Assignment 2 Solution

Hamrish Saravanakumar

February 25, 2021

This report discusses the testing phase for the implementation of Assignment 2, specifically the code coverage of `CiricleT.py`, `TriangleT.py`, `BodyT.py`, and `Scene.py`. The assignment specifications are critiqued and the discussion questions are answered.

1 Testing of the Original Program

The following assignment exposed students to using `pytest` in order to implement an effective `test_driver.py` to verify the code coverage of the python files written. The assignment specification required the complete testing of the methods in the python files `CiricleT.py`, `TriangleT.py`, `BodyT.py`, and `Scene.py`. Every method, including the getters and setters were tested using pytest to assert the expected output compared to the output of the written python files.

The `CircleT.py` file only has getter methods, so one test case was written to ensure that the CircleT class successfully gets the `cm_x`, `cm_y`, `mass` and `m_inert` functions. It is important to note that assignment specification clearly stated that a `ValueError` should be raised for a radius or mass value that isn't greater than 0. For this reason, two additional test cases were written to test edge cases that should raise a ValueError before even compiling, due to invalid state variables. Out of these 6 test cases, all 6 were passed. The same test cases were written for `TriangleT.py`, except the ValueError should have been raised for a side length or mass value that isn't greater than 0. Out of these 6 test cases, all 6 were passed.

The first four test cases for `BodyT.py` were identical to the previous two python files, as the four getter methods were tested. However, the assignment specifications had two exceptions that required a `ValueError` to be raised. The BodyT class should not have created a BodyT member if the length of the sequences passed for the x coordinates of the center of mass, y coordinate of the center of mass, and mass of the body were not equal to one another. In addition, if at least one of the values in the sequence of masses was less than or equal to zero, a ValueError should also be raised. Two separate test cases were written to test these edge cases. Out of 6 written tests, all 6 passed.

The first six test cases for `Scene.py` test the setters and getters of the shape, the unbalanced forces and initial velocities using a Scene that is defined with state variables that would allow the Scene object to construct without any issues. The getters were tested by asserted the object of type Shape or tuple that would be expected to be outputted for each respective method. The setter methods were tested by changing a particular property of the Scene object, and using the getter method to assert the appropriate, newly changed object of type Shape or tuple that would be expected to be outputted for each respective method. Testing the `sim` function is not as straight forward, and I was unable to effectively implement a test case for this method. However, I attempted to assert the expected values for both outputs "t" and "wsol" in two separate test cases. This would have been done by using the relative error formula to determine whether the calculated solution was close enough to the expected solution in order to confidently claim that a test case has passed. In each case, the outputted sequence was compared to the expected output, and the two sequences were subtracted from one another. The max value of this sequence was taken, and divided by the max value of the expected solution in order to obtain a relative error value. If this value was less than a value denoted by epsilon, which was chosen to be 10^-5, the two sequences are considered close enough to be equal, and thus the test case would pass. Therefore, out of the 6 test cases that were created with confidence, all 6 passed. For the two test cases that were created to test the `sim` function, both tests failed, but this was a result of not having a fully implemented test case rather than having any fault with the code itself.

Although `test_driver.py` was not required to have test cases for `Plot.py`, it's validity was confirmed used the given `test_expt` file by running `make test` and comparing the generated graph with the Figure 1 in the A2 tex file.

2 Results of Testing Partner's Code

The testing of the partner code was conducted by copying the `CiricleT.py`, `TriangleT.py`, `BodyT.py`, and `Scene.py` files over to the local src folder where my `Plot.py`, `Shape.py` and `test_driver.py` were used in order to run similar automated testing using pytest. This method allowed me to simply run `make test` without having to modeify the `Makefile`. The test cases that my `test_driver.py` ran were explained in Section 1 in order to provide as close to a 100% code coverage as possible. After running `make test`, a graph that looked identical to the plot provided in the assignment specification was created. This was expected because `plot.py` was called upon through `test_expt.py`. After this, pytest ran as expected without any preliminary errors, and passed 24 out of 26 cases. This was expected because the final 2 test cases for the sim method were not written correctly, as mentioned in Section 1. Upon examining the partner code, all of the test cases ran as expected because the code was implemented in an almost identical fashion.

