Assignment 2 Solution

Bhavna Pereira

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

For `Triangle.py`, `Circle.py`, and `BodyT.py`, I tested each getter method once to ensure their validity. I ensured to assert that a `ValueError` is raised when a circle has a radius and mass of less than 0 or when a triangle has a side length or mass less than 0. A unit test was also created to ensure proper calculation for the `m_inert()` methods within CircleT.py and TriangleT.py. As explained in **`7b`** further down, I unit tested the getter and setter methods found within `Scene.py` once each. I had not found a proper way to test `sim()` within `Scene.py` and thus had no unit test for it. However, each of my 25 test cases passed.

2 Results of Testing Partner's Code

Upon testing my partners test files, 18/25 test cases passed. 6 out of the 7 tests that failed were due to Assertion Errors. This can be explained due to the absence of an error message in my partners code. My code displays a message Invalid Input whenever a `ValueError` is raised; my partners does not. The structure of my unit test asserts that the string displayed during a `ValueError` is equal to Invalid Input, which passes when testing my own files. It does not pass for my partners code, as there is no message displayed. When altered to match my display message, the cases pass for my partners code as well. The only other failed test occurs in a calculation error, either within my code or in my partners file. The value returned for the moment of inertia within my partners `BodyT.py` file is 2.0, whereas mine is 80.0 for the same input. The difference lies within `__mmom__(x, y, m)`. My partner returns a total, divided by the sum of m, while my function returns the total.

3 Critique of Given Design Specification

The design specification was consistent in that the MIS formatting provided information exactly relevant and translatable to semi-formal language. This goes hand in hand with essentiality, as whatever was necessary was indicated within each module, including necessary assumptions and exceptions. Essentiality can be improved however, due to the fact that students are left to make any other assumption according to what they feel fit. This however, adds to generality, as it maintains a sense of open-endedness for the student. For the remainder of the design specification, I felt that despite the use of natural language, the Module Interface Specification was understandable

4 Answers

a) It is important to unit test getters and setters to ensure that the modules and functions that rely on it will work. A fault in a getter or setter method can indicate a fault in state variable instantiation or inaccurate calculation with said value.

b) It can be tested by defining functions `Fx(t)` and `Fy(t)` prior to unit testing, each which returns a value, given an integer input. From there, a Scene object can be created using an arbitrary input in order to test if the getter and setter methods work. For example, `Fx(t)` and `Fy(t)` can return the value 34 if `t` is less than 19, or else it returns 0. Then, creating an object `Object = Scene(s, Fx(4), Fy(20), Vx, Vy)`, you can test the setter method using `Object.set_unbal_forces(Fx(4), Fy(20)` and asserting that `Object.get_unbal_forces()` is equal to (34, 0). For the sake of this example, the other parameters are assumed to be properly inputted, however the point is to highlight the getters and setters of Fx and Fy.

c) By saving a file for the correct plot, plot.py can be tested using manual inputs of `w` and `t` of the same length. Each test can then be saved as its own file and its contents then be compared to that of the correct plot file. If the contents match, the test will pass. If not, the test does not pass

d) `close_enough()`: `seq of` $\mathbb{R} \times$ `seq of` $\mathbb{R} \rightarrow \mathbb{B}$

e) It is appropriate to have exceptions for non-positive values of shape dimensions and mass, as an object cannot physically have a shape dimension or mass less than 0.

