one for the expected image and one for the generated image, allows us to use the `compare_images method`. This method takes three parameters, two image paths and a tolerance value to the comparison of the two images. If the images are equal within the tolerance, a `None` type is returned. Assert that `None` is returned to pass the test case and make testing automated.

d) The solution below uses the same value for ϵ as used in my implementation (2e-3).

close_enough : seq of $\mathbb{R} \times$ seq of $\mathbb{R} \rightarrow$ Bool
close_enough$(x_{\text{calc}}, x_{\text{true}}) \equiv |\frac{norm(difference(x_{\text{calc}}, x_{\text{true}}))}{norm(x_{\text{true}})}| < \epsilon$

Local Functions:

norm : seq of $\mathbb{R} \rightarrow \mathbb{R}$
norm$(s) \equiv (\exists x \in s | (\forall y \in s \cdot x \geq y))$

difference : seq of $\mathbb{R} \times$ seq of $\mathbb{R} \rightarrow$ seq of $\mathbb{R}$
difference$(s_1, s_2) \equiv [i : \mathbb{N} | i \in [0..|s_1| - 1] : s_{1\mathrm{i}} - s_{2\mathrm{i}}]$

e) There should not be any exceptions raised for negative x and y coordinates of the centre of mass. This is due to the fact that x and y coorindates only tell us the location of the centre of mass which is possibly anywhere in the coordinate system. Negative axis are still included in the 2D coordinate system, therefore raising an exception for this is not required.

f) `TriangleT` has a state invariant that $s > 0 \wedge m > 0$. This invariant is always satisfied by the given specification as the constructor for TriangleT always first performs a check on the side length (s) and mass (m) to make sure they are both greater than 0. If not, a `Value Error` is raised and an object is not created. Therefore, informally proven that this state invariant is always satisfied.

g) Python list comprehension statement that generates a list of the square root of all odd integers between 5 and 19:

```
import math
[math.sqrt(x) for x in range(5, 20) if x%2 != 0]
```

h) Python function that takes in a string and returns a string but with all upper case letters removed:

```
def removeUpperCase(str1):
    removed = [x for x in str1 if x.islower()]
    return ''.join(removed)
```

i) The principle of abstraction and generality are related as generality is used to solve a more general problem than the problem at hand while abstraction is to focus on what is important while ignoring what is irrelevant. Abstraction is often used to extract a more general solution from a specific solution. Using abstraction we leave out details that are not important which allows us to solve a more general problem that can be used in different solutions. Therefore, using abstraction helps us imlplement generality and they both help us achieve reusability of code.

j) When a module uses many other modules, we call this fan-out. On the other hand, when a module is used by many other modules, we call this fan-in. Fan-in is better than fan-out as a module that uses many others tends to be more fragile as it's correctness depends on different modules. However, when a module is used by many other modules, it promotes the reusability of code as different modules are able to use it. Therefore, in general fan-in is better.

