Assignment 2 Solution

Cassidy Baldin

February 25, 2021

This report discusses the testing phase for the `CircleT`, `TriangleT`, `BodyT`, and `Scene` classes written for Assignment 2. It does not discuss testing for the `Shape` interface or the `Plot` function, as these were to be tested manually in this specification. It also discusses the results of running the same tests on the partner files. The assignment specifications are then critiqued and the requested discussion questions are answered.

1 Testing of the Original Program

Tests were written such that each method that was implemented into the design had an appropriate amount of test cases that I felt covered the edge/boundary cases for each method respectively. These tests were written using pytest as a way to check and tally the results of the testing. The breakdown of all test cases and rationale are below:

For class `CircleT`:

To test methods `cm_x` and `cm_y`, I tested two cases for each as I thought it would either get the required self.variable or it would not. The first case was when the value of cm was a positive number, and the other was when it was 0. Since these were basic methods, I did not think that there would be much variance in the results of it.
To test methods `mass` and `m_inert`, I also did two cases that were pretty basic, just to see if it would return the correct values as it should as `mass` was a simple variable return and `m_inert` is a relatively simple calculation. I also used the approximation function to test if the `m_inert` method was correct as it returns float values and there could be precision errors. I decided to check if the value was off by 1e-3, as I thought that was a reasonable margin of error.
To test for the exceptions that might be raised in this class, I created cases where the mass, radius and both simultaneously were negative or zero values to make sure the exception held as both radius and mass should be greater than zero according to the specification.

If the mass was found to be less than or equal to zero, a ValueError was thrown and caught and raised in pytest. I also could have used testing cases where the parameters `xs` and `ys` were negative values, but all that would change would be to add another case for the `cm_x` and `cm_y` methods to the testing file. Since the method was simply returning the value it was set to, I thought it would be sufficient with the cases I had.

For class `TriangleT`:

To test methods `cm_x`, `cm_y`, `mass` and `m_inert`, I tested using the same ideas as the `CircleT` class, as these specifications were almost identical in terms of the methods being used and the implementation of these methods. The only difference was the use of side instead of radius, and the inertia was divided by 12 instead of 2.
The testing of the exceptions was the same as well, as both the side length and mass must be greater than zero in this specification. Again, I could have added cases for testing negative values of `cm_x` and `cm_y`, but since these were basic getter methods I thought that two cases would suffice.

For class `BodyT`:

To test methods `cm_x`, `cm_y`, `mass` and `m_inert`, I tested using the same ideas as the `CircleT` class, as these specifications were almost identical in terms of the methods being used and the implementation of these methods. The difference was with how they were setting these values, as they were reliant on local methods calculating the correct information to set the values. However, since for this assignment we were not testing local methods specifically, the only way to tell if the implementation was correct was to test the getter methods, which is what I did. These were tested the same was as `CircleT` as they were similar getter methods.
To test the two different exceptions raised in this specification, I first tested to make sure that the length of the sequences was the same by using an if statement to check if the lengths of input lists `xs`, `ys` and `ms` were the same. This was because in the specification the lengths must be the same so that the center of mass coordinates, mass and inertia of the body are successfully and correctly returned. If the length of these three sequences were not equal, it would throw a ValueErorr. To test if all mass values were greater than zero, I used a loop with an if statement that checked all values in the `ms` list. If the value in this mass list was less than or equal to zero, it would throw a ValueError.

For class `Scene`:

To test the getters `get_shape`, and `get_init_velo`, I used the same ideas as `CircleT`, as

these were basic tests for basic getter methods. The only differences were the names of the getters and `get_init_velo` returned tuples of both the x and y values of the velocity instead of just one single value. So, for these tests, I simply checked if the values I was testing were equal to the ones I input into the tests.
To test the setters `set_shape`, and `set_init_velo`, I used a combination of the setters and their respective getters to set the value to the new one and return that value to show that it had indeed been mutated. Then after it had been set to one value, I checked if it could be set back to the original value, then checked again with the getter. This test was not only checking the capability of the implementation to set a new shape of velocity, but also testing the ability of the getters once more. Since this test relies on the getters to work correctly, there is some margin or error for this test to fail if the getters also fail, but I am not sure if there is any other way to check if the value of the object has changed from outside of the class as the variables in the `init` constructor are private outside of the class.
To test `sim`...

2 Results of Testing Partner's Code

After testing my partners code using my testing file, they passed all 56 cases! This might have been a result of the A2 specification being less ambiguous than A1, allowing the designs of both my partner and I to be relatively similar.

For instance, in the `CircleT` class, they were nearly identical in terms of implementation, besides the variable names being different from one another. The only other difference was that they raised a value error before assigning the self parameters, while in my implementation, this exception was checked after they were assigned. My partner's implementation could save a bit of time and memory, as it will throw this exception immediately while my design does not. This was similar to the `TriangleT` class implementation.
In the `BodyT` class, the `__init__` method was done in a similar way with both of us checking that the length of the sequences were equal and checking if all masses in the sequence were greater than zero. A difference between the implementation in this class was the `__sum__` method, as in my partner's code they included it but I just used the built in functionality of python to sum all the values in a sequence using the `sum` method. Since they were local functions, I assumed that this would be allowed for the implementation, as it was not a direct method that was needed in the specification and was just a helper method for the calculation of the total mass of the body. The other two local methods `__cm__` and `__mmom` were implemented in a similar manner.
In the `Scene` class, all the methods were implemented in the same basic way, apart from

