2AA4 Assignment Two Report

Susan Fayez
fayezs

February 25 2017

1 My Code

1.1 pointADT.py

```
##  @file pointADT.py
#   @title pointADT
#   @author Susan Fayez
#   @date 2/14/2017

##  @brief importing from libraries needed
import math

##  @brief A class for a point ADT
#   @details This class represents a point with x and y coordinates
class PointT():

    ##  @brief Constructor for PointT
    #   @details Only allows numeric types for x and y. If anything
    #   else entered, a warning is printed
    #   @param x The x-coordinate of the point
    #   @param y The y-coordinate of the point
    def __init__(self, x, y):
        t1 = type(x)
        t2 = type(y)
        if((t1 == int or  t1 == long or t1 == float) and (t2 == int
        or t2 == long or t2 == float)):
            self.xc = x
            self.yc = y
        else:
            self.xc = None
            self.yc = None
            print "Invalid type entered."
```

```
    ##  @brief Getter method for the x-coordinate
    #   @return The x-coordinate of the point
    def xcrd(self):
        return self.xc

    ##  @brief Getter method for the y-coordinate
    #   @return The y-coordinate of the point
    def ycrd(self):
        return self.yc

    ##  @brief Calculates the distance between two points
    #   @param p The other point we want to find the distance from
    #   @return The distance between the two points
    def dist(self, p):
        return math.sqrt((self.xc - p.xcrd())**2 + (self.yc - p.ycrd())**2)

    ##  @brief Rotates the point about the origin
    #   @param angle The angle the point is rotated by in radians
    def rot(self, angle):
        self.xc = (math.cos(angle) - math.sin(angle)) * self.xc
        self.yc= (math.cos(angle) + math.sin(angle)) * self.yc
```

1.2 lineADT.py

```
##  @file lineADT.py
#   @title lineADT
#   @author Susan Fayez
#   @date 2/14/2017

##  @brief importing from libraries needed
from pointADT import *

##  @brief A class for a line ADT
#   @details This class represents a line with two points (type PointT)
class LineT():

    ##  @brief Constructor for LineT
    #   @param p1 The beginning point of the line
    #   @param p2 The end point of the line
    def __init__(self, p1, p2):
        if(isinstance(p1, PointT) and isinstance(p2, PointT)):
            self.b = p1
            self.e = p2
        else:
            print "Invaild type entered."
```

```
        self.b = None
        self.e = None

    ##  @brief Private method that calculates the average of two numbers
    #   @param x1 The first number
    #   @param x2 The second number
    #   @return The average of the two numbers


    ##  @brief Getter method for the beginning point of the line
    #   @return The beginning point of the line
    def beg(self):
        return self.b

    ##  @brief Getter mothod for the end point of the line
    #   @return The end point of the line
    def end(self):
        return self.e

    ##  @brief Returns the length of the line
    #   @details uses dist() from pointADT.py to calculate the distance
    #   between the end point and the beginning point.
    #   @return the length of the line
    def len(self):
        return self.b.dist(self.e)

    ##  @brief returns the midpoint of the line
    #   @details Implements __avg() to find the mid x-coordinate and mid
    #   y-coordinate and uses them to create a new point to return
    #   @return The midpoint of the line, type PointT
    def mdpt(self):
        def __avg(x1, x2):
            return (x1 + x2) / 2.0
        x = __avg(self.b.xcrd(), self.e.xcrd())
        y = __avg(self.b.ycrd(), self.e.ycrd())
        return PointT(x, y)

    ##  @brief rotates the line about the origin
    #   @details Implements rot() from pointADT.py to rotate the
    #   beginning point and the end point.
    #   @param angle The angle by which to rotate the line, in radians
    def rot(self, angle):
        self.b.rot(angle)
        self.e.rot(angle)
```