The testing was far more efficient for Assignment 2 when comparing this exercise to the testing conducted in Assignment 1. In order to examine why this was the case, it is important to differentiate between formal and natural language. A general MIS reduces the discrepancies by using formal language, specifically mathematical notation in order to define functions, variables, parameters, exceptions, etc. This was not the case in Assignment 1, as the assignment specifications were written in Natural Language. This leaves a lot of decisions up to the implementer of the code, which proved to be the case as there were many different possible implementations of the code in Assignment 1. Specifically, my partner and I had interpreted the specifications in A1 very differently, which is why the testing was not as straight forward. By implementing an MIS to specify the desired implementation, the discrepancies that may arise from user interpretation is reduced significantly, which makes the testing exercise much more straight forward.

3 Critique of Given Design Specification

The given design specification enables the programmer to create modules for simulating the physics of a scene where a shape moves through 2D space. This was done through the use of a module interface specification (MIS), that prioritizes the use of formal language to provide the properties of the modules. This includes the functions, state variables, invariants, exceptions, assumptions, etc. The use of formal language is effective in reducing the number of discrepancies that an individual often experiences when forced with making implementation decisions. Despite the standard that a MIS written in formal language often establishes, reading a specification that utilizes mathematical notation to define functions is much more complicated than reading a specification that is written in natural language, which made the process of understanding the implementation of the code quite challenging. However, this can not be commented as a drawback of the specification, as the mathematical notation allows for a greater standardization when it comes to the implementation of these modules. `CircleT.py` and `TriangleT.py` allows for greater freedom in other classes. Specifically in `Scene.py`, a sequence of members of these data types is able to be created to be analyzed.

Despite the nature of the formal language used to create the module interface specification for this assignment, there are a few changes to the design that can be proposed as a means of improving the effectiveness of the specification. In regards to `Scene.py`, this module could have been implemented as an abstract object instead of an abstract data type. In this case, specific functions could have been defined using @staticmethod to define them on the class level rather than having calculations within the `init` method to define the state variables. This would allow for functionality when it comes to each function, and can be implemented because there is no practical use of having multiple Scene objects, especially in the context of this assignment. In addition, with `Scene.py`, if the nsteps parameter is equal to 1, the method will throw a `ZeroDivisionError`. There was no exception listed in the specification for this case, nor was an assumption made for the value of nsteps. In order to improve this, the constructor should raise a `ValueError` when this is the case, rather than waiting for the `ZeroDivisionError` to occur. Additionally, the `Shape.py` module would serve more effectively as an abstract object rather than an interface. Although this module is used effectively by modules such as `TriangleT.py`, `CircleT.py`, and `BodyT.py`, defining `Shape.py` as an abstract object would have enabled the programmer to make use of inheritance of the Shape class in order to make use of python's object oriented programming capabilities.

The effectiveness of the specification can also be analyzed based on the interface's ability to provide the programmer checks for potential exceptions. The programmer should be able to check the exception themselves without having to raise it. This is not necessarily the case with the given specification. A majority of the exceptions are resolved by raising a `ValueError`, but this does not provide the programmer themselves to choose what they want to do if this exception is raised. It can be argued that `ValueError` may act as a check for the programmer. This is because it prevents an unexpected error to occur later on in the program. An example is if a `ValueError` is raised for a mass value of 0, a `DivisionByZero` error is prevented from occurring later on in the program. However, in order to improve the programmer's freedom to deal with such checks as they wish, the exception should have been stated, but it should have been up to the programmer to either implement a function that handles the exception, or to simply raise the appropriate error.

Finally, the quality criteria discussed in class can be used as a means of determining the quality of the specification. Based on the observations made regarding some of the needed improvements to the specification, the current implementation is not general because it doesn't allow for increased freedom of the user to implement methods to handle exceptions as they wish, and is not essential because the repetition of the methods used in `Shape.py` that are used in modules such as `TriangleT.py` and `CircleT.py` offers the same service in multiple ways. By separating the setter and getters in `Scene.py` the specification is minimal, and can also be considered opaque because local functions that are not necessarily needed for the implementation of the class themselves are defined as private methods. Overall, there are components in which the specification falls short in generality and essentiality, however the overall use of formal language throughout the MIS provided for an implementation that created standardization among all programmers.