the `ode` local function. In my implementation, it was a part of the `sim` method, while in my partner's code it was a private method separate from the `sim` method. This does not change the results of the output of this function, but it does mean that if hypothetically you wanted another method that could use this `ode` method, in my case I would need to create another method inside of the new one, while my partner could just use the one that they already have as it is separate from an already existing function. I also realized while looking at my code again that I forgot to add doxygen comments for the parameters for the input variables `w, t` in the `ode` method to explain to the user what each parameter was specifying. Another difference in our implementation is that my partner imported `Shape` and used instantiated it in the `Scene` class while I did not. This is because I did not think that it was needed in the `Scene` class, as it was not directly using the interface. The input to the class had a relationship to `Shape` like the first parameter representing the shape that was to be in the scene, but the class itself did not need it from my understanding. Also, in the specification, it was stated that modules like `CircleT` inherit `Shape` and this was not the case for `Scene`.

Having seen the similarities between our designs, it can be easy to see why my partner had passed all my test cases. If I had been able to fully test the `sim` method, I am sure they would have passed the tests, as their implementation was very similar to mine overall. I also ran their code using the `test_expt.py` file and it produced the correct graph using my plot method, further confirming my statement.

3 Critique of Given Design Specification

4 Answers

a) I think that getter and setters methods should not be tested as much, as if they are simply returning a state variable in its normal form without exception to the values like in this specification, then all the test will show is that same value returned. Since getter methods are usually relatively simple, meaning that all they do is return a value of the class, it does not need to be tested greatly, as not much can really go wrong or break the program. If you have a certain exception that must be thrown in the constructor or setter method where the getting of that value using the getter must not occur, then you could test this case, but that would not be testing the getter method so much as it would be testing the exception case within the class. In the case where the getter or setter method is used as part of a wider test for example if you needed to get the value of a variable of the class to be used in another method, then this could be a way to indirectly test your getter method, but imply testing if a value can be returned is not much of a test. That is why I think that getters and setters should not be unit tested if they are very simple like they were in this specification.

b) Since the functions that are taken into the constructor in the class `Scene` must be defined outside of the class, to test the getters and setters for these state variables, you could run tests that include function definitions in them, as well as all other necessary components of building the `Scene` object. For example, to test `get_unbal_forces`, you could define two functions for the forces in each direction, then create a shape object to input into a new `Scene` object that can be created in this test. Then you can successfully test the getter method for the functions. This example can be seen below in `Ftest`. A way to test the setters would be to copy the last test but define a new function inside the test case, then set the forces to this new function using the setter method. Then you would use a getter to make sure the test ran correctly. This example can be seen below in `Ftest2`. If you try to test these functions when they are not inside of the test, it will throw a `NameError`, as the function is not in the scope of the test and is therefore not recognized. This type of test would clutter the testing file as new forces and objects need to be defined with each test, causing an excess amount of objects to be made.

```
def test_Ftest(self):
    def Fx(t):
        return 0
    def Fy(t):
        return -9.81
    self.c = CircleT(1.0, 10.0, 0.5, 5.0)
    self.s1 = Scene(self.c, Fx, Fy, 0, 0)
    assert self.s1.get_unbal_forces() == (Fx, Fy)

def test_Ftest2(self):
    def Fx(t):
        return 0
    def Fy(t):
        return -9.81
    def Fz(t):
        return 1
    self.c = CircleT(1.0, 10.0, 0.5, 5.0)
    self.s1 = Scene(self.c, Fx, Fy, 0, 0)
    self.s1.set_unbal_forces(Fx, Fz)
    assert self.s1.get_unbal_forces() == (Fx, Fz)
```

c) c) The assignment does not require automated tests for Plot.py. If automated tests were required how might you do them? Hint: matplotlib can generate a file for any plots that you might build.

d) Close_Enough Module

Close_Enough

Uses

None

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
close_enough	x_{calc} : seq of $\mathbb{R}$, y_{true} : seq of $\mathbb{R}$	$\mathbb{B}$	ValueError

Semantics

State Variables

None

State Invariant

None

Assumptions

It assumes that both x_{calc} and y_{true} sequences are of the same length and not empty.

Access Routine Semantics

close_enough($x_{\text{calc}}, y_{\text{true}}$):

- transition: Implements the formula below to determine if two sequences are close to being equal, based on 1e-03, as this is a reasonably close margin of error.

$$\frac{||x_{\text{calc}} - y_{\text{true}}||}{||y_{\text{true}}||} < \text{1e-03} \tag{1}$$

- output: $out := \frac{\text{abs}(\text{sub}(x_{\text{calc}}, y_{\text{true}}))}{\text{abs}(y_{\text{true}})} < \text{1e-03}$
- exception: $(\neg(\text{abs}(y_{\text{true}}) \neq 0) \Rightarrow \text{ValueError})$

