Assignment 2 Solution

Alan Scott

February 24, 2021

This report discusses the testing phase 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

Both the CircleT and TriangleT programs were put through the same tests. For each method, the output of each getter method were tested against the value that was inputted while creating the object (the exception being for the moments of inertia, which were calculated based on formulas). Only a generic input was necessary for each of these methods, as there was no restrictions or possibilities of error when running each method. However, there were restrictions placed in the constructors. This was tested by checking for a raised exception during the construction of the objects. The BodyT module was subjected to very similar tests. Each of the getter methods only had a single test due to a lack of restrictions on their operation. The constructor had two restrictions on its operations, and both were tested, one with a case with mismatching input lengths, and another with negative masses. These tests checked for a ValueError, which was the expected output for incorrect entries. The last module tested was Scene. The getter methods were tested with generic cases, since they did not have restrictions on their operations, or the potential to create errors. The setter methods were tested using the getter methods. In this case, they were simple comparisons of the outputted values or functions. The testing of the sim() method took a different form to the rest of the functions. The method was run on a provided input function, and the output was recorded to two separate lists. Various different elements in this list were compared to expected results of previous runs. This process was aided by a function which determined if the values were acceptably close to one another, as floating point errors can often cause calculations to result in very small inaccuracies.

2 Results of Testing Partner's Code

Immediately upon the execution of the partner code, the compiler returned a TypeError stating "Can't instantiate abstract class Scene with abstract methods cm_x, cm_y, m_inert, mass". To fix this to continue with coding, I added placeholders for these abstract methods to make it run for the sake of testing. It is likely that this difference arises in the coding of Shape.py, however, I was not provided with this file, so I cannot say for certain. Other than that, the modules passed expected tests.

3 Critique of Given Design Specification

4 Answers

a) Getters and setters should ideally be unit tested for the sake of completeness, it is not always necessary. For getter and setter methods that directly access or change a state variable, it is not really necessary. However, for methods that perform operations on the input values or output values, such as the moment of inertia methods, it is necessary to unit test, since they perform some level of calculation.

b) To test the values of F_x and F_y, you could test them through the forces getter. You could assign the outputs to two variables, then test each of these variables as functions. So "x,y = get_unbal_forces()" then test them using "x(t)" and "y(t)".

c) If you wanted to compare the outputs of two plot functions, you could could generate an output file from both. You could then compare the contents of the files together, and see if they have identical contents.

d) close_enough: $\mathbb{R} \times \mathbb{R} \to \mathbb{B}$
close_enough$(x_{calc}, x_{true}) \equiv \frac{|x_{calc} - x_{true}|}{|x_{true}|} < \epsilon$

e) It is fine for coordinates to be negative since the simulation is based on the change in coordinate values, so the actual values themselves don't matter as long as the reference point is kept constant.

f) The TriangleT object is initialized with the condition $(s > 0 \wedge m > 0)$, so it is known initially that the invariant holds. The TriangleT class contains no methods which can alter the state of the object (ie no mutator methods), nor does it inherit any. Since the invariant holds in the initial case, and there exists no way to modify the state of the object, it must logically follow that this invariant must always hold for TriangleT.

g)
```
sqrt_list = [math.sqrt(x) for x in range(5,20)]
```

h)
```
def remove_upper(string):
    return "".join([letter for letter in string if letter.islower()])
```

i) The process of abstraction creates a model of some program, where some irrelevant details are left out. Generality involves solving more general problems which can apply to various different problems. For this reason, the more abstract a program is, the more general it tends to be as well.

j) Typically, a module should be designed such that it has low coupling, and does not need to rely on many other modules. The better scenario would be for a module to be used by other modules, rather than one module using many others. This is because the individual modules would only be using a single module, which would be an instance of low coupling (good), whereas one module using many other would be high coupling (bad).