E Code for Shape.py

```python
## @file Shape.py
#  @author Mehak Khan
#  @brief An interface for modules that implement shape entities
#  @date February 2nd, 2021

from abc import ABC, abstractmethod

## @brief Shape provides an interface for shape entities
#  @details The methods in the interface are abstract and have to
#   be overriden by modules to be implemented


class Shape(ABC):

    @abstractmethod
    ## @brief A generic method for x coordinate of center of mass
    #  @return A real number indicating the x coordinate of center of mass
    def cm_x(self):
        pass

    @abstractmethod
    ## @brief A generic method for y coordinate of center of mass
    #  @return A real number indicating the y coordinate of center of mass
    def cm_y(self):
        pass

    @abstractmethod
    ## @brief A generic method for the mass of a 2D shape
    #  @return A real number indicating the mass of a 2D shape
    def mass(self):
        pass

    @abstractmethod
    ## @brief A generic method for the moment of inertia of a 2D shape
    #  @return A real number indicating the moment of inertia of a 2D shape
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Mehak Khan
#  @brief Contains the CircleT type to represent circles as a shape
#  @date February 2nd, 2021

from Shape import Shape

## @brief CircleT is a class the implements an ADT for the concept of
#   shapes moving through 2D space
#  @details The ADT contains the x, y coordinate of center of mass,
#   mass and radius of the circle


class CircleT(Shape):

    ## @brief Constructor for CircleT
    #  @details The constructor assumes that the arguments provided to the access program will
    #  of the correct type.
    #  @throws ValueError If the radius or mass arguments provided are not greater than 0
    #  @param x Representing x coordinate of center of mass
    #  @param y Representing y coordinate of center of mass
    #  @param r Representing radius of circle
    #  @param m Representing mass of circle
    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 the x coordinate of center of mass
    #  @return Real number representing x coordinate of center of mass
    def cm_x(self):
        return self.x

    ## @brief Getter for the y coordinate of center of mass
    #  @return Real number representing the y coordinate of center of mass
    def cm_y(self):
        return self.y

    ## @brief Getter for the mass of the circle
    #  @return Real number representing mass of circle
    def mass(self):
        return self.m

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

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Mehak Khan
#  @brief Contains the TriangleT type to represent triangles as a shape
#  @date February 2nd, 2021

from Shape import Shape

## @brief TriangleT is a class the implements an ADT for the concept of
#   shapes moving through 2D space
#  @details The ADT contains the x, y coordinate of center of mass,
#   mass and side length of the triangle


class TriangleT(Shape):

    ## @brief Constructor for TriangleT
    #  @details The constructor assumes that the arguments provided to the access program will
    #  of the correct type. The class also assumes that all triangles are equilateral
    #  therefore only one side length is provided.
    #  @throws ValueError If the side length or mass arguments provided are not greater than 0
    #  @param x Representing x coordinate of center of mass
    #  @param y Representing y coordinate of center of mass
    #  @param r Representing side length of triangle
    #  @param m Representing mass of triangle
    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 x coordinate of center of mass
    #  @return Real number representing the x coordinate of center of mass
    def cm_x(self):
        return self.x

    ## @brief Getter for the y coordinate of center of mass
    #  @return Real number representing y coordinate of center of mass
    def cm_y(self):
        return self.y

    ## @brief Getter for the mass of the triangle
    #  @return Real number representing mass of triangle
    def mass(self):
        return self.m

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

H Code for BodyT.py

```python
## @file BodyT.py
#  @author Mehak Khan
#  @brief  Contains the BodyT type to represent sequence of masses
#  @date February 2nd, 2021

from Shape import Shape

## @brief ShapeT is responsible for representing a system of shapes
#  @details ShapeT is a class that implements the ADT for the concept of shapes in 2D space.
#   The ADT contains the x, y coordinate of center of mass,
#   moment of inertia and mass of the body


class BodyT(Shape):

    ## @brief constructor for BodyT
    #  @details The constructor builds a body with the x, y coordinates of center of mass,
    #   mass moment of inertia, and mass of the sequence of masses
    #  @throws ValueError If the lengths of the sequences provided as arguments are not equal
    #   Else if any mass value in the sequence of mass values is not greater than 0
    #  @param x Sequeunce of real numbers representing x coordinate of center of mass
    #  @param y Sequence of real numbers representing  y coordinate of center of mass
    #  @param m Sequence of real numbers representing masses
    def __init__(self, x, y, m):
        if (not (len(x) == len(y) == len(m))):
            raise ValueError
        elif any(u <= 0 for u in m):
            raise ValueError

        self.cmx = self.__cm(x, m)
        self.cmy = self.__cm(y, m)
        self.m = self.__sum(m)
        squared_val = self.cmx * self.cmx + self.cmy * self.cmy
        self.moment = self.__mmom(x, y, m) - self.__sum(m) * squared_val

    ## @brief Getter for the x coordinate of center of mass
    #  @return Real number representing x coordinate of center of mass
    def cm_x(self):
        return self.cmx

    ## @brief Getter for the y coordinate of center of mass
    #  @return Real number representing y coordinate of center of mass
    def cm_y(self):
        return self.cmy

    ## @brief Getter for the mass of the body
    #  @return Real number representing the mass of the body
    def mass(self):
        return self.m

    ## @brief Getter for the moment of inertia of the body
    #  @return Real number representing the moment of inertia of the body
    def m_inert(self):
        return self.moment

    ## @brief Sum up values in a list
    #  @param m List to sum up values of
    #  @return Real number representing the sum of the values in the list
    def __sum(self, m):
        total = 0
        for u in m:
            total = total + u
        return total

