Assignment 2 Solution

Mohammad Omar Zahir - zahirm1

February 25, 2021

This report discusses the testing phase for Shape.py, CircleT.py, TriangleT.py, BodyT.py, Scene.py, and Plot.py. 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 testing for the modules in this assignment was done quite similarly to the first one, where many test cases were carefully chosen to account for both regular and average cases, as well as more specific boundary and edge cases. For the regular cases, common values in an appropriate range from both positive and negative values were given, as would be the case in most scenarios.

To test the rigour of the modules, it was important to cover as many boundary and edge cases as possible, to ensure that the modules behaved reasonably well under extreme circumstances. One of these such tests were done on the exception cases, to ensure that the state invariants were properly being checked for. For this reason, most of the modules contained at least two exception tests, one for every condition of the invariant, as well as testing for them together as well. Furthermore, the types of values given to the functions were not only regular, but also given at very large and small values, both negative and positive, in the hopes of testing any irregular behavior in those cases.

In terms of other testing decisions, many functions like getters and setters, were tested below, with the rationale provided below. Similarly, many usual and general use test cases were tested, like those for objects falling under the effect of gravity (freefall), as well as very familiar projectile and ballistic motions. The advantages from testing these scenarios were that the results that were produced were quite easily verifiable, as the motion of such objects have a very distinguished result. This philosophy was especially taken into

consideration when testing the *Plot* module manually, which allowed us to identify the correct results of the testing almost immediately.

Certain testing techniques that I had not yet used before were also introduced for the test cases, the most notable strategy of which was the method to test the results of the *Scene* module. Due to the complicated nature of the *Scene* and it's outputs, a testing technique was derived where, through the use of for loops, all the expected outputs were efficiently tested, and the method could be repeated for other results of *Scene* quite easily.

2 Results of Testing Partner's Code

After importing the partner files, and using my own implementations for *Shape.py* and *Plot.py*, I found that all 40 of the automated tests created in my $test_driver.py$ had all passed. Similarly, all the manual tests that I had created for *Plot.py* were also shown to be identical to my results. Upon closer inspection into my partner's code, apart from a few differences in design decisions, our code was nearly identical in implementation, which is why they also produced identical results.

Comparing the same partner testing from the first assignment, I found there to be a drastic difference in both the code implementations and results. In the previous assignment, my partner's code and my own were quite different, despite having the identical, albeit informal, specification sheet, which is likely what contributed to the difference. In this assignment, however, it was found that nearly all of our implementations were identical, from the input specifications to the output formatting, which is a direct consequence of the mathematically rigid specifications that were provided. Furthermore, the results of testing my partner code in the previous assignments resulted in many failed test cases, mainly due to misinterpretations that I had assumed that my partner had made when designing his module, while this time, all of the test cases passed. Overall, taking away from the comparison of this exercise from both assignments I found that a more mathematically rigid specification is absolutely necessary for producing reliable and consistent code.

3 Critique of Given Design Specification

The design specification given in this assignment was starkly different to the one given in the first assignment. The major difference, as discussed above, was the mathematically rigid module specifications that were given, which left no room for ambiguity. Thanks to their rigidness, and apart from a few design difference, my code and my partners code

performed identical, having passed all the tests-cases and producing the same graphs. Some aspects of the specifications that I appreciated included the consistency of the specification, which made implementing the code quite efficient and intuitive. Some minor examples that were appreciated include the uniform pattern in terms of variable inputs, the ordering of certain variable declarations inside methods, the order by which exception handling was implemented, as well as certain consistent naming conventions. Although these properties have minimal impact on the module itself, it can make a difference for overall cleanliness and even incorporates for designing for change as things will be easier to follow if looked at again. It is also worth noting that the specifications supported high cohesion within the modules, meaning that modules had methods that had related functions and were performing tasks that were focused on a central purpose. Each shape module, for example, had methods that specifically pertained to the correct functioning and outcome of the final product for the module, and were all working closely together to achieve that result, like how Scene had the sim function, as well as getters and setters for the force and velocity.