1.3 circleADT.py

```
##  @file circleADT.py
#   @title circleADT
#   @author Susan Fayez
#   @date 2/14/2017

##  @brief importing from libraries needed
from lineADT import *

##  @brief A class for a circle ADT
#   @details This class represents a circle with a point for the center
#   and a number for the radius
class CircleT():

    ##  @brief Constructor for CircleT
    #   @details Constructor accepts a point for the centre of the circle
    #   and a number for the radius. Throws an exception if zero is
    #   entered, takes the absolute value of the radius and prints a
    #   warning if a negative radius is entered.
    #   @param cin The center of the radius of type PointT
    #   @param rin The numeric value of the radius
    def __init__(self, cin, rin):
        t = type(rin)
        if(rin == 0):
            print "Radius cannot be zero."
            self.c = None
            self.r = None
        if(isinstance(cin, PointT) and (t == int or t == long or t ==
        float) and rin != 0):
            self.c = cin
            self.r = abs(rin)
            if(rin < 0):
                print("Radius can't be negative. Absolute value of
                input taken.")
        else:
            print "Invalid type entered."
            self.c = None
            self.r = None

    ##  @brief Private method that returns a lambda function that
    #   calculates the gravitational constant divided by the input
    #   squared
    #   @details Not implemented in this class, but here to show what
    #   the lambda function for force should look like
    def __f(self):
```

```
    return lambda r: (6.672e-11) / (r ** 2)

##  @brief Getter method for the center point of the circle
#   @return The center point of the circle, type PointT
def cen(self):
    return self.c

##  @brief Getter method for the radius of the circle
#   @return The radius of the circle
def rad(self):
    return self.r

##  @brief Calculates the area of the circle.
#   @return The area of the circle
def area(self):
    return math.pi * self.r ** 2

##  @brief Determines whether or not the circle intersects with
#   a given circle
#   @details Function accepts an instance of CircleT and
#   determines whether or not the circles intersect by checking
#   if the distance between the centers of the circles is greater
#   than the sum of the radii.
#   @param circleTwo An instance of CircleT
#   @return a boolean value on whether or not the circles intersect
def intersect(self, ci):

    ##  @brief Checks whether a given point is inside a given circle
    #   @details Uses dist() from pointADT.py to check whether
    #   the distance from the given point to the center of the
    #   given circle is less than the radius of the circle. If it
    #   is, the method returns True, if not it returns False. Not
    #   implemented in this class, but can be used to check
    #   intersection of circles
    #   @param p instance of PointT
    #   @param c instance of CircleT
    #   @return Boolean depending on whether the point is in
    #   the circle
    def insideCircle(p, c):
        if (p.dist(c.cen()) <= c.rad()):
            return True
        else:
            return False
    if (self.c.dist(ci.cen()) <= (self.r + ci.rad())):
        return True
    else:
```

```
            return False

    ##  @brief Creates a line connecting the centers of the
    #   circle and an input circle
    #   @return A line connnecting the centers of the circles,
    #   type LineT
    def connection(self, ci):
        return LineT(self.c, ci.cen())

    ##  @brief Calculates the force of gravity between two circles
    #   @details Accepts a lambda function and multiplies the result of
    #   entering the distance between the two circles in the lambda
    #   function by the areas of the circles.
    #   @param f lambda function that calculates the gravitational
    #   constant divided by the input squared
    #   @return The force of gravity between the two circles
    def force(self, f):
        return lambda x: self.area() * x.area() *
        f(self.connection(x).len())
```

1.4 deque.py

```
##  @file deque.py
#   @title DequeCircleModule
#   @author Susan Fayez
#   @date 2/15/2017

##  @brief importing from libraries needed
from circleADT import *

##  @brief A class for a deque abstract object
#   @details This class represents a deque by storing it in a list s
class Deq:

    ##  @brief declaring constants and state variables
    MAX_SIZE = 20

    s = []

    ##  @brief Constructor for the deque
    #   @details Initializes the deque before any other access
    #   program is called
    @staticmethod
    def init():
        Deq.s = []
```

```
##  @brief Adds an element to the back of the deque
#   @details raises an error if user attempts to push an
#   element onto the deque when it is full
@staticmethod
def pushBack(c):
    if (len(Deq.s) >= Deq.MAX_SIZE):
        raise Full("Maxium size exceeded")

    Deq.s.append(c)

##  @brief Adds an element to the front of the deque
#   @details raises an error if user attempts to push an
#   element onto the deque when it is full
@staticmethod
def pushFront(c):
    if (len(Deq.s) >= Deq.MAX_SIZE):
        raise Full("Maxium size exceeded")

    Deq.s.insert(0, c)

##  @brief Deletes an element from the back of the deque
#   @details raises an error if user attempts to pop an
#   element from the deque when it is empty
@staticmethod
def popBack():
    if(len(Deq.s) == 0):
        raise Empty("No elements in the deque")

    del Deq.s[len(Deq.s) - 1]

##  @brief Deletes an element from the front of the deque
#   @details raises an error if user attempts to pop an
#   element from the deque when it is empty
@staticmethod
def popFront():
    if(len(Deq.s) == 0):
        raise Empty("No elements in the deque")

    del Deq.s[0]

##  @brief Retrieves an element from the back of the deque
#   @details raises an error if user attempts to retrieve an
#   element from the deque when it is empty
@staticmethod
def back():
    if(len(Deq.s) == 0):
```

```
            raise Empty("No elements in the deque")

        return Deq.s[len(Deq.s) - 1]

    ##  @brief Retrieves an element from the front of the deque
    #   @details raises an error if user attempts to retrieve an
    #   element from the deque when it is empty
    @staticmethod
    def front():
        if(len(Deq.s) == 0):
            raise Empty("No elements in the deque")

        return Deq.s[0]

    ##  @brief Retrieves the size of the deque
    @staticmethod
    def size():
        return len(Deq.s)

    ##  @brief Determines whether or not the circles in the
    #   dueque intersect
    #   @details Raises an error if user attempts to call this
    #   access program when the deque is empty. If not, compares
    #   each element in the deque to one another. Returns false
    #   if any elements intersect one another. Returns true if no
    #   circles in the set intersect.
    @staticmethod
    def disjoint():
        if(len(Deq.s) == 0):
            raise Empty("No elements in the deque")

        for i in range(len(Deq.s)):
            for j in range(len(Deq.s)):
                if (Deq.s[i].intersect(Deq.s[j]) and i != j):
                    return False
        return True

    ##  @brief Calculates the sum of the gravitational forces
    #   between the circles in the deque and the first circle in
    #   the x direction
    #   @details Raises an error if user attempts to call this
    #   access program when the deque is empty. If not, implements
    #   __Fx() to calculate the sum of the gravitational forces in
    #   the x direction
    @staticmethod
    def sumFx(f):
```

```
        if(len(Deq.s) == 0):
            raise Empty("No elements in the deque")

        ##  @brief Local function returns the x-component of
        #   a force between two circles
        #   @param F The force between the two circles
        #   @param ci The first circle, type CircleT
        #   @param cj The second circle, type CircleT
        def xcomp(F, ci, cj):
            return F * ((ci.cen().xcrd() - cj.cen().xcrd()) /
            ci.connection(cj).len())

        ##  @brief Local function returns the x-component of
        #   a force between two circles
        #   @details Implements __xcomp to find the x component of
        #   the force by calling the force method from circleADT.py
        #   for the force and entering the circles as the other
        #   two parameters
        #   @param f The lambda function needed by force() to calculate
        #   the gravitational force
        #   @param ci The first circle
        #   @param cj The second circle
        def Fx(f, ci, cj):
            return xcomp(ci.force(f)(cj), ci, cj)

        sum = 0;
        for i in range(len(Deq.s)):
            if (i != 0):
                sum += Fx(f, Deq.s[i], Deq.s[0])
        return sum

##  @brief Class defines the deque full exception
class Full(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)

##  @brief Class defines the deque empty exception
class Empty(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)
```