    ## @brief Calculate the center of mass of point masses
    #  @param z Sequence of real numbers representing coordinates
    #  @param m Sequence of real numbers representing mass
    #  @return cmx Real number representing the center of mass
    def __cm(self, z, m):
        total = 0
        for i in range(len(m)):
            total = total + (z[i] * m[i])
        return total / self.__sum(m)

    ## @brief Calculate the moment of inertia of point masses
    #  @param x The x coordinate of the centre of mass
```

```
#  @param y The y coordinate of the centre of mass
#  @param m Sequence of real numbers representing mass of shapes
#  @return mmom Real number representing the moment of inertia
def __mmom(self, x, y, m):
    total = 0
    for i in range(len(m)):
        total = total + (m[i] * ((x[i] * x[i]) + (y[i] * y[i])))
    return total
```

I Code for Scene.py

```python
## @file Scene.py
#  @author Mehak Khan
#  @brief Contains the Scene type for simulation of masses
#  @date 3rd February, 2021

from scipy.integrate import odeint


## @brief Scene class is responsible for representing simulation of
#   objects in 2D space
#  @details Implements the ADT for Scene class. The ADT contains
#   the shapes in 2D space, velocities in x, y directions and
#   forces in x, y directions.


class Scene():

    ## @brief Constructor for Scene class
    #  @param s Shape object
    #  @param Fx Unbalanced force function in x direction
    #  @param Fy Unbalanced force function in y direction
    #  @param Vx Initial velocity in x direction
    #  @param Vy Initial velocity in 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 object state variable
    #  @return Shape object
    def get_shape(self):
        return self.s

    ## @brief Getter for the unbalanced forces
    #  @return A tuple of the unbalanced force functions in x and y direction
    def get_unbal_forces(self):
        return self.Fx, self.Fy

    ## @brief Getter for the inital velocities
    #  @return A tuple of the initial velocities in x and y direction
    def get_init_velo(self):
        return self.Vx, self.Vy

    ## @brief Mutator to set the shape object
    #  @param s1 new shape object
    def set_shape(self, s1):
        self.s = s1

    ## @brief Mutator to set the unbalanced forces
    #  @param fx the unabalanced force function in x direction
    #  @param fy the unbalanced force function in y direction
    def set_unbal_forces(self, fx, fy):
        self.Fx = fx
        self.Fy = fy

    ## @brief Mutator to set the intial velocities
    #  @param vx the initial velocity in x direction
    #  @param vy the initial velocity in y direction
    def set_init_velo(self, vx, vy):
        self.Vx = vx
        self.Vy = vy

    ## @brief Calculates ode for simulation of 2D objects in space
    #  @param t_final Real number representing final time
    #  @param n_steps Natural number representing number of time intervals
    #  @return A tuple of sequence of real numbers representing time and four sequences
    #   of sequences of real numbers representing the ode integration
    def sim(self, t_final, n_steps):
        t = []
        for i in range(n_steps):
            t.append((i * t_final) / (n_steps - 1))

        ode_seq = odeint(self.__ode, [self.s.cm_x(), self.s.cm_y(), self.Vx, self.Vy], t)
```

```
    return t, ode_seq

## @brief A local function for ordinary differential equation
#  @param w A sequence of real numbers representing position and velocity
#  @param t A real number representing time
#  @return A sequence of real numbers used to calculate ode integration
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 Mehak Khan
#  @brief Implements a library module for plotting points in 2D space
#  @date 3rd February 2021
#  @details The library module aids in plotting points on a graph with
#   the use of python's matplotlib

import matplotlib.pyplot as plt


## @brief Plots 3 graphs of motion simulation
#  @param w The list for the x and y coordinate values
#  @param t The list for the time values
def plot(w, t):
    if (not (len(w) == len(t))):
        raise ValueError
    x = []
    y = []
    for i in range(len(w)):
        x.append(w[i][0])
        y.append(w[i][1])

    graphs, ax = plt.subplots(3)
    graphs.suptitle("Motion Simulation")

    ax[0].plot(t, x)
    ax[1].plot(t, y)
    ax[2].plot(x, y)

    plots = ax.flat
    plots[0].set(ylabel='x(m)')
    plots[1].set(ylabel='y(m)')
    plots[2].set(ylabel='y(m)')
    plots[2].set(xlabel='x(m)')
    plt.show()
```

K Code for test_driver.py