4 Answers

a) The purpose of unit tests are to examine the behaviour of the code in a meaningful was. Although an argument exists that states that testing the getter and setter methods are tedious and are not worth the time, it is important to consider the following claim: If it is possible for a developer to write the code incorrectly, it should be tested. Regardless of how elementary the implementation of such methods may be, there is still room for error that must be tested in order to guarantee 100% code coverage.

b) Two functions would need to be defined within the automated test case for both F_x and F_y. These functions would return a value for the unbalanced forces in the x and y direction based on the value of the input of each function; t. For example, the function $F_x(t)$ would return 1 if the time is less than 5, or else it will return 0. Likewise, the function $F_y(t)$ could be defined such that -9.81 (the force of gravity) is returned if the time is less than 5, or will return 9.81 otherwise. The automated test of the getter method should assert that a tuple of the two valid forces are being returned from the calculated solution based on the value of t. The automated test of the setter method should change the input value of the time for each function, and use the getter method to assert that a tuple of the two new valid forces are being returned based on the newly set value of t. Although these test cases were not required to be implemented in the `test_driver`, an example of how these test cases could be implemented as described previously can be found in this tester.

c) In order to implement an effective automated test, it is required to have an expected solution, the calculated solution, and a means of comparing the two solutions to determine if they are equivalent. The `Plot.py` module utilizes `matplotlib` in order to display three unique graphs. Although a quantifiable output is not necessarily created from `Plot.py`, the documentation of `matplotlib` provides an effective means of extracting a calculated and expected solution. Using the line of code `plt.savefig("figurename.png")`, the graph can be saved as a file. If both the expected and calculated solutions of this test can be represented by a png file, comparing the two image files can simply be done by reading the two files, and asserting if the contents are equal.

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

- output: $out := \frac{max(diff(x_{\text{calc}}, x_{\text{true}}))}{max(x_{\text{true}})} < \epsilon$
- exception: $(\neg(max(x_{\text{true}}) \neq 0) \Rightarrow \text{ValueError} | \neg(|x_{\text{calc}}| = |x_{\text{true}}|) \Rightarrow \text{ValueError})$

Local Functions:

max : seq of $\mathbb{R} \to \mathbb{R}$
$\text{max}(xs) \equiv (a : \mathbb{R} | \exists a \in xs | (\forall b \in xs \cdot a \geq b))$

diff : seq of $\mathbb{R} \times$ seq of $\mathbb{R} \to$ seq of $\mathbb{R}$
$\text{diff}(xs, ys) \equiv [i : \mathbb{R} | i \in [0..|xs| - 1] : xs_i - ys_i]$

e) The given MIS enables the programmer to write a python program to simulate the physics of a scene where a shape moves through 2D space. Therefore, the properties of the measurements and formulas used throughout the specification must be valid in the real world where they are applied. The reason why exceptions existed for non positive dimensions and masses is because such properties would be invalid according to the laws of physics. In regards to the coordinate system that represents the center of mass, similar pre-existing theorems and laws can be referenced in order to decide whether there should be exceptions for negative coordinates. According to the properties of the Cartesian Coordinate system, an x value and y value defined as a coordinate can be any rational number. Therefore, there should not be an exception for negative coordinates, as this would unnecessarily prevent the programmer from working with cases of center of masses that would otherwise be valid in 2D space.

f) In constructing the TriangleT class using the `init` method, an exception is raised. Specifically, if the inputted side length or mass of the triangle are not greater than zero, a `ValueError` should be raised. This directly proves that the state invariant holds before the access program is even called. This is crucial, as it means that the constructor will not build a member of the TriangleT class unless the exception does not occur. Since the `init` method does in fact have a transition, it can be proved that the invariant is always satisfied based on the constructor itself. If a TriangleT is in fact instantiated, the properties defined by the state variables must hold. If the inputs and the state variables hold for the state invariant, then it is proven that the invariant will always hold based on the given specification.

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

h)
```
def remove_uppercase(s):
    return ''.join(i for i in s if not i.isupper())
```

i) While abstraction reduces complexity by removing details that are irrelevant, generalization actually reduces the complexity by replacing multiple entities which perform similar functions with a single construct. For example, a system that manages team basketball statistics will have a lot of properties such as point differential, home wins, away wins, conference wins, etc. However, for the purpose of displaying the standings of the teams, only the number of games played, number of wins and number of losses are needed. This means that the team's stats were abstracted and only the properties that were required for the context of the application were needed. On the other hand, generalization will not aim to remove any detail, but instead to make functionality applicable to a more generic range of items. This allows for a generic system that is applicable to far more types.

j) High coupling implies that the module knows an extensive amount about the inner working of other modules. This makes changes very difficult to coordinate between modules, thus making them brittle and fragile. For example, if Module A knows too much about Module B, changes to the internals of Module B may break functionality in Module A. This is why the high coupling case where a module is used by many other modules.