1.5 testCircleDeque.py

```
##  @file testCircleDeque.py
#   @title testCircleDeque
#   @author Susan Fayez
#   @date 2/18/2017

##  @brief importing from libraries needed
import unittest
from deque import *

##  @brief Class contains test cases for PointT
class PointTests(unittest.TestCase):

    ##  @brief Runs before test cases to initialize points
    def setUp(self):
        p1 = PointT(1, -5)
        p2 = PointT(4.76, -0.663)
        p3 = PointT(24L, 5L)
        p4 = PointT("hello", False)

        self.points = [p1, p2, p3, p4]

    ##  @brief Runs after test cases to remove points
    def tearDown(self):
        self.points = None

    ##  @brief Tests xcrd()
    def testXcrd(self):
        self.assertTrue(self.points[0].xcrd() == 1)
        self.assertTrue(self.points[1].xcrd() == 4.76)
        self.assertTrue(self.points[2].xcrd() == 24L)
        self.assertTrue(self.points[3].xcrd() == None)

    ##  @brief Tests ycrd()
    def testYcrd(self):
        self.assertTrue(self.points[0].ycrd() == -5)
        self.assertTrue(self.points[1].ycrd() == -0.663)
        self.assertTrue(self.points[2].ycrd() == 5L)
        self.assertTrue(self.points[3].ycrd() == None)

    ##  @brief Tests dist()
    def testDist(self):
        self.assertAlmostEqual(self.points[0].dist(self.points[1]),
        5.7399624563232114, None, None, 0.1)
        self.assertAlmostEqual(self.points[0].dist(self.points[2]),
```

```
            25.079872407968907, None, None, 0.1)
            self.assertAlmostEqual(self.points[1].dist(self.points[2]),
            20.056100543226247, None, None, 0.1)

        ##  @brief Tests rot()
        #   @details Rotates each point and checks whether the new
        #   x and y coordinates
        #   are the expected values
        def testRot(self):
            self.points[0].rot(3 * math.pi)
            self.points[1].rot(0)
            self.points[2].rot(-math.pi / 2.0)

            self.assertAlmostEqual(self.points[0].xcrd(),
            -1.0000000000000004, None, None, 0.1)
            self.assertTrue(self.points[1].xcrd() == 4.76)
            self.assertTrue(self.points[2].xcrd() == 24.0)

    ##  @brief Class contains tests for LineT
    class LineTests(unittest.TestCase):

        ##  @brief Runs before test cases to initialize lines
        def setUp(self):
            p1 = PointT(1, -5)
            p2 = PointT(4.76, -0.663)
            p3 = PointT(24L, 5L)
            p4 = PointT(-27.4, 2L)
            p5 = PointT(54, 0.234)
            p6 = PointT(8.223, -8L)

            l1 = LineT(p1, p2)
            l2 = LineT(p3, p4)
            l3 = LineT(p5, p6)
            l4 = LineT("world", complex(8, 1))
            l5 = LineT(p1, p1)

            self.lines = [l1, l2, l3, l4, l5]

        ##  @brief Runs after test cases to remove lines
        def tearDown(self):
            self.lines = None

        ##  @brief Tests beg()
        #   @details Compares the x and y coordinates of the beginning
        #   point to the expected values
        def testBeg(self):
```

```
        self.assertTrue(self.lines[0].beg().xcrd() == 1 and
        self.lines[0].beg().ycrd() == -5)
        self.assertTrue(self.lines[1].beg().xcrd() == 24L and
        self.lines[1].beg().ycrd() == 5L)
        self.assertTrue(self.lines[2].beg().xcrd() == 54 and
        self.lines[2].beg().ycrd() == 0.234)
        self.assertIsNone(self.lines[3].beg())

    ##  @breif Tests end()
    #   @details Comapres the x and y coordinates of the end point
    #   to the expected values
    def testEnd(self):
        self.assertTrue(self.lines[0].end().xcrd() == 4.76 and
        self.lines[0].end().ycrd() == -0.663)
        self.assertTrue(self.lines[1].end().xcrd() == -27.4 and
        self.lines[1].end().ycrd() == 2L)
        self.assertTrue(self.lines[2].end().xcrd() == 8.223 and
        self.lines[2].end().ycrd() == -8L)
        self.assertIsNone(self.lines[3].end())

    ##  @brief Tests len()
    def testLen(self):
        self.assertAlmostEqual(self.lines[0].len(), 5.7399624563232114,
        None, None, 0.1)
        self.assertAlmostEqual(self.lines[1].len(), 51.487474204897644,
        None, None, 0.1)
        self.assertAlmostEqual(self.lines[2].len(), 46.511638167237244,
        None, None, 0.1)

    ##  @brief Tests mdpt()
    #   @details Compares the x and y coordinates of the midpoint
    #   to the expected values
    def testMdpt(self):
        self.assertAlmostEqual(self.lines[0].mdpt().xcrd(), 2.88,
        None, None, 0.1)
        self.assertAlmostEqual(self.lines[0].mdpt().ycrd(), -2.8315,
        None, None, 0.1)

        self.assertAlmostEqual(self.lines[1].mdpt().xcrd(), -1.7,
        None, None, 0.1)
        self.assertAlmostEqual(self.lines[1].mdpt().ycrd(), 3.5,
        None, None, 0.1)

        self.assertAlmostEqual(self.lines[2].mdpt().xcrd(), 31.1115,
        None, None, 0.1)
        self.assertAlmostEqual(self.lines[2].mdpt().ycrd(), -3.883,
```

```
            None, None, 0.1)

            self.assertTrue(self.lines[4].mdpt().xcrd() == 1)
            self.assertTrue(self.lines[4].mdpt().ycrd() == -5)

        ##  @brief Tests rot()
        #   @details Rotates each point then compares the x and y
        #   coordinates of the beginning and end points to the expected values
        def testRot(self):
            self.lines[0].rot(math.pi / 4)
            self.lines[1].rot(0)
            self.lines[2].rot(-50)

            self.assertAlmostEqual(self.lines[0].beg().xcrd(),
            1.11022302463e-16, None, None, 0.1)
            self.assertAlmostEqual(self.lines[0].end().xcrd(),
            5.28466159722e-16, None, None, 0.1)