```
## @file test_All.py
#  @author Mehak Khan
#  @brief Testing driver
#  @date 5th February, 2021
#  @details This file tests CircleT.py, TriangleT.py, Scene.py, and BodyT.py using pytest.

import pytest
from CircleT import *
from TriangleT import *
from Scene import *
from BodyT import *
import math


class TestCircle:

    def setup_method(self, method):
        self.c1_x = 2.0
        self.c1_y = 5.0
        self.c1_r = 15.0
        self.c1_m = 3.0
        self.c1 = CircleT(self.c1_x, self.c1_y, self.c1_r, self.c1_m)

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

    def test_getter_cm_x(self):
        assert self.c1.cm_x() == self.c1_x

    def test_getter_cm_y(self):
        assert self.c1.cm_y() == self.c1_y

    def test_getter_mass(self):
        assert self.c1.mass() == self.c1_m

    def test_inertia(self):
        assert self.c1.m_inert() == 337.5

    def test_exception_mass_zero(self):
        with pytest.raises(ValueError):
            CircleT(1.0, 3.0, 4.0, 0)

    def test_exception_mass_neg(self):
        with pytest.raises(ValueError):
            CircleT(1.0, 3.0, 4.0, -5)

    def test_exception_radius_neg(self):
        with pytest.raises(ValueError):
            CircleT(1.0, 3.0, -10, 20)


class TestTriangle:

    def setup_method(self, method):
        self.t1_x = 3.6
        self.t1_y = 7.8
        self.t1_s = 4.7
        self.t1_m = 1.5
        self.t1 = TriangleT(self.t1_x, self.t1_y, self.t1_s, self.t1_m)

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

    def test_getter_cm_x(self):
        assert self.t1.cm_x() == self.t1_x

    def test_getter_cm_y(self):
        assert self.t1.cm_y() == self.t1_y

    def test_getter_mass(self):
        assert self.t1.mass() == self.t1_m

    def test_inertia(self):
        assert self.t1.m_inert() == pytest.approx(2.76125)

    def test_exception_mass_zero(self):
```

```
        with pytest.raises(ValueError):
            TriangleT(1.0, 3.0, 4.0, 0)

    def test_exception_mass_neg(self):
        with pytest.raises(ValueError):
            TriangleT(1.0, 3.0, 4.0, -5)

    def test_exception_side_neg(self):
        with pytest.raises(ValueError):
            TriangleT(1.0, 3.0, -6, 20)


class TestBody:

    def setup_method(self, method):
        self.b1_x = [3.6, 4.8, 9.5, 2.3]
        self.b1_y = [7.8, 6.6, 3.7, 2.2]
        self.b1_m = [8.9, 9.9, 1.2, 0.5]
        self.b1 = BodyT(self.b1_x, self.b1_y, self.b1_m)

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

    def test_cm_x(self):
        assert self.b1.cm_x() == pytest.approx(4.493170732)

    def test_cm_y(self):
        assert self.b1.cm_y() == pytest.approx(6.843902439)

    def test_mass(self):
        assert self.b1.mass() == pytest.approx(20.5)

    def test_m_inert(self):
        assert self.b1.m_inert() == pytest.approx(71.88753166)

    def test_exception_unequal_length(self):
        with pytest.raises(ValueError):
            BodyT([1, 2], [1, 2], [1])

    def test_2_exception_unequal_length(self):
        with pytest.raises(ValueError):
            BodyT([1], [1], [1, 2])

    def test_exception_zero_mass(self):
        with pytest.raises(ValueError):
            BodyT([1, 2, 3], [2, 3, 4], [5, 9, 0])

    def test_exception_neg_mass(self):
        with pytest.raises(ValueError):
            BodyT([1, 2, 3], [2, 3, 4], [-5, 9, 7])


class TestScene:

    def Fx(self, t):
        return 0

    def Fy(self, t):
        return -self.g * self.m

    def Fx_2(self, t):
        return 3 if t < 5 else 0

    def Fy_2(self, t):
        return -self.g * self.m if t < 3 else self.g * self.m

    def setup_method(self, method):
        self.g = 9.81  # gravity
        self.m = 1  # mass
        self.e = 2e-3  # small

