2ME3 Assignment 2

Justin Staples (staplejw)

February 27, 2017

Contents

1 Testing the Modules **1**
- 1.1 Point Module Tests . . . 1
- 1.2 Line Module Tests . . . 1
- 1.3 Circle Module Tests . . . 2
- 1.4 Deque Module Tests . . . 2
- 1.5 Results with Original circleADT.py . . . 3
- 1.6 Results with Partners circleADT.py . . . 3

2 Discussion **3**
- 2.1 Learning Achievements . . . 4
- 2.2 Original Modules . . . 4
- 2.3 Partners Modules . . . 4
- 2.4 Specification . . . 5
- 2.5 PyUnit . . . 5
- 2.6 Specification of totalArea . . . 5
- 2.7 Specification of averageRadius . . . 6
- 2.8 Critique of Circle Module . . . 6
- 2.9 Disjoint . . . 7

A Code for pointADT.py **8**

B Code for lineADT.py **9**

C Original Code for circleADT.py **10**

D Code for deque.py **11**

E Code for testCircleDeque.py **13**

F Code for Makefile **18**

G Partners Code for circleADT.py **19**

The purpose of this report is to document the results of creating and testing the following Python modules: `pointADT.py`, `lineADT.py`, `circleADT.py` and `deque.py`. The report contains an analysis of the original test results, as well as the results from a secondary set of tests, using a `circleADT.py` that was provided by a partner. All source files (original and partner's), as well as the corresponding Makefile, can be found in the appendices (A - G).

1 Testing the Modules

The main idea behind selecting test cases was to ensure that the tests show the core functionality of each method. For methods that are simple or methods that have relatively trivial implementations, only a single test case was chosen to show that the method behaves exactly as intended in a common case. However, for many methods, extra 'edge' cases were added (as specified by the interface). As well, all of methods in the `deque.py` module that contain exceptions were tested to show that they raise the correct exception in the right circumstance.

It is assumed that the person using the module would understand the different data types available and would not attempt to input incorrect data types into the constructor methods or the double ended queue. For this reason, the modules are not considered robust and no effort was made to test this kind of behaviour. The tests just focus on ensuring that the methods give the correct output, assuming the correct input type is given.

The tests were conducted using PyUnit, a unit testing framework that allows test classes to be created. Within these classes, individual tests can be created to test one method at a time.

1.1 Point Module Tests

The methods from this module are all relatively straightforward. As a set up routine, a number of instances of PointT were created and used to test the methods. The getter methods (`xcrd` and `ycrd`), as well as the `dist` method were just tested once to show basic functionality. However, there were a few extra cases added for the `rot` method to show that it functions well for a variety of angles. Tests were conducted that used positive angles, negative angles, an angle of 0 radians and an angle greater than π radians.

1.2 Line Module Tests

As a set up routine, a number of instances of LineT were created and used to test the methods. Similar to the tests in `pointADT.py`, the getter methods for this module (`beg`

and `end`) are just tested once to show they behave properly. `len` is tested twice, once with a line of non-zero length (where the beginning and ends points are different) and another with a line of zero length (where the beginning points are the same). Here, `mdpt` is tested in a very similar way, using one typical line and another where the length is zero. The `rot` method for this module is tested in the exact same way as in the pointADT.py module, using a variety of different angles.

1.3 Circle Module Tests

As a set up routine, a number of instances of CircleT were created and used to test the methods. The set up routine for this test class also includes a few anonymous functions that are, in particular, used for testing `force`. The getter methods for this module (`cen` and `rad`) are tested just once, as is `area` to show basic behaviour. There were multiple test cases considered for `intersect`. There are multiple configurations for pairs of circles, and it was decided to pick tests that show one pair that do intersect and one pair that do not. The first pair of circles intersect at a point, where as the second pair of circles do not intersect at all. A simple test for `connection` was conducted to check that the correct instance of LineT was returned. The `force` method was tested using the anonymous functions defined in the set up routine, alongside a few instances of CircleT. Here, two tests were conducted that show that the method returns the correct value for gravitational force. The two tests choose different circles with varying radii and distances to each other. Each test is also conducted with a different lambda function to show that it works in a variety of circumstances.

1.4 Deque Module Tests

This module has the exact same set up routine as `circleADT.py`. All of the methods for changing or accessing the front or back of the queue (push, pop, get) were tested multiple times. One test with just one cirlce in the queue and another with three circles in the queue to show that the methods work for any amount of circles. All of these methods were also tested with a full or empty queue where appropriate to make sure that they raise the correct exceptions. The `size` method was tested with an empty queue to make sure that it returned a size of 0. It was also tested with a non-zero number of circles. This was to show that the size method returns the correct value for any number of elements in the queue. Because `disjoint` analyzes all the elements in the queue, there are a lot of possible cases to test. First, the method was tested with an empty queue to show that it raises the EMPTY exception. Next, it was tested with just a single circle. Based on the specification of `disjoint` the method should return `True` for a queue with just one circle. A further exlpanation of this output is given in the discussion section. Two more

tests were conducted for this method, one where all the circles in the queue were disjoint and one where they were not. Since the method returns a boolean, it is essential to have a test case for each boolean value. For `sumFx` one simple test was done using one of the anonymous functions given in the set up routine. It used three circles to check that the method gives the correct sum of forces in the x direction. More tests could have been added to this method because there are so many possible inputs. It would have been good to test circles that are not just on the x-axis, but rather all over the xy-plane. It also would have been good to test a symmetrical set of circles to show that the gravitational force sums to zero. In the interest of time, these tests were not implemented.