Local Functions

sub : seq of $\mathbb{R}$, seq of $\mathbb{R} \rightarrow \mathbb{R}$
$\text{sub}(x, y) \equiv [(+i : \mathbb{N} | i \in [0..|x| - 1] : x_i - y_i)]$

abs : seq of $\mathbb{R} \rightarrow \mathbb{R}$
$\text{abs}(z) \equiv (+i : \mathbb{N} | i \in [0..|z| - 1] : z_i > max \Rightarrow max = z_i)$,
where max = current maximum of sequence

e) The given specification has exceptions for non-positive values of shape dimensions and mass but not for the x and y coordinates of the center of mass because of what they represent in terms of physical space. In the case of shape dimension like radius and side length they must be non-negative as in the physical world you cannot have a negative side length as you cannot physically measure that or have a negative amount of space. If it has zero length, then it is a point or it would not exist, but it is definitely not considered a shape. In the case of mass, an object having zero or negative mass is not possible, unless you count anti-matter as part your design. However, all matter has mass, so to have a non-positive value of mass would not be possible. There does not need to be exceptions for coordinates though, as they do not represent physical properties, but rather positions in relation to objects in space. The center of mass can be calculated using vector addition of position vectors which point to the center of mass of each body which was seen in the `BodyT` class. The center of mass is calculated as the sum of all x coordinates multiplied by corresponding y coordinates, divided by the total mass of the system. If a coordinate is positive, it means that it is to the right relative to the center of the system. If it is negative, it is left relative to the center of the system. This is similar to how vectors work, as positive is usually up and right, while negative is usually down and left. This is the same logic applied to the coordinates of mass. Therefore, there does not need to be exceptions for negative coordinates of the center of mass.

f) In the class `TriangleT`, the state invariant is that $s > 0 \wedge m > 0$. In order for this to be satisfied by the given specification, this would mean that every access program/method in the class will hold both before and after each method is run. In this class, the only place for this state invariant to be violated is in the `__init__` method, as this is where the values of s and m could be less than 0. All of the other methods in this class are getters, meaning that they simply return the value of the parameter in the constructor. There is no way for the value of the side or mass parameters to change in these methods. In the given specification, in the `__init__` method it says to throw a `ValueError` if the state invariant is true. This will cause the program to not finish making the object if either the side or mass is less than or equal to zero. Since this exception exists in the constructor and the value of the self parameters cannot be changed by any mutators in the class, this proves that the state invariant is always satisfied by the given specification.

g) List comprehension statement:

```
sq_list = [i**(1/2) for i in range(5, 20, 2)]
```

where `sq_list` is a list of the square roots of all odd ints between 5 and 19 (inclusive).

h) A python function that takes a string and returns the string, but with all upper case letters removed is:

```
def no_cap(string):
    new = string
    for i in string:
        if (ord(i) >= 65) and (ord(i) <= 90):
            new = string.replace(i, '')
        string = new
    return string
```

i) i) How are principles of abstraction and generality related?

j) If we have high coupling between modules, the better scenario would be to have a module that is used by many other modules. If you have a module that uses many other modules, this means that you would need all of the other modules to work properly and be implemented before you are able to use the top level module. This could allow for many issues as if one of those modules has a bug or does not run properly, then the top module will not work properly either as a result. If you have a module that is used by many other modules, this would mean that as long as the low level is functioning properly, then the other modules that use it will be able to operate effectively as well. But if there is a problem with the low level module, it will affect the other modules as well. This method relies on program correctness and robustness of only one module as opposed to the correctness of many modules as seen in the first example. The second scenario would be better in general, as it would be easier to fix just the one low level module than to try to figure out which of the lower level modules in the first scenario needed to be fixed to make the top level module run. Since many modules feed into the top level module in the first scenario, if it does not work, you will not necessarily always know which of the lower modules is not working correctly in order to fix it. This is why the second scenario would be better in general.

E Code for Shape.py

```python
## @file Shape.py
#  @author Cassidy Baldin
#  @brief Contains an interface for the shape of the object
#  @date February 12th, 2021

from abc import ABC, abstractmethod


##  @brief Shape is used as an interface for the shape of the object
class Shape(ABC):
    @abstractmethod
    ## @brief cm_x returns the x value of the center of mass
    #  @return value representing the center of mass of the x value
    def cm_x(self):
        pass

    @abstractmethod
    ## @brief cm_y returns the y value of the center of mass
    #  @return value representing the center of mass of the y value
    def cm_y(self):
        pass

    @abstractmethod
    ## @brief mass returns the mass of the object
    #  @return value representing the mass of the object
    def mass(self):
        pass

    ## @brief m_inert returns the inertia of the object
    #  @return value representing the inertia of the object
    def m_inter(self):
        pass
```

F Code for CircleT.py

```
## @file CircleT.py
# @author Cassidy Baldin
# @brief Contains a module for a circle
# @date February 12th, 2021


from Shape import Shape


## @brief CircleT is used as a constructor for a circle
class CircleT(Shape):
    ## @brief constructor for method CircleT
    # @details Assumes that arguments provided to the access programs
    #          will be of the correct type
    # @param xs value representing the x value of the center of mass
    # @param ys value representing the y value of the center of mass
    # @param rs value representing the radius of the circle
    # @param ms value representing mass of circle
    # @throws ValueError if either radius or mass are less than zero
    def __init__(self, xs, ys, rs, ms):
        self.__x = xs
        self.__y = ys
        self.__r = rs
        self.__m = ms

        if not ((self.__r > 0) and (self.__m > 0)):
            raise ValueError("Radius and Mass must be greater than zero")

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

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

    ## @brief mass returns the mass of the object
    # @return value representing the mass of the object
    def mass(self):
        return self.__m

    ## @brief m_inert returns the moment of inertia of the object
    # @return value representing the moment of inertia of the object
    def m_inert(self):
        return (self.__m * self.__r**2) / 2
```