E Code for Shape.py

```
## @file Shape.py
#  @author Hamrish Saravanakumar
#  @brief Provides the Shape Interface Module
#  @date February 16 2021

from abc import ABC, abstractmethod

## @brief Shape contains 4 generic functions to define the properties of the shape.


class Shape(ABC):

    @abstractmethod
    ## @brief A generic method to determine the center of mass of the x coordinate.
    #  @return A real value representing the center of mass of the x coordinate.
    def cm_x(self):
        pass

    @abstractmethod
    ## @brief A generic method to determine the center of mass of the y coordinate.
    #  @return A real value representing the center of mass of the y coordinate.
    def cm_y(self):
        pass

    @abstractmethod
    ## @brief A generic method to determine the of mass.
    #  @return A real value representing the mass.
    def mass(self):
        pass

    @abstractmethod
    ## @brief A generic method to determine the moment of inertia.
    #  @return A real value representing the moment of inertia.
    def m_inert(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
#  @author Hamrish Saravanakumar
#  @brief Provides the CircleT ADT class that represents a circle
#  @date February 16 2021

from Shape import *

## @brief An abstract data type that represents a circle.
#  @details It is assumed that the arguments provided to the access programs will
#           be of the correct type.


class CircleT(Shape):
    ## @brief CircleT constructor.
    #  @details Initializes a CircleT object using a center of mass, radius and mass.
    #  @param Xs The x coordinate of the center of mass.
    #  @param Ys The y coordinate of the center of mass.
    #  @param Rs The radius of the circle.
    #  @param Ms The mass of the circle.
    #  @throws A value error is thrown if the radius or mass are less than or equal to zero.
    def __init__(self, Xs, Ys, Rs, Ms):
        if Rs <= 0 or Ms <= 0:
            raise ValueError
        self.x = Xs
        self.y = Ys
        self.r = Rs
        self.m = Ms

    ## @brief Obtain value of x coordinate of center of mass.
    #  @return Value of x coordinate of center of mass.
    def cm_x(self):
        return self.x

    ## @brief Obtain value of y coordinate of center of mass.
    #  @return Value of y coordinate of center of mass.
    def cm_y(self):
        return self.y

    ## @brief Obtain value of mass.
    #  @return Value of mass.
    def mass(self):
        return self.m

    ## @brief Obtain value of moment of inertia.
    #  @return Value of moment of intertia.
    def m_inert(self):
        return (self.m * (self.r**2)) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Hamrish Saravanakumar
#  @brief Provides the TriangleT ADT class that represents a triangle
#  @date February 16 2021

from Shape import *

## @brief An abstract data type that represents a triangle.
#  @details It is assumed that the arguments provided to the access programs will
#           be of the correct type.


class TriangleT(Shape):

    ## @brief TriangleT constructor.
    #  @details Initializes a TriangleT object using a center of mass,
    #           the side length, and mass.
    #  @param Xs The x coordinate of the center of mass.
    #  @param Ys The y coordinate of the center of mass.
    #  @param Ss The side length of the triangle.
    #  @param Ms The mass of the triangle.
    #  @throws A value error is thrown if the side length or mass are less
    #          than or equal to zero.
    def __init__(self, Xs, Ys, Ss, Ms):
        if Ss <= 0 or Ms <= 0:
            raise ValueError
        self.x = Xs
        self.y = Ys
        self.s = Ss
        self.m = Ms

    ## @brief Obtain value of x coordinate of center of mass.
    #  @return Value of x coordinate of center of mass.
    def cm_x(self):
        return self.x

    ## @brief Obtain value of y coordinate of center of mass.
    #  @return Value of y coordinate of center of mass.
    def cm_y(self):
        return self.y

    ## @brief Obtain value of mass.
    #  @return Value of mass.
    def mass(self):
        return self.m

    ## @brief Obtain value of moment of inertia.
    #  @return Value of moment of intertia.
    def m_inert(self):
        return (self.m * (self.s**2)) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Hamrish Saravanakumar
#  @brief Provides the BodyT ADT class that represents a sequence of masses in space
#  @date February 16 2021

from Shape import *

## @brief An abstract data type that represents a sequences of masses in space.
#  @details It is assumed that the arguments provided to the access programs will
#           be of the correct type.


class BodyT(Shape):
    ## @brief BodyT constructor.
    #  @details Initializes a BodyT object described by a center of mass and mass.
    #  @param Xs The x coordinate of the center of mass.
    #  @param Ys The y coordinate of the center of mass.
    #  @param Ms The mass of the circle.
    #  @throws A value error is thrown if the length of the three argument lists
    #          are not equal.
    #  @throws A value error is thrown if any value in the list of masses is less
    #          than or equal to zero.
    def __init__(self, Xs, Ys, Ms):
        if not (len(Xs) == len(Ys) == len(Ms)):
            raise ValueError
        for i in Ms:
            if i < 0:
                raise ValueError
        self.cmx = BodyT.__cm(Xs, Ms)
        self.cmy = BodyT.__cm(Ys, Ms)
        self.m = sum(Ms)
        value = self.cmx**2 + self.cmy**2
        self.moment = BodyT.__mmom(Xs, Ys, Ms) - (sum(Ms) * value)

    ## @brief Obtain value of x coordinate of center of mass.
    #  @return Value of x coordinate of center of mass.
    def cm_x(self):
        return self.cmx

    ## @brief Obtain value of y coordinate of center of mass.
    #  @return Value of y coordinate of center of mass.
    def cm_y(self):
        return self.cmy

    ## @brief Obtain value of mass.
    #  @return Value of mass.
    def mass(self):
        return self.m

    ## @brief Obtain value of moment of mass.
    #  @return Value of moment of mass.
    def m_inert(self):
        return self.moment

    @staticmethod
    def __cm(z, m):
        return sum([z[i] * m[i] for i in range(len(m))]) / sum(m)

    @staticmethod
    def __mmom(x, y, m):
        return sum([m[i] * (x[i]**2 + y[i]**2) for i in range(len(m))])
```