1.5 Results with Original circleADT.py

PyUnit considers each method one test. Over the four modules, 25 total tests were designed to test all of the methods at least once. In many cases, methods were given more than one test.

The main result of running the testing module is that the modules (`lineADT.py`, `pointADT.py`, `circleADT.py` and `deque.py`) passed all 25 tests.

Overall, the tests that were selected were just aiming to show that the functions work in basic circumstances. The tests were designed to be thorough, but not exhaustive. In the interest of time, most methods were not tested to the extreme or given many rare cases. They were just tested to show that they met the specification. To be even more certain that the modules are correct, more tests could have been selected that test the methods more rigorously.

1.6 Results with Partners circleADT.py

The main result of running the testing module with the partners circle module is that the modules (`lineADT.py`, `pointADT.py`, `circleADT.py` and `deque.py`) passed all 25 tests.

2 Discussion

This section outlines some of the shortcomings of the original program code, the partners implementation, as well as the specification of the modules given in the assignment. Some possible improvements are mentioned, along with a brief discussion of the lessons learned during this exercise.

2.1 Learning Achievements

I really enjoyed working on this assignment because it introduced me to a completely new programming paradigm, functional programming. Learning about anonymous functions, as well as the different functional methods like filter, map, and reduce were really interesting to me. It felt really powerful to learn about these and made me realize just how many different ways you can program. I hope to expand on this knowledge in the future. I also enjoyed learning about PyUnit. It proved very useful for testing the methods and I like how it gave a summary of the output, saying which tests passed or failed and why. On top of these things, I would say that my overall skill in programming in python has improved. I have learned more about how to handle exceptions in python, and I think my understanding of import statements has also grown.

2.2 Original Modules

I am confident in the code that I have written and I do not have many negatives to comment on, but I do think I could have done some things differently in the future. All of the getter methods for my modules return a reference to the objects instance variables. This is not a problem when the instance variables are primitives like floats, but in `lineADT.py` or `circleADT.py`, the instance variables are objects. My methods return a reference to these objects instead of returning a new object. This could create some strange behaviour if the objects instance variables are changed at some point, which could be undesirable. The safe thing to do would be to return a new object.

Overall, I would say that my implementations were concise and efficient, but I do think some improvements could have been made. For the `disjoint` method in `deque.py`, I would have liked to use map/filter/reduce to give the output, but decided to implement it as a nested for loop. I do think my current implementation is understandable/readable, but it is a bit lengthy.

My `deque.py` module has a global variable at the top of the file, G. I was using this for some preliminary tests earlier on in the development and forgot to remove it. It does not really create any problems, but it is unused and unnecessary in this module.

2.3 Partners Modules

I did not notice any major problems with my partner's code. I would say that our style and implementations were extremely similar. My partner did not use a hard text wrap of 80 columns for their code, which some could argue is not the best for readability. As well, the partner's circle module has import statements for both the line and point module, which I consider a little bit redundant because the line module already imports all the

methods from point itself, so it is not necessary to have this. These are extremely small details and I would say that overall the partner's code is very well done.

2.4 Specification

I actually really loved having the formal specifiation. It made everything extremely clear. I always knew what to do and there were basically zero ambiguities. In Assignment 1, there were many instances where I had to make a design decision myself (like deciding how to rank or which definition of intersect to use), which is bound to lead to inconsistencies in the implementations. With the formal specification, it was perfectly clear. It was basically automatic. Having the formal specification made everything easier because I just followed the instructions and did what I had to do. There was no confusion because there was no other way to do it other than what the specification asked for. It made the mapping from specification to code quite mechanical, which I enjoyed. I didn't have any real problems with the specification. At first I found the notion of local functions a little bit confusing and was not sure how to handle these or if they were necessary. However, after Dr. Smith's clariciation that they did not have to be implemented, it made a lot more sense. Overall, having a strong understanding of discrete mathematics made the specification very enjoyable to read and work with.

2.5 PyUnit

I definitely preferred using PyUnit compared to writing my own custom test module. PyUnit allows you to create different test classes, where each class represents a module that you would like to test. Within these test classes, you can have many different methods that tests the methods you wrote. This structure is extremely useful for organizing your tests. It is very convenient to have each PyUnit test method correspond to one of the methods you are testing. You can have many tests for one method all within one testing method. Another major advantage of using PyUnit is that it prints out a summary of the test results. The summary shows you which test pass and fail, and for the ones that fail, it gives you a reason why. It also shows the time that it took to run all of the tests.

