Assignment 2 Solution

Name: Mingzhe Wang MacID: wangm235

February 25, 2021

This report discusses the testing phase for `CircleT`, `TriangleT`, `BodyT`, and `SceneT`. 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 Assumptions and Exceptions

For assignment 2, because we used a formal MIS to specify the assumptions and exceptions, the number of them is much smaller than assignment 1. Specifically, they are:

Assumptions

- For all the modules, the arguments provided to the access programs will be of the correct type.

Exceptions

- For `__init__` in `CircleT`, `ValueError` is raised if radius or mass is not positive.
- For `__init__` in `BodyT`, `ValueError` is raised if the three sequences do not have the same length.
- For `__init__` in `BodyT`, `ValueError` is raised if not all the masses of points are positive.
- For `__init__` in `TriangleT`, `ValueError` is raised if side length or mass is not positive.

2 Test Cases and Rationale

2.1 Test Cases

For assignment 2, there are totally 27 test cases performed, in each test cases there could be multiple assertions that can test the performance of the program to different inputs. However, it should be noticed that each test case is designed to test one branch/exception of a given method. Through this, it is easy to determine which statement is not reliable in a single method.

2.2 Rationale

Part of rationales of test cases selection are consisted with Assignment 1, they are:

- The test cases should cover every branches of the tested method. If a method or constructor could raise certain exceptions, then these exceptions could be treated as branches, which means the test cases should also cover them. For example, there should be at least 3 test cases for testing `__init__` in `BodyT`. (two for the exception branches and one for the normal case.)
- For float numbers comparison, normally we only need to perform approximate equality under certain tolerated difference. Because we are using pytest for Assignment 2, they are dealt with function `approx()`.
- Test cases that can be verified in another way easily should be chosen. Because they can be calculated manually quickly, however they may not be that easy for computers to calculate. Hence, they can help us verify the reliability of our program fast.

  For example, test cases like the following were chosen:

  ```
  BodyT([-1.0, 1.0, 0.0], [0.0, 0.0, math.sqrt(3)],
  [2.8, 2.8, 2.8])
  ```

  Another example is that for testing `sim` in `Scene`, we choose to simulate common cases likes falling motion and projectile motion.
- Test cases that use different mathematical types of numbers, such as negative numbers, rational numbers, irrational numbers, numbers expressed in scientific notation etc. should be chose. Because they can test the performance of program for different type of input.

- Test cases that use extremely large or small numbers as input should be chosen. For example, test cases like `TriangleT(7.0313, 302, 0.00002, -0.000001)` and `CircleT(-2.3e10, 2.3395934, math.sqrt(2), 5.3e10)` were chosen because they help test the reliability of the program in edge cases or boundary cases.

In addition, as what we have learned for the unit test part, there are many other goals we are trying to pursue in this assignment, which are:

- Test all the requirements in each function. As we are using a formal MIS for this assignment, when designing the test cases, we should go through all the requirements in the MIS, checking both sytax and semantic for any violation.
- Cover edge cases that cause unintended consequences, especially for empty lists or 0 inputs. For example, `CircleT(0.0, 0.0, 0.0, 0.0)`, `TriangleT(0.0, 0.0, 0.0, 0.0)`, and `BodyT([0.0], [0.0], [0.0, 0.0])` was chosen to test if the behavior of our program in this edge case is reasonable.
- Try to have an acceptable amount of code coverage. In my implementation, the cover of my testing cases is as the following, which I believe is an acceptable amount.

```
Name                 Stmts   Miss  Cover
----------------------------------------
src/BodyT.py            34      0   100%
src/CircleT.py          17      0   100%
src/Plot.py             16     14    12%
src/Scene.py            30      0   100%
src/Shape.py            10      4    60%
src/TriangleT.py        17      0   100%
src/test_All.py        205      2    99%
src/test_expt.py        32      0   100%
----------------------------------------
TOTAL                  361     20    94%
```

- For function coverage, statement coverage, branch coverage, and conditional coverage, most of these principles are fully included in the previous rationale. For example, for testing `__init__` in `BodyT`, we have `test_init_exception_len_not_equal`, `test_init_exception_m_not_pos`, and a normal test case in the `setup_method`.

Finally, for testing some complex method such as `sim`, the testing process is:

```
    def test_sim_scene1(self):
        t, wsol = self.scene1.sim(10, 101)
        t_counter = 0
        for t_s, wsol_s in zip(t, wsol):
            wsol_true = [10.0, 10.0 - 0.5 * 9.81 * t_counter**2,
                0.0, - 9.81 * t_counter]
            assert t_s == approx(t_counter)
            assert TestScene.__compare_two_seqs(wsol_s,
                wsol_true, 0.001)
            t_counter += 0.1
```

First, generate two sequences, which are the expected solution and the program's output. (Each of these sequences consists of time t and corresponding variable values). Then for each time t, compare two sequences of corresponding variable values by the relative error formula that mentioned in the assignment instruction.

By designing the test like this, we can compare all the output values in an iteration to avoid manually or randomly selecting some time t and its corresponding variable values, which could not guarantee the accuracy of the tests.

3 Results of Testing the Original Program

```
pytest --cov src
============================= test session starts ==============================
platform linux -- Python 3.6.9, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /nfs/u50/wangm235/wangm235/A2, inifile:
plugins: cov-2.5.1
collected 27 items

src/test_All.py ...........................
[100%]

----------- coverage: platform linux, python 3.6.9-final-0 -----------
Name                Stmts   Miss  Cover
---------------------------------------
src/BodyT.py           34      0   100%
src/CircleT.py         17      0   100%
src/Plot.py            16     16     0%
src/Scene.py           30      0   100%
```

```
src/Shape.py              10        4     60%
src/TriangleT.py          17        0    100%
src/test_All.py          204        2     99%
src/test_expt.py          32        0    100%
------------------------------------------------
TOTAL                    360       22     94%


============================ 27 passed in 0.92 seconds ============================
```

When running the tests on my code, there is no exception raised. My code passes all the 27 tests.