I Code for Scene.py

```
## @file Scene.py
#  @author Hamrish Saravanakumar
#  @brief Scene Module for a Shape moving through 2D Space
#  @date February 16 2021

import scipy.integrate

## @brief Class that implements a representation of a shape moving through 2D space.


class Scene:
    ## @brief Constructor method for class Scene
    #  @details Initialized by a Shape, an unbalanced force function in the x and y direction,
    #           and the initial velocity in the x and y direction.
    #  @param s_ A shape.
    #  @param Fx_ Unbalanced force function in the x direction.
    #  @param Fy_ Unbalanced force function in the y direction.
    #  @param Vx_ Initial velocity in the x durection.
    #  @param Vy_ Initial velocity in the y direction.
    def __init__(self, s_, fx_, fy_, vx_, vy_):
        self.s = s_
        self.fx = fx_
        self.fy = fy_
        self.vx = vx_
        self.vy = vy_

    ## @brief Getter method to obtain shape.
    #  @return The shape.
    def get_shape(self):
        return self.s

    ## @brief Getter method to obtain unbalanced force functions in both directions.
    #  @return Tuple consisting of unbalanced force functions in the x and y directions.
    def get_unbal_forces(self):
        return self.fx, self.fy

    ## @brief Getter method to obtain initial velocities in both directions.
    #  @return Tuples consisting of initial velocities in the x and y directions.
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief Setter method to modify the shape.
    def set_shape(self, s_):
        self.s = s_

    ## @brief Setter method to modify the unbalanced force functions in either direction.
    def set_unbal_forces(self, fx_, fy_):
        self.fx = fx_
        self.fy = fy_

    ## @brief Setter method to modify the initial velocities in either direction.
    def set_init_velo(self, vx_, vy_):
        self.vx = vx_
        self.vy = vy_

    ## @brief Function that determines the position and velocity
    #  @details Position and velocity over time are determined specifically based on the
    #           forces and the initial values for position and velocity.
    #  @param final_t The value for the final time
    #  @param nsteps The value for the number of steps that the time interval
    #         will be divided into.
    #  @return A simulation of the scene as a sequence of times, and a sequence of sequences
    #          for the position and velocity over time.
    def sim(self, final_t, nsteps):
        t = []
        for i in range(nsteps):
            t.append((i * final_t) / (nsteps - 1))
        sequence = [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy]
        ode = scipy.integrate.odeint(self.__ode, sequence, t)
        return t, ode

    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 Hamrish Saravanakumar
#  @brief Plot module used to plot functions.
#  @date February 16 2021

import matplotlib.pyplot as plt

## @brief Plot will plot three separate graphs of data points
#  @details The first graph will plot x versus t, the second graph will plot y versus t,
#           and the third graph will plot y versus x.
#  @param w Sequence of sequences that represent the x and y values to be plotted against.
#  @param t Sequence of times


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

K Code for test_driver.py