PyUnit gives you access to a wide range of functions for testing, like `assertTrue` and `assertEquals`. It even has built in functions for testing to see if certain exceptions are raised (`assertRaises`)! This was extremely useful for this assignment. These methods are part of a library of functions that are very reliable. They are standard and reusable.

2.6 Specification of totalArea

The specification for the output of `totalArea` is as follows:

$$out := +(i : \mathbb{N} | i \in ([0..|s| - 1]) : s[i].\text{area}())$$

2.7 Specification of averageRadius

The specification for the output of `averageRadius` is as follows:

$$out := \frac{+(i : \mathbb{N} | i \in ([0..|s| - 1]) : s[i].\text{rad}())}{\text{Deq\_size}()}$$

2.8 Critique of Circle Module

I will give a short critique of the circle module based on module qualities discussed in class.

- Consistent: This module is consistent. The naming conventions and overall style of the module are common for all methods and are easy to follow.
- Essential: This module is actually not essential. `connection` creates a line between the centers of the two circles. `len`, from the line module, could be used to find the length of this line. Once this is known, the length could be compared against the radii of the circles to determine intersection. Because `intersect` could be provided by a combination of other routines, the module is not essential.
- General: It could be argued that the interface is not that general because it only offers one way to do each service. In python, there are many different ways to read from a file, which means that it is general in this sense, because the designers have tried to anticipate many different ways users might want to read from a file. In the circle module, there is only one way to access each service, so it could be said that it is not very general in that way.
- Minimal: This module is minimal because each access routine only offers one independent service.
- Opaque: This module is not entirely opaque. It is implemented in python, and in python, variables that you declare private are not truly private, meaning that the user still has access to these.

2.9 Disjoint

The specification for `disjoint` is a universal quantification. Because there exist no values for i and j that satisfy the range, this is a quantification over an empty set. In this case, the function should just return the identity of the quantifier operator. The identity of conjuction (and) is True, and so the function should output True when there is only one circle in the queue. This is similar to summing all the values in an empty set, which would return the identity of +, which is 0. This makes sense because what the disjoint function asks is "are there any pairs of circles that are overlapping?". Because there are no pairs at all, the answer is no, and so the set is disjoint. This is the same result as calculated by my code.

A Code for pointADT.py

```
## @file pointADT.py
#  @title pointADT
#  @author Justin Staples
# staplejw
# 001052815

import math

## @brief This class represents a 2D point.
#  @details This class represents a two-dimensional point
#  as an abstract data type. The point is characterized
#  with an 'x' and 'y' coordinate.
class PointT:
    ## @brief This is the constructor for PointT
    #  @details The constructor takes in two parameters. 'x'
    #  and 'y' serve as the coordinates for the point.
    #  @param x x-coordinate of the point.
    #  @param y y-coordinate of the point.
        def __init__(self, x, y):
                self.xc = x
                self.yc = y

        ## @brief Getter method for the x-coordinate.
        #  @return Returns the x-coordinate of an intance of PointT.
        def xcrd(self):
                return self.xc

        ## @brief Getter method for the y-coordinate.
        #  @return Returns the y-coordinate of an intance of PointT.
        def ycrd(self):
                return self.yc

        ## @brief A method for calculating the distance between two instances
        #  of PointT.
        #  @details Uses the x-coordinates and y-coordinates of both points,
        #  along with the pythagorean theorem, to find the distance between
        #  points.
        #  @param p Distance requires two instance of PointT to calculate the
        #  distance, and p serves as the second point.
        #  @return The distance bewteen the two points.
        def dist(self, p):
                return math.sqrt((self.xc - p.xcrd())**2 + (self.yc - p.ycrd())**2)

        ## @brief Rotates the point around the origin (0, 0)
        #  @details Uses the transition matrix for rotation to rotate the
        #  @param phi The angle of rotation, given in radians.
        def rot(self, phi):
                temp1 = self.xc
                temp2 = self.yc
                self.xc = math.cos(phi)*temp1 - math.sin(phi)*temp2
                self.yc = math.sin(phi)*temp1 + math.cos(phi)*temp2
```