4 Results of Testing Partner's Code

```
pytest --cov partner
================================ test session starts ================================
platform linux -- Python 3.6.9, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /nfs/u50/wangm235/wangm235/A2, inifile:
plugins: cov-2.5.1
collected 27 items

partner/test_All.py ...........................
[100%]

-------------- coverage: platform linux, python 3.6.9-final-0 --------------
Name                         Stmts     Miss   Cover
----------------------------------------------------
partner/BodyT.py                35        0    100%
partner/CircleT.py              17        0    100%
partner/Scene.py                27        0    100%
partner/Shape.py                10        4     60%
partner/TriangleT.py            17        0    100%
partner/test_All.py            204        2     99%
partner/test_expt.py            32        0    100%
----------------------------------------------------
TOTAL                          342        6     98%
```

```
========================== 27 passed in 1.10 seconds ==========================
```

When running the tests on partner's code, there is no exception raised. Partner's code passes all the 27 tests.

This time, both my and my partner's code passes all the tests, which I think is largely owe to the formal MIS specification. This kind of specification increases the performance of the programs by providing the programmer with lots of useful information. For example, in the syntax field, it specifies all the types and exceptions that the implementation should have; while in the semantics field, it even lists all the concrete details of the operations by using formal mathematical notations, etc. All of these designs act like an common standard to decrease the divergences when someone is trying to implement it. Hence, it guarantees an good-performance implementation.

In addition, I would like to provide some suggestions for my partner's code:

- Currently, all the functions do not have doxygen comments of the type `@details`. He(she) needs to add this kind of comment in the future, because it provides the client of our programmers much useful information especially when they are dealing with complex problems.
- `__sum` in `BodyT` could be dropped, because there is already an original function `sum` in Python that can perform this task.

Comparing assignment 2 with assignment 1, this time my partner's code passes all the tests rather than failing 3 cases last time. What I leaned from this assignment is that in the future when dealing with large-scale programming project that needs programmers' collaboration, an MIS is almost a necessary tool to ensure the quality of the project.

5 Critique of Given Design Specification

5.1 What I like

In general, I like this formal MIS specification given for assignment 2.

- The input, output, and desired behavior is specified clearly. The mathematical notations and the formal language it used decrease possible ambiguity when programmer wants to implement it.

- Its content is consistent, if we know how to interpret some previous parts to our programming language, then it's not hard for us to interpret any parts that come after them.
- Module design in this specification is consistent. For example, each of the `CircleT`, `TriangleT`, or `BodyT` as a whole object follows the interface `Shape`, sharing the same behaviors.
- Module design in this specification is high cohesion and low coupling. For example, for `CircleT`, `TriangleT`, and `BodyT`, it only depends on an shared interface `Shape` and is independent of each other. And for each of these modules, its content is high cohesion – it provides the basic physic information of that object, such as the position and the mass etc.
- Generality is another thing I like this MIS. At first, the specification only tries to implement a circle and triangle object. However, it later defines an `BodyT` object that consists of finite mass points. That kind of generality provides us the possibility to save all the scenes we may see in the future. (Most object can be defined as a set of mass points in physics.

5.2 What areas need improvement

- The consistency of `CircleT`, `TriangleT`, and `BodyT` should be improved. Because if we start from the module name, we will think that the `BodyT` is an general version of `CircleT` and `TriangleT`. However, `CircleT` or `TriangleT` actually consists of all the mass points in that shape, while `BodyT` consists of finite mass points that we define. For example, the `m_inert` formula for triangles isn't based on point masses at the vertices of the triangle, but rather a thin piece of material with the mass uniformly distributed across the entire triangle. This kind of inconsistency could bring us some kinds of confusion, thus it needs to be improved. My suggestion is to define another module that consists of all the points surrounded by lines connecting all the vertices to make this solution more general.
- The local functions in `Scene` module can be extremely hard to interpret. I think a large part of the confusion comes from the `ode` function's semantic:

  $\text{ode} : (\text{seq } [4] \text{ of } \mathbb{R}) \times \mathbb{R} \rightarrow \text{seq } [4] \text{ of } \mathbb{R}$
  $\text{ode}(w, t) \equiv [w[2], w[3], F_x(t)/s.\text{mass}(), F_y(t)/s.\text{mass}()]$

  In the second line, the specification implicitly uses a variable w of the type seq [4] of $\mathbb{R}$

where it does not tell the programmer what more detailed meaning about w. I think some additional comments should be added to this part to avoid creating unnecessary confusion for programmers.

5.3 Change proposal

The thing I want to propose to change is the `TriangleT` module. Although it is called TriangleT, it is actually an object of shape equilateral triangle. We do not need to narrow our design to such a range, in fact, the `TriangleT` should be type of any triangles, and its constructor can be overloading such that we can use the combination of angles and side lengths to form our triangles. That could make the implementation more complex, but that also makes this module more general to solve many future programs that are about triangle objects.

5.4 Does the interface provide checks?

Yes. In the `syntax` field, the specification list the type of all the input and output and the exception behavior for each access program; In the `semantic` field, the specification provide `state invariant` that can be useful for proving program correctness and many other detailed statements; also, in other fields like `Uses`, `template`, `exported`, it also provide programmer a reference to check which module they want to import/export and which function should they consider to be having been implemented. I think all of them provide some kinds of checks that could help programmers to avoid generating an exception. To do this, we just need to go through the MIS again searching for any violation after we have implemented the program.