```
## @file test_driver.py
#  @author Hamrish Saravanakumar
#  @brief Test driver
#  @date February 16 2021
#  @details Presents test cases for BodyT.py, CircleT.py, Scene.py & TriangleT.py

from Shape import *
from CircleT import *
from TriangleT import *
from BodyT import *
from Scene import *

from pytest import *


class TestCircleT:

    def test_normalcm_x(self):
        circle1 = CircleT(3.0, 5.0, 6.0, 9.0)
        assert circle1.cm_x() == 3.0

    def test_normalcm_y(self):
        circle1 = CircleT(3.0, 5.0, 6.0, 9.0)
        assert circle1.cm_y() == 5.0

    def test_normalmass(self):
        circle1 = CircleT(3.0, 5.0, 6.0, 9.0)
        assert circle1.mass() == 9.0

    def test_normalm_inert(self):
        circle1 = CircleT(3.0, 5.0, 6.0, 9.0)
        assert circle1.m_inert() == 162.0

    def test_invalidRadius(self):
        with raises(ValueError):
            CircleT(2.0, 5.0, -2.0, 3.0)

    def test_invalidMass(self):
        with raises(ValueError):
            CircleT(2.0, 5.0, 2.0, 0.0)


class TestTriangleT:

    def test_normalcm_x(self):
        triangle1 = TriangleT(3.0, 5.0, 6.0, 9.0)
        assert triangle1.cm_x() == 3.0

    def test_normalcm_y(self):
        triangle1 = TriangleT(3.0, 5.0, 6.0, 9.0)
        assert triangle1.cm_y() == 5.0

    def test_normalmass(self):
        triangle1 = TriangleT(3.0, 5.0, 6.0, 9.0)
        assert triangle1.mass() == 9.0

    def test_normalm_inert(self):
        triangle1 = TriangleT(3.0, 5.0, 6.0, 9.0)
        assert triangle1.m_inert() == 27.0

    def test_invalidSideLength(self):
        with raises(ValueError):
            TriangleT(2.0, 5.0, -2.0, 3.0)

    def test_invalidMass(self):
        with raises(ValueError):
            TriangleT(2.0, 5.0, 2.0, 0.0)


class TestBodyT:

    def test_normalcm_x(self):
        body1 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
        assert body1.cm_x() == 0

    def test_normalcm_y(self):
```

```
        body1 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
        assert body1.cm_y() == 0

    def test_normalmass(self):
        body1 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
        assert body1.mass() == 40.0

    def test_normalm_inert(self):
        body1 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
        assert body1.m_inert() == 80.0

    def test_invalidSequenceLengths(self):
        with raises(ValueError):
            BodyT([7, 8, 8, 7, 5, 2], [7, 7, 8, 8], [10, 10, 10, 10])

    def test_invalidMass(self):
        with raises(ValueError):
            BodyT([7, 8, 8, 7], [7, 7, 8, 8], [-10, 10, 0, 10])


class TestScene:

    def test_normalget_shape(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        assert scene1.get_shape() == circle1

    def test_normalget_unbal_forces(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        assert scene1.get_unbal_forces() == (1, -9.81)

    def test_normalget_init_velo(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        assert scene1.get_init_velo() == (0, 0)

    def test_normalset_shape(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        triangle1 = TriangleT(3.0, 5.0, 6.0, 9.0)
        scene1.set_shape(triangle1)
        assert scene1.get_shape() == triangle1

    def test_normalset_unbal_forces(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
```

```
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        scene1.set_unbal_forces(fx(7), fy(7))
        assert scene1.get_unbal_forces() == (0, 9.81)

    def test_normalset_init_velo(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        new_vx, new_vy = 1, 1
        scene1.set_init_velo(new_vx, new_vy)
        assert scene1.get_init_velo() == (1, 1)

    def test_normalsim_t(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        t, w = scene1.sim(2, 5)
        calculated_t = [0.0, 0.5, 1.0, 1.5, 2.0]
        sub = [x1 - x2 for (x1, x2) in zip(t, calculated_t)]
        norm_t = max(sub)
        val = norm_t / max(calculated_t)
        assert val <= 0.000001

    def test_normalsim_w(self):
        circle1 = CircleT(1.0, 10.0, 0.5, 1.0)

        def fx(t):
            return 1 if t < 5 else 0

        def fy(t):
            return -9.81 if t < 5 else 9.81
        vx, vy = 0, 0
        scene1 = Scene(circle1, fx(3), fy(3), vx, vy)
        t, w = scene1.sim(2, 5)
        calculated_w = [[1, 10, 0, 0]
                        [1.625, 8.77375, 2.5, -4.905]
                        [3.5, 5.095, 5, -9.81]
                        [6.625, -1.03625, 7.5, -14.715]
                        [11, -9.62, 10, -19.62]]
        sub = [x1 - x2 for (x1, x2) in zip(w, calculated_w)]
        norm_w = max(sub)
        val = norm_w / max(calculated_w)
        assert val <= 0.000001

def remove_uppercase(string):
   remove_uppercase = ''.join(c for i in string if not i.isupper())
   return string

remove_uppercase("AoisHAIsa")
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Laura Southwood (MacId: southwol)
#  @brief implements an ADT for circles
#  @date February 16, 2021

from Shape import Shape


## @brief CircleT is used to represent a circle shape
## @details circles have an (x,y) coordinate, radius, and mass.
# This module assumes that arguments provided to access programs are correct type
class CircleT(Shape):