B Code for lineADT.py

```
## @file lineADT.py
#  @title lineADT
#  @author Justin Staples
# staplejw
# 001052815

from pointADT import *

## @brief This class represents a line.
#  @details This class represents a two-dimensional line
#  as an abstract data type. The line is characterized
#  with two points, a beginning and and end.
class LineT:
        ## @brief This is the constructor for LineT
    #  @details The constructor takes in two parameters. 'p1'
    #  and 'p2' serve as the beginning and end of the line.
    #  @param p1 The beginning point of the line.
    #  @param p2 The end point of the line.
        def __init__(self, p1, p2):
                self.b = p1
                self.e = p2

        ## @brief Getter method for the beginning point.
        #  @return Returns the beginning point of an instance of LineT.
        def beg(self):
                return self.b

        ## @brief Getter method for the end point.
        #  @return Returns the end point of an instance of LineT.
        def end(self):
                return self.e

        ## @brief A method for calculating the length of a line.
        #  @details Uses the 'dist' method defined in pointADT.
        #  @return The length of the line.
        def len(self):
                return self.beg().dist(self.end())

        ## @brief A method for finding the midpoint of a line.
        #  @details Uses the 'dist' method defined in pointADT.
        #  @return The length of the line.
        def mdpt(self):
                xmid = (self.beg().xcrd() + self.end().xcrd())/2.0
                ymid = (self.beg().ycrd() + self.end().ycrd())/2.0
                return PointT(xmid, ymid)

        ## @brief This method will rotate a line some angle around the origin.
        #  @details This method rotates the beginning and end points of the line,
        #  effectively rotating the whole line.
        #  @param phi The angle of rotation, given in radians.
        def rot(self, phi):
                self.beg().rot(phi)
                self.end().rot(phi)
```

C Original Code for circleADT.py

```
## @file circleADT.py
#  @title circleADT
#  @author Justin Staples
# staplejw
# 001052815

from lineADT import *


## @brief This class represents a circle.
#  @details This class represents a circle
#  as an abstract data type. The circle is characterized
#  with a center point and radius.
class CircleT:
        ## @brief This is the constructor for CircleT
    #  @details The constructor takes in two parameters. 'cin'
    #  is the point that represents the center of the circle
    #  and 'rin' serve as the beginning and end of the line.
    #  @param cin The center point of the circle.
    #  @param rin The radius of the circle.
        def __init__(self, cin, rin):
                self.c = cin
                self.r = rin

        ## @brief Getter method for the center point.
        #  @return Returns the center point of an instance of CircleT.
        def cen(self):
                return self.c

        ## @brief Getter method for the radius.
        #  @return Returns the radius of an instance of CircleT.
        def rad(self):
                return self.r


        ## @brief A method for calculating the area of a circle.
        #  @details Uses the formula for area of a circle.
        #  @return Returns the area of the circle.
        def area(self):
                return math.pi * self.r ** 2

        ## @brief A method for determining if two circles intersect.
        #  @details Uses the distance between the circles and
        #  the radii of the circle to determine if two circles
        #  share any common points.
        #  @param ci This circle acts as the circle to compare to
        #  test for intersection.
        #  @return Returns a boolean value. True if the circles
        #  intersect and False otherwise.
        def intersect(self, ci):
                return self.cen().dist(ci.cen()) <= (self.rad() + ci.rad())

        ## @brief A method for creating a connection line between
        #  the centers of two circles.
        #  @details Creates a line that where the beginning and end
        #  points are determined by the center points of each circle.
        #  @param ci This cirlce will be the circle that is connected to self.
        #  @return Returns an instance of LineT, where the beginning
        #  and end points are given by the centers of the circle.
        def connection(self, ci):
                return LineT(self.cen(), ci.cen())

        ## @brief This method creates a function that can be used
        #  to calculate the graviational force between two circles.
        #  @param f This function takes in a real as an argument and
        #  returns a real. This function is used to replace the 1/r**2
        #  term in universational law of gravitation, letting the user
        #  create custom functions for gravitational force.
        #  @return Returns the gravitational force between two instances
        #  CircleT, using the area of the circles as masses and the input
        #  function f.
        def force(self, f):
                return lambda x: self.area() * x.area() * f(self.connection(x).len())
```