6 Answers

a) I think that depends on the state variables that getters and setters interact with. If the state variables are simply passed values from the constructor without any operation, logic, or type check, then they may not require testing. However, when the the state variables are transited by doing some operations on the arguments, then we need to design tests for these kind of getters or setters. For example, the state variable `m_inert` in `BodyT` is documented as $moment := \text{mmom}(x_s, y_s, m_s) - \text{sum}(m_s)(\text{cm}(x_s, m_s)^2 + \text{cm}(y_s, m_s)^2)$ in MIS. To check the reliability of this transition, we need to design a test for the getter `m_inert()`. Another cases we need to test getters and setters should be in some other programming language, unlike Python, that highly depends on the type of parameter, if we declaim the type of state variables in MIS, then we need tests to make sure that logic is being tested.

b) I would like to set a test to check if the returned function arguments of getters have the type of $\mathbb{R} \rightarrow \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$ and if the given function arguments of setters have the type of $\mathbb{R}, \mathbb{R}$. In addition, we may make a plot to manually test if this force function is as what we expect.

c) First, using `matplotlib` generate two figures, one based on the expected data, one bases on the data output by the program. Second, use `plt.savefig()` command to save these two figures as two .png files. Third, define a function that takes two images as parameter, converts them to two `pandas` objects of type float, and use some kind of equations to calculate a value representing their relative error. If that error is smaller than our acceptable value, then the test passes. Otherwise, we will say the test fails.

d) abs : $\mathbb{R} \rightarrow \mathbb{R}$
abs$(x) \equiv (x \geq 0 \Rightarrow x | x < 0 \Rightarrow -x)$

max : seq of $\mathbb{R} \rightarrow \mathbb{R}$
max$(x_s) \equiv m$, such that $(\forall i : \mathbb{N} | i \in [0..|x_s| - 1] : m \geq (x_s)_i)$

(Note: we assume $|x_{cal}| = |x_{true}|$)
close_enough : seq of $\mathbb{R} \times$ seq of $\mathbb{R} \rightarrow \mathbb{B}$
close_enough$(x_{cal}, x_{true}) \equiv$

$$\frac{\max(i : \mathbb{N} | i \in [0..|x_{cal}| - 1] : \text{abs}(x_{cal} - x_{true}))}{\max(i : \mathbb{N} | i \in [0..|x_{cal}| - 1] : \text{abs}(x_{true}))} < \epsilon$$

, where ϵ is a small number that we define.

e) No, there should not be exceptions for negative coordinates. Because in physics, we can choose any object as our reference frame, and even negative coordinates are meaningful. For example, to trace a robot's motion in a given room, we can select the middle of the room as the origin of the coordinate system, then even the negative coordinates records some places where the robot is. Therefore, there is no need to throw exceptions for negative coordinates.

f) Because only the routine `TriangleT` actually has the transition operation for state variables s and m, we only need to prove that $s > 0 \wedge m > 0$ holds true after assignment statements.

$$\begin{aligned}
& s > 0 \wedge m > 0 \\
\Rightarrow\ & [s := s_s] \langle \mathit{Assignment} \rangle \\
& s_s > 0 \wedge m > 0 \\
\Rightarrow\ & [m := m_s] \langle \mathit{Assignment} \rangle \\
& s_s > 0 \wedge m_s > 0 \\
\equiv\ & \langle \text{Left identity of implication} \rangle \\
& \text{True} \Rightarrow s_s > 0 \wedge m_s > 0 \\
\equiv\ & \langle \text{Contrapositive; Def. of False} \rangle \\
& \neg(s_s > 0 \wedge m_s > 0) \Rightarrow \text{False} - \text{This is exception.}
\end{aligned}$$

Therefore, we prove that state variable will never be violated.

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

h)
```
from functools import reduce

def remove_upper(s):
    res = [c for c in s if c.islower()]
    s2 = reduce(lambda x, y: x + y, res, '')
    return s2
```

i) According to Fundamentals of Software Engineering, abstraction is naturally combined with the generality principle. For example, producing intermediate code for an abstract machine, rather than producing object code directly for a concrete one, allows us to build a general compiler that can be adapted, with minor modifications, to the production of code for different machines, thus enhancing the reusability quality.

j) The scenario that a module is used by many other modules would be better. Because in this scenario, we only need to be careful about modification in that single module. However, if a module uses many other modules, then we need to be careful about modification in all other modules. If any of those modules are modified, there is a possibility that the single module depending on them could not work properly. That can make the maintenance or improvement of the project unachievable.

G Code for Shape.py

```
## @file Shape.py
#  @author Mingzhe Wang (wangm235)
#  @brief A shape inerface.
#  @date 2021 Feb 16

from abc import ABC, abstractmethod


## @brief An interface specifying which methods should a Shape type has.
#  @details A Shape should have cm_x, cm_y, mass, and m_inert getters.
class Shape(ABC):

    ## @brief The implementation should get the x-coordinate of mass center.
    #  @details This method should return a real number.
    @abstractmethod
    def cm_x(self):
        pass

    ## @brief The implementation should get the y-coordinate of mass center.
    #  @details This method should return a real number.
    @abstractmethod
    def cm_y(self):
        pass

    ## @brief The implementation should get the mass.
    #  @details This method should return a real number.
    @abstractmethod
    def mass(self):
        pass

    ## @brief The implementation should get the moment of inertia.
    #  @details This method should return a real number.
    @abstractmethod
    def m_inert(self):
        pass
```