However, it is not necessary to have exceptions for x and y coordinates for the objects centre of mass, as the values represent a point in space in which the object lies, with respect to a point of origin.

f) It can be ensured that this invariant is satisfied through the use of exception handling. If and only iff both the inputted sidelengths *and* mass are positive, the constructor will create state variables. If even one of these conditions do not hold, the constructor method will not build and the rest of the methods will not work. However, due to this exception handling, this invariant will always be satisfied; the state invariables built will always be valid to use for the remaining methods in the class.

g)

h)
```
def get_lower(a):
        new = ""
        for i in a:
                if i.islower():
                        new += i
        return new
```

i) Abstraction removes irrelevant details, while generalization outlines general properties of an object.

j) It would be a better case of high coupling when one module is being used by many other modules. Due to the intertwining of modules, making a change to just one module would have less of an effect to other modules than changes to many modules would have on one dependent module. By changing something within one module, it is likely that the necessary changes to modules depending on it will be similar. However changing many modules that one module is dependent on risks having a detrimental effect on the functionality of that module.

E Code for Shape.py

```
## @file Shape.py
#  @author Bhavna Pereira
#  @brief Shape ADT Class implementations
#  @date 16/02/2021

from abc import ABC, abstractmethod

## @brief The class outlines shape properties
#  @details The class contains four methods to detail the x and y coordinates
#           of the center of mass, along with the mass and the m_inert


class Shape(ABC):
    ## @brief Abstract method to pass the x coordinate of the center of mass
    #  @details defines and passes on this property for use by other modules

    @abstractmethod
    def cm_x(self):
        pass

    ## @brief Abstract method to pass the y coordinate of the center of mass
    #  @details defines and passes on this property for use by other modules
    @abstractmethod
    def cm_y(self):
        pass

    ## @brief Abstract method to pass the mass of the shape
    #  @details defines and passes on this property for use by other modules
    @abstractmethod
    def mass(self):
        pass

    ## @brief Abstract method to pass the maoment of inertia
    #  @details defines and passes on this property for use by other modules
    @abstractmethod
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Bhavna Pereira
#  @brief Class implementation of CircleT with the use of Shape.py
#  @date 16/02/2021

from Shape import *

## @brief The following class outlines properties of a circle in motion
#  @details The properties include x and y coordinate of Center of Mass,
#           the mass, and the moment of inertia


class CircleT(Shape):
    ## @brief This method is a constructor for CircleT
    #  @details The method ensures that the radius and mass
    #           are greater than zero, and if they are, the x and y
    #           coordinates, along with the radius and mass are stated
    #  @param x represents the x coordinate of the center of mass (Real Number)
    #  @param y represents the x coordinate of the center of mass (Real Number)
    #  @param r representst the radius of the circle (Real Number)
    #  @param m represents the mass of the circle (Real Number)

    def __init__(self, x, y, r, m):
        if r < 0 or m < 0:
            raise ValueError('Invalid Input')
        else:
            self.__x = x
            self.__y = y
            self.__r = r
            self.__m = m

    ## @brief this method returns the value of the x coordinate of the circle's
    #         center of mass
    #  @returns Returns the state variable of the correct type
    def cm_x(self):
        return self.__x

    ## @brief this method returns the value of the y coordinate of the circle's
    #         center of mass
    #  @returns Returns the state variable of the correct type
    def cm_y(self):
        return self.__y

    ## @brief this method returns the value of the mass of the circle
    #  @returns Returns the state variable of the correct type
    def mass(self):
        return self.__m

    ## @brief this method calculates the moment of inertia of the circle
    #  @details using the mass, multiplied by the square of the radius
    #           and dividind the product by 2, this method returns the value of the