D Code for deque.py

```
from circleADT import *

G = 6.672e-11

## @brief This class represents a double ended queue for cirlces.
#  @details This class represents a double ended queue for circles,
#  as an abstract object. The maximum size for the queue is 20 circles.
class Deq:

        # the maximum size of the queue is 20 circles.
        MAX_SIZE = 20

        # the state variable that is used to represent the queue is a list
        s = []

        ## @brief This is the initializer method the deque.
        #  It initializes the queue to an empty queue.
        @staticmethod
        def init():
                Deq.s = []

        ## @brief This method will push an instance of CircleT to the
        #  the back of the queue.
        #  @param c This instance of CircleT will be pushed to the back of the queue.
        @staticmethod
        def pushBack(c):
                if (len(Deq.s) == Deq.MAX_SIZE):
                        raise FULL("The queue is full, maximum size cannot be exceeded")
                back = [c]
                Deq.s = Deq.s + back

        ## @brief This method will push an instance of CircleT to the
        #  the front of the queue.
        #  @param c This instance of CircleT will be pushed to the front of the queue.
        @staticmethod
        def pushFront(c):
                if (len(Deq.s) == Deq.MAX_SIZE):
                        raise FULL("The queue is full, maximum size cannot be exceeded")
                front = [c]
                Deq.s = front + Deq.s

        ## @brief This method will remove the instance of CircleT
        #  at the very back of the queue.
        @staticmethod
        def popBack():
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                Deq.s = Deq.s[0:len(Deq.s) - 1]

        ## @brief This method will remove the instance of CircleT
        #  at the very front of the queue.
        @staticmethod
        def popFront():
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                Deq.s = Deq.s[1:len(Deq.s)]

        ## @brief This method can be used to access the back
        #  element in the queue.
        #  @return Returns the backmost element in the queue.
        @staticmethod
        def back():
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                return Deq.s[len(Deq.s) - 1]

        ## @brief This method can be used to access the front
        #  element in the queue.
        #  @return Returns the frontmost element in the queue.
        @staticmethod
        def front():
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                return Deq.s[0]

        ## @brief This method gives the current size of the
```

```
        #  queue, by returning the number of elements in the queue.
        #  @return Returns the number of elements in the queue.
        @staticmethod
        def size():
                return len(Deq.s)

        ## @brief This method determines if all the circle
        #  currently in the queue are disjoint from eachother.
        #  @details Considers all possible pairs of circles from the
        #  queue and if none of the pairs intersect, then the whole set
        #  is considered disjoint.
        #  @return Returns a boolean value. True if all the circles are
        #  disjoint and False otherwise.
        @staticmethod
        def disjoint():
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                for i in range(0, len(Deq.s)):
                        for j in range(0, len(Deq.s)):
                                if i != j:
                                        if Deq.s[i].intersect(Deq.s[j]):
                                                return False
                return True

        ## @brief This method sums up all the x components
        #  of all the gravitational forces contributed by the
        #  circles in the queue on the front circle (other than the
        #  front circle).
        #  @param f This is a function that takes in a real and returns
        #  a real. It passes this function to the force() method from
        #  circleADT.py and uses the output of this function to find
        #  the forces and the xcomponents from that.
        #  @return Returns a single real, which is a sum of all the
        #  xcomponents of all forces acting on the front circle.
        @staticmethod
        def sumFx(f):
                if (len(Deq.s) == 0):
                        raise EMPTY("The queue is empty")
                xdelta = lambda ci: ci.c.xc - Deq.s[0].c.xc
                distance = lambda ci: ci.connection(Deq.s[0]).len()
                xcomp = lambda ci: ci.force(f)(Deq.s[0])*xdelta(ci)/distance(ci)
                return reduce(lambda x, y: x + y, map(xcomp, Deq.s[1:len(Deq.s)]), 0)

## @brief This in an exception class called FULL.
#  It is thrown when the queue has reached its max size
#  and certain methods are called that would attempt to
#  violate the state invariant.
class FULL(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return str(self.value)

## @brief This in an exception class called EMPTY.
#  It is thrown when the queue has a size of 0 and certain
#  methods are called. Many methods cannot return a value
#  when the queue does not have any elements in it.
class EMPTY(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return str(self.value)
```