Through an essentiality perspective, most of the methods were purposeful and in use by performing essential functions, either by performing calculations, or getting and setting state variables. That being said, certain methods, the local functions in particular, could be deemed unessential as their computations can be done inside other major methods. One example in particular is the sum function, which does not need a separate method and can be done in python natively.
Furthermore, I also believe that the modules had high coupling, whereas low coupling is usually preferred, with the fact that *Shape.py* was being used by nearly all the methods. Although this is discussed further in part j of question 7, I believe that this implementation can be justified in our case since *Shape* was quite an important interface, and all the resulting shape modules used it as a template.
While the specification had some aspects of minimality, as most methods were following a single objective, it cannot be defined as minimal overall. This is due to the fact that certain setters were changing more than one state variable, like the setters for forces in *Scene.py*, which violates the principle of minimality. This decision can be justified, however, because if we were to create separate mutators, this could lead to increased amount of code use, when the current implementation compactly and effectively can get the same implementation done.
It is also worth noting that the specifications did not incorporate opacity, mainly due to the fact that the implementation was done in Python. Python does not explicitly support the functionality for private methods in the same way that languages like Java do, meaning that we can not enforce information hiding, and thus, opacity. A simple solution for maintaining opacity could be to implement the modules in a language like Java where

these functionalities are supported.
Moreover, when looking for generality in the specification we can see that some modules are not as general as they can be. One such example is $TriangleT$ which generates a equilateral triangle object for simulation, as it takes only one side length and projects it for all sides. While this does serve our purpose, it would appeal to a larger variety of use cases if the functionality for scalene and isosceles triangles were added as well, widening our application scope. This could be done seamlessly by having a single side length as the default input, but if only two more are provided, a custom triangle object can be made. While this may change some of our implementation, as the exceptions and calculations may be altered, the resulting module could appeal to a larger use case.

Apart from the principles discussed above, while I greatly appreciate the mathematical rigidness of the specifications, due to its functional nature some implementations were hard to grasp initially. In particular, I found some trouble with the *sim* method from *Scene.py*, where it was not immediately clear what some of the variables like *odeint* and *ode* represented, or how they behaved. While these mathematical notations are essential, as they leave no room for ambiguity, I believe that an improvement that could be made is to complement these notations with a brief example in plain English, which could help explain some of the more complicated aspects of the model.

4 Answers

a) By definition, getters and setters serve the simple purpose to protect your data, meaning your state variables can only be accessed or altered through them. For that reason, it might not immediately make sense for someone to run test cases for them as they usually contain a single line of code for a return statement. In most, cases there is no need as getters and setters are quite simple and if there are restrictions on deadlines or there is confidence that they were implemented properly than it would not be required to test them. However, I believe it may be important to test these methods for a few reasons, one such being consistency. Because nearly every other aspect of the code implementation is being checked for correctness, most of them being more difficult tests to do in general, it would not make sense for something like setters and getters to be glossed over. This brings me to my second point where I argue that there are important situations where it may be necessary to test a setter and getter. The existence of setter and getter methods imply that other integral methods in the module are using them to change the state variables. This would mean that if the methods do not work as intended, this would essentially destroy the productivity of the entire module.

b) In my test cases, I went through the liberty of testing my setters and getters for the Fx and Fy functions, mainly for the reasons stated in part a). Two separate functions were specifically made for the testing of the setters and getters. For testing the getters, the output of the get function was compared directly to the known functions that were having the getter called on them, with a pass assuring that the getter was working as intended. To test the setter, we would have to leverage the functionality of the getter, which we have confirmed to be working correctly. This was done by first using the setter to set the function used by an object to another distinct function specifically, and then use the getter and compare the changed version directly with the one that it is expected to have. These testing implementations as described can be seen in the code for further clarification.

c) If automated tests were required to be done on Plot.py, matplotlib allows for the saving of generated graphs as image files using the $.savefig$ method. With this, we can generate the expected graphs separately using matplotlib, and then compare these two images with a pixel-by-pixel difference by converting the images into a sequence of pixel arrays, which can be done using *imread* which can be found in the *scipy* module.