    ## @brief constructor for CircleT class
    # @details a circle's x,y coordinates refer to its center.
    # this method raises a ValueError if rs <= 0 or if ms <= 0
    # @param xs a real number (int or float) representing the x location
    # @param ys a real number (int or float) representing the y location
    # @param rs a real number (int or float) representing the radius
    # @param ms a real number (int or float) representing the mass
    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 a method for getting the center of mass x location
    # @return a real number (int or float), the x coordinate of the center of mass
    def cm_x(self):
        return self.x

    ## @brief a method for getting the center of mass y location
    # @return a real number (int or float), the y coordinate of the center of mass
    def cm_y(self):
        return self.y

    ## @brief a method for getting the mass of the circle
    # @return a real number (int or float), the mass of the circle
    def mass(self):
        return self.m

    ## @brief a method for getting moment of inertia of the circle
    # @return a real number (int or float), 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 Laura Southwood (MacId: southwol)
#  @brief implements an ADT for triangles
#  @date February 16, 2021

from Shape import Shape


## @brief TriangleT is used to represent a triangle shape
## @details triangles have an (x,y) coordinate, side length (equilateral), and mass
# This module assumes that arguments provided to access programs are correct type
class TriangleT(Shape):

    ## @brief constructor for TriangleT class
    # @details a triangle's x,y coordinates refer to its center.
    # this method raises a ValueError if ss <= 0 or if ms <= 0
    # @param xs a real number (int or float) representing the x location
    # @param ys a real number (int or float) representing the y location
    # @param ss a real number (int or float) representing the side length (all 3 same)
    # @param ms a real number (int or float) representing the mass
    def __init__(self, xs, ys, ss, ms):
        if not(ss > 0 and ms > 0):
            raise ValueError
        self.x = xs
        self.y = ys
        self.s = ss
        self.m = ms

    ## @brief a method for getting the center of mass x location
    # @return a real number (int or float), the x coordinate of the center of mass
    def cm_x(self):
        return self.x

    ## @brief a method for getting the center of mass y location
    # @return a real number (int or float), the y coordinate of the center of mass
    def cm_y(self):
        return self.y

    ## @brief a method for getting the mass of the triangle
    # @return a real number (int or float), the mass of the triangle
    def mass(self):
        return self.m

    ## @brief a method for getting moment of inertia of the triangle
    # @return a real number (int or float), 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 Laura Southwood (MacId: southwol)
#  @brief implements an ADT for other shapes (bodies)
#  @date February 16, 2021

from Shape import Shape


## @brief BodyT is used to represent a shape made up of point masses
## @details it will have a calculated (x,y) coordinate for center of mass,
# as well as an overall mass and a moment of inertia.
# This module assumes that arguments provided to access programs are correct type
class BodyT(Shape):

    ## @brief constructor for BodyT class
    # @details State variables are calcualted from sequences of point masses.
    # This method raises a ValueError if the sequences differ in length,
    # or if any mass values in the sequence are <= 0. This ensures the state invariant
    # @param xs a sequence of real numbers (int or float) representing x locations
    # @param ys a sequence of real numbers (int or float) representing y locations
    # @param ms a sequence of real numbers (int or float) representing masses of points
    def __init__(self, xs, ys, ms):
        if not(len(xs) == len(ys) and len(ys) == len(ms)):
            raise ValueError
        for miu in ms:
            if not miu > 0:
                raise ValueError
        self.cmx = self.cm(xs, ms)
        self.cmy = self.cm(ys, ms)
        self.m = self.sum(ms)
        mmom = self.mmom(xs, ys, ms)
        self.moment = mmom - self.sum(ms) * (self.cm(xs, ms)**2 + self.cm(ys, ms)**2)
        if not self.moment >= 0:
            raise ValueError

    ## @brief a method for getting the center of mass x location
    # @return a real number (int or float), the x coordinate of the center of mass
    def cm_x(self):
        return self.cmx

    ## @brief a method for getting the center of mass y location
    # @return a real number (int or float), the y coordinate of the center of mass
    def cm_y(self):
        return self.cmy

    ## @brief a method for getting the mass of the body
    # @return a real number (int or float), the mass of the body
    def mass(self):
        return self.m