E Code for testCircleDeque.py

```
## @file testCircleDeque.py
#  @title Testing Module
#  @author Justin Staples
#  @details This module consists of four classes, each
#  dedicated to testing one of the pointADT, lineADT,
#  circleADT and deque modules. The tests are done using
#  PyUnit, a unit testing frame work that tests one
#  method at a time. There is at least one test for every
#  method. A more thorough description of the results of
#  testing and rationale behind test cases will be given
#  in the final lab report.
# staplejw
# 001052815

from deque import *
import unittest

## @brief This class tests all the methods in the pointADT
#  module.
class pointTests(unittest.TestCase):

        def setUp(self):
                self.p1 = PointT(1, 2)
                self.p2 = PointT(0, 0)
                self.p3 = PointT(1, 0)

        def tearDown(self):
                self.p1 = None
                self.p2 = None
                self.p3 = None

        def test_xcrd(self):
                self.assertTrue(self.p1.xcrd() == 1)

        def test_ycrd(self):
                self.assertTrue(self.p1.ycrd() == 2)

        def test_dist(self):
                self.assertTrue(self.p1.dist(self.p2) == math.sqrt(5))

        def test_rot(self):
                # no rotation
                x = PointT(self.p3.xcrd(), self.p3.ycrd())
                x.rot(0)
                self.assertAlmostEqual(x.xcrd(), 1, None, None, 0.001)
                self.assertAlmostEqual(x.ycrd(), 0, None, None, 0.001)
                # positive angle
                x = PointT(self.p3.xcrd(), self.p3.ycrd())
                x.rot(math.pi/4)
                self.assertAlmostEqual(x.xcrd(), math.sqrt(2)/2, None, None, 0.001)
                self.assertAlmostEqual(x.ycrd(), math.sqrt(2)/2, None, None, 0.001)
                # negative angle
                x = PointT(self.p3.xcrd(), self.p3.ycrd())
                x.rot(-math.pi/2)
                self.assertAlmostEqual(x.xcrd(), 0, None, None, 0.001)
                self.assertAlmostEqual(x.ycrd(), -1, None, None, 0.001)
                # greater than pi
                x = PointT(self.p3.xcrd(), self.p3.ycrd())
                x.rot(5*math.pi/4)
                self.assertAlmostEqual(x.xcrd(), -math.sqrt(2)/2, None, None, 0.001)
                self.assertAlmostEqual(x.ycrd(), -math.sqrt(2)/2, None, None, 0.001)

## @brief This class tests all the methods in the lineADT
#  module.
class lineTests(unittest.TestCase):

        def setUp(self):
                self.l1 = LineT(PointT(0, 0), PointT(0, 0))
                self.l2 = LineT(PointT(-1, 2), PointT(3, 4))
                self.l3 = LineT(PointT(0, 0), PointT(1, 4))
                self.l4 = LineT(PointT(-3, -3), PointT(-3, 3))
                self.l5 = LineT(PointT(0, 1), PointT(1, 0))

        def tearDown(self):
                self.l1 = None
                self.l2 = None
```

```
            self.l3 = None
            self.l4 = None
            self.l5 = None

        def test_beg(self):
            self.assertTrue(self.l1.beg().xcrd() == 0 and self.l1.beg().ycrd() == 0)
            self.assertTrue(self.l2.beg().xcrd() == -1 and self.l2.beg().ycrd() == 2)


        def test_end(self):
            self.assertTrue(self.l1.end().xcrd() == 0 and self.l1.end().ycrd() == 0)
            self.assertTrue(self.l2.end().xcrd() == 3 and self.l2.end().ycrd() == 4)

        def test_len(self):
            self.assertTrue(self.l1.len() == 0)
            self.assertTrue(self.l2.len() == math.sqrt(20))

        def test_mdpt(self):
            self.assertTrue(self.l1.mdpt().xcrd() == 0 and self.l1.mdpt().ycrd() == 0)
            self.assertTrue(self.l2.mdpt().xcrd() == 1 and self.l2.mdpt().ycrd() == 3)

        def test_rot(self):
            # no rotation
            self.l2.rot(0)
            self.assertAlmostEqual(self.l2.beg().xcrd(), -1, None, None, 0.001)
            self.assertAlmostEqual(self.l2.beg().ycrd(), 2, None, None, 0.001)
            self.assertAlmostEqual(self.l2.end().xcrd(), 3, None, None, 0.001)
            self.assertAlmostEqual(self.l2.end().ycrd(), 4, None, None, 0.001)
            # positive angle
            self.l3.rot(math.pi/2)
            self.assertAlmostEqual(self.l3.beg().xcrd(), 0, None, None, 0.001)
            self.assertAlmostEqual(self.l3.beg().ycrd(), 0, None, None, 0.001)
            self.assertAlmostEqual(self.l3.end().xcrd(), -4, None, None, 0.001)
            self.assertAlmostEqual(self.l3.end().ycrd(), 1, None, None, 0.001)
            # negative angle
            self.l4.rot(-math.pi/2)
            self.assertAlmostEqual(self.l4.beg().xcrd(), -3, None, None, 0.001)
            self.assertAlmostEqual(self.l4.beg().ycrd(), 3, None, None, 0.001)
            self.assertAlmostEqual(self.l4.end().xcrd(), 3, None, None, 0.001)
            self.assertAlmostEqual(self.l4.end().ycrd(), 3, None, None, 0.001)
            # greater than pi
            self.l5.rot(5*math.pi/4)
            self.assertAlmostEqual(self.l5.beg().xcrd(), 0.7071, None, None, 0.001)
            self.assertAlmostEqual(self.l5.beg().ycrd(), -0.7071, None, None, 0.001)
            self.assertAlmostEqual(self.l5.end().xcrd(), -0.7071, None, None, 0.001)
            self.assertAlmostEqual(self.l5.end().ycrd(), -0.7071, None, None, 0.001)

## @brief This class tests all the methods in the circleADT
#  module.
class circleTests(unittest.TestCase):

        def setUp(self):
            self.c1 = CircleT(PointT(0, 0), 1)
            self.c2 = CircleT(PointT(3, 0), 1)
            self.c3 = CircleT(PointT(6, 0), 1)
            self.c4 = CircleT(PointT(0, 3), 1)
            self.c5 = CircleT(PointT(3, 4), 2)
            self.c6 = CircleT(PointT(8, 0), 1)
            self.f1 = lambda r: 1/r**2
            self.f2 = lambda r: r + 1

        def tearDown(self):
            self.c1 = None
            self.c2 = None
            self.c3 = None
            self.c4 = None
            self.c5 = None
            self.c6 = None

        def test_cen(self):
            self.assertTrue(self.c5.cen().xcrd() == 3 and self.c5.cen().ycrd() == 4)

        def test_rad(self):
            self.assertTrue(self.c5.rad() == 2)

        def test_area(self):
            self.assertAlmostEqual(self.c1.area(), 3.14159, None, None, 0.001)

        def test_intersect(self):
            # c3 and c6 do intersect at a point
```

```
                self.assertTrue(self.c3.intersect(self.c6))
                # c1 and c2 do not intersect, so assertFalse is used