        self.c1 = CircleT(2.7, 7.8, 4.5, 1.0)
        self.t1 = TriangleT(7.7, 4.5, 3.9, 1.0)
        self.b1 = BodyT([1, 3, 4, 0.5], [2, 7, 8, 0.3], [6, 2, 1, 0.2])

        theta = 30
        v = 5.7

        self.s1_Vx = v * math.cos(theta)
        self.s1_Vy = v * math.sin(theta)
```

```python
        self.s1 = Scene(self.c1, self.Fx, self.Fy, self.s1_Vx, self.s1_Vy)
        self.s2 = Scene(self.t1, self.Fx, self.Fy, 0, 0)
        self.s3 = Scene(self.b1, self.Fx_2, self.Fy_2, 0, 0)

        self.time1, self.ode1 = self.s1.sim(5, 5)
        self.time2, self.ode2 = self.s2.sim(6, 5)
        self.time3, self.ode3 = self.s3.sim(7, 5)

    def teardown_method(self, method):
        self.s1 = None
        self.t1 = None
        self.c1 = None

    def test_getter_s_s1(self):
        assert self.s1.get_shape() == self.c1

    def test_getter_s_s2(self):
        assert self.s2.get_shape() == self.t1

    def test_getter_s_s3(self):
        assert self.s3.get_shape() == self.b1

    def test_getter_vx_s1(self):
        assert self.s1.get_init_velo() == (self.s1_Vx, self.s1_Vy)

    def test_getter_vx_s2(self):
        assert self.s2.get_init_velo() == (0, 0)

    def test_getter_vx_s3(self):
        assert self.s3.get_init_velo() == (0, 0)

    def test_setter_s1(self):
        self.s1.set_shape(self.t1)
        assert self.s1.get_shape() == self.t1

    def test_setter_s2(self):
        self.s2.set_shape(self.b1)
        assert self.s2.get_shape() == self.b1

    def test_setter_s3(self):
        self.s3.set_shape(self.c1)
        assert self.s3.get_shape() == self.c1

    def test_setter_init_velo_s1(self):
        self.s1.set_init_velo(0, 0)
        assert self.s1.get_init_velo() == (0, 0)

    def test_setter_init_velo_s1_setback(self):
        self.s1.set_init_velo(self.s1_Vx, self.s1_Vy)
        assert self.s1.get_init_velo() == (self.s1_Vx, self.s1_Vy)

    def calculate_sequence_error(self, cal_seq, true_seq):
        new_seq = []
        for i in range(len(cal_seq)):
            new_seq.append(cal_seq[i] - true_seq[i])
        return new_seq

    def test_sim_t1(self):
        t_expected = [0.0, 1.25, 2.5, 3.75, 5.0]
        t_new = self.calculate_sequence_error(self.time1, t_expected)
        assert abs(max(t_new, key=abs)) / abs(max(t_expected, key=abs)) < self.e

    def test_sim_t2(self):
        t_expected = [0.0, 1.5, 3.0, 4.5, 6.0]
        t_new = self.calculate_sequence_error(self.time2, t_expected)
        assert abs(max(t_new, key=abs)) / abs(max(t_expected, key=abs)) < self.e

    def test_sim_t3(self):
        t_expected = [0.0, 1.75, 3.5, 5.25, 7.0]
        t_new = self.calculate_sequence_error(self.time3, t_expected)
        assert abs(max(t_new, key=abs)) / abs(max(t_expected, key=abs)) < self.e

    def test_sim_ode1_rx(self):
        rx_expected = [2.7, 3.7990, 4.8981, 5.9971, 7.0961]
        rx = self.ode1[0:, 0]
        rx_new = self.calculate_sequence_error(rx, rx_expected)
        assert abs(max(rx_new, key=abs)) / abs(max(rx_expected, key=abs)) < self.e

    def test_sim_ode1_ry(self):
```

```
        ry_expected = [7.8, -6.8960, -36.9045, -82.2254, -142.8589]
        ry = self.ode1[0:, 1]
        ry_new = self.calculate_sequence_error(ry, ry_expected)
        assert abs(max(ry_new, key=abs)) / abs(max(ry_expected, key=abs)) < self.e

    def test_sim_ode1_vx(self):
        vx_expected = [0.8792, 0.8792, 0.8792, 0.8792, 0.8792]
        vx = self.ode1[0:, 2]
        vx_new = self.calculate_sequence_error(vx, vx_expected)
        assert abs(max(vx_new, key=abs)) / abs(max(vx_expected, key=abs)) < self.e