    ## @brief a method for getting moment of inertia of the body
    # @return a real number (int or float), the moment of inertia of the body
    def m_inert(self):
        return self.moment

    ## @brief a method for calculating the sum of all elements in a given list
    # @param ms a sequence of real numbers (int or float) to be added together
    # @return a real number (int or float), the sum of all elements in list
    def sum(self, ms):
        total = 0
        for miu in ms:
            total += miu
        return total

    ## @brief a method for calculating the center of mass in 1 dimension
    # @param z a sequence of real numbers (int or float) of locations
    # @param m a sequence of real numbers (int or float) of masses
    # @return a real number (int or float), the center of mass in the z dimension
    def cm(self, z, m):
        total = 0
        for i in range(len(m)):
            total += z[i] * m[i]
        return total / self.sum(m)

    ## @brief a method for calculating the sum of all point mass moment of inertia
    # @param x a sequence of real numbers (int or float) of x locations
```

```
# @param y a sequence of real numbers (int or float) of y locations
# @param m a sequence of real numbers (int or float) of masses
# @return a real number (int or float), the sum of all point mass moments of inertia
def mmom(self, x, y, m):
    total = 0
    for i in range(len(m)):
        total += m[i] * (x[i]**2 + y[i]**2)
    return total
```

O Code for Partner's Scene.py

```
## @file Scene.py
#  @author Laura Southwood (MacId: southwol)
#  @brief implements an ADT for a scene of a shape
#  @date February 16, 2021
#  @details only models 1 shape at a time

from Shape import Shape
from scipy.integrate import odeint


## @brief creates a scene with 1 Shape, initial velocity, and unbalanced forces
# @details velocity and forces are in their x and y components
class Scene:

    ## @brief constructor for Scene class
    # @param s a Shape
    # @param Fx a function (R -> R), the unbalanced force in the x direction
    # @param Fy a function (R -> R), the unbalanced force in the y direction
    # @param vx a real number (int or float), the initial velocity in the x direction
    # @param vy a real number (int or float), the initial velocity in the y direction
    def __init__(self, s_prime, Fx_prime, Fy_prime, vx_prime, vy_prime):
        self.s = s_prime
        self.Fx = Fx_prime
        self.Fy = Fy_prime
        self.vx = vx_prime
        self.vy = vy_prime

    ## @brief a method for getting the scene's shape
    # @return a Shape, the one in the scene
    def get_shape(self):
        return self.s

    ## @brief a method for getting the unbalanced forces acting on the shape
    # @return two functions (R -> R), the first one acting in the x direction,
    # the second acting in the y direction
    def get_unbal_forces(self):
        return self.Fx, self.Fy

    ## @brief a method for getting the initial velocity of the shape
    # @return two real numbers (int or float), the first one being inital velocity
    # in the x direction, the second being initial velocity in the y direction.
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief a method for setting the scene's shape
    # @param s_prime a Shape, a new one to be replaced in the scene
    def set_shape(self, s_prime):
        self.s = s_prime

    ## @brief a method for setting the unbalanced forces acting on the shape
    # @param Fx_prime a function (R -> R) to replace the x direction force
    # @param Fy_prime a function (R -> R) to replace the y direction force
    def set_unbal_forces(self, Fx_prime, Fy_prime):
        self.Fx = Fx_prime
        self.Fy = Fy_prime

    ## @brief a method for setting the initial velocity of the shape
    # @param vx_prime a real number (int or float) to replace the x direction initial velocity
    # @param vy_prime a real number (int or float) to replace the y direction initial velocity
    def set_init_velo(self, vx_prime, vy_prime):
        self.vx = vx_prime
        self.vy = vy_prime

    ## @brief simulates the scene (calculates the ode)
    # @param t_final a real number (int or float)
    # @param nsteps a natural number (int)
    # @return a sequence of real numbers, time values increasing incrementally up to t_final.
    # Also returns a sequence of sequences ([rx, ry, vx, vy] real number values over time)
    def sim(self, t_final, nsteps):
        t = []
        for i in range(nsteps):
            t.append((i * t_final) / (nsteps - 1))

        ## @brief uses x and y position and velocity (and time),
        # to get x and y velocity and acceleration
        # @param w a sequence of 4 elements [rx, ry, vx, vy]
```

```
# @param t a real number (int or float), represents time
# @return a sequence of 4 elements [vx, vy, deriv(vx/t), deriv(vy/t)]
def ode(w, t):
    return [w[2], w[3], self.Fx(t) / self.s.mass(), self.Fy(t) / self.s.mass()]

return t, odeint(ode, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
```