                self.assertFalse(self.c1.intersect(self.c2))

        def test_connection(self):
                l = self.c3.connection(self.c4)
                self.assertTrue(l.beg().xcrd() == 6 and l.beg().ycrd() == 0)
                self.assertTrue(l.end().xcrd() == 0 and l.end().ycrd() == 3)

        def test_force(self):
                self.assertAlmostEqual(self.c1.force(self.f1)(self.c6), 0.1542, None, None, 0.001)
                self.assertAlmostEqual(self.c1.force(self.f2)(self.c5), 236.8705, None, None, 0.001)

## @brief This class tests all the methods in the deque
#  module.
class dequeTests(unittest.TestCase):

        def setUp(self):
                self.c1 = CircleT(PointT(0, 0), 1)
                self.c2 = CircleT(PointT(3, 0), 1)
                self.c3 = CircleT(PointT(6, 0), 1)
                self.c4 = CircleT(PointT(0, 3), 1)
                self.c5 = CircleT(PointT(3, 4), 2)
                self.c6 = CircleT(PointT(8, 0), 1)
                self.f1 = lambda r: 1/r**2
                self.f2 = lambda r: r + 1

        def tearDown(self):
                self.c1 = None
                self.c2 = None
                self.c3 = None
                self.c4 = None
                self.c5 = None
                self.c6 = None

        def test_init(self):
                Deq.init()
                self.assertTrue(Deq.size() == 0)

        def test_pushBack(self):
                # pushBack tested with just one element in the queue
                Deq.init()
                Deq.pushBack(self.c1)
                self.assertTrue(Deq.back().cen().xcrd() == 0)
                self.assertTrue(Deq.back().cen().ycrd() == 0)
                self.assertTrue(Deq.back().rad() == 1)
                # same test, but with three circles in the queue
                Deq.init()
                Deq.pushBack(self.c3)
                Deq.pushBack(self.c4)
                Deq.pushBack(self.c5)
                self.assertTrue(Deq.back().cen().xcrd() == 3)
                self.assertTrue(Deq.back().cen().ycrd() == 4)
                self.assertTrue(Deq.back().rad() == 2)
                # filling the queue to test FULL exception
                Deq.init()
                for i in range(0, 20):
                        Deq.pushBack(self.c2)
                self.assertRaises(FULL, Deq.pushBack, self.c2)

        def test_pushFront(self):
                # pushFront tested with just one element in the queue
                Deq.init()
                Deq.pushFront(self.c1)
                self.assertTrue(Deq.front().cen().xcrd() == 0)
                self.assertTrue(Deq.front().cen().ycrd() == 0)
                self.assertTrue(Deq.front().rad() == 1)
                # same test, but with three circles in the queue
                Deq.init()
                Deq.pushFront(self.c3)
                Deq.pushFront(self.c4)
                Deq.pushFront(self.c5)
                self.assertTrue(Deq.front().cen().xcrd() == 3)
                self.assertTrue(Deq.front().cen().ycrd() == 4)
                self.assertTrue(Deq.front().rad() == 2)
                # filling the queue to test FULL exception
                Deq.init()
                for i in range(0, 20):
                        Deq.pushFront(self.c2)
                self.assertRaises(FULL, Deq.pushBack, self.c2)
```

```
def test_popBack(self):
        # push two elements at the back, then pop the back and check the first element
        Deq.init()
        Deq.pushBack(self.c2)
        Deq.pushBack(self.c6)
        Deq.popBack()
        self.assertTrue(Deq.back().cen().xcrd() == 3)
        self.assertTrue(Deq.back().cen().ycrd() == 0)
        self.assertTrue(Deq.back().rad() == 1)
        # popping an empty queue to test EMPTY exception
        Deq.init()
        self.assertRaises(EMPTY, Deq.popBack)

def test_popFront(self):
        # push two elements to the front, then pop the front and check the first element
        Deq.init()
        Deq.pushFront(self.c4)
        Deq.pushFront(self.c5)
        Deq.popFront()
        self.assertTrue(Deq.front().cen().xcrd() == 0)
        self.assertTrue(Deq.front().cen().ycrd() == 3)
        self.assertTrue(Deq.front().rad() == 1)
        # popping an empty queue to test EMPTY exception
        Deq.init()
        self.assertRaises(EMPTY, Deq.popFront)

def test_back(self):
        # push three circles and check the back element
        Deq.init()
        Deq.pushBack(self.c3)
        Deq.pushBack(self.c4)
        Deq.pushBack(self.c5)
        self.assertTrue(Deq.back().cen().xcrd() == 3)
        self.assertTrue(Deq.back().cen().ycrd() == 4)
        self.assertTrue(Deq.back().rad() == 2)
        # asking for the back element of an empty queue gives EMPTY exception
        Deq.init()
        self.assertRaises(EMPTY, Deq.back)

def test_front(self):
        # push three circles and check the front element
        Deq.init()
        Deq.pushFront(self.c3)
        Deq.pushFront(self.c4)
        Deq.pushFront(self.c5)
        self.assertTrue(Deq.front().cen().xcrd() == 3)
        self.assertTrue(Deq.front().cen().ycrd() == 4)
        self.assertTrue(Deq.front().rad() == 2)
        # asking for the front element of an empty queue gives EMPTY exception
        Deq.init()
        self.assertRaises(EMPTY, Deq.front)

def test_size(self):
        Deq.init()
        self.assertTrue(Deq.size() == 0)
        Deq.pushFront(self.c2)
        Deq.pushBack(self.c1)
        self.assertTrue(Deq.size() == 2)

def test_disjoint(self):
        # disjoint with 0 circles should raise EMPTY exception
        Deq.init()
        self.assertTrue(EMPTY, Deq.disjoint)
        # disjoint with 1 circle should return True
        Deq.pushBack(self.c1)
        self.assertTrue(Deq.disjoint())
        # disjoint with multiple disjoint circles should return True
        Deq.pushBack(self.c2)
        Deq.pushBack(self.c3)
        Deq.pushBack(self.c4)
        Deq.pushBack(self.c5)
        self.assertTrue(Deq.disjoint())
        # now adding a circle that breaks disjoint condition should return False
        Deq.pushBack(self.c6)
        self.assertFalse(Deq.disjoint())

def test_sumFx(self):
        # sumFx with empty queue should raise EMPTY exception
        Deq.init()
```

```
                self.assertRaises(EMPTY, Deq.sumFx, self.f1)
                Deq.pushFront(self.c1)
                Deq.pushBack(self.c2)
                Deq.pushBack(self.c5)
                self.assertAlmostEqual(Deq.sumFx(self.f1), 2.0441, None, None, 0.001)

if __name__ == '__main__':
        unittest.main()
```