    def test_sim_ode1_vy(self):
        vy_expected = [-5.6318, -17.8818, -30.1318, -42.3818, -54.6318]
        vy = self.ode1[0:, 3]
        vy_new = self.calculate_sequence_error(vy, vy_expected)
        assert abs(max(vy_new, key=abs)) / abs(max(vy_expected, key=abs)) < self.e

    def test_sim_ode2_rx(self):
        rx_expected = [7.7, 7.7, 7.7, 7.7, 7.7]
        rx = self.ode2[0:, 0]
        rx_new = self.calculate_sequence_error(rx, rx_expected)
        assert abs(max(rx_new, key=abs)) / abs(max(rx_expected, key=abs)) < self.e

    def test_sim_ode2_ry(self):
        ry_expected = [4.5, -6.525, -39.6, -94.725, -171.9]
        ry = self.ode2[0:, 1]
        ry_new = self.calculate_sequence_error(ry, ry_expected)
        assert abs(max(ry_new, key=abs)) / abs(max(ry_expected, key=abs)) < self.e

    def test_sim_ode2_vx(self):
        vx_expected = [0.0, 0.0, 0.0, 0.0, 0.0]
        vx = self.ode2[0:, 2]
        vx_new = self.calculate_sequence_error(vx, vx_expected)
        assert abs(max(vx_new, key=abs)) == 0.0

    def test_sim_ode2_vy(self):
        vy_expected = [0.0, -14.7, -29.4, -44.1, -58.8]
        vy = self.ode2[0:, 3]
        vy_new = self.calculate_sequence_error(vy, vy_expected)
        assert abs(max(vy_new, key=abs)) / abs(max(vy_expected, key=abs)) < self.e

    def test_sim_ode3_rx(self):
        rx_expected = [1.75, 2.24932, 3.74728, 6.23370, 9.086956]
        rx = self.ode3[0:, 0]
        rx_new = self.calculate_sequence_error(rx, rx_expected)
        assert abs(max(rx_new, key=abs)) / abs(max(rx_expected, key=abs)) < self.e

    def test_sim_ode3_ry(self):
        ry_expected = [3.70217, 2.071060, -2.555978, -5.58519, -5.35217]
        ry = self.ode3[0:, 1]
        ry_new = self.calculate_sequence_error(ry, ry_expected)
        assert abs(max(ry_new, key=abs)) / abs(max(ry_expected, key=abs)) < self.e

    def test_sim_ode3_vx(self):
        vx_expected = [0.0, 0.570652, 1.14130, 1.63043, 1.63043]
        vx = self.ode3[0:, 2]
        vx_new = self.calculate_sequence_error(vx, vx_expected)
        assert abs(max(vx_new, key=abs)) / abs(max(vx_expected, key=abs)) < self.e

    def test_sim_ode3_vy(self):
        vy_expected = [0.0, -1.8641, -2.66304, -0.79891, 1.065217]
        vy = self.ode3[0:, 3]
        vy_new = self.calculate_sequence_error(vy, vy_expected)
        assert abs(max(vy_new, key=abs)) / abs(max(vy_expected, key=abs)) < self.e
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Samia Anwar
#  @brief Contains a CircleT type to represent a circle with a mass on a plane
#  @date February 2, 2021

from Shape import Shape

## @brief CircleT is used to represent a circle on a plane with a mass
#  to calculate its moment of inertia


class CircleT(Shape):
    ## @brief constructor for class CircleT, represents circles as their
    #  cartesian coordinates of the center, their radius, and their mass
    #  @param x is a real number representation of the x coordinate of the
    #  centre of the circle
    #  @param y is a real number representation of the y coordinate of the centre of
    #  the circle
    #  @param r is a real number representation of the radius of the circle
    #  @param m is a real number representation of the mass of the circle
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    #  @throws ValueError raised if either the mass or radius is defined to be less than
    #  or equal to zero
    def __init__(self, x, y, r, m):
        if (m <= 0 or r <= 0):
            raise ValueError
        self.x = x
        self.y = y
        self.r = r
        self.m = m

    ## @brief returns the x coordinate of the center of the circle
    #  @return real number representation of x-coordinate of the centre of the circle
    def cm_x(self):
        return self.x