E Code for Shape.py

```
## @file Shape.py
#  @author Alan Scott
#  @brief Parent class for various shapes
#  @date 16/02/2021

from abc import ABC, abstractmethod

class Shape(ABC):

    ## @brief Inheritable function for the x center of mass
    #  @details This function returns the x value of the center of mass. This function is
    #           inherited by any children of the Shape class.
    @abstractmethod
    def cm_x(self):
        pass

    ## @brief Inheritable function for the y center of mass
    #  @details This function returns the y value of the center of mass. This function is
    #           inherited by any children of the Shape class.
    @abstractmethod
    def cm_y(self):
        pass

    ## @brief Inheritable function for the mass of the shape
    #  @details This function returns the value of the mass. This function is inherited
    #           by any children of the Shape class.
    @abstractmethod
    def mass(self):
        pass

    ## @brief Inheritable function for the moment of intertia
    #  @details This function returns the value of the mass. This function is inherited
    #           by any children of the Shape class.
    @abstractmethod
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Alan Scott
#  @brief Class representing a Circle Shape
#  @date 11/02/21

from Shape import Shape


class CircleT(Shape):
    x, y, r, m = 0.0, 0.0, 0.0, 0.0

    ## @brief CicleT Constructor
    #  @details This constructor creates the CircleT object. It takes in values
    #           of the x and y coordinates, the radius and the mass. If the values
    #           of the mass or radius are not above zero, then a ValueError is thrown.
    #  @throws ValueError when the radius or mass are not above zero.
    #  @param xs x value of the circle
    #  @param ys y value of the circle
    #  @param rs radius of the circle
    #  @param mass of the circle
    def __init__(self, xs, ys, rs, ms):
        if (not (rs > 0 and ms > 0)):
            raise ValueError
        self.x = xs
        self.y = ys
        self.r = rs
        self.m = ms

    ## @brief Return function for the x center of mass
    #  @details This function is inherited from the parent class, and returns the
    #           x value of the center of mass.
    def cm_x(self):
        return self.x

    ## @brief Return function for the y center of mass
    #  @details This function is inherited from the parent class, and returns the
    #           y value of the center of mass.
    def cm_y(self):
        return self.y

    ## @brief Return function for the mass of the circle
    #  @details This function is inherited from the parent class, and returns the
    #           mass of the circle.
    def mass(self):
        return self.m

    ## @brief Return function for the moment of intertia of the circle
    #  @details This function is inherited from the parent class, and returns the
    #           moment of inertia of the circle.
    def m_inert(self):
        return (self.m * self.r**2) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Alan Scott
#  @brief Triangle Shape Class
#  @date 16/02/21

from Shape import Shape


class TriangleT(Shape):
    x, y, s, m = 0.0, 0.0, 0.0, 0.0

    ## @brief TriangleT Constructor
    #  @details This constructer creates a TriangleT type object. It takes in parameters
    #           for the x and y coordinates, side lengths and mass. If the side lengths
    #           or mass are below zero, then it throws a ValueError.
    #  @throws ValueError when side lengths or mass are not greater than zero
    #  @param xs x coordinate of the triangle
    #  @param ys y coordinate of the triangle
    #  @param ss side lengths of the triangle
    #  @param ms mass of the triangle
    def __init__(self, xs, ys, ss, ms):
        if (not (ss > 0 and ms > 0)):
            raise ValueError
        self.x, self.y, self.s, self.m = xs, ys, ss, ms