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

- output: $out := \dfrac{\text{norm}(\text{difference}(x_{\text{calc}}, x_{\text{true}}))}{\text{norm}(x_{\text{true}})} < \epsilon$
- exception: Not mentioned

Local Functions

difference : seq of $\mathbb{R}$ $\times$ seq of $\mathbb{R}$ $\to$ seq of $\mathbb{R}$
difference$(a, b) \equiv [x : \mathbb{N} \mid x \in [0..|a_{\text{s}}| - 1] : a_{\text{x}} - b_{\text{x}}]$
norm : seq of $\mathbb{R} \to \mathbb{R}$
norm$(a) \equiv \exists x \in a \mid \forall y \in a \mid |x| \geq |y| : x$

For further clarification, the norm function looks for the largest element in the list, and returns that as a single output.

e) In the given specification, we have restrictions for the mass and shape dimensions simply because we are modelling reality, and it is simply impossible for objects that are modelled using shapes to have negative mass or dimensions. The position coordinates of x and y, however, are things that should have the possibility of both positive and negative values. One such reason for negative values is for positions with a certain perspective. For example, if you are modelling the position of an object above sea level, the advantages of having a negative y-coordinate would be to represent the height below sea-level, with the positive representing the height above. Similarly,

negative positions allow you to simulate the movement of an object on the entire 2D plane with no restrictions, allowing you to see the full extent of how much an object has moved relative to it's original position when placed at the origin. As such, it would be counter-intuitive to limit our scope of the 2D-plane for our applications by checking for negative position coordinates.

f) We can informally prove that the invariant for TriangleT will always be satisfied because of the exception check during the initialization, where s and m are both checked to be greater than 0, and a $ValueError$ is thrown otherwise. Furthermore, because we know that there are no other mutators in the module, meaning that the values for s and m cannot be changed after initialization, we can confirm that if the object passes the initialization, there is no way for the invariant to not hold given our current implementation.

g)
```
[x**(1/2) for x in range(5,20) if x%2 != 0]
```

h)
```
def no_upper(s):
    l = []
    for c in s:
        if not c.isupper():
            l += [c]
    return ''.join(l)
```

i) Abstraction is the software engineering principle that pertains to distancing yourself from unimportant details that complicate the problem, and instead 'abstracting' them away and getting a simpler foundation for the solution. Generality is the principle that deals with solving a more 'general' version of an existing problem rather than a very specific or detailed one. Although they are separate principles, it is clear that these principles can overlap in more than a few cases, as often generality is a by-product of abstraction. Abstraction can often lead to a more general version of the problem as we are removing details that can be overlooked for the simpler implementation. For example, if there is a complex problem at hand and abstraction is suggested to make the problem easier, it can often be the case that the problem you are looking at may have many specific state variables and scenarios, and solving a more general case of the problem could give you the initial foundation to solve the harder solution. In that situation, you would be abstracting away the unimportant details of the extremely specific situation by solving a general version of the problem, making generality an application of abstraction.

j) Generally, in software engineering, coupling refers to how dependent individual modules are to one another, where low coupling refers to increased independence, and high coupling implies more dependence. As we have learnt, it is ideal for a good implementation to have low coupling, as this means that if a module does have to be changed, we do not need to worry about the impacts on other modules since they are designed to be independent. In this scenario, if we are forced to have high coupling, I believe that it is better for many modules to be dependant on a single module rather than a single module being dependant on many others. As testing for the other modules happens, the one module that is depended on will continuously be modified and rigorously tested. On the other hand, if we were to allow a single module to use many others, in the case that there are bugs in the dependant module, it may take time to determine the exact module, from the many that it is depending on, to find the issue. To see an example of the better approach, we can look at our specifications of this assignment. In the assignment we have one module *Shape.py*, which is being depended on by most of the other modules, and the benefits of this can clearly be seen throughout the assignment. If it were the opposite, it would result in a lot of backtracking between the depended and dependant modules. The Fan-in approach also has the added benefit of code reuse, as can be seen from the assignment, that all the modules using *Shape.py* use the abstract methods that are set by it. The Fan-out implementation also has the added drawback of being fragile because of the 'all-or-nothing' characteristic - nearly all the other modules must be implemented fully before we can work on our single module. It is also worth noting from the example in our assignment that if the 'Fan-in' approach is being used, it is advisable that the central module should be kept as simple and minimal as possible, as we can see in the example of *Shape.py* being a template class.