    ## @brief returns the y coordinate of the center of the circle
    #  @return real number representation of x-coordinate of the centre of the circle
    def cm_y(self):
        return self.y

    ## @brief returns the mass of the circle
    #  @return real number representation of mass of the circle
    def mass(self):
        return self.m

    ## @brief returns the mass of the circle based on a formula using the initialised
    #  mass and radius values
    #  @return real number representation of moment of inertia of the circle
    def m_inert(self):
        return (self.m * self.r * self.r) / 2
```

M Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Samia Anwar
#  @brief Contains a TriangleT type to represent an equilateral triangle
#  with a mass on a plane
#  @date Feb 2/2021

from Shape import Shape

## @brief TriangleT is used to represent an equilateral Triangle on a plane with a mass
#  to eventually calculate its moment of inertia when called on


class TriangleT(Shape):
    ## @brief constructor for class TriangleT, represents a triangle as its
    #  cartesian coordinates of the center, its side length, and its mass
    #  @param x is a real number representation of the x coordinate of the
    #  centre of the triangle
    #  @param y is a real number representation of the y coordinate of the centre of
    #  the triangle
    #  @param s is a real number representation of all sides of the equilateral triangle
    #  @param m is a real number representation of the mass of the triangle
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    #  @throws ValueError raised if either the mass or side length is defined to be less than
    #  or equal to 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 returns the x coordinate of the center of the triangle
    #  @return real number representation of x-coordinate of the centre of the triangle
    def cm_x(self):
        return self.x

    ## @brief returns the y coordinate of the center of the triangle
    #  @return real number representation of x-coordinate of the centre of the triangle
    def cm_y(self):
        return self.y

    ## @brief returns the mass of the triangle
    #  @return real number representation of mass of the triangle
    def mass(self):
        return self.m

    ## @brief returns the mass of the triangle based on a formula using the initialised
    #  mass and side length values
    #  @return real number representation of moment of inertia of the triangle
    def m_inert(self):
        return (self.m * self.s * self.s / 12)
```

N Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Samia Anwar
#  @brief Contains a generic BodyT type which has properties of a Shape
#  @date Feb 2/2021

from Shape import Shape

##  @brief Objects of this class represent body of points with mass
#   cartesian placement of physical structures, their masses, and their moments of inertia


class BodyT(Shape):

    ## @brief Constructor method for class BodyT, initialises a Body from their
    #  x, y, and mass values
    #  @param x is the x-coordinates of an object on the cartesian plane, represented
    #  as a sequence of real numbers
    #  @param y is the y-coordinates of an object on the cartesian plane, represented
    #  as a sequence of real numbers
    #  @param m is the mass of each part of an object, represented as a sequence of real
    #  numbers, corresponding to the indices in the x and y lists
    #  @details the constructor method conducts calculations based on the given parameters
    #  to create a numerical self object corresponding to the moment of inertia of the whole
    #  object, the x-y coordinates of the centre of mass of the whole system and the mass of
    #  the whole system
    #  @throws ValueError if parameters are not sequences of the same length, and if members
    #  of sequence m are less than or equal to zero
    def __init__(self, x, y, m):
        if not (len(x) == len(y) and len(x) == len(m)):
            raise ValueError
        for i in m:
            if i <= 0:
                raise ValueError
        self.cmx = self.__cm__(x, m)
        self.cmy = self.__cm__(y, m)
        self.m = self.__sum__(m)
        self.moment = self.__mmom__(x, y, m) - self.m * (self.cmx ** 2 + self.cmy ** 2)

    ## @brief returns the value of the x coordinate of the object's center of mass
    #  @return a real number representation of the x-coordinate
    def cm_x(self):
        return self.cmx

    ## @brief returns the value of the y coordinate of the object's center of mass
    #  @return a real number representation of the y-coordinate of the object's center of mass
    def cm_y(self):
        return self.cmy

    ## @brief returns the value of the total mass of the object
    #  @return a real number representation of the total mass of the object
    def mass(self):
        return self.m

    ## @brief returns the value of the object's moment of inertia
    #  @return real number representation of the object's total moment of inertia
    def m_inert(self):
        return self.moment

