Line Puzzle Classes

In [1]:
```python
# from cmu_cs3_utils import testFunction

class Line:
    label = None

    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.label = Line.label

    def __repr__(self):
        return (f'Line from ({self.x1},{self.y1}) to ({self.x2},{self.y2})' +
                f' with label={self.label}')

    def __eq__(self, other):
        return (isinstance(other, Line) and
                ((self.x1 == other.x1 and
                  self.y1 == other.y1 and
                  self.x2 == other.x2 and
                  self.y2 == other.y2) or
                 (self.x1 == other.x2 and
                  self.y1 == other.y2 and
                  self.x2 == other.x1 and
                  self.y2 == other.y1)))

class Puzzle:
    def __init__(self, lineList):
        self.lines = []
        self.points = set()
        for x1, y1, x2, y2 in lineList:
            newLine = Line(x1, y1, x2, y2)
            self.lines.append(newLine)
            self.points.add((x1, y1))
            self.points.add((x2, y2))
        self.lastLabelValue = 0
        self.currentPoint = None

    def getClosestPoint(self, mouseX, mouseY):
        closestPoint = None
        closestDist = 10**7
        for (x, y) in self.points:
            currDist = distance(mouseX, mouseY, x, y)
            if currDist < closestDist:
                closestDist = currDist
                closestPoint = (x, y)
        return closestPoint

    def addUserPoint(self, targetX, targetY):
        if (targetX, targetY) not in self.points:
            return
        else:
            targetPoint = (targetX, targetY)
            if self.currentPoint == None:
                self.currentPoint = targetPoint
            else:
                x1, y1 = self.currentPoint
                x2, y2 = targetX, targetY
                targetLine = Line(x1, y1, x2, y2)
                line = self.getLine(x1, y1, x2, y2)
                if line != None and line.label == None:
                    self.lastLabelValue += 1
                    line.label = self.lastLabelValue
                    self.currentPoint = targetPoint

    def getLine(self, x1, y1, x2, y2):
        targetLine = Line(x1, y1, x2, y2)
        for line in self.lines:
            if line == targetLine:
                return line
        return None

    def isSolved(self):
        for line in self.lines:
            if line.label == None:
                return False
        return True

def distance(x1, y1, x2, y2):
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5

# @testFunction
def testLineClass():
    # Line(x1, y1, x2, y2) represents a line from (x1, y1) to (x2, y2):
    line1 = Line(1,2,3,4)
    assert((line1.x1 == 1) and (line1.y1 == 2) and
           (line1.x2 == 3) and (line1.y2 == 4))

    # Line instances have a label, which starts out as None:
    assert(line1.label == None)

    # You will need to write __repr__ for this:
    assert(str(line1) == 'Line from (1,2) to (3,4) with label=None')

    line1.label = 5
    assert(str(line1) == 'Line from (1,2) to (3,4) with label=5')

    line2 = Line(3,4,1,2)
    assert(str(line2) == 'Line from (3,4) to (1,2) with label=None')
    line2.label = 6
    assert(str(line2) == 'Line from (3,4) to (1,2) with label=6')

    line3 = Line(2,1,4,3)
    assert(str(line3) == 'Line from (2,1) to (4,3) with label=None')
    line3.label = 7
    assert(str(line3) == 'Line from (2,1) to (4,3) with label=7')

    # You will need to write __eq__ for this next part.
    # Two lines are equal if they have the same start and end points,
    # even if they are in a different order.
    # For example, a line from (1,2) to (3,4) is equal to
    # a line from (3,4) to (1,2).
    # Also, ignore the line labels, so two lines can be equal even with
    # different labels.
    assert(line1 == line1)
    assert(line1 == line2)
    assert(line1 != line3)
    assert(line1 != 'ack') # do not crash here

# @testFunction
def testPuzzleClass():
    # To create a Puzzle instance, pass in a list of the endpoints of lines.
    # So here, the first line is from (200, 100) to (100, 300):
```

```
puzzle = Puzzle([[200, 100, 100, 300],
                 [100, 300, 300, 300],
                 [300, 300, 200, 100]])