H Code for CircleT.py

```
## @file CircleT.py
#  @author Mingzhe Wang (wangm235)
#  @brief A circle ADT implementing the shape inerface.
#  @date 2021 Feb 16

from Shape import Shape


## @brief A class representing a circle.
#  @details This circle should implement all the methods specified in Shape interface.<br/>
#           Assumptions: The arguments provided to the access programs will be of the correct
#           type.
class CircleT(Shape):

    ## @brief Constructor for CircleT.
    #  @details Constructor taking in x-coordinate, y-coordinate, radius, and mass for a
    #           circle object.
    #  @param x A real number representing the x-coordinate of the circle center.
    #  @param y A real number representing the y-coordinate of the circle center.
    #  @param r A real number representing the radius of the circle object.
    #  @param m A real number representing the mass of the circle object.
    #  @throws ValueError If radius or mass is not positive.
    def __init__(self, x, y, r, m):
        if not (r > 0.0 and m > 0.0):
            raise ValueError
        ## X-coordinate of the circle center.
        self.x = x
        ## Y-coordinate of the circle center.
        self.y = y
        ## Radius of the circle object.
        self.r = r
        ## Mass of the circle object.
        self.m = m

    ## @brief Returns the x-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the x-coordinate of the centre of mass.
    def cm_x(self):
        return self.x

    ## @brief Returns the y-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the y-coordinate of the centre of mass.
    def cm_y(self):
        return self.y

    ## @brief Returns the mass of the circle object.
    #  @return A real number representing the mass of the circle object.
    def mass(self):
        return self.m

    ## @brief Returns the the moment of inertia of the circle object.
    #  @details For a circle object, the moment of inertia is equal to (m * r**2) / 2.
    #  @return A real number representing the moment of inertia of the circle object.
    def m_inert(self):
        return self.m * self.r**2 / 2.0
```

I Code for TriangleT.py

```
## @file TriangleT.py
#  @author Mingzhe Wang (wangm235)
#  @brief An equilateral triangle ADT implementing the shape inerface.
#  @date 2021 Feb 16

from Shape import Shape


## @brief A class representing an equilateral triangle.
#  @details This equilateral triangle should implement all the methods specified in Shape int
#           erface.<br/> Assumptions: The arguments provided to the access programs will be of
#            the correct type.
class TriangleT(Shape):

    ## @brief Constructor for TriangleT.
    #  @details Constructor taking in x-coordinate, y-coordinate, side length, and mass for an
    #  equilateral triangle object.
    #  @param x A real number representing the x-coordinate of the triangle centre.
    #  @param y A real number representing the y-coordinate of the triangle centre.
    #  @param s A real number representing the side length of the triangle.
    #  @param m A real number representing the mass of the triangle object.
    #  @throws ValueError If side length or mass is not positive.
    def __init__(self, x, y, s, m):
        if not (s > 0 and m > 0):
            raise ValueError
        ## X-coordinate of the triangle centre.
        self.x = x
        ## Y-coordinate of the triangle centre.
        self.y = y
        ## Side length of the triangle.
        self.s = s
        ## Mass of the triangle object.
        self.m = m

    ## @brief Returns the x-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the x-coordinate of the centre of mass.
    def cm_x(self):
        return self.x

    ## @brief Returns the y-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the y-coordinate of the centre of mass.
    def cm_y(self):
        return self.y

    ## @brief Returns the mass of the triangle object.
    #  @return A real number representing the mass of the triangle object.
    def mass(self):
        return self.m

    ## @brief Returns the the moment of inertia of the triangle object.
    #  @details For a triangle object, the moment of inertia is equal to (m * s**2)/ 12.
    #  @return A real number representing the moment of inertia of the triangle object.
    def m_inert(self):
        return self.m * self.s**2 / 12.0
```

J Code for BodyT.py

```
## @file BodyT.py
#  @author Mingzhe Wang (wangm235)
#  @brief A body ADT implementing shape interface.
#  @date 2021 Feb 16

from Shape import Shape


## @brief A class representing an object consisting of finite points.
#  @details This object should implement all the methods specified in Shape interface.<br/>
#           Assumptions: The arguments provided to the access programs will be of the correct
#           type.
class BodyT(Shape):

    ## @brief Constructor for BodyT.
    #  @details Constructor taking in a sequence of x-coordinate, a sequence of y-coordinate,
    #           and a sequence of masses for an objetc consisting of finite mass points.
    #  @param xs A seq of real number representing the x-coordinates of the object's mass
    #            points.
    #  @param ys A seq of real number representing the y-coordinates of the object's mass
    #            points.
    #  @param ms A seq of real number representing the masses of the object's mass points.
    #  @throws ValueError If the three seqs do not have the same length.
    #  @throws ValueError If not all the masses of points are positive.
    def __init__(self, xs, ys, ms):
        if not (len(xs) == len(ys) == len(ms)):
            raise ValueError
        elif not self.__is_m_pos(ms):
            raise ValueError
        else:
            ## X-coordinate of the mass centre.
            self.cmx = self.__cm(xs, ms)
            ## Y-coordinate of the mass centre.
            self.cmy = self.__cm(ys, ms)
            ## Total mass of the object.
            self.m = sum(ms)
            ## Moment of inertia of the object.
            self.moment = self.__mmom(xs, ys, ms) - sum(ms) \
                * (self.__cm(xs, ms)**2 + self.__cm(ys, ms)**2)

    ## @brief Returns the x-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the x-coordinate of the centre of mass.
    def cm_x(self):
        return self.cmx

    ## @brief Returns the y-coordinate of the centre of mass.
    #  @details The centre of mass is calculated under the physic law.
    #  @return A real number representing the y-coordinate of the centre of mass.
    def cm_y(self):
        return self.cmy

    ## @brief Returns the mass of the object.
    #  @details The mass of the object is equal to the sum of masses of all points.
    #  @return A real number representing the mass of the object.
    def mass(self):
        return self.m

    ## @brief Returns the the moment of inertia of the object.
    #  @details For an object consisting of finite mass points, the moment of inertia is calc
    #           ulated using discrete quantifiers.
    #  @return A real number representing the moment of inertia of the triangle object.
    def m_inert(self):
        return self.moment

    # private method for check the positivity of a seq.
    def __is_m_pos(self, ms):
        for m in ms:
            if m <= 0:
                return False
        return True

    # private method for calculating the mass centre in one coordinate.
    def __cm(self, zs, ms):
        numerator = 0
        for i in range(len(ms)):
            numerator += zs[i] * ms[i]
        return numerator / sum(ms)

    # private method used for calculating mass moment.
    def __mmom(self, xs, ys, ms):
        quant = 0
        for i in range(len(ms)):
            quant += ms[i] * (xs[i]**2 + ys[i]**2)
        return quant
```