E Code for Shape.py

```
## @file Shape.py
#  @author Mohammad Omar Zahir, zahirm1
#  @brief This class acts as a blueprint interface module for Shapes to create other
# Shape classes.
#  @date Feb 11, 2021

from abc import ABC, abstractmethod


## @brief Creating an abstract data type for working with Shape objects and their
# basic properties. The methods are abstract and meant to be overwritten by the
# inheriting modules.
class Shape(ABC):

    ## @brief Method for determining the x-coordinate of the center of mass.
    @abstractmethod
    def cm_x(self):
        pass

    ## @brief Method for determining the y-coordinate of the center of mass.
    @abstractmethod
    def cm_y(self):
        pass

    ## @brief Method for determining the mass.
    @abstractmethod
    def mass(self):
        pass

    ## @brief Method for determining the moment of inertia.
    @abstractmethod
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a circle
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a circle
#  @details represention a circle with properties center of mass x and y coordinates,
#   radius r, and mass m
class CircleT(Shape):

    ## @brief Constructor for CircleT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param r is the radius
    #  @param m is the mass
    def __init__(self, x, y, r, m):
        if r <= 0 or m <= 0:
            raise ValueError("r and m are less than or equal to 0")
        self.__x = x
        self.__y = y
        self.__r = r
        self.__m = m

    ## @brief returns the x component of the center of mass
    #  @return x
    def cm_x(self):
        return self.__x

    ## @brief returns the y component of the center of mass
    #  @return y
    def cm_y(self):
        return self.__y

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the inertia
    #  @return mr^2/2
    def m_inert(self):
        return (self.__m) * ((self.__r)**2) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a triangle
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a triangle
#  @details represention a triangle with properties center of mass x and y coordinates,
#   sidelength s, and mass m
class TriangleT(Shape):

    ## @brief Constructor for TriangleT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param s is the sidelength
    #  @param m is the mass
    def __init__(self, x, y, s, m):
        if s <= 0 or m <= 0:
            raise ValueError("s and m are less than or equal to 0")
        self.__x = x
        self.__y = y
        self.__s = s
        self.__m = m

    ## @brief returns the x component of the center of mass
    #  @return x
    def cm_x(self):
        return self.__x

    ## @brief returns the y component of the center of mass
    #  @return y
    def cm_y(self):
        return self.__y

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the inertia
    #  @return ms^2/12
    def m_inert(self):
        return (self.__m) * ((self.__s)**2) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a body
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a body
#  @details represention a body with properties center of mass x and y coordinates,
#   mass m, and moment
class BodyT(Shape):

    ## @brief Constructor for BodyT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param m is the mass
    def __init__(self, x, y, m):
        if len(x) != len(y) or len(x) != len(m) or len(y) != len(m):
            raise ValueError("lengths not equal")
        for i in m:
            if i <= 0:
                raise ValueError("not all masses are more than 0")
        self.__cmx = cmm(x, m)
        self.__cmy = cmm(y, m)
        self.__m = summ(m)
        self.__moment = mmomm(x, y, m) - summ(m) * (cmm(x, m)**2 + cmm(y, m)**2)

    ## @brief returns the x component of the center of mass
    #  @return cmx
    def cm_x(self):
        return self.__cmx

    ## @brief returns the y component of the center of mass
    #  @return cmy
    def cm_y(self):
        return self.__cmy

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the moment
    #  @return moment
    def m_inert(self):
        return self.__moment


## @brief local function
#  @return summation
def summ(m):
    summation = 0
    for i in m:
        summation += i
    return summation


## @brief local function
#  @return center of mass
def cmm(z, m):
    total = 0
    for i in range(len(m)):
        total += z[i] * m[i]
    return total / summ(m)


## @brief local function
#  @return momemt
def mmomm(x, y, m):
    mom = 0
    for i in range(len(m)):
        mom += m[i] * ((x[i]**2) + (y[i]**2))
    return mom
```