            self.assertAlmostEqual(self.lines[0].beg().ycrd(),
            -7.07106781187, None, None, 0.1)
            self.assertAlmostEqual(self.lines[0].end().ycrd(),
            -0.937623591853, None, None, 0.1)

            self.assertTrue(self.lines[1].beg().xcrd() == 24L)
            self.assertTrue(self.lines[1].end().xcrd() == -27.4)
            self.assertTrue(self.lines[1].beg().ycrd() == 5L)
            self.assertTrue(self.lines[1].end().ycrd() == 2L)

            self.assertAlmostEqual(self.lines[2].beg().xcrd(),
            37.9399234386, None, None, 0.1)
            self.assertAlmostEqual(self.lines[2].end().xcrd(),
            5.77740723028, None, None, 0.1)
            self.assertAlmostEqual(self.lines[2].beg().ycrd(),
            0.287197766434, None, None, 0.1)
            self.assertAlmostEqual(self.lines[2].end().ycrd(),
            -9.81872705757, None, None, 0.1)

    ##  @brief Class contains tests for CircleT
    class CircleTests(unittest.TestCase):

        ##  @brief Runs before test cases to initialize circles
        def setUp(self):
            p1 = PointT(0, 0)
            p2 = PointT(0, 10)
            p3 = PointT (-80.662, -45L)
            p4 = PointT (-15.234, 50L)
```

```
        c1 = CircleT(p1, 6)
        c2 = CircleT(p1, -9)
        c3 = CircleT(p2, 4)
        c4 = CircleT(p3, 2.726)
        c5 = CircleT(p3, 2L)
        c6 = CircleT(p4, -0.342)
        c7 = CircleT(p1, 0)
        c8 = CircleT("heck", [])

        self.circles = [c1, c2, c3, c4, c5, c6, c7, c8]

    ##  @brief Runs after test cases to remove circles
    def tearDown(self):
        self.circles = None

    ##  @brief Tests cen()
    #   @details Compares the x and y coordinates of the center
    #   point to the expected values
    def testCen(self):
        self.assertTrue(self.circles[0].cen().xcrd() == 0)
        self.assertTrue(self.circles[0].cen().ycrd() == 0)

        self.assertTrue(self.circles[5].cen().xcrd() == -15.234)
        self.assertTrue(self.circles[5].cen().ycrd() == 50L)

        self.assertIsNone(self.circles[6].cen())

        self.assertIsNone(self.circles[7].cen())

    ##  @brief Tests rad()
    def testRad(self):
        self.assertTrue(self.circles[0].rad() == 6)
        self.assertTrue(self.circles[5].rad() == 0.342)
        self.assertIsNone(self.circles[6].rad())
        self.assertIsNone(self.circles[7].rad())

    ## @brief Tests area()
    def testArea(self):
        self.assertAlmostEqual(self.circles[1].area(),
        254.46900494077323, None, None, 0.1)
        self.assertAlmostEqual(self.circles[2].area(),
        50.26548245743669, None, None, 0.1)
        self.assertAlmostEqual(self.circles[4].area(),
        12.566370614359172, None, None, 0.1)

    ##  @brief Tests intersect()
```

```
    def testIntersect(self):
        self.assertTrue(self.circles[0].intersect(self.circles[1]))
        self.assertTrue(self.circles[0].intersect(self.circles[2]))
        self.assertFalse(self.circles[3].intersect(self.circles[5]))

    ##  @brief Tests connection()
    #   @details Creates lines for the connections and compares the
    #   x and y coordinates of the begining and end points to the
    #   expected values
    def testConnection(self):
        l1 = self.circles[0].connection(self.circles[1])
        l2 = self.circles[2].connection(self.circles[3])
        l3 = self.circles[4].connection(self.circles[5])

        self.assertTrue(l1.beg().xcrd() == 0)
        self.assertTrue(l1.beg().ycrd() == 0)
        self.assertTrue(l1.end().xcrd() == 0)
        self.assertTrue(l1.end().ycrd() == 0)

        self.assertTrue(l2.beg().xcrd() == 0)
        self.assertTrue(l2.beg().ycrd() == 10)
        self.assertTrue(l2.end().xcrd() == -80.662)
        self.assertTrue(l2.end().ycrd() == -45L)

        self.assertTrue(l3.beg().xcrd() == -80.662)
        self.assertTrue(l3.beg().ycrd() == -45L)
        self.assertTrue(l3.end().xcrd() == -15.234)
        self.assertTrue(l3.end().ycrd() == 50L)

    ##  @brief Tests force()
    #   @details Creates the lambda function and passes it to
    #   force for tests
    def testForce(self):
        f = lambda r: (6.672e-11) / (r ** 2)

        self.assertAlmostEqual(self.circles[2].force(f)(self.circles[1]),
        8.5341600731e-09, None, None, 0.1)
        self.assertAlmostEqual(self.circles[0].force(f)(self.circles[3]),
        2.06486568875e-11, None, None, 0.1)
        self.assertAlmostEqual(self.circles[4].force(f)(self.circles[5]),
        2.31540111708e-14, None, None, 0.1)

##  @brief Class contains tests for Deq
class DequeTests(unittest.TestCase):

    ##  @brief Runs before the test cases to initialize deque
```

```
def setUp(self):
    p1 = PointT(0, 0)
    p2 = PointT(0, 10)
    p3 = PointT (-80.662, -45L)
    p4 = PointT (-15.234, 50L)

    c1 = CircleT(p1, 6)
    c2 = CircleT(p1, -9)
    c3 = CircleT(p2, 4)
    c4 = CircleT(p3, 2.726)
    c5 = CircleT(p3, 2L)
    c6 = CircleT(p4, -0.342)

    self.circles = [c1, c2, c3, c4, c5, c6]

    Deq.init()

##  @brief Runs after test cases to remove deque
def tearDown(self):
    self.circles = None
    Deq.s = None

##  @brief Tests pushBack()
#   @details Pushes circles onto the deque. Creates an array for
#   the expected result and compares the deque to it
def testDeq_pushBack(self):
    for i in range(6):
        Deq.pushBack(self.circles[0])
        Deq.pushBack(self.circles[2])
        Deq.pushBack(self.circles[4])

    Deq.pushBack(self.circles[5])
    Deq.pushBack(self.circles[5])

    a = []

    for i in range(6):
        a.append(self.circles[0])
        a.append(self.circles[2])
        a.append(self.circles[4])