    #           moment of inertia
    def m_inert(self):
        return (self.__m * (self.__r**2)) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Bhavna Pereira
#  @brief Class implementation of TriangleT with the use of Shape.py
#  @date 16/02/2021

from Shape import *

## @brief The following class outlines properties of a triangle in motion
#  @details The properties include x and y coordinate of Center of Mass,
#           the mass, and the moment of inertia


class TriangleT(Shape):
    ## @brief This method is a constructor for TriangleT
    #  @details The method ensures that the radius and mass
    #           are greater than zero, and if they are, the x and y
    #           coordinates, along with the radius and mass are stated
    #  @param x represents the x coordinate of the center of mass (Real Number)
    #  @param y represents the y coordinate of the center of mass (Real Number)
    #  @param s represents the sidelength of the triangle (Real Number)
    #  @param m represents the mass of the objet (Real Number)
    def __init__(self, x, y, s, m):
        if s < 0 or m < 0:
            raise ValueError('Invalid Input')
        else:
            self.__x = x
            self.__y = y
            self.__s = s
            self.__m = m

    ## @brief this method returns the value of the x coordinate of the triangle's
    #         center of mass
    #  @returns Returns the state variable of the correct type
    def cm_x(self):
        return self.__x

    ## @brief this method returns the value of the y coordinate of the triangle's
    #         center of mass
    #  @returns Returns the state variable of the correct type
    def cm_y(self):
        return self.__y

    ## @brief this method returns the value of the mass of the triangle
    #  @returns Returns the state variable of the correct type
    def mass(self):
        return self.__m

    ## @brief this method calculates the moment of inertia of the triangle
    #  @details using the mass, multiplied by the square of the sidelength
    #           and dividing the product by 12, this method 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 Bhavna Pereira
#  @brief Class implementation of BodyT with the use of Shape.py
#  @date 16/02/2021

from Shape import *


## @brief The following class outlines properties of all other shapes in motion
#  @details The properties include x and y coordinate of Center of Mass,
#  the mass, and the moment of inertia


class BodyT(Shape):
    ## @brief This method is a constructor for BodyT
    #  @details The method ensures that all masses in sequence
    #  are greater than zero, and if the length of the sequences
    #  of all parameters are equal. If they are, the x and y
    #  coordinates and the mass are stated using local functions
    #  @param x represents the x coordinate of the center of mass (Real Number)
    #  @param y represents the y coordinate of the center of mass (Real Number)
    #  @param m represents the mass of the object (Real Number)
    def __init__(self, x, y, m):
        for i in m:
            if i < 0:
                raise ValueError('Invalid Input')
        if len(x) == len(y) and len(y) == len(m):
            self.__cmx = cm(x, m)
            self.__cmy = cm(y, m)
            self.__m = sum(m)
            self.__moment = mmom(x, y, m) - sum(m) * (cm(x, m)**2 + cm(y, m)**2)
        else:
            raise ValueError('Invalid Input')

    ## @brief this method returns the value of the x coordinate of the shape's
    #          center of mass
    #  @returns Returns the state variable of the correct type
    def cm_x(self):
        return self.__cmx

    ## @brief this method returns the value of the y coordinate of the shape's
    #          center of mass
    #  @returns Returns the state variable of the correct type (Real Number)
    def cm_y(self):
        return self.__cmy

    ## @brief this method returns the value of the mass of the shape
    #  @returns Returns the state variable of the correct type
    def mass(self):
        return self.__m

    ## @brief this method calculates the moment of inertia of the triangle
    #  @details this method returns a value based on the calculated moment of
    #           inertia through local function mmom(x,y,z)
    def m_inert(self):
        return self.__moment


def cm(z, m):
    total = 0
    for i in range(0, len(m)):
        val = z[i] * m[i]
        total += val
    return total / sum(m)


def mmom(x, y, m):
    total = 0
    for i in range(0, len(m)):
        val = m[i] * (x[i]**2 + y[i]**2)
        total += val
    return total
```