G Code for TriangleT.py

```
## @file TriangleT.py
#  @author Cassidy Baldin
#  @brief Contains a module for a triangle
#  @date February 12th, 2021

from Shape import Shape


##  @brief TriangleT is used as a constructor for a triangle
class TriangleT(Shape):
    ## @brief constructor for method BodyT
    #  @details Assumes that arguments provided to the access programs
    #           will be of the correct type
    #  @param xs value representing the x value of the center of mass
    #  @param ys value representing the y value of the center of mass
    #  @param ss value representing the side length of the triangle
    #  @param ms value representing mass of the triangle
    #  @throws ValueError if either side length or mass are less than zero
    def __init__(self, xs, ys, ss, ms):
        self.__x = xs
        self.__y = ys
        self.__s = ss
        self.__m = ms

        if not ((self.__s > 0) and (self.__m > 0)):
            raise ValueError("Side and Mass must be greater than zero")

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

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

    ## @brief mass returns the mass of the object
    #  @return value representing the mass of the object
    def mass(self):
        return self.__m

    ## @brief m_inert returns the moment of inertia of the object
    #  @return value representing the moment of inertia of the object
    def m_inert(self):
        return (self.__m * self.__s**2) / 12
```

H Code for BodyT.py

```
## @file BodyT.py
#  @author Cassidy Baldin
#  @brief Contains a module for an unspecified body in space
#  @date February 12th, 2021

from Shape import Shape


##  @brief BodyT is used as a constructor for a body of unknown shape in space
class BodyT(Shape):
    ## @brief constructor for method BodyT
    #  @details Assumes that arguments provided to the access programs
    #           will be of the correct type
    #  @param xs value representing the x value of the center of mass
    #  @param ys value representing the y value of the center of mass
    #  @param ms value representing mass of the object
    #  @throws ValueError if the length of all input sequences are not equal
    #  @throws ValueError if values in sequence ms are less than zero
    def __init__(self, xs, ys, ms):
        if not (len(xs) == len(ys) == len(ms)):
            raise ValueError("Sequences must be of the same length")
        for i in range(0, len(ms)):
            if not (ms[i] > 0):
                raise ValueError("Mass must be greater than zero")

        self.__cmx = self.__cm__(xs, ms)
        self.__cmy = self.__cm__(ys, ms)
        self.__m = sum(ms)
        self.__moment = self.__mmom__(xs, ys, ms) \
            - sum(ms) * (self.__cm__(xs, ms)**2 + self.__cm__(ys, ms)**2)

    ## @brief cm returns the center of mass of the object
    #  @return value representing the center of mass of the object
    def __cm__(self, z, m):
        cm = 0
        for i in range(0, len(m)):
            cm = cm + (z[i] * m[i])
        return cm / sum(m)

        ## @brief mmom returns the value of the moment of inertia of the body
        #  @return value of the moment of inertia of the body
    def __mmom__(self, x, y, m):
        mmom = 0
        for i in range(0, len(m)):
            mmom = mmom + m[i] * (x[i]**2 + y[i]**2)
        return mmom

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

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

    ## @brief mass returns the mass of the object
    #  @return value representing the mass of the object
    def mass(self):
        return self.__m

    ## @brief m_inert returns the moment of inertia of the object
    #  @return value representing the moment inertia of the object
    def m_inert(self):
        return self.__moment
```

I Code for Scene.py

```
## @file Scene.py
#  @author Cassidy Baldin
#  @brief Contains a module to construct motion simulation
#  @date February 12th, 2021

from scipy import integrate


##  @brief Scene is used as a way to construct a motion simulation
class Scene:
    ## @brief constructor for method Scene
    #  @param s_prime value representing the shape
    #  @param Fx_prime value unbalanced force function in x direction
    #  @param Fy_prime value unbalanced force function in y direction
    #  @param vx_prime value representing initial velocity in x direction
    #  @param vy_prime value representing initial velocity in 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 gets the shape of the body
    #  @return value representing the shape of the body
    def get_shape(self):
        return self.__s

    ## @brief gets the unbalanced forces of the body in the x and y direction
    #  @return value of the unbalanced forces in the x direction
    def get_unbal_forces(self):
        return self.__Fx, self.__Fy

    ## @brief gets the initial velocity of the body in the x and y direction
    #  @return value of initial velocity in the x direction
    def get_init_velo(self):
        return self.__vx, self.__vy

    ## @brief sets the shape of the body
    #  @param s_prime value representing the shape
    def set_shape(self, s_prime):
        self.__s = s_prime

    ## @brief sets the unbalanced forces of the body
    #  @param Fx_prime value unbalanced force function in x direction
    #  @param Fy_prime value unbalanced force function in y direction
    def set_unbal_forces(self, Fx_prime, Fy_prime):
        self.__Fx = Fx_prime
        self.__Fy = Fy_prime

    ## @brief sets the initial velocity of the body
    #  @param vx_prime value representing initial velocity in x direction
    #  @param vy_prime value representing initial velocity in y direction
    def set_init_velo(self, vx_prime, vy_prime):
        self.__vx = vx_prime
        self.__vy = vy_prime

    ## @brief simulates the  solution to the ode function to simulate motion
    #  @param t_final value representing the final time
    #  @param n_steps value representing the number of steps to divide the time interval
    def sim(self, t_final, nsteps):
        ## @brief calculates the resulting ode of the input
        #  @return solution to resulting ode
        def __ode(w, t):
            return [w[2], w[3],
                    self.__Fx(t) / self.__s.mass(), self.__Fy(t) / self.__s.mass()]

        t = []
        for i in range(0, nsteps):
            t.append((i * t_final) / (nsteps - 1))

        return t, integrate.odeint(__ode, [self.__s.cm_x(), self.__s.cm_y(),
                                           self.__vx, self.__vy], t)
```