    a.append(self.circles[5])
    a.append(self.circles[5])

    self.assertTrue(Deq.s == a)
    with self.assertRaises(Full):
```

```
        Deq.pushBack(self.circles[3])

##  @brief Tests pushFront()
#   @details Pushes circles onto the deque. Creates an array
#   for the expected result and compares the deque to it
def testDeq_pushFront(self):
    for i in range(6):
        Deq.pushFront(self.circles[1])
        Deq.pushFront(self.circles[3])
        Deq.pushFront(self.circles[5])

    Deq.pushFront(self.circles[0])
    Deq.pushFront(self.circles[0])

    a = []

    for i in range(6):
        a.insert(0, self.circles[1])
        a.insert(0, self.circles[3])
        a.insert(0, self.circles[5])

    a.insert(0, self.circles[0])
    a.insert(0, self.circles[0])

    self.assertTrue(Deq.s == a)
    with self.assertRaises(Full):
        Deq.pushFront(self.circles[4])

##  @brief Tests popBack()
#   @details Pushes circles onto the deque. Pops from the
#   back once and creates an array for the expected result
#   and compares the deque to it
def testDeq_popBack(self):
    for i in range(6):
        Deq.pushBack(self.circles[0])
        Deq.pushBack(self.circles[2])
        Deq.pushBack(self.circles[4])

    Deq.popBack()

    a = []

    for i in range(6):
        a.append(self.circles[0])
        a.append(self.circles[2])
        a.append(self.circles[4])
```

```
        del a[len(a) - 1]

        self.assertTrue(Deq.s == a)

    ##  @brief Tests popFront()
    #   @details Pushes circles onto the deque. Pops from the
    #   front once and creates an array for the expected result
    #   and compares the deque to it
    def testDeq_popFront(self):
        for i in range(6):
            Deq.pushFront(self.circles[1])
            Deq.pushFront(self.circles[3])
            Deq.pushFront(self.circles[5])

        Deq.popFront()

        a = []

        for i in range(6):
            a.insert(0, self.circles[1])
            a.insert(0, self.circles[3])
            a.insert(0, self.circles[5])

        del a[0]

        self.assertTrue(Deq.s == a)

    ##  @brief Tests back()
    #   @details Pushes circles onto the deque. Creates an array
    #   for the expected result and compares the back values
    def testDeq_back(self):
        for i in range(6):
            Deq.pushBack(self.circles[0])
            Deq.pushBack(self.circles[2])
            Deq.pushBack(self.circles[4])

        a = []

        for i in range(6):
            a.append(self.circles[0])
            a.append(self.circles[2])
            a.append(self.circles[4])

        self.assertTrue(Deq.back() == a[len(a)-1])
```

```
##  @brief Tests front()
#   @details Pushes circles onto the deque. Creates an
#   array for the expected result and compares the front values
def testDeq_front(self):
    for i in range(6):
        Deq.pushFront(self.circles[1])
        Deq.pushFront(self.circles[3])
        Deq.pushFront(self.circles[5])

    a = []

    for i in range(6):
        a.insert(0, self.circles[1])
        a.insert(0, self.circles[3])
        a.insert(0, self.circles[5])

    self.assertTrue(Deq.front() == a[0])

##  @brief Tests back()
#   @details Pushes circles onto the deque. Creates an
#   array for the expected result and compares the sizes
def testDeq_size(self):
    for i in range(6):
        Deq.pushFront(self.circles[1])
        Deq.pushBack(self.circles[5])
        Deq.pushFront(self.circles[2])

    a = []

    for i in range(6):
        a.insert(0, self.circles[1])
        a.append(self.circles[5])
        a.insert(0, self.circles[2])

    self.assertTrue(Deq.size() == len(a))

##  @brief Tests disjoint()
#   @details Pushes circles that don't intersect onto the deque.
#   Checks disjoint() returns True. Pops all circles off the
#   deque. Pushes circles that do intersect onto the deque.
#   Checks that disjoint() returns True. Pops all but one circle.
#   Checks that disjoint() returns True. Pops the last circle
#   and checks that calling disjoint() raises an error
def testDeq_disjoint(self):
    Deq.pushFront(self.circles[0])
    Deq.pushBack(self.circles[3])
```

```
        Deq.pushFront(self.circles[5])

        self.assertTrue(Deq.disjoint())

        Deq.popBack()
        Deq.popBack()
        Deq.popBack()

        Deq.pushFront(self.circles[0])
        Deq.pushBack(self.circles[1])
        Deq.pushFront(self.circles[1])

        self.assertFalse(Deq.disjoint())

        Deq.popFront()
        Deq.popFront()

        self.assertTrue(Deq.disjoint())

        Deq.popFront()

        with self.assertRaises(Empty):
            Deq.disjoint()

    ##  @brief Tests sumFx()
    #   @details Defines the lambda function for force() and
    #   pushes circles onto the deque. Creates an array of
    #   expected values and computes the sum of the forces
    #   in the x direction. Compares result to sumFx()
    def testDeq_sumFx(self):
        f = lambda r: (6.672e-11) / (r ** 2)


        Deq.pushFront(self.circles[0])
        Deq.pushFront(self.circles[2])
        Deq.pushFront(self.circles[3])
        Deq.pushFront(self.circles[5])

        a = []

        a.insert(0, self.circles[0])
        a.insert(0, self.circles[2])
        a.insert(0, self.circles[3])
        a.insert(0, self.circles[5])

        sfx = 0
```

```
        Deq.sumFx(f)

        for i in range(len(a)):
            if (i != 0):
                sfx += a[i].force(f)(a[0]) * ((a[i].cen().xcrd() -
                a[0].cen().xcrd()) / a[i].connection(a[0]).len())
        self.assertTrue(sfx == Deq.sumFx(f))

##  @brief Runs the test cases
if __name__ == '__main__':
    unittest.main()
```