I Code for Scene.py

```
## @file Scene.py
#  @author Bhavna Pereira
#  @brief Class implementation of Scene with the use of Shape.py
#  @date 16/02/2021
#  @details The class makes use of the shape of the object, its
#           unbalanced forces in order to create data points to track its
#           motion


import scipy.integrate

## @brief The following class outlines properties of an object in motion
#  @details The properties include the shape, its initial velocities, and
#           the forces acting upon it


class Scene:
    ## @brief This method is a constructor for Scene
    #  @details The method creates state variables for the shape type,
    #             its velocities, and the forces acting upon it
    #  @param s represents the shape Fx, Fy, vx, vy
    #  @param Fx represents a function for horizontal forces
    #  @param Fy represents a function for vertical forces
    #  @param vx represents a real number horizontal velocity
    #  @param vy represents a real number vertical velocity
    def __init__(self, s, Fx, Fy, vx, vy):
        self.__s = s
        self.__Fx = Fx
        self.__Fy = Fy
        self.__vx = vx
        self.__vy = vy

    ## @brief this method returns the shape of the object
    #  @returns Returns the state variable of the correct type
    def get_shape(self):
        return self.__s

    ## @brief this method returns the forces acting upon the objects
    #          both horizontally and vertically
    #  @returns Returns the state variable of the correct type
    def get_unbal_forces(self):
        return self.__Fx, self.__Fy

    ## @brief this method returns the velocity of the object
    #          both horizontally and vertically
    #  @returns Returns the state variable of the correct type
    def get_init_velo(self):
        return self.__vx, self.__vy

    ## @brief this method sets the property of the shape to a
    #          variable to be used in calculation
    def set_shape(self, s):
        self.__s = s

    ## @brief this method sets the property of the forces to
    #          variables to be used in calculation
    def set_unbal_forces(self, Fx, Fy):
        self.__Fx = Fx
        self.__Fy = Fy

    ## @brief this method sets the property of the velocities to
    #          variables to be used in calculation
    def set_init_velo(self, vx, vy):
        self.__vx = vx
        self.__vy = vy

    ## @brief this method extracts data points to represent the track of
    #          the object in motion
    #  @param tfinal represents the final time
    #  @param nsteps represents number of movements
    #  @returns the method returns the time and the respective track of the
    #             object for each moment of its course in motion
    def sim(self, tfinal, nsteps):
        t = []
        for i in range(nsteps):
            t.append((i * tfinal) / (nsteps - 1))
            val = [self.__s.cm_x(), self.__s.cm_y(), self.__vx, self.__vy]
        return (t, scipy.integrate.odeint(self.ode, val, t))

    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 Bhavna Pereira
#  @brief Function to plot objects in motion
#  @date 16/02/2021
#  @details Using the matplotlib library, the function
#           provides a visual representation of objects in motion

import matplotlib.pyplot as plt

## @brief This method sets up and graphs the track of objects in motion
#  @details the method creates three subplots to represent to compare
#           the x coordinate of the CM against the time of motion, the y coordinate
#           of the CM against the time of motion, and the x coordinate versus the
#           y coordinate
#  @param w represents a sequence of sequences
#  @param t represents the sequence of time values


def plot(w, t):
    if len(w) != len(t):
        raise ValueError
    else:
        fig, axs = plt.subplots(3)
        fig.suptitle('Motion Simulation')
        axs[0].set(ylabel='x(m)')
        axs[1].set(ylabel='y(m)')
        axs[2].set(ylabel='y(m)')
        axs[2].set(xlabel='x(m)')
        x = []
        y = []
        for i in range(0, len(w)):
            x.append(w[i][0])
        for i in range(0, len(w)):
            y.append(w[i][1])
        axs[0].plot(t, x)
        axs[1].plot(t, y)
        axs[2].plot(x, y)
        plt.show()
```