    ## @brief Inherited function for center of mass x
    #  @details This function is inherited from the Shape class. It returns the x value
    #           of the center of mass.
    def cm_x(self):
        return self.x

    ## @brief Inherited function for center of mass y
    #  @details This function is inherited from the Shape class. It returns the y value
    #           of the center of mass.
    def cm_y(self):
        return self.y

    ## @brief Inherited function for mass of the triangle
    #  @details This function is inherited from the Shape class. It returns the value
    #           of the center of mass.
    def mass(self):
        return self.m

    ## @brief Inherited function for moment of inertia of the triangle
    #  @details This function is inherited from the Shape class. It returns the value
    #           of the moment of inertia.
    def m_inert(self):
        return (self.m * self.s**2) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Alan Scott
#  @brief Body class
#  @date 16/02/21

from Shape import Shape


class BodyT(Shape):

    ## @brief BodyT Constructor
    #  @details This constructor creates an the BodyT object from the parameters of
    #           the center of mass and the mass. If the values of the mass and the
    #           centers of mass are not of equal size, a Value error is thrown. The
    #           center of mass, total mass, and the moment of inertia are calculated
    #           in the constructor using class functions.
    #  @throws ValueError when the lengths of the x, y and mass are not the same.
    #  @param xs The set of x coordinates of the centers of mass
    #  @param ys The set of y coordinates of the centers of mass
    #  @param xs The set of masses
    def __init__(self, xs, ys, ms):
        if (not (len(xs) == len(ys) and (len(ys) == len(ms)))):
            raise ValueError
        for i in range(len(ms)):
            if (ms[i] <= 0):
                raise ValueError
        self.cmx = self.__cm__(xs, ms)
        self.cmy = self.__cm__(ys, ms)
        self.m = sum(ms)
        self.moment = self.__mmom__(xs, ys, ms) - sum(ms)
        self.moment *= (self.__cm__(xs, ms)**2 + self.__cm__(ys, ms)**2)

    ## @brief Inherited cm_x function from Shape
    #  @details This function is inherited from the parent class. It returns the x value
    #           of the center of mass.
    def cm_x(self):
        return self.cmx

    ## @brief Inherited cm_y function from Shape
    #  @details This function is inherited from the parent class. It returns the y value
    #           of the center of mass.
    def cm_y(self):
        return self.cmy

    ## @brief Inherited mass function from Shape
    #  @details This function is inherited from the parent class. It returns the value of
    #           the mass.
    def mass(self):
        return self.m

    ## @brief Inherited m_inert function from Shape
    #  @details This function is inherited from the parent class. It returns the value
    #           of the center of moment of intertia.
    def m_inert(self):
        return self.moment

    ## @brief Centre of mass calculation
    #  @details This function takes two arrays and finds the sum of the elements
    #           of two equally sized arrays. It then divides this sum by the sum of
    #           the mass array.
    #  @param z First array
    #  @param m Second array
    def __cm__(self, z, m):
        sumout = 0.0
        for i in range(len(m)):
            sumout += z[i] * m[i]
        return sumout / sum(m)

    ## @brief Pythagorean summation
    #  @details This function sums the square of each element of the x and y arrays
    #           multiplied by the respictive value of the mass.
    #  @param x Array of x values
    #  @param y Array of y values
    #  @param m Array of masses
    def __mmom__(self, x, y, m):
        sumout = 0.0
```

```
for i in range(len(m)):
    sumout += m[i] * (x[i]**2 + y[i]**2)
return sumout
```