1.6 Makefile

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

SRC = src/testCirclesDeque.py

.PHONY: all test doc clean

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

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


all: test doc

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

2 Partner's Code

2.1 Original Code

```
#Michael Balas
#400023244
import pointADT
import lineADT
```

```
import math
## @file circleADT.py
#  @title CircleADT
#  @author Michael Balas
#  @date 2/2/2017

## @brief This class represents a circle ADT.
#  @details This class represents a circle ADT, with point cin (x, y)
#  defining the centre of the circle, and r defining its radius.
class CircleT(object):

        ## @brief Constructor for CircleT
        #  @details Constructor accepts one point and a number (radius)
        #  to construct a circle.
        #  @param cin is a point (the centre of the circle).
        #  @param rin is any real number (represents the radius
        #   of the circle).
        def __init__(self, cin, rin):
                self.c = cin
                self.r = rin

        ## @brief Returns the centre of the circle
        #  @return The point located at the centre of the circle
        def cen(self):
                return self.c

        ## @brief Returns the radius of the circle
        #  @return The radius of the circle
        def rad(self):
                return self.r

        ## @brief Calculates the area of the circle
        #  @return The area of the circle
        def area(self):
                return math.pi*(self.r)**2

        ## @brief Determines whether the circle intersects another circle
        #  @details This function treats circles as filled objects:
        #  circles completely inside other circles are considered
        #  as intersecting, even though their edges do not cross.
        #  The set of points in each circle includes the boundary
        #  (closed sets).
        #  @param ci Circle to test intersection with
        #  @return Returns true if the circles intersect; false if not
        def intersect(self, ci):
                xDist = self.c.xc - ci.c.xcrd()
```

```
            yDist = self.c.yc - ci.c.ycrd()
            centerDist = math.sqrt(xDist ** 2 + yDist ** 2)
            rSum = self.r + ci.rad()
            return rSum >= centerDist

    ## @brief Creates a line between the centre of two circles
    #  @details This function constructs a line beginning at
    #  the centre of the first circle, and ending at the centre
    #  of the other circle.
    #  @param ci Circle to create connection with
    #  @return Returns a new LineT that connects the centre of
    #   both circles
    def connection(self, ci):
            return lineADT.LineT(self.c, ci.cen())

    ## @brief Determines the force between two circles given
    #  some parameterized gravitational law
    #  @details This functions calculates the force between two
    #  circles of unit thickness with a density of 1 (i.e. the
    #  mass is equal to the area). Any expression can be substituted
    #  for the gravitational law, f(r), or G/(r**2).
    #  @param f Function that parameterizes the gravitational law.
    #  Takes the distance between the centre of the circles and
    #  can apply expressions to it (e.g. multiply the universal
    #  gravitation constant, G, by the inverse of the squared
    #   distance between the circles).
    #  @return Returns the force between two circles
    def force(self, f):
            return lambda x: self.area() * x.area() *
            f(self.connection(x).len())
```