    ## @brief Calculates the sum of values in a list of real numbers
    #  @param a is the list composed of real numbers to be added together
    #  @return a real number representation of the sum of the list
    def __sum__(self, a):
        s = 0
        for u in a:
            s = s + u
        return s

    ## @brief Calculates the center of mass of an object on one cartesian axis
    #  @param a is the list composed of real number masses corresponding to parts of an object
    #  @param z is the list composed of real number x-coordinates  corresponding
    #  to parts of an object
    #  @return a real number representation of the center of mass of an object in parts
    def __cm__(self, z, a):
        s = 0
        for i in range(len(a)):
            s = s + (z[i] * a[i])
```

```
    return (s / self.__sum__(a))

## @brief Calculates some real number value in the moment of inertia equation
#  @param x is the list of x-coordinates of the parts of a system of objects
#  @param y is the list of y-coordinates of the parts of a system of objects
#  @param m is the list of masses of the parts of a system of objects
#  @returns real number representaion of the sum of m * (x^2 + y^2) at each
#  index of the corresponding lists
def __mmom__(self, x, y, m):
    s = 0
    for i in range(len(m)):
        s = s + m[i] * (x[i] * x[i] + y[i] * y[i])
    return s
```

O Code for Partner's Scene.py

```python
## @file Scene.py
#  @author Samia Anwar
#  @brief Generic module to represent forces and velocity on an object
#  @date Feb 2, 2021
#  @details Simulates motion of an object based on force and initial velocity

from Shape import Shape
from scipy.integrate import odeint

## @brief This module takes in a Shape object and generates seqeuences of numbers to simulate
#  its motion given a force acting upon it and its initial velocity


class Scene(Shape):
    ## @brief constructor for class Scene, represents the motion acted upon a given shape
    #  @param ds is a Shape object defined elsewhere in the code and contains x-y coordinates
    #  for center of mass, a total mass and a moment of inertia
    #  @param dfx is the formula for the x-direction force acted upon the object
    #  @param dfy is the formula for the y-direction force acted upon the object
    #  @param dvx is a real number representation of the starting velocity of the object
    #  in the x-plane
    #  @param dvy is a real number representation of the starting velocity of the object
    #  in the y-plane
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    def __init__(self, ds, dfx, dfy, dvx, dvy):
        self.s = ds
        self.fx = dfx
        self.fy = dfy
        self.vx = dvx
        self.vy = dvy

    ## @brief Returns the shape object associated with the Scene
    #  @return shape object and all of its parametres
    def get_shape(self):
        return self.s

    ## @brief returns the force equations in the x and y direction
    #  @return x and y direction force equations as python functions
    def get_unbal_forces(self):
        return self.fx, self.fy

    ## @brief returns the x and y direction values of velocity
    #  @return x and y direction real number values of velocity
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief changes the shape specified in the Scene
    #  @param s_new is an Shape object containing the specified parameters
    def set_shape(self, s_new):
        self.s = s_new

    ## @brief changes the x and y direction force functions specified in the Scene
    #  @param fx_n is a python function representing the new x-direction force function
    #  @param fy_n is a python function representing the new y-direction force function
    def set_unbal_forces(self, fx_n, fy_n):
        self.fx = fx_n
        self.fy = fy_n

    ## @brief changes the x and y direction initial velocities specified in the Scene
    #  @param vx_n is a real number velocity values representing the new x-direction velocity
    #  @param vy_n is a real number velocity values representing the new y-direction velocity
    def set_init_velo(self, vx_n, vy_n):
        self.vx = vx_n
        self.vy = vy_n

    ## @brief Integrates the given functions based on initial velocity and a step value
    #  @param tf is a real number used in the numerator of the calculations
    #  @param nsteps is a natural number used in the denominator of the calculations
    #  @assumption assume that nsteps is never equal to one
    #  @return two sequences of real numbers
    def sim(self, tf, nsteps):
        t = []
        for i in range(nsteps):
            t.append((i * tf) / (nsteps - 1))
        return t, odeint(self.__ode__, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
```

```
## @brief Generates an array for computation in odeint method in sim()
#  @param w is a sequence with 4 values
#  @param t is a real number used as an input for the given force equations
#  @return an array with 4 elements inside
def __ode__(self, w, t):
    return [w[2], w[3], self.fx(t) / self.s.mass(), self.fy(t) / self.s.mass()]
```