J Code for Plot.py

```
## @file Plot.py
#  @author Cassidy Baldin
#  @brief Contains a module for plotting a motion simulation
#  @date February 12th, 2021

from matplotlib.pyplot import *


def plot(w, t):
    ## @brief plots values of x, y, and t based on input, where w, t is the output from
    #         the sim() method in Scene class that represents the motion of a projectile.
    #         X represents position of the projectile in x direction, Y represents position
    #         of the projectile in the y direction. T represents time.
    #  @details Assumes that the sequence will be built in order of increasing i values.
    if not (len(w) == len(t)):
        raise ValueError("Sequences must be of the same length")

    x, y = [], []

    for i in range(0, len(w)):
        x.append(w[i][0])
        y.append(w[i][1])

    fig, (ax1, ax2, ax3) = subplots(3)
    fig.suptitle("Motion Simulation")
    ax1.plot(t, x)
    ax1.set(ylabel="x(m)")
    ax2.plot(t, y)
    ax2.set(ylabel="y(m)")
    ax3.plot(x, y)
    ax3.set(ylabel="y(m)")
    ax3.set(xlabel="x(m)")
    show()
```

K Code for test_driver.py

```
## @file test_driver.py
#  @author Cassidy Baldin
#  @brief A module for testing modules CircleT, TriangleT, BodyT, and Scene.py
#  @date February 12th, 2021

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

from pytest import *


class TestCircleT:
    def setup_method(self, method):
        self.c1 = CircleT(1.0, 10.0, 0.5, 5.0)
        self.c2 = CircleT(0, 0, 1.0, 1.0)

    def teardown_method(self, method):
        self.c1 = None
        self.c2 = None

    def test_cm_x(self):
        assert self.c1.cm_x() == 1.0

    def test_cm_x_zero(self):
        assert self.c2.cm_x() == 0

    def test_cm_y(self):
        assert self.c1.cm_y() == 10.0

    def test_cm_y_zero(self):
        assert self.c2.cm_y() == 0

    def test_mass1(self):
        assert self.c1.mass() == 5.0

    def test_mass2(self):
        assert self.c2.mass() == 1.0

    def test_m_inert1(self):
        assert self.c1.m_inert() == 0.625

    def test_m_inert2(self):
        assert self.c2.m_inert() == 0.5

    def test_neg_mass(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, 0.5, -5.0)

    def test_neg_rad(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, -0.5, 5.0)

    def test_neg_both(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, -0.5, -5.0)

    def test_zero_mass(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, 0.5, 0)

    def test_zero_rad(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, 0, 5.0)

    def test_zero_both(self):
        with raises(ValueError):
            CircleT(1.0, 10.0, 0, 0)


class TestTriangleT:
    def setup_method(self, method):
        self.t1 = TriangleT(1.0, 10.0, 0.5, 24.0)
        self.t2 = TriangleT(0, 0, 1.0, 1.0)
```

```python
    def teardown_method(self, method):
        self.t1 = None
        self.t2 = None

    def test_cm_x(self):
        assert self.t1.cm_x() == 1.0

    def test_cm_x_zero(self):
        assert self.t2.cm_x() == 0

    def test_cm_y(self):
        assert self.t1.cm_y() == 10.0

    def test_cm_y_zero(self):
        assert self.t2.cm_y() == 0

    def test_mass1(self):
        assert self.t1.mass() == 24.0

    def test_mass2(self):
        assert self.t2.mass() == 1.0

    def test_m_inert1(self):
        assert self.t1.m_inert() == 0.5

    def test_m_inert(self):
        assert self.t2.m_inert() == approx(0.08333, abs=1e-3)

    def test_neg_mass(self):
        with raises(ValueError):
            TriangleT(1.0, 10.0, 0.5, -5.0)

    def test_neg_side(self):
        with raises(ValueError):
            TriangleT(1.0, 10.0, -0.5, 5.0)

    def test_neg_both(self):
        with raises(ValueError):
            TriangleT(1.0, 10.0, -0.5, -5.0)

    def test_zero_mass(self):
        with raises(ValueError):
            TriangleT(1.0, 10.0, 0.5, 0)

    def test_zero_side(self):
        with raises(ValueError):
            TriangleT(1.0, 10.0, 0, 5.0)

    def test_zero_both(self):
        with raises(ValueError):
            TriangleT(0, 0, 0, 0)


class TestBodyT:
    def setup_method(self, method):
        self.b1 = BodyT([5, -7.5, -9.5, 11], [12, 6.5, -1, -10], [10, 10, 50, 30])
        self.b2 = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])

    def teardown_method(self, method):
        self.b1 = None
        self.b2 = None

    def test_cm_x(self):
        assert self.b1.cm_x() == -1.7

    def test_cm_x_zero(self):
        assert self.b2.cm_x() == 0

    def test_cm_y(self):
        assert self.b1.cm_y() == -1.65

    def test_cm_y_zero(self):
        assert self.b2.cm_y() == 0

    def test_mass1(self):
        assert self.b1.mass() == 100

    def test_mass2(self):
        assert self.b2.mass() == 40.0
```

```python
    def test_m_inert1(self):
        assert self.b1.m_inert() == 13306.25

    def test_m_inert2(self):
        assert self.b2.m_inert() == 80.0

    def test_len_xs(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1, 1], [1, 1, -1, -1], [10, 10, 10, 10])

    def test_len_ys(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1, 1], [10, 10, 10, 10])