I Code for Scene.py

```
## @file Scene.py
#  @author Simone Ocvirk
#  @brief sets the scene
#  @date February 15th 2021
#  @details

# from Shape import Shape
from scipy.integrate import odeint


## @brief sets the scene
#  @details makes sure the vibes right
class Scene():

    ## @brief Constructor for BodyT
    #  @param s is a Shape
    #  @param Fx is the x component of the unbalanced force
    #  @param Fy is the y component of the unbalanced force
    #  @param vx is the x component of the velocity
    #  @param vy is the y component of the 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 returns the shape
    #  @return s
    def get_shape(self):
        return self.__s

    ## @brief returns the x and y components of the unbalanced force as a tuple
    #  @return (Fx, Fy)
    def get_unbal_forces(self):
        return (self.__Fx, self.__Fy)

    ## @brief returns the x and y components of the velocity as a tuple
    #  @return (vx, vy)
    def get_init_velo(self):
        return (self.__vx, self.__vy)

    ## @brief sets the shape as a new shape
    def set_shape(self, s):
        self.__s = s

    ## @brief sets the x and y components of the unbalanced force
    def set_unbal_forces(self, Fx, Fy):
        self.__Fx = Fx
        self.__Fy = Fy

    ## @brief sets the x and y components of the velocity
    def set_init_velo(self, vx, vy):
        self.__vx = vx
        self.__vy = vy

    ## @brief returns two sequences as a tuple
    #  @return (sequence1, sequence2)
    def sim(self, tf, steps):
        ## @brief local function
        #  @return something
        def ode(w, t):
            return (w[2], w[3], self.__Fx(t) / self.__s.mass(),
                    self.__Fy(t) / self.__s.mass())
        t = []
        for i in range(steps):
            t.append(i * tf / (steps - 1))
        return (t, odeint(ode, [self.__s.cm_x(), self.__s.cm_y(),
                self.__vx, self.__vy], t))
```

J Code for Plot.py

```
## @file Plot.py
#  @author Mohammad Omar Zahir, zahirm1
#  @brief This class implements the plot function to plot graphs from the data generated
#  by the Scene class.
#  @date Feb 11, 2021

import matplotlib.pyplot as plt


## @brief Method to plot the graph of various data values.
# @param w The window environment.
# @param t The set of time values.
# @throws ValueError When the length of the window environment is not the same as
# the data values for time, t
def plot(w, t):

    if not (len(w) == len(t)):
        raise ValueError
    x = []
    y = []
    for i in range(len(w)):
        x += [w[i][0]]
        y += [w[i][1]]
    fig, axs = plt.subplots(3)
    fig.suptitle('Motion Simulation')
    axs[0].plot(t, x)
    axs[1].plot(t, y)
    axs[2].plot(x, y)

    axs.flat[0].set(xlabel='t (m)', ylabel='x (m)')
    axs.flat[1].set(xlabel='t (m)', ylabel='y (m)')
    axs.flat[2].set(xlabel='x (m)', ylabel='y (m)')
    plt.subplots_adjust(hspace=0.5)
    plt.show()
```