K Code for Scene.py

```
## @file Scene.py
#  @author Mingzhe Wang (wangm235)
#  @brief A scene ADT that simulates the movement of an obeject of type Shape.
#  @date 2021 Feb 16
#  @details The object can be a circle, an equilateral triangle, or an general object consist
#           ing of finite mass points.s

from scipy.integrate import odeint


## @brief A class representing a scene for simulation.
#  @details A scene contains the detailed information of the movement of an obejct, such as sh
#           ape, unbalanced force, init_velocity, etc.
class Scene:

    ## @brief Constructor for Scene.
    #  @details Constructor taking in a shape type, two functions representing the unbalanced
    #           forces with time t in x and y coordinate, an horizontal initial velocity, and
    #           an vertial initial velocity to simulate an scene.
    #  @param s A Shape object representing the type of the object shape.
    #  @param fx A function of type real number -> real number representing the unbalanced for
    #            ces with time t in x-coordinate.
    #  @param fy A function of type real number -> real number representing the unbalanced for
    #            ces with time t in y-coordinate.
    #  @param vx A real number representing the horizontal initial velocity.
    #  @param vy A real number representing the vertial initial velocity.
    def __init__(self, s, fx, fy, vx, vy):
        ## Shape
        self.s = s
        ## unbalanced force function in x dir
        self.fx = fx
        ## unbalanced force function in y dir
        self.fy = fy
        ## initial velocity in x dir
        self.vx = vx
        ## initial velocity in y dir
        self.vy = vy

    ## @brief Returns the Shape object of the scene.
    #  @details The shape object records the information of the object.
    #  @return A Shape representing the object that is focused in this simulation.
    def get_shape(self):
        return self.s

    ## @brief Returns the tuple of the functions of unbalanced forces.
    #  @details The first element is the function of unbalanced force in x-coordinate.<br/>
    #           The second element is the function of unbalanced force in y-coordinate.
    #  @return A tupe of two functions of real number -> real number representing functions of
    #          unbalanced forces.
    def get_unbal_forces(self):
        return (self.fx, self.fy)

    ## @brief Returns the tuple of initial velocities.
    #  @details The first element is the initial velocities in x-coordinate.<br/>
    #           The second element is the initial velocities in y-coordinate.
    #  @return A tupe of two real number representing the unbalanced forces.
    def get_init_velo(self):
        return (self.vx, self.vy)

    ## @brief Set a new object for this scene.
    #  @param s A Shape representing the object that is focused in this simulation.
    def set_shape(self, s):
        self.s = s

    ## @brief Set a new pair of functions of unbalanced forces for this scene.
    #  @param fx A function of real number -> real number representing unbalanced force in x-c
    #            oordinate.
    #  @param fy A function of real number -> real number representing unbalanced force in y-c
    #            oordinate.
    def set_unbal_forces(self, fx, fy):
        self.fx = fx
        self.fy = fy

    ## @brief Set a new pair of initial velocities for this scene.
    #  @param fx A real number representing the initial velocity in x-coordinate.
    #  @param fy A real number representing the initial velocity in y-coordinate.
```

```
def set_init_velo(self, vx, vy):
    self.vx = vx
    self.vy = vy

## @bried Begin a simulation in an time interval.
#  @details The time of simulation will start from 0 to t_fin in a n_step interval.
#  @param t_fin A real number representing the final time.
#  @param  n_step A natural number representing the number of time intervals.
#  @return A tuple (seq of real numbers, seq of 4-length-seq of real numbers). First elem-
#          ent represents seq of time; second element represents [x, y, vx, vy] in diffe-
#          rent time t.
def sim(self, t_fin, n_step):
    t = []
    for i in range(n_step):
        t.append((i * t_fin) / (n_step - 1))
    sec_ag = odeint(self.__ode, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
    return (t, sec_ag)

# A local fucntion used for solving the ode.
def __ode(self, w, t):
    return [w[2], w[3], self.fx(t) / self.s.mass(), self.fy(t) / self.s.mass()]
```

L Code for Plot.py

```
## @file Plot.py
#  @author Mingzhe Wang (wangm235)
#  @brief A plot script.
#  @date 2021 Feb 16
#  @details Plot used to output the simulation result in three figures.

import matplotlib.pyplot as plt


## @brief This plot method will plot the t vs x, t vs y, x vs y figures in a vertical dire
#         ction as showed in the assignment requirement.
#  @details Implementation of Plot.py is facilitated by the matplotlib package.
#  @param w A seq of 4-length-seq of real number representing [x, y, vx, vy] in different time
#           t.
#  @param t A seq of real number representing the dicrete time.
#  @throws ValueError If the two input seqs has different lengths.
def plot(w, t):
    if not (len(w) == len(t)):
        raise ValueError
    x = [w_ele[0] for w_ele in w]
    y = [w_ele[1] for w_ele in w]

    fig, axs = plt.subplots(nrows=3, ncols=1)
    fig.suptitle('Motion Simulation')
    axs[0].plot(t, x)
    axs[1].plot(t, y)
    axs[2].plot(x, y)
    axs[0].set(ylabel='x (m)')
    axs[1].set(ylabel='y (m)')
    axs[2].set(ylabel='y (m)')
    axs[2].set(xlabel='x (m)')
    plt.show()
```