    def test_len_ms(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10, 1])

    def test_neg_mass(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, -10, 10])

    def test_zero_mass(self):
        with raises(ValueError):
            BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 0])

    def test_empty(self):
        with raises(ValueError):
            BodyT([], [1, 1, -1, -1], [10, 10, 10, 10])

    def test_zero_xy(self):
        b3 = BodyT([0, 0, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1])
        assert b3.m_inert() == 0


class TestSceneT:
    def setup_method(self, method):
        def Fx(t):
            return 0

        def Fy(t):
            return -9.81

        self.c = CircleT(1.0, 10.0, 0.5, 5.0)
        self.c2 = CircleT(1.0, 10.0, 0.5, 1.0)
        self.t = TriangleT(1.0, 10.0, 0.5, 24.0)
        self.b = BodyT([1, -1, -1, 1], [1, 1, -1, -1], [10, 10, 10, 10])
        self.s1 = Scene(self.c, Fx, Fy, 0, 0)
        self.s2 = Scene(self.t, Fx, Fy, 10, -10)
        self.s3 = Scene(self.c2, Fx, Fy, 0, 0)

        # def test_Ftest(self):
        #     assert self.s1.get_unbal_forces() == (Fx, Fy)

        # def test_Ftest2(self):
        #     def Fz(t):
        #         return 1
        #     self.s1.set_unbal_forces(Fx, Fz)
        #     assert self.s1.get_unbal_forces() == (Fx, Fz)

    def teardown_method(self, method):
        self.c = None
        self.c2 = None
        self.t = None
        self.s1 = None
        self.s2 = None

    def test_Ftest(self):
        def Fx(t):
            return 0
        def Fy(t):
            return -9.81
        self.c = CircleT(1.0, 10.0, 0.5, 5.0)
        self.s1 = Scene(self.c, Fx, Fy, 0, 0)
        assert self.s1.get_unbal_forces() == (Fx, Fy)

    def test_Ftest2(self):
        def Fx(t):
            return 0
        def Fy(t):
```

```
            return -9.81
        def Fz(t):
            return 1
        self.c = CircleT(1.0, 10.0, 0.5, 5.0)
        self.s1 = Scene(self.c, Fx, Fy, 0, 0)
        self.s1.set_unbal_forces(Fx, Fz)
        assert self.s1.get_unbal_forces() == (Fx, Fz)

    def test_get_shape1(self):
        assert self.s1.get_shape() == self.c

    def test_get_shape2(self):
        assert self.s2.get_shape() == self.t

    def test_get_init_velo1(self):
        assert self.s1.get_init_velo() == (0, 0)

    def test_get_init_velo12(self):
        assert self.s2.get_init_velo() == (10, -10)

    def test_set_shape1(self):
        self.s1.set_shape(self.b)
        assert self.s1.get_shape() == self.b

    def test_undo_set1(self):
        self.s1.set_shape(self.c)
        assert self.s1.get_shape() == self.c

    def test_set_shape2(self):
        self.s2.set_shape(self.b)
        assert self.s2.get_shape() == self.b

    def test_undo_set2(self):
        self.s2.set_shape(self.t)
        assert self.s2.get_shape() == self.t

    def test_set_init_velo1(self):
        self.s1.set_init_velo(25, 12)
        assert self.s1.get_init_velo() == (25, 12)

    def test_undo_init_velo1(self):
        self.s1.set_init_velo(0, 0)
        assert self.s1.get_init_velo() == (0, 0)

    def test_set_init_velo2(self):
        self.s2.set_init_velo(300, 3)
        assert self.s2.get_init_velo() == (300, 3)

    def test_undo_init_velo2(self):
        self.s2.set_init_velo(10, -10)
        assert self.s2.get_init_velo() == (10, -10)

    def test_sim_t(self):
        t, wsol = self.s3.sim(10, 10)
        exp_t = [0.0, 1.111, 2.222, 3.333, 4.444, 5.555, 6.666, 7.777, 8.888, 10.0]

        diff_t = []
        for i in range(0, len(t)):
            diff_t.append(t[i] - exp_t[i])

        norm_t = 0
        for i in range(0, len(diff_t)):
            if abs(i) > norm_t:
                norm_t = i

        norm_exp = 0
        for i in range(0, len(exp_t)):
            if abs(i) > norm_exp:
                norm_exp = i

        assert (norm_t / norm_exp) == approx(1, rel=1e-3)

    # I couldn't figure out how to effectively test this method :(
    # def test_sim_wsol(self):
    #     t, wsol = self.s3.sim(10, 10)
    #     exp_wsol = [[1, 10, 0, 0],
    #                 [1, 3.94444444, 0, -10],
    #                 [1, -14.22222222, 0, -21.8],
    #                 [1, -44.5, 0, -32.7],
    #                 [1, -86.88888889, 0, -43.6],
```

```
#                    [1, -141.38888889, 0, -54.5],
#                    [1, -208, 0, -65.4],
#                    [1, -286.72222222, 0, -76.3],
#                    [1, -377.55555556, 0, -87.2],
#                    [1, -480.5, 0, -98.1]]

#       wsol_all = []
#       for i in wsol:
#           for j in range(len(i)):
#               wsol_all.append(i[j])

#       exp_all = []
#       for i in exp_wsol:
#           for j in range(len(i)):
#               exp_all.append(i[j])

#       diff_wsol = []
#       for i in wsol_all:
#           diff_wsol.append(wsol_all[i] - exp_all[i])

#       norm_wsol = 0
#       for i in range(0, len(diff_wsol)):
#           if abs(i) > norm_wsol:
#               norm_wsol = i

#       norm_exp = 0
#       for i in range(0, len(exp_all)):
#           if abs(i) > norm_exp:
#               norm_exp = i

#       assert (norm_wsol / norm_exp) == approx(1, rel=1e-3)
```