2.2 Updated Code

```
#Michael Balas
#400023244

from lineADT import *
import math
## @file circleADT.py
#  @title CircleADT
#  @author Michael Balas
#  @date 2/2/2017

## @brief This class represents a circle ADT.
#  @details This class represents a circle ADT, with point
#  cin (x, y) defining the centre of the circle, and r defining
```

```
#  its radius.
class CircleT():

        ## @brief Constructor for CircleT
        #  @details Constructor accepts one point and a number
        #  (radius) to construct a circle.
        #  @param cin is a point (the centre of the circle).
        #  @param rin is any real number (represents the radius
        #   of the circle).
        def __init__(self, cin, rin):
                self.c = cin
                self.r = rin

        ## @brief Returns the centre of the circle
        #  @return The point located at the centre of the circle
        def cen(self):
                return self.c

        ## @brief Returns the radius of the circle
        #  @return The radius of the circle
        def rad(self):
                return self.r

        ## @brief Calculates the area of the circle
        #  @return The area of the circle
        def area(self):
                return math.pi*(self.r)**2

        ## @brief Determines whether the circle intersects
        #  another circle
        #  @details This function treats circles as filled
        #  objects: circles completely inside other circles are
        #  considered as intersecting, even though their edges do
        #  not cross. The set of points in each circle includes
        #  the boundary (closed sets).
        #  @param ci Circle to test intersection with
        #  @return Returns true if the circles intersect;
        #  false if not
        def intersect(self, ci):
                xDist = self.c.xc - ci.c.xcrd()
                yDist = self.c.yc - ci.c.ycrd()
                centerDist = math.sqrt(xDist ** 2 + yDist ** 2)
                rSum = self.r + ci.rad()
                return rSum >= centerDist

        ## @brief Creates a line between the centre of two circles
```

```
#  @details This function constructs a line beginning
#  at the centre of the first circle, and ending at the
#  centre of the other circle.
#  @param ci Circle to create connection with
#  @return Returns a new LineT that connects the centre
#  of both circles
def connection(self, ci):
        return LineT(self.c, ci.cen())

## @brief Determines the force between two circles given
#  some parameterized gravitational law
#  @details This functions calculates the force between
#  two circles of unit thickness with a density of 1 (i.e.
#  the mass is equal to the area). Any expression can be
#  substituted for the gravitational law, f(r), or G/(r**2).
#  @param f Function that parameterizes the gravitational
#  law. Takes the distance between the centre of the circles
#  and can apply expressions to it (e.g. multiply the
#  universal gravitation constant, G, by the inverse of the
#  squared distance between the circles).
#  @return Returns the force between two circles
def force(self, f):
        return lambda x: self.area() * x.area() *
        f(self.connection(x).len())
```

3 Testing

3.1 Test Cases

When choosing the test cases for PointT, I wanted to show that the constructor would accept numeric inputs of type int, long, and float and reject other types such as str and bool. I ran each method with the points to show that all the numeric worked and the edge case always returned None values. In testing the method rot(), I made sure to rotate the points by angles greater than pi, that are negative, and that are zero to ensure that they all work as well.

Once again, when choosing the test cases for LineT, I wanted to show that the constructor accepts inputs of type PointT (that were constructed with numeric types) and rejects other types (such as str and complex numbers). I also made sure to include a line that started and ended on the same point, which was only used in testing mdpt() to make sure it returned the one point. The rest of the

methods test each of the other lines, making sure they run properly and the edge case always returns None values. Again, for testing rot(), I made sure to rotate the points by angles greater than pi, that are negative, and that are zero to ensure that they all work as well.

Once again, when choosing the test cases for CircleT, I wanted to show that the constructor accepts inputs of types PointT (that were constructed with numeric types) and numeric types, while rejecting other types (such as str and list). I also wanted to show that the constructor rejects zero radius circles and accepts negative radius circles by taking the absolute value of the input. In selecting the circles, I wanted to have circles that overlap, circles that intersect at one point, and circles that don't intersect at all. For the getter methods, I tested them with two regular circles, making sure they return the right values, and the two edge case circles, making sure they return non values. For area(), connection(), and force(), I picked three regular circles to make sure they return the correct values. For intersect() I tested the circles that overlap, the circles that intersect at one point, and the circles that don't intersect at all.

For the test cases of Deque, I decided to use the same tests circles from CircleT, excluding the edge cases, as I thought it might be redundant and unnecessary to test the constructor again in that way. For testing pushBack(), popBack, pushFront(), and popFront(), I use each of the methods and compare the resulting deque to a list that contains the expected values. I also test that pushing to a full deque raises and error and popping from an empty deque raises an error. For testing back() and front(), I create a deque and use the methods and compare the result to the expected value. For testing disjoint(), I test it with a deque that is a disjoint set, a deque that is not a disjoint set, a deque with one element, and an empty deque. For testing sumFx(), I pushed all the circles (except for ones that shared a center point to avoid a divide by zero error) on to deque and compared the result to expected result.