K Code for test_driver.py

```
## @file test_driver.py
#  @author Mohammad Omar Zahir, zahirm1
#  @brief Test cases for the various python modules in Assignment 2
#  @date Feb 16, 2021
#  @details

import pytest
from CircleT import CircleT
from TriangleT import TriangleT
from BodyT import BodyT
from Scene import Scene
from Plot import plot
import math

g = 9.81


def Fx(t):
    return 0


def Fy(t):
    return -g * 8


def Fx2(t):
    return 10


def Fy2(t):
    return -g * 16


epsilon = 0.0001


class TestCircleT:

    def setup_method(self, method):
        self.circle1 = CircleT(1, 1, 1, 1)
        self.circle2 = CircleT(-100, -10, 5, 10)

    def tear_down(self, method):
        self.circle1 = None
        self.circle2 = None

    def test_xcoord(self):
        assert self.circle1.cm_x() == 1

    def test_xcoord2(self):
        assert self.circle2.cm_x() == -100

    def test_ycoord(self):
        assert self.circle1.cm_y() == 1

    def test_ycoord2(self):
        assert self.circle2.cm_y() == -10

    def test_mass(self):
        assert self.circle1.mass() == 1

    def test_mass2(self):
        assert self.circle2.mass() == 10

    def test_m_inert(self):
        assert self.circle1.m_inert() == 0.5

    def test_m_inert2(self):
        assert self.circle2.m_inert() == 125

    # Exception Cases
    def test_ValueError(self):
        with pytest.raises(ValueError):
            CircleT(1, 1, 0, 1)

    def test_Value_Error2(self):
```

```
        with pytest.raises(ValueError):
            CircleT(1, 1, 1, 0)


class TestTriangleT:

    def setup_method(self, method):
        self.triangle1 = TriangleT(1, 1, 1, 1)
        self.triangle2 = TriangleT(1000, -2500, 7, 9)

    def tear_down(self, method):
        self.triangle1 = None
        self.triangle2 = None

    def test_xcoord(self):
        assert self.triangle1.cm_x() == 1

    def test_xcoord2(self):
        assert self.triangle2.cm_x() == 1000

    def test_ycoord(self):
        assert self.triangle1.cm_y() == 1

    def test_ycoor2(self):
        assert self.triangle2.cm_y() == -2500

    def test_mass(self):
        assert self.triangle1.mass() == 1

    def test_mass2(self):
        assert self.triangle2.mass() == 9

    def test_m_inert(self):
        assert self.triangle1.m_inert() == 1 / 12

    def test_m_inert2(self):
        assert self.triangle2.m_inert() == 147 / 4

    # Exception Cases
    def test_ValueError(self):
        with pytest.raises(ValueError):
            TriangleT(1, 1, 0, 1)

    def test_ValueError2(self):
        with pytest.raises(ValueError):
            TriangleT(1, 1, 1, 0)


class TestBodyT:

    def setup_method(self, method):
        self.body1 = BodyT([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12])
        self.body2 = BodyT([0, 0, 0, 0], [0, 0, 0, 0], [1, 2, 3, 4])

    def tear_down(self, method):
        self.body1 = None
        self.body2 = None

    def test_cmx1(self):
        assert self.body1.cm_x() == 110 / 42

    def test_cmx2(self):
        assert self.body2.cm_x() == 0

    def test_cmy1(self):
        assert self.body1.cm_y() == 278 / 42

    def test_cmy2(self):
        assert self.body2.cm_y() == 0

    def test_mass1(self):
        assert self.body1.mass() == 42

    def test_mass2(self):
        assert self.body2.mass() == 10

    def test_m_inert1(self):
        assert abs(self.body1.m_inert() - 103.809523) < epsilon

    def test_m_inert2(self):
```

```
        assert self.body2.m_inert() == 0

    # Exception Cases
    def test_ValueError(self):
        with pytest.raises(ValueError):
            BodyT([1, 2], [1, 3, 4], [1, 2])

    def test_ValueError2(self):
        with pytest.raises(ValueError):
            BodyT([1, 2], [1, 3], [0, 2])


class TestSceneT():

    def setup_method(self, method):
        self.circle1 = CircleT(1, 2, 4, 9)
        self.triangle1 = TriangleT(2, 7, 3, 4)
        self.scene1 = Scene(self.circle1, Fx, Fy, 0, 0)
        self.scene2 = Scene(self.triangle1, Fx, Fy, 0, 0)

    def teardown_method(self, method):
        self.circle1 = None
        self.triangle1 = None