K Code for test_All.py

```
## @file test_All.py
#  @author Bhavna Pereira
#  @brief This file is used to test methods relating to A2
#  @date 16/02/2021
#  @details This file tests methods within CircleT.py, TriangleT.py,
#           BodyT.py, and Scene.py

import pytest
from CircleT import CircleT
from TriangleT import TriangleT
from BodyT import BodyT
from Scene import Scene

goodcircle = CircleT(10, 11, 12, 13)


def test_xcoordcircle():
    expectedxcoord = 10
    assert goodcircle.cm_x() == expectedxcoord


def test_ycoordcircle():
    expectedycoord = 11
    assert goodcircle.cm_y() == expectedycoord


def test_mass():
    expectedmass = 13
    assert goodcircle.mass() == expectedmass


def test_minert():
    expected_inert = 936
    assert goodcircle.m_inert() == expected_inert


def test_validrad():
    with pytest.raises(ValueError) as e:
        CircleT(10, 12, -1, 13)
    assert "Invalid Input" == str(e.value)


def test_validmass():
    with pytest.raises(ValueError) as e:
        CircleT(10, 11, 12, -1)
    assert "Invalid Input" == str(e.value)


goodtriangle = TriangleT(10, 20, 30, 40)


def test_xcoordtriangle():
    expectedxcoord = 10
    assert goodtriangle.cm_x() == expectedxcoord


def test_ycoordtriangle():
    expectedycoord = 20
    assert goodtriangle.cm_y() == expectedycoord


def test_masstriangle():
    expectedmass = 40
    assert goodtriangle.mass() == expectedmass


def test_minerttriangle():
    expectedminert = 3000
    assert goodtriangle.m_inert() == expectedminert


def test_validsidelength():
    with pytest.raises(ValueError) as e:
        TriangleT(10, 20, -1, 40)
    assert "Invalid Input" == str(e.value)
```

```
def test_validmass_triangle():
    with pytest.raises(ValueError) as e:
        TriangleT(10, 20, -1, 40)
    assert "Invalid Input" == str(e.value)


b = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])


def test_cm_x_body():
    expectedxcoord = 0
    assert b.cm_x() == expectedxcoord


def test_cm_y_body():
    expectedycoord = 0
    assert b.cm_y() == expectedycoord


def test_mass_body():
    expectedmass = 40
    assert b.mass() == expectedmass


def test_minert_body():
    expectedminert = 80
    assert b.m_inert() == expectedminert


def test_valid_body():
    with pytest.raises(ValueError) as e:
        BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, -1])
    assert "Invalid Input" == str(e.value)


def test_valid_body2():
    with pytest.raises(ValueError) as e:
        BodyT([1, -1, -1, 1], [1, 1, -1], [10, 10, 10, 10])
    assert "Invalid Input" == str(e.value)


triangle = TriangleT(1, 2, 3, 4)


def Fx(t):
    return 34 if t < 19 else 0


def Fy(t):
    return 17


goodscene = Scene(triangle, Fx(4), Fy(4), 10, 12)


def test_set_shape():
    goodscene.set_shape(triangle)
    assert goodscene.get_shape() == (triangle)


def test_set_forces():
    goodscene.set_unbal_forces(Fx(4), Fy(4))
    assert goodscene.get_unbal_forces() == (34, 17)


def test_set_forces2():
    goodscene.set_unbal_forces(Fx(35), Fy(4))
    assert goodscene.get_unbal_forces() == (0, 17)


def test_set_velocities():
    goodscene.set_init_velo(10, 12)
    assert goodscene.get_init_velo() == (10, 12)


def test_get_shape():
    expectedshape = triangle
    assert goodscene.get_shape() == expectedshape
```

```
def test_get_forces():
    expectedforces = (0, 17)
    assert goodscene.get_unbal_forces() == expectedforces


def test_get_velo():
    expectedvelocities = (10, 12)
    assert goodscene.get_init_velo() == expectedvelocities
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Nathan Uy
#  @brief Contains the CircleT module which creates circles.
#  @date 02/16/2021

from Shape import Shape

## @brief An abstract data type that represents circles.


class CircleT(Shape):

    ## @brief A CircleT constructor.
    # @details Initializes a CricleT object with its x, y position, radius, and mass.
    # @param x A float that represents the x-coordinate of the circle.
    # @param y A float that represents the y-coordinate of the circle.
    # @param r A float that represents the radius of the circle.
    # @param m A float that represents the mass of the circle.
    # @throws ValueError Throws ValueError if the radius and mass
    # of the circle are less than or equal to 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 Gets the x-coordinate of the circle.
    # @return A float that represents the x-coordinate of the circle.
    def cm_x(self):
        return self.x

    ## @brief Gets the y-coordinate of the circle.
    # @return A float that represents the y-coordinate of the circle.
    def cm_y(self):
        return self.y

    ## @brief Gets the mass of the circle.
    # @return A float that represents the mass of the circle.
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the circle.
    # @return A float that represents the moment of inertia of the circle.
    def m_inert(self):
        return (self.m * self.r**2) / 2
```

M Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Nathan Uy
#  @brief Contains the TriangleT module which creates triangles.
#  @date 02/16/2021

from Shape import Shape

## @brief An abstract data type that represents equilateral triangles.


class TriangleT(Shape):

    ## @brief A TriangleT constructor.
    # @details Initializes a TriangleT object with its x, y position, side, and mass.
    # @param x A float that represents the x-coordinate of the triangle.
    # @param y A float that represents the y-coordinate of the triangle.
    # @param r A float that represents the side length of the triangle.
    # @param m A float that represents the mass of the triangle.
    # @throws ValueError Throws ValueError if the side length and mass
    # of the triangle are 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 Gets the x-coordinate of the triangle.
    # @return A float that represents the x-coordinate of the triangle.
    def cm_x(self):
        return self.x

    ## @brief Gets the y-coordinate of the triangle.
    # @return A float that represents the y-coordinate of the triangle.
    def cm_y(self):
        return self.y

    ## @brief Gets the mass of the triangle.
    # @return A float that represents the mass of the triangle.
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the triangle.
    # @return A float that represents the moment of inertia of the triangle.
    def m_inert(self):
        return (self.m * self.s**2) / 12
```

N Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Nathan Uy
#  @brief Contains the BodyT module which creates bodies.
#  @date 02/16/2021

from functools import reduce
from Shape import Shape

## @brief An abstract data type that represents a body.


class BodyT(Shape):

    ## @brief A BodyT constructor.
    # @details Initializes a BodyT object with its center of mass wrt x and y,
    # its mass, and its moment.
    # @param x A sequence of floats that represents the x-coordinate of each shape.
    # @param y A sequence of floats that represents the y-coordinate of each shape.
    # @param m A sequence of floats that represents the mass of each shape.
    # @throws ValueError Throws ValueError if the length of x, y and m are not equal
    # or if any of the mass is less than or equal to zero.
    def __init__(self, x, y, m):
        if not(len(x) == len(y) == len(m)) or \
           not(reduce(lambda x, y: x and (y > 0), m, True)):
            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.__sum__(m) * \
            (self.__cm__(x, m)**2 + self.__cm__(y, m)**2)

    ## @brief Gets the x component of the body's center of mass.
    # @return A float that represents the x component of the body's center of mass.
    def cm_x(self):
        return self.cmx

    ## @brief Gets the y component of the body's center of mass.
    # @return A float that represents the y component of the body's center of mass.
    def cm_y(self):
        return self.cmy

    ## @brief Gets the mass of the body.
    # @return A float that represents the mass of the body.
    def mass(self):
        return self.m

    ## @brief Gets the moment of inertia of the body.
    # @return A float that represents the moment of inertia of the body.
    def m_inert(self):
        return self.moment

    ## @brief Calculates the overall mass of the body.
    # @param m A sequence of floats that represents the
    # mass of each shape in the body.
    # @return A float that represents the total mass of the shapes in the body.
    def __sum__(self, m):
        return reduce(lambda x, y: x + y, m, 0)

    ## @brief Calculates the center of mass of the body.
    # @param z A sequence of floats that represents the x position
    # of each shape in the body.
    # @param m A sequence of floats that represents the
    # mass of each shape in the body.
    # @return A float that represents the center of mass of the body.
    def __cm__(self, z, m):
        accum = 0
        for i in range(len(m)):
            accum += z[i] * m[i]
        return accum / self.__sum__(m)

    ## @brief Calculates the mass moment of inertia of the body.
    # @param x A sequence of floats that represents the x position of each
    # shape in the body.
    # @param y A sequence of floats that represents the y position of each
    # shape in the body.
    # @param m A sequence of floats that represents the mass of each
    # shape in the body.
    # @return A float that represents the mass moment of inertia of the body.
    def __mmom__(self, x, y, m):
        accum = 0
        for i in range(len(m)):
            accum += m[i] * (x[i]**2 + y[i]**2)
        return accum / self.__sum__(m)

b = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
print(b.m_inert())
```