L Code for Partner's CircleT.py

```
## @file CircleT.py
#  @author Samia Anwar
#  @brief Contains a CircleT type to represent a circle with a mass on a plane
#  @date February 2, 2021

from Shape import Shape

## @brief CircleT is used to represent a circle on a plane with a mass
#  to calculate its moment of inertia


class CircleT(Shape):
    ## @brief constructor for class CircleT, represents circles as their
    #  cartesian coordinates of the center, their radius, and their mass
    #  @param x is a real number representation of the x coordinate of the
    #  centre of the circle
    #  @param y is a real number representation of the y coordinate of the centre of
    #  the circle
    #  @param r is a real number representation of the radius of the circle
    #  @param m is a real number representation of the mass of the circle
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    #  @throws ValueError raised if either the mass or radius is defined to be less than
    #  or equal to zero
    def __init__(self, x, y, r, m):
        if (m <= 0 or r <= 0):
            raise ValueError
        self.x = x
        self.y = y
        self.r = r
        self.m = m

    ## @brief returns the x coordinate of the center of the circle
    #  @return real number representation of x-coordinate of the centre of the circle
    def cm_x(self):
        return self.x

    ## @brief returns the y coordinate of the center of the circle
    #  @return real number representation of x-coordinate of the centre of the circle
    def cm_y(self):
        return self.y

    ## @brief returns the mass of the circle
    #  @return real number representation of mass of the circle
    def mass(self):
        return self.m

    ## @brief returns the mass of the circle based on a formula using the initialised
    #  mass and radius values
    #  @return real number representation of moment of inertia of the circle
    def m_inert(self):
        return (self.m * self.r * self.r) / 2
```

M Code for Partner's TriangleT.py

```python
## @file TriangleT.py
#  @author Samia Anwar
#  @brief Contains a TriangleT type to represent an equilateral triangle
#  with a mass on a plane
#  @date Feb 2/2021

from Shape import Shape

## @brief TriangleT is used to represent an equilateral Triangle on a plane with a mass
#  to eventually calculate its moment of inertia when called on


class TriangleT(Shape):
    ## @brief constructor for class TriangleT, represents a triangle as its
    #  cartesian coordinates of the center, its side length, and its mass
    #  @param x is a real number representation of the x coordinate of the
    #  centre of the triangle
    #  @param y is a real number representation of the y coordinate of the centre of
    #  the triangle
    #  @param s is a real number representation of all sides of the equilateral triangle
    #  @param m is a real number representation of the mass of the triangle
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    #  @throws ValueError raised if either the mass or side length is defined to be less than
    #  or equal to zero
    def __init__(self, x, y, s, m):
        if (not (s > 0 and m > 0)):
            raise ValueError
        self.x = x
        self.y = y
        self.s = s
        self.m = m

    ## @brief returns the x coordinate of the center of the triangle
    #  @return real number representation of x-coordinate of the centre of the triangle
    def cm_x(self):
        return self.x

    ## @brief returns the y coordinate of the center of the triangle
    #  @return real number representation of x-coordinate of the centre of the triangle
    def cm_y(self):
        return self.y

    ## @brief returns the mass of the triangle
    #  @return real number representation of mass of the triangle
    def mass(self):
        return self.m

    ## @brief returns the mass of the triangle based on a formula using the initialised
    #  mass and side length values
    #  @return real number representation of moment of inertia of the triangle
    def m_inert(self):
        return (self.m * self.s * self.s / 12)
```