        self.scene1 = None
        self.scene = None

    def test_get_shape(self):
        assert self.scene1.get_shape() == self.circle1

    def test_get_shape2(self):
        assert self.scene2.get_shape() == self.triangle1

    def test_get_unbal_forces(self):
        assert self.scene1.get_unbal_forces() == (Fx, Fy)

    def test_get_init_velo(self):
        assert self.scene1.get_init_velo() == (0, 0)

    def test_set_shape(self):
        self.scene1.set_shape(self.triangle1)
        assert self.scene1.get_shape() == self.triangle1

    def test_set_unbal_forces(self):
        self.scene1.set_unbal_forces(Fx2, Fy2)
        assert self.scene1.get_unbal_forces() == (Fx2, Fy2)

    def test_set_init_velo(self):
        self.scene1.set_init_velo(2, 10)
        assert self.scene1.get_init_velo() == (2, 10)

    def test_sim(self):
        t, wsol = self.scene1.sim(5, 3)
        t_expec = [0, 2.5, 5]
        wsol_expec = [[1, 2, 0, 0], [1, -25.25, 0, -21.8], [1, -107, 0, -43.6]]

        for i in range(len(t)):
            if t_expec[i] == 0:
                assert t[i] == t_expec[i]
            else:
                assert (t[i] - t_expec[i]) / t_expec[i] < epsilon

        for j in range(len(wsol)):
            for i in range(len(wsol[j])):
                if wsol_expec[j][i] == 0:
                    assert wsol[j][i] == wsol_expec[j][i]
                else:
                    assert (wsol[j][i] - wsol_expec[j][i]) / wsol_expec[j][i] < epsilon

    def test_sim2(self):
        t, wsol = self.scene2.sim(4, 5)
        t_expec = [0.0, 1.0, 2.0, 3.0, 4.0]
        wsol_expec = [[2, 7, 0, 0], [2, -2.81, 0, -19.62], [2, -32.24, 0, -39.24],
                      [2, -81.29, 0, -58.86], [2, -149.96, 0, -78.48]]

        for i in range(len(t)):
            if t_expec[i] == 0:
                assert t[i] == t_expec[i]
            else:
                assert (t[i] - t_expec[i]) / t_expec[i] < epsilon
```

```
        for j in range(len(wsol)):
            for i in range(len(wsol[j])):
                if wsol_expec[j][i] == 0:
                    assert wsol[j][i] == wsol_expec[j][i]
                else:
                    assert (wsol[j][i] - wsol_expec[j][i]) / wsol_expec[j][i] < epsilon


class TestPlotT():

    def setup_method(self, method):
        self.circle = CircleT(2, -4, 7, 8)
        self.triangle = TriangleT(-5, -1, 6, 16)
        self.body = BodyT([0, -3, 2, -1], [4, -5, 7, 2], [2, 2, 2, 2])
        self.scene = Scene(self.circle, Fx, Fy, 0, 0)

    def teardown_method(self, method):
        self.circle = None
        self.triangle = None
        self.body = None
        self.scene = None

    def test_plot(self):
        t, wsol = self.scene.sim(5, 10)
        plot(wsol, t)

        self.scene.set_shape(self.body)
        t, wsol = self.scene.sim(3, 40)
        plot(wsol, t)

        self.scene.set_init_velo(2 * math.cos(math.pi / 2), 2 * math.sin(math.pi / 2))
        self.scene.set_unbal_forces(Fx2, Fy2)
        self.scene.set_shape(self.triangle)
        t, wsol = self.scene.sim(2, 30)
        plot(wsol, t)
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a circle
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a circle
#  @details represention a circle with properties center of mass x and y coordinates,
#   radius r, and mass m
class CircleT(Shape):

    ## @brief Constructor for CircleT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param r is the radius
    #  @param m is the mass
    def __init__(self, x, y, r, m):
        if r <= 0 or m <= 0:
            raise ValueError("r and m are less than or equal to 0")
        self.__x = x
        self.__y = y
        self.__r = r
        self.__m = m

    ## @brief returns the x component of the center of mass
    #  @return x
    def cm_x(self):
        return self.__x

    ## @brief returns the y component of the center of mass
    #  @return y
    def cm_y(self):
        return self.__y

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the inertia
    #  @return mr^2/2
    def m_inert(self):
        return (self.__m) * ((self.__r)**2) / 2
```

M Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a triangle
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a triangle
#  @details represention a triangle with properties center of mass x and y coordinates,
#   sidelength s, and mass m
class TriangleT(Shape):

    ## @brief Constructor for TriangleT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param s is the sidelength
    #  @param m is the mass
    def __init__(self, x, y, s, m):
        if s <= 0 or m <= 0:
            raise ValueError("s and m are less than or equal to 0")
        self.__x = x
        self.__y = y
        self.__s = s
        self.__m = m

    ## @brief returns the x component of the center of mass
    #  @return x
    def cm_x(self):
        return self.__x

    ## @brief returns the y component of the center of mass
    #  @return y
    def cm_y(self):
        return self.__y

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the inertia
    #  @return ms^2/12
    def m_inert(self):
        return (self.__m) * ((self.__s)**2) / 12
```

N Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Simone Ocvirk
#  @brief defines the properties of a body
#  @date February 15th 2021

from Shape import Shape


## @brief defines the properties of a body
#  @details represention a body with properties center of mass x and y coordinates,
#   mass m, and moment
class BodyT(Shape):

    ## @brief Constructor for BodyT
    #  @param x is the x component of the center of mass
    #  @param y is the y component of the center of mass
    #  @param m is the mass
    def __init__(self, x, y, m):
        if len(x) != len(y) or len(x) != len(m) or len(y) != len(m):
            raise ValueError("lengths not equal")
        for i in m:
            if i <= 0:
                raise ValueError("not all masses are more than 0")
        self.__cmx = cmm(x, m)
        self.__cmy = cmm(y, m)
        self.__m = summ(m)
        self.__moment = mmomm(x, y, m) - summ(m) * (cmm(x, m)**2 + cmm(y, m)**2)

    ## @brief returns the x component of the center of mass
    #  @return cmx
    def cm_x(self):
        return self.__cmx

    ## @brief returns the y component of the center of mass
    #  @return cmy
    def cm_y(self):
        return self.__cmy

    ## @brief returns the mass
    #  @return m
    def mass(self):
        return self.__m

    ## @brief returns the moment
    #  @return moment
    def m_inert(self):
        return self.__moment


## @brief local function
#  @return summation
def summ(m):
    summation = 0
    for i in m:
        summation += i
    return summation


## @brief local function
#  @return center of mass
def cmm(z, m):
    total = 0
    for i in range(len(m)):
        total += z[i] * m[i]
    return total / summ(m)


## @brief local function
#  @return momemt
def mmomm(x, y, m):
    mom = 0
    for i in range(len(m)):
        mom += m[i] * ((x[i]**2) + (y[i]**2))
    return mom
```

O Code for Partner's Scene.py

```
## @file Scene.py
#  @author Simone Ocvirk
#  @brief sets the scene
#  @date February 15th 2021
#  @details

# from Shape import Shape
from scipy.integrate import odeint


## @brief sets the scene
#  @details makes sure the vibes right
class Scene():

    ## @brief Constructor for BodyT
    #  @param s is a Shape
    #  @param Fx is the x component of the unbalanced force
    #  @param Fy is the y component of the unbalanced force
    #  @param vx is the x component of the velocity
    #  @param vy is the y component of the 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 returns the shape
    #  @return s
    def get_shape(self):
        return self.__s

    ## @brief returns the x and y components of the unbalanced force as a tuple
    #  @return (Fx, Fy)
    def get_unbal_forces(self):
        return (self.__Fx, self.__Fy)

    ## @brief returns the x and y components of the velocity as a tuple
    #  @return (vx, vy)
    def get_init_velo(self):
        return (self.__vx, self.__vy)

    ## @brief sets the shape as a new shape
    def set_shape(self, s):
        self.__s = s

    ## @brief sets the x and y components of the unbalanced force
    def set_unbal_forces(self, Fx, Fy):
        self.__Fx = Fx
        self.__Fy = Fy

    ## @brief sets the x and y components of the velocity
    def set_init_velo(self, vx, vy):
        self.__vx = vx
        self.__vy = vy

    ## @brief returns two sequences as a tuple
    #  @return (sequence1, sequence2)
    def sim(self, tf, steps):
        ## @brief local function
        #  @return something
        def ode(w, t):
            return (w[2], w[3], self.__Fx(t) / self.__s.mass(),
                    self.__Fy(t) / self.__s.mass())
        t = []
        for i in range(steps):
            t.append(i * tf / (steps - 1))
        return (t, odeint(ode, [self.__s.cm_x(), self.__s.cm_y(),
                self.__vx, self.__vy], t))
```