3.2 My Results

My code passed each test without raising any errors. This is obviously the case, as the test cases were written with my exact code in mind. There are definitely test cases I could have included that my code would not have passed. An example of this would be entering non CircleT types into the deque, as my code does not raise an errors when this happens.

3.3 Partner Results

Upon my first attempt to run my partner's code, the program raised a lot of errors. I realized that this was due to my partner implementing his code with new style classes, while I implemented mine with with old style classes. In order to get a better comparison of our interpretations of the MIS, I edited his code to fix the inconsistencies (both versions are listed in the beginning of the report).

The test cases ran without raising any exceptions and 20 cases passed while 4 cases failed. The first failure was the test case for cen() of CircleT. This failed because my partner's code did not assign None values to invalid inputs into the constructor as I did. The second failure was the test case for intersect() of CircleT. This is likely because his code did not account for negative radius inputs, therefore throwing off the calculation for distances. The third failure was the test case for rad() of CircleT. Again, this was liekly due to his code not setting the absolute value of the input as the radius in the constructor. The final failure was the test case for disjoint() of Deque. Again, this was likely due to his code not accounting for negative radii, therefore throwing off the checks for intersect().

4 Shortcomings

The process of writing this report brought to my attention various mistakes in the submitted version of my code. Upon running my code again to compare to my partner's, I realized that the name of my source code in my makefile was incorrect. This was due to my error in misnaming testCircleDeque.py as

testCirclesDeque.py in the development stage. When I went to move the files into the submission directory, the spelling mistake was brought to my attention and I renamed the file, but neglected to update the name of the source code in the makefile.

Another mistake in my makefile is that the rule doc fails to compile the documentation. With this error I'm not exactly sure what went wrong, as I did include the rule doc, but it failed to compile the documentation each time I tried testing it.

A mistake I noticed in lineADT.py was in the method mdpt(). I defined the local function avg() as a private and local method when it only needed to be local. This was because I misinterpreted the MIS in the beginning and implemented it as a private method outside of mdpt() and when I moved it to be local, I forgot to remove the underscores.

Similarly, in deque.py, I refer to xcomp() and Fx() as private methods in the comments. Again, this is because I misinterpreted the MIS in the beginning, this time remembering to remove the underscores in the actual functions, but neglecting to update the comments.

One aspect of my code that wasn't exactly an error, but definitely could have been improved is my handling of edge case inputs for constructors. Instead of creating objects with None value attributes, a better implementation would be to simply raised an error. The reason I didn't do this is that the MIS specifies that the constructors raise no errors. Perhaps a more correct implementation would have been to not handle invalid inputs at all.

In running my partner's files, I also noticed that his area() test case for CircleT should have failed. His code does not take the absolute value of the radius and one of the circles for that test case has a negative radius. This means that my partner's code should return a negative area, thereby failing that test case. It passes, however, and I believe that this is due to the test case using self.assertAlmostEqual() to account for floating point numbers.

5 Reflection

This exercise was another lesson in the pitfalls of specifications. The slightest amount of vagueness when designing the specifications of a project leaves it open to different interpretations and implementations. This is exemplified in the differences between my and my partner's codes. Though neither of us violate the MIS, we had some major inconsistencies in our interpretations and implementations of the assignment, as discussed in the Testing section.

Surprisingly, the discrepancies between my code and my partner's code were significantly less in the first assignment than in this one. That being said, the first assignment was just as open, if not more open to inconsistencies. The fact that the first assignment yielded less inconsistencies could be due to the assignment being much simpler and sheer luck. This does show, however, that formal notation when creating specifications doesn't necessarily yield more consistent implementations. In fact, when developing the codes for each assignment, I actually preferred informal specifications, as it was much easier to interpret.

A huge asset to this assignment was the use of PyUnit over manual test cases. It was helpful in developing and testing the code as it was much easier to implement and is much more ogranized. With PyUnit, when a test case fails, it tells you exactly which assertion of which test case caused the failure, making fixing mistakes a lot simpler than searching for line numbers. Furthermore, a test case failure does not crash the entire program, it continues to test the other cases.

6 Discussion

6.1 Access Program Specifications

Deq_totalArea()

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

Deq_averageRadius()

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

6.2 Critiques

The in terms of the quality of the specification, Circle Module's interface does follow most of quality criteria. It is consistent in its naming conventions, ordering of parameters, and exception handling (in that it does not deal with them). The interface is also general in that it can be applied to a variety of different projects (games, etc..). It is minimal because none of the access methods have multiple uses. The interface can also be considered opaque, as any extra methods are to be kept private (though Python implementation does not fully support information hiding). Where the interface fails is that it is not essential. The method intersect() can be implemented with connection() from the Line Module.

6.3 Disjoint

The mathematical specification for Deq.disjoint() states that for all i (i being the set of numbers from 0 to the length of s minus one) and for all j (j being the set of numbers from zero to the length of s minus one) and for every combination of i and j where i and j are not equal, s[i] does not intersect s[j] (s being the deque of circles). With this definition, the output of the mathematical specification of Deq.disjoint() for a deque with one element must be true. This makes sense because the specification basically states that no circles in the deque intersect with other circles in the deque. Since there are no other circles, it must return true. This aligns with the output of my code.