I Code for Scene.py

```
## @file Scene.py
#  @author Alan Scott
#  @brief Scene class
#  @date 16/02/21
#  @details This class represents a "scene" with a shape. It then simulates the motion
#           of the body given forces, initial velocity and other factors.

from Shape import Shape
from scipy import integrate as odei


class Scene(Shape):

    ## @brief Scene constructor
    #  @details This method constructs the Scene object from a shape, forces and velocity.
    #  @param sprime Initial Shape
    #  @param fxprime Initial horizontal force
    #  @param sprime Initial vertical force
    #  @param sprime Initial horizontal velocity
    #  @param sprime Initial horizontal velocity
    def __init__(self, sprime, fxprime, fyprime, vxprime, vyprime):
        self.s, self.fx, self.fy = sprime, fxprime, fyprime
        self.vx, self.vy = vxprime, vyprime

    g = 9.81  # accel due to gravity (m/s^2)
    m = 1  # mass (kg)

    ## @brief Horizontal force function
    #  @details This function calculates the horizontal force.
    #  @param t The time given
    def __Fx__(self, t):
        return 5 if t < 5 else 0

    ## @brief Vertical force function
    #  @details This function calculates the vertical force.
    #  @param t The time given
    def __Fy__(self, t):
        return -self.g * self.m if t < 3 else self.g * self.m

    ## @brief Inherited cm_x function
    #  @details This function is declared here to prevent exceptions from abstract
    #           functions from the parent class. This method is a placeholder.
    def cm_x(self):
        pass

    ## @brief Inherited cm_y function
    #  @details This function is declared here to prevent exceptions from abstract
    #           functions from the parent class. This method is a placeholder.
    def cm_y(self):
        pass

    ## @brief Inherited mass function
    #  @details This function is declared here to prevent exceptions from abstract
    #           functions from the parent class. This method is a placeholder.
    def mass(self):
        pass

    ## @brief Inherited m_inert function
    #  @details This function is declared here to prevent exceptions from abstract
    #           functions from the parent class. This method is a placeholder.
    def m_inert(self):
        pass

    ## @brief Shape return function
    #  @details This function returns the Shape currently stored in the class.
    def get_shape(self):
        return self.s

    ## @brief Unbalanced forces return function
    #  @details This function returns the values of the x and y unbalanced forces.
    def get_unbal_forces(self):
        return self.fx, self.fy

    ## @brief Initial velocity return function
    #  @details This function returns the values of the x and y values of the initial
    #           velocities.
```

```
def get_init_velo(self):
    return self.vx, self.vy

## @brief Shape setter function
#  @details This function mutates the Shape stored in the class.
#  @param sprime New shape
def set_shape(self, sprime):
    self.s = sprime

## @brief Unbalanced forces setter function
#  @details This function mutates the unbalanced forces stored in the class.
#  @param fxprime New horizontal force
#  @param fyprime New vertical force
def set_unbal_forces(self, fxprime, fyprime):
    self.fx, self.fy = fxprime, fyprime

## @brief Initial velocity setter function
#  @details This function mutates the initial velocities stored in the class.
#  @param fxprime New horizontal velocity
#  @param fyprime New vertical velocity
def set_init_velo(self, vxprime, vyprime):
    self.vx, self.vy = vxprime, vyprime

## @brief Physics simulation function
#  @details This function simulates the motion of the body. It does so by integrating the
#           function over t through an approximation.
#  @param tfinal Final value of t
#  @param nsteps Number of increments
def sim(self, tfinal, nsteps):
    t = []
    for i in range(nsteps):
        t.append((i * tfinal) / (nsteps - 1))
    out = odei.odeint(self.__ode__, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
    return t, out

## @brief Ordinary differential equation constructor
#  @details This function creates an ordinary differential equation.
#  @param w Input array
#  @param t Given time
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 Alan Scott
#  @brief Function for generating matplot plots
#  @date 16/02/21
#  @details This file contains a single function which plots a series of x, y, and y
#           relations on 3 graphs on 1 window.

import matplotlib.pyplot as plt


## @brief Plot function
#  @details This function plots 3 different graphs of x vs t, y vs t and x vs y. It does so
#           on 1 window using 3 vertically oriented subplots.
#  @param w 2D array containing pairs of x and y coordinates
#  @param t Array contain all the values of t
def plot(w, t):
    x = []
    y = []
    for item in w:
        x.append(item[0])
        y.append(item[1])
    plt.figure("Motion Simulation")
    plt.subplot(311)
    plt.xlabel("x(t)")
    plt.ylabel("t")
    plt.plot(t, x)
    plt.subplot(312)
    plt.xlabel("y(t)")
    plt.ylabel("t")
    plt.plot(t, y)
    plt.subplot(313)
    plt.xlabel("x(t)")
    plt.ylabel("y(t)")
    plt.plot(x, y)

    plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9, top=0.9, wspace=0.2, hspace=0.8)

    plt.show()
```

K Code for test_All.py

```
## @file test_All.py
#  @author
#  @brief
#  @date
#  @details

from CircleT import CircleT

total = 0
g = 9.81  # accel due to gravity (m/s^2)
m = 1  # mass (kg)

def Fx(t):
    return 5 if t < 5 else 0


def Fy(t):
    return -g * m if t < 3 else g * m

circle = CircleT(1.0, 10.0, 0.5, 1.0)

def test_circle():
    assert circle.cm_x() == 1.0
    assert circle.cm_y() == 10.0
    assert circle.mass() == 1.0
    assert circle.m_inert() == (1.0*0.5**2)/2

test_circle()
```

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

```
## @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()]
```