M Code for test_All.py

```
## @file test_All.py
#  @author Mingzhe Wang (wangm235)
#  @brief Test all the functions in this assignment.
#  @date 2021 Feb 8
#  @details pass.

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


class TestCircleT:

    def setup_method(self, method):
        self.circle1 = CircleT(0.0, 0.0, 4.0, 2.0)
        self.circle2 = CircleT(-2.3e10, 2.3395934, math.sqrt(2), 5.3e10)

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

    def test_init_exception_both_invalid(self):
        with pytest.raises(ValueError):
            CircleT(0.0, 0.0, 0.0, 0.0)

    def test_init_exception_second_invalid(self):
        with pytest.raises(ValueError):
            CircleT(7.03, 5.02, 1.0, -123.1)

    def test_init_exception_first_invalid(self):
        with pytest.raises(ValueError):
            CircleT(4.213, 3.4122, -1e10, 1e10)

    def test_cm_x(self):
        assert self.circle1.cm_x() == approx(0.0)
        assert self.circle2.cm_x() == approx(-2.3e10)

    def test_cm_y(self):
        assert self.circle1.cm_y() == approx(0.0)
        assert self.circle2.cm_y() == approx(2.3395934)

    def test_mass(self):
        assert self.circle1.mass() == approx(2.0)
        assert self.circle2.mass() == approx(5.3e10)

    def test_m_inert(self):
        assert self.circle1.m_inert() == approx(16.0)
        assert self.circle2.m_inert() == approx(5.3e10)


class TestTriangleT:

    def setup_method(self, method):
        self.triangle1 = TriangleT(4.6e3, 9.123, math.sqrt(6), 12.0)
        self.triangle2 = TriangleT(5.02, 23.1, 1.0, 5.389)

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

    def test_init_exception_both_invalid(self):
        with pytest.raises(ValueError):
            TriangleT(0.0, 0.0, 0.0, 0.0)

    def test_init_exception_second_invalid(self):
        with pytest.raises(ValueError):
            TriangleT(7.0313, 302, 0.00002, -0.000001)

    def test_init_exception_first_invalid(self):
        with pytest.raises(ValueError):
            TriangleT(4.213, math.pi, -1.0 * 23, 12.123)
    def test_cm_x(self):
        assert self.triangle1.cm_x() == approx(4.6e3)
        assert self.triangle2.cm_x() == approx(5.02)

    def test_cm_y(self):
        assert self.triangle1.cm_y() == approx(9.123)
        assert self.triangle2.cm_y() == approx(23.1)

    def test_mass(self):
        assert self.triangle1.mass() == approx(12.0)
        assert self.triangle2.mass() == approx(5.389)

    def test_m_inert(self):
        assert self.triangle1.m_inert() == approx(6.0)
        assert self.triangle2.m_inert() == approx(5.389 / 12)


class TestBodyT:

    def setup_method(self, method):
        self.body1 = BodyT([-1.0, 1.0, 1.0, -1.0], [-1.0, -1.0, 1.0, 1.0], [2.33412,
                           2.33412, 2.33412, 2.33412])
        self.body2 = BodyT([-1.0, 1.0, 0.0], [0.0, 0.0, math.sqrt(3)], [2.8, 2.8, 2.8])
        self.body3 = BodyT([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.5, 2.83, 3.2])

    def teardown_method(self, method):
        self.body1 = None
        self.body2 = None
        self.body3 = None

    def test_init_exception_len_not_equal(self):
        with pytest.raises(ValueError):
            BodyT([1.0, 2.0, 3.12], [0.12], [1.0, 5.3, 7.9])
        with pytest.raises(ValueError):
            BodyT([0.0], [0.0], [0.0, 0.0])

    def test_init_exception_m_not_pos(self):
        with pytest.raises(ValueError):
            BodyT([1.2, math.pi, 12.0], [2.3, 12.3, 5.31], [2.3, 12.3, -1e10])
        with pytest.raises(ValueError):
            BodyT([-12.2, 0.0], [-5.7, 0.1], [1.4, 0.0])

    def test_cm_x(self):
        assert self.body1.cm_x() == approx(0.0)
        assert self.body2.cm_x() == approx(0.0)
        assert self.body3.cm_x() == approx(0.0)

    def test_cm_y(self):
        assert self.body1.cm_y() == approx(0.0)
        assert self.body2.cm_y() == approx(1 * math.sqrt(3) / 3.0)
        assert self.body3.cm_y() == approx(0.0)

    def test_mass(self):
        assert self.body1.mass() == approx(4 * 2.33412)
        assert self.body2.mass() == approx(3 * 2.8)
        assert self.body3.mass() == approx(1.5 + 2.83 + 3.2)

    def test_m_inert(self):
        self.body4 = BodyT([-1.0, 1.0, 0.0], [0.0, 0.0, math.sqrt(3)], [1.0, 1.0, 1.0])
        assert self.body1.m_inert() == approx(8 * 2.33412)
        assert self.body2.m_inert() == approx(2.8 * 4.0)
        assert self.body3.m_inert() == approx(0.0)
        assert self.body4.m_inert() == approx(4.0)


class TestScene:

    # No test for get_unbal_forces() and set_unbal_forces(), as specified in the requirement.
    def setup_method(self, method):

        # global settings with some function only for s1 and s2.
        g = 9.81
        m = 1.0

        def F_0(t):
            return 0.0

        def F_g_s1s2(t):
            return - g * m
        def F_x_s2(t):
            return 10.923

        # scene1: falling movement for a circle object