O Code for Partner's Scene.py

```python
## @file Scene.py
#  @author Nathan Uy
#  @brief Contains the Scene module which creates scenes.
#  @date 02/16/2021
#  @details A Scene has a shape, unbalance force functions in x and y
#  and initial velocities in x and y.

from scipy.integrate import odeint
from Shape import Shape

## @brief An abstract data type that represents a scene.


class Scene:

    ## @brief A Scene constructor.
    # @details Initializes a Scene object with s, fx, fy, vx, and vy.
    # @param s A shape object.
    # @param fx The unbalanced force function in the x direction.
    # @param fy The unbalanced force function in the y direction.
    # @param vx A float that represents the x direction of initial velocity.
    # @param vy A float that represents the x direction of initial velocity.
    def __init__(self, s, fx, fy, vx, vy):
        self.s = s
        self.fx = fx
        self.fy = fy
        self.vx = vx
        self.vy = vy

    ## @brief Gets the shape of the Scene object.
    # @return A Shape object.
    def get_shape(self):
        return self.s

    ## @brief Gets the unbalanced force functions in the x and y direction.
    # @return 2 functions that represent the unbalanced force functions
    # in the x and y direction, respectively.
    def get_unbal_forces(self):
        return (self.fx, self.fy)

    ## @brief Gets the initial velocity in the x and y direction.
    # @return 2 floats that represent the initial velocity in the
    # x and y direction, respectively.
    def get_init_velo(self):
        return (self.vx, self.vy)

    ## @brief Sets the Shape of the Scene object.
    # @param s A Shape object that will replace the Scene's current Shape.
    def set_shape(self, s):
        self.s = s

    ## @brief Sets the unbalanced force functions of the Scene object.
    # @param newfx A function that will replace the Scene's current
    # unbalanced force function in x direction.
    # @param newfy A function that will replace the Scene's current
    # unbalanced force function in y direction.
    def set_unbal_forces(self, newfx, newfy):
        self.fx = newfx
        self.fy = newfy

    ## @brief Sets the initial velocities of the Scene object.
    # @param newvx A float that will replace the Scene's current
    # initial velocity in the x direction.
    # @param newvy A float that will replace the Scene's current
    # initial velocity in the y direction.
    def set_init_velo(self, newvx, newvy):
        self.vx = newvx
        self.vy = newvy

    ## @brief Calculates the interval of t and the odeint of the object.
    # @details Uses the scipy library to calculate odeint
    # @param tfinal A float that represents the final time.
    # @param nsteps An integer greater than or equal to zero
    # that represents the number of partitions of t.
    # @return A sequence of floats and a sequence of sequence of 4 floats
    # that represents the t interval and the odeint respectively.
```

```
def sim(self, tfinal, nsteps):
    t = list(map(lambda i: i * tfinal / (nsteps - 1), range(nsteps)))
    return t, odeint(self.__ode__, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)

## @brief Calculates the derivative of seq at t.
# @param seq A sequence of 4 floats.
# @param t A float that represents the time.
# @return A sequence of 4 floats which represents the
# derivative of seq at t.
def __ode__(self, seq, t):
    new_seq = [seq[2], seq[3], self.fx(t) / self.s.mass(), self.fy(t) / self.s.mass()]
    return new_seq
```