F Code for Makefile

```
PY = python
PYFLAGS =
DOC = doxygen
DOCFLAGS =
DOCCONFIG = doxConfig

SRC = src/testCircleDeque.py

test:
	$(PY) $(PYFLAGS) $(SRC)

doc:
	$(DOC) $(DOCFLAGS) $(DOCCONFIG)
	cd latex && $(MAKE)

clean:
	rm -rf html
	rm -rf latex
```

G Partners Code for circleADT.py

```
## @file circleADT.py
#  @title circleADT
#  @author Jenny Feng Chen (fengchej)
#  @date 2/19/2017

from pointADT import*
from lineADT import*
import math

## @brief This class represents a circle.
#  @details This class represent a circle with a center represented by PointT and a radius.
class CircleT(object):

    ## @brief This is a constructor for CircleT.
    #  @details This is a constructor for CircleT that takes two parameters and assigns one to center
        of circle and the radius of circle.
    #  @param cin is an instance of PointT object.
    #  @param rin is a positive real number.
    def __init__(self, cin, rin):
        self.c = cin
        self.r = float(rin)

    ## @brief This method returns the center point of the circle.
    #  @return the center of the circle.
    def cen(self):
        return self.c

    ## @brief This method returns the radius of the circle.
    #  @return the radius of the circle.
    def rad(self):
        return self.r

    ## @brief This method determines the area of the circle.
    #  @return the area of the circle.
    def area(self):
        return self.r**2 * math.pi

    ## @brief This method check whether two circles intersect.
    #  @details This method treat circles as filled objects. The set of points in each circle
    #  includes the boundary (closed sets).
    #  @param ci is an instance of the CircleT.
    #  @return true if the circles intersect; false if not.
    def intersect(self,ci):
        intersect = (self.cen().xcrd()-ci.cen().xcrd())**2 + (self.cen().ycrd()-ci.cen().ycrd())**2 <=
            (self.rad() + ci.rad())**2
        return intersect

    ## @brief This method creates a new line between the center of two circles.
    #  @param ci is an instance of the circle.
    #  @return line which is a new instance of the LineT object.
    def connection(self, ci):
        return LineT(self.cen(), ci.cen())

    ## @brief This method calculates the gravitational force between two circles.
    #  @details This method takes a function and return a function. It is basically calculates the
        gravitational force between two circles using the universal gravitational constant.
    #  @param f is a function that takes a real number an returns a real number.
    #  @return f1 a function that takes an instance of CircleT and returns a real number.
    def force(self, f):
        f1 = lambda x: x.area()*self.area()* f(self.connection(x).len())
        return f1
```