N Code for Partner's BodyT.py

```python
## @file BodyT.py
#  @author Samia Anwar
#  @brief Contains a generic BodyT type which has properties of a Shape
#  @date Feb 2/2021

from Shape import Shape

##  @brief Objects of this class represent body of points with mass
#   cartesian placement of physical structures, their masses, and their moments of inertia


class BodyT(Shape):

    ## @brief Constructor method for class BodyT, initialises a Body from their
    #  x, y, and mass values
    #  @param x is the x-coordinates of an object on the cartesian plane, represented
    #  as a sequence of real numbers
    #  @param y is the y-coordinates of an object on the cartesian plane, represented
    #  as a sequence of real numbers
    #  @param m is the mass of each part of an object, represented as a sequence of real
    #  numbers, corresponding to the indices in the x and y lists
    #  @details the constructor method conducts calculations based on the given parameters
    #  to create a numerical self object corresponding to the moment of inertia of the whole
    #  object, the x-y coordinates of the centre of mass of the whole system and the mass of
    #  the whole system
    #  @throws ValueError if parameters are not sequences of the same length, and if members
    #  of sequence m are less than or equal to zero
    def __init__(self, x, y, m):
        if not (len(x) == len(y) and len(x) == len(m)):
            raise ValueError
        for i in m:
            if i <= 0:
                raise ValueError
        self.cmx = self.__cm__(x, m)
        self.cmy = self.__cm__(y, m)
        self.m = self.__sum__(m)
        self.moment = self.__mmom__(x, y, m) - self.m * (self.cmx ** 2 + self.cmy ** 2)

    ## @brief returns the value of the x coordinate of the object's center of mass
    #  @return a real number representation of the x-coordinate
    def cm_x(self):
        return self.cmx

    ## @brief returns the value of the y coordinate of the object's center of mass
    #  @return a real number representation of the y-coordinate of the object's center of mass
    def cm_y(self):
        return self.cmy

    ## @brief returns the value of the total mass of the object
    #  @return a real number representation of the total mass of the object
    def mass(self):
        return self.m

    ## @brief returns the value of the object's moment of inertia
    #  @return real number representation of the object's total moment of inertia
    def m_inert(self):
        return self.moment

    ## @brief Calculates the sum of values in a list of real numbers
    #  @param a is the list composed of real numbers to be added together
    #  @return a real number representation of the sum of the list
    def __sum__(self, a):
        s = 0
        for u in a:
            s = s + u
        return s

    ## @brief Calculates the center of mass of an object on one cartesian axis
    #  @param a is the list composed of real number masses corresponding to parts of an object
    #  @param z is the list composed of real number x-coordinates  corresponding
    #  to parts of an object
    #  @return a real number representation of the center of mass of an object in parts
    def __cm__(self, z, a):
        s = 0
        for i in range(len(a)):
            s = s + (z[i] * a[i])
```

```
    return (s / self.__sum__(a))

## @brief Calculates some real number value in the moment of inertia equation
#  @param x is the list of x-coordinates of the parts of a system of objects
#  @param y is the list of y-coordinates of the parts of a system of objects
#  @param m is the list of masses of the parts of a system of objects
#  @returns real number representaion of the sum of m * (x^2 + y^2) at each
#  index of the corresponding lists
def __mmom__(self, x, y, m):
    s = 0
    for i in range(len(m)):
        s = s + m[i] * (x[i] * x[i] + y[i] * y[i])
    return s
```

O Code for Partner's Scene.py

```python
## @file Scene.py
#  @author Samia Anwar
#  @brief Generic module to represent forces and velocity on an object
#  @date Feb 2, 2021
#  @details Simulates motion of an object based on force and initial velocity

from Shape import Shape
from scipy.integrate import odeint

## @brief This module takes in a Shape object and generates seqeuences of numbers to simulate
#  its motion given a force acting upon it and its initial velocity


class Scene(Shape):
    ## @brief constructor for class Scene, represents the motion acted upon a given shape
    #  @param ds is a Shape object defined elsewhere in the code and contains x-y coordinates
    #  for center of mass, a total mass and a moment of inertia
    #  @param dfx is the formula for the x-direction force acted upon the object
    #  @param dfy is the formula for the y-direction force acted upon the object
    #  @param dvx is a real number representation of the starting velocity of the object
    #  in the x-plane
    #  @param dvy is a real number representation of the starting velocity of the object
    #  in the y-plane
    #  @details the units of these real number representations is at the discretion
    #  of the user and is no way controlled or represented in this python implementation
    def __init__(self, ds, dfx, dfy, dvx, dvy):
        self.s = ds
        self.fx = dfx
        self.fy = dfy
        self.vx = dvx
        self.vy = dvy

    ## @brief Returns the shape object associated with the Scene
    #  @return shape object and all of its parametres
    def get_shape(self):
        return self.s

    ## @brief returns the force equations in the x and y direction
    #  @return x and y direction force equations as python functions
    def get_unbal_forces(self):
        return self.fx, self.fy

    ## @brief returns the x and y direction values of velocity
    #  @return x and y direction real number values of velocity
    def get_init_velo(self):
        return self.vx, self.vy

    ## @brief changes the shape specified in the Scene
    #  @param s_new is an Shape object containing the specified parameters
    def set_shape(self, s_new):
        self.s = s_new

    ## @brief changes the x and y direction force functions specified in the Scene
    #  @param fx_n is a python function representing the new x-direction force function
    #  @param fy_n is a python function representing the new y-direction force function
    def set_unbal_forces(self, fx_n, fy_n):
        self.fx = fx_n
        self.fy = fy_n

    ## @brief changes the x and y direction initial velocities specified in the Scene
    #  @param vx_n is a real number velocity values representing the new x-direction velocity
    #  @param vy_n is a real number velocity values representing the new y-direction velocity
    def set_init_velo(self, vx_n, vy_n):
        self.vx = vx_n
        self.vy = vy_n

    ## @brief Integrates the given functions based on initial velocity and a step value
    #  @param tf is a real number used in the numerator of the calculations
    #  @param nsteps is a natural number used in the denominator of the calculations
    #  @assumption assume that nsteps is never equal to one
    #  @return two sequences of real numbers
    def sim(self, tf, nsteps):
        t = []
        for i in range(nsteps):
            t.append((i * tf) / (nsteps - 1))
        return t, odeint(self.__ode__, [self.s.cm_x(), self.s.cm_y(), self.vx, self.vy], t)
```

```python
## @brief Generates an array for computation in odeint method in sim()
#  @param w is a sequence with 4 values
#  @param t is a real number used as an input for the given force equations
#  @return an array with 4 elements inside
def __ode__(self, w, t):
    return [w[2], w[3], self.fx(t) / self.s.mass(), self.fy(t) / self.s.mass()]
```