        self.circle1 = CircleT(10.0, 10.0, 1.0, m)
        self.scene1 = Scene(self.circle1, F_0, F_g_s1s2, 0.0, 0.0)

        # scene2: projectile movement for a triange object
        self.triangle1 = TriangleT(0.0, 0.0, 1.2, m)
        self.scene2 = Scene(self.triangle1, F_x_s2, F_g_s1s2, math.sqrt(3), 0.0)

        # scene3: general projectile movement for an object.
        # The settings can be changed later and need to update the corresponding test result.
        # all settings for scene3 are here
        m_in_ys = 2.3
        v = 5.6
        theta = math.pi / 4.0
        xs = [-1, 1, 0]
        ys = [0, 0, math.sqrt(3)]

        # input corresponding the setting
        vx = math.cos(theta) * v
        vy = math.sin(theta) * v
        ms = [m_in_ys, m_in_ys, m_in_ys]

        def F_g_s3(t):
            return - g * len(ms) * m_in_ys

        self.body1 = BodyT(xs, ys, ms)
        self.scene3 = Scene(self.body1, F_0, F_g_s3, vx, vy)

    def teardown_method(self, method):
        self.circle1 = None
        self.scene1 = None
        self.triangle1 = None
        self.scene2 = None
        self.body1 = None
        self.scene3 = None

    def test_get_shape(self):
        assert self.scene1.get_shape() == self.circle1
        assert self.scene2.get_shape() == self.triangle1
        assert self.scene3.get_shape() == self.body1

    def test_get_init_velo(self):
        assert self.scene1.get_init_velo()[0] == approx(0.0)
        assert self.scene1.get_init_velo()[1] == approx(0.0)
        assert self.scene2.get_init_velo()[0] == approx(math.sqrt(3))
        assert self.scene2.get_init_velo()[1] == approx(0.0)
        assert self.scene3.get_init_velo()[0] == approx((math.sqrt(2) / 2) * 5.6)
        assert self.scene3.get_init_velo()[1] == approx((math.sqrt(2) / 2) * 5.6)

    def test_set_shape(self):
        self.triangle1 = TriangleT(0.0, 0.0, 1.0, 5.82)
        self.scene1.set_shape(self.triangle1)
        assert self.scene1.get_shape() == self.triangle1
        self.body1 = BodyT([-1.0, 1.0, 0.0], [0.0, 0.0, math.sqrt(3)], [2.8, 2.8, 2.8])
        self.scene2.set_shape(self.body1)
        assert self.scene2.get_shape() == self.body1
        self.triangle1 = TriangleT(0.12, 9.12, 1.0, 5.23)
        self.scene3.set_shape(self.triangle1)
        assert self.scene3.get_shape() == self.triangle1

    def test_set_init_velo(self):

        self.scene1.set_init_velo(5.6, 1e10)
        assert self.scene1.get_init_velo()[0] == approx(5.6)
        assert self.scene1.get_init_velo()[1] == approx(1e10)
        self.scene2.set_init_velo(-0.12, 9.123)
        assert self.scene2.get_init_velo()[0] == approx(-0.12)
        assert self.scene2.get_init_velo()[1] == approx(9.123)
        self.scene3.set_init_velo(-1e10, 2.3)
        assert self.scene3.get_init_velo()[0] == approx(-1e10)
        assert self.scene3.get_init_velo()[1] == approx(2.3)

    @staticmethod
    def __compare_two_seqs(s_calc, s_true, epi):
        if len(s_calc) != len(s_true):
            raise ValueError("Must be two same-length seqs")
        max_abs_numer = 0.0
        max_abs_denom = 0.0
        for i in range(len(s_calc)):
            sub = s_calc[i] - s_true[i]
            abs_sub = abs(sub)
            abs_true = abs(s_true[i])
            if abs_sub > max_abs_numer:
                max_abs_numer = abs_sub
            if abs_true > max_abs_denom:
                max_abs_denom = abs_true
        if max_abs_numer / max_abs_denom < epi:
            return True
        else:
            return False

    def test_sim_scene1(self):
        t, wsol = self.scene1.sim(10, 101)
        t_counter = 0
        for t_s, wsol_s in zip(t, wsol):
            wsol_true = [10.0, 10.0 - 0.5 * 9.81 * t_counter**2, 0.0, - 9.81 * t_counter]
            assert t_s == approx(t_counter)
            assert TestScene.__compare_two_seqs(wsol_s, wsol_true, 0.001)
            t_counter += 0.1

    def test_sim_scene2(self):
        t, wsol = self.scene2.sim(10, 101)
        t_counter = 0
        for t_s, wsol_s in zip(t, wsol):
            wsol_true = [
                math.sqrt(3) * t_counter + 0.5 * 10.923 * t_counter**2,
                -0.5 * 9.81 * t_counter**2, math.sqrt(3) + 10.923 * t_counter,
                -9.81 * t_counter
            ]
            assert t_s == approx(t_counter)
            assert TestScene.__compare_two_seqs(wsol_s, wsol_true, 0.001)
            t_counter += 0.1

    def test_sim_scene3(self):
        t, wsol = self.scene3.sim(10, 101)
        t_counter = 0
        for t_s, wsol_s in zip(t, wsol):
            wsol_true = [
                (math.sqrt(2) / 2) * 5.6 * t_counter,
                math.sqrt(3) / 3 + math.sqrt(2) * 5.6 / 2 * t_counter - 4.905 * t_counter**2,
                (math.sqrt(2) / 2) * 5.6,
                (math.sqrt(2) / 2) * 5.6 - 9.81 * t_counter
            ]
            assert t_s == approx(t_counter)
            assert TestScene.__compare_two_seqs(wsol_s, wsol_true, 0.001)
            t_counter += 0.1
```

N Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Steven Kostiuk
#  @brief Contains the CircleT type to represent shapes that are circles
#  @date 2021-02-16
from Shape import Shape

## @brief CircleT is used to represent shapes that are circles


class CircleT(Shape):
    ## @brief Constructor for class CircleT
    #  @param xs X value of the center of mass
    #  @param ys Y value of the center of mass
    #  @param rs Radius of the circle
    #  @param ms Mass of the circle
    #  @throws ValueError Thrown if the mass or radius of circle is less than or equal to zero
    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 Getter method that gets the x value of the center of mass
    #  @return x value of the center of mass
    def cm_x(self):
        return self.x

    ## @brief Getter method that gets the y value of the center of mass
    #  @return y value of the center of mass
    def cm_y(self):
        return self.y

    ## @brief Getter method that gets the mass of the circle
    #  @return mass of the circle
    def mass(self):
        return self.m

    ## @brief Getter method that gets the moment of inertia of the circle
    #  @return moment of inertia of the circle
    def m_inert(self):
        return (self.m * (self.r ** 2)) / 2
```

O Code for Partner's TriangleT.py

```
## @file TriangleT.py
#  @author Steven Kostiuk
#  @brief Contains the TriangleT type to represent shapes that are triangles
#  @date 2021-02-16
from Shape import Shape

## @brief TriangleT is used to represent shapes that are triangles


class TriangleT(Shape):
    ## @brief Constructor for class TriangleT
    #  @param xs X value of the center of mass
    #  @param ys Y value of the center of mass
    #  @param rs Side of the triangle
    #  @param ms Mass of the triangle
    #  @throws ValueError Thrown if the mass or side of triangle is less than or equal to zero
    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 Getter method that gets the x value of the center of mass
    #  @return x value of the center of mass
    def cm_x(self):
        return self.x

    ## @brief Getter method that gets the y value of the center of mass
    #  @return y value of the center of mass
    def cm_y(self):
        return self.y

    ## @brief Getter method that gets the mass of the triangle
    #  @return mass of the triangle
    def mass(self):
        return self.m

    ## @brief Getter method that gets the moment of inertia of the triangle
    #  @return moment of inertia of the triangle
    def m_inert(self):
        return (self.m * (self.s ** 2)) / 12
```

P Code for Partner's BodyT.py

```
## @file BodyT.py
#  @author Steven Kostiuk
#  @brief Contains the BodyT type to represent physics bodies
#  @date 2021-02-16
from Shape import Shape

## @brief BodyT is used to represent a combination of shapes
#  that form a physics body


class BodyT(Shape):
    ## @brief Constructor for class TriangleT
    #  @param xs List of x values of the center of mass
    #  @param ys List of Y values of the center of mass
    #  @param ms List of masses
    #  @throws ValueError Thrown if the lengths of the 3 lists are not the same
    #  @throws ValueError Thrown if one of the masses 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 u in ms:
            if not(u > 0):
                raise ValueError
        self.cmx = self.__cm(xs, ms)
        self.cmy = self.__cm(ys, ms)
        self.m = self.__sum(ms)
        self.moment = self.__mmom(xs, ys, ms) - self.__sum(ms) * \
            (self.__cm(xs, ms) ** 2 + self.__cm(ys, ms) ** 2)

    ## @brief Getter method that gets the x value of the center of mass
    #  @return x value of the center of mass
    def cm_x(self):
        return self.cmx

    ## @brief Getter method that gets the y value of the center of mass
    #  @return y value of the center of mass
    def cm_y(self):
        return self.cmy

    ## @brief Getter method that gets the mass of the physics body
    #  @return mass of the physics body
    def mass(self):
        return self.m

    ## @brief Getter method that gets the moment of inertia of the physics body
    #  @return moment of inertia of the physics body
    def m_inert(self):
        return self.moment

    ## @brief Calculates the sum of a list
    #  @param m List of masses
    #  @return total mass of the physics body
    def __sum(self, m):
        result = 0
        for i in m:
            result = result + i
        return result

    ## @brief Calculates the center of mass of the physics body
    #  @param z List of coordinates that represent the x or y values of center of mass
    #  @param m List of masses
    #  @returns x or y coordinate of the center of mass of the physics body
    def __cm(self, z, m):
        result = 0
        for i in range(len(m)):
            result = result + z[i] * m[i]
        return result / self.__sum(m)

    ## @brief Calculates the first part of the moment of inertia of the physics body
    #  @param x List of x values of the center of mass
    #  @param y List of y values of the center of mass
    #  @param m List of masses
    #  @return first part of the moment of inertia of the physics body
    def __mmom(self, x, y, m):
        result = 0
```

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

Q Code for Partner's Scene.py

```
## @file Scene.py
#  @author Steven Kostiuk
#  @brief Contains the Scene type which is used to simulate a shape's movement
#  @date 2021-02-16
#  @details Simulates a shape's movement according to initial velocities
#  and unbalanced forces functions
from scipy.integrate import odeint

## @brief Scene is used to simulate a shape's movement


class Scene():
    ## @brief Constructor for class TriangleT
    #  @param s Shape
    #  @param Fx Unbalanced forces function for x coordinates
    #  @param Fy Unbalanced forces function for y coordinates
    #  @param vx Initial x velocity
    #  @param vy Initial y 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 Getter method to get the shape
    #  @return shape
    def get_shape(self):
        return self.s

    ## @brief Getter method to get the unbalanced forces functions
    #  @return unbalanced forces functions
    def get_unbal_forces(self):
        return self.Fx, self.Fy

    ## @brief Getter method to get the initial velocities of the shape
    #  @return initial velocities
    def get_init_velo(self):
        return self.vx, self.vy

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

    ## @brief Setter method to set the unbalanced forces functions
    #  @param Fx Unbalanced forces function for x coordinates
    #  @param Fy Unbalanced forces function for y coordinates
    def set_unbal_forces(self, Fx, Fy):
        self.Fx = Fx
        self.Fy = Fy

    ## @brief Setter method to set the initial velocities of the shape
    #  @param vx Initial x velocity
    #  @param vy Initial y velocity
    def set_init_velo(self, vx, vy):
        self.vx = vx
        self.vy = vy

    ## @brief Simulation method that simulates the movement of a shape
    #  @param tfinal Final time which the simulation runs until
    #  @param nsteps Number of steps which the time will be divided into
    #  @return sequence of real numbers representing the time and a
    #  sequence containing 4 sequences of real numbers representing x and y values
    def sim(self, tfinal, nsteps):
        t = [(i * tfinal) / (nsteps - 1) for i in range(nsteps)]
        return t, odeint(self.__ode, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)

    ## @brief Ordinary differential equation method
    #  @param w List of center of masses and initial velocities
    #  @param t List of time in natural ascending order
    #  @return List of initial velocities and the results of
    #  the unbalanced forces functions
    def __ode(self, w, t):
        return [w[2], w[3], self.Fx(t) / self.s.mass(), self.Fy(t) / self.s.mass()]
```