# The Puzzle constructor has to create Line instances for each tuple
# (x1, y1, x2, y2) in its lineList argument.
assert(puzzle.lines == [Line(200,100,100,300),
                        Line(100,300,300,300),
                        Line(300,300,200,100)])

# The Puzzle constructor also has to create self.points, which is
# a set of all the points that occur in any of the Line instances:
assert(puzzle.points == {(200,100), (100,300), (300,300)})

# The Puzzle constructor also initializes self.lastLabelValue to 0
# and self.currentPoint to None.
assert(puzzle.lastLabelValue == 0)
assert(puzzle.currentPoint == None)

# The puzzle is not solved at first. It is only solved once every line
# instance in the puzzle has a label that is not None.
assert(puzzle.isSolved() == False)

# puzzle.getClosestPoint(targetX, targetY)
# This method takes a point and returns the closest point in the
# set puzzle.points to that target point.
# If there are two equidistant points, you may return either one
x,y = puzzle.getClosestPoint(180, 80)
assert((x,y) == (200,100))

# puzzle.addUserPoint(targetX, targetY)
# This method responds to the user adding the point (targetX, targetY).
# If the point is not in the puzzle's set of points, do nothing.
# If the point is in the puzzle's set of points, then:
#     If there is no current point, then this is the first point that was
#         added, so there are no lines label yet, so just set the puzzle's
#         current point to this point.
#     Else, if there is a current point:
#         Here, the user is trying to add a new line.
#         If there is a line in the puzzle from the current point
#         to this target point, and that line does not yet have a label,
#         then set that line's label to the next label value
#         (using self.lastLabelValue) and
#         make the target point the new current point.

# Here we add the first point (so no lines are labeled yet):
puzzle.addUserPoint(200, 100)
assert(puzzle.lastLabelValue == 0)
assert(puzzle.currentPoint == (200, 100))
assert(puzzle.isSolved() == False)

# Here we add the second point, which labels the line
# from (300, 300) to (200, 100).  Since this is the third
# line in our constructor's lineList, it has an index of 2,
# which is why we test that puzzle.lines[2].label == 1:
puzzle.addUserPoint(300, 300)
assert(puzzle.lastLabelValue == 1)
assert(puzzle.currentPoint == (300, 300))
assert(puzzle.lines[2].label == 1)
assert(puzzle.isSolved() == False)

# Next we add the third point, (100, 300), which labels
# the line from (100, 300) to (300, 300), which is lines[1],
# to 2:
    puzzle.addUserPoint(100, 300)
    assert(puzzle.lastLabelValue == 2)
    assert(puzzle.currentPoint == (100, 300))
    assert(puzzle.lines[1].label == 2)
    assert(puzzle.isSolved() == False)

    # Next we try to add the point (300, 300), which would add
    # the line from (100, 300) to (300, 300), but that line is
    # already labeled, so we can't do this.  Thus, this has
    # no effect:
    puzzle.addUserPoint(300, 300)
    assert(puzzle.lastLabelValue == 2)
    assert(puzzle.currentPoint == (100, 300))
    assert(puzzle.lines[1].label == 2)
    assert(puzzle.isSolved() == False)

    # Next we add the point (200, 100), which labels the line
    # from (200, 100) to (100, 300), which is lines[0], to 3:
    puzzle.addUserPoint(200, 100)
    assert(puzzle.lastLabelValue == 3)
    assert(puzzle.currentPoint == (200, 100))
    assert(puzzle.lines[0].label == 3)

    # And now all the lines are labeled, so the puzzle is solved:
    assert(puzzle.isSolved() == True)

def main():
    testLineClass()
    testPuzzleClass()

main()
```

Polynomial

In [2]:

```
# from cmu_cs3_utils import testFunction
import copy

class Polynomial:
    def __init__(self, coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return coefficientsToString(self.coeffs)

    def __eq__(self, other):
        return (isinstance(other, Polynomial) and
                self.coeffs == other.coeffs)

    def __hash__(self):
        return hash(str(self))

    def evalAt(self, x):
        result = 0
        terms = len(self.coeffs)
        for index in range(terms):
            coeff = self.coeffs[-index-1]
            power = index
            result += coeff * (x ** power)
        return result

    def plus(self, other):
        result = []
        length = min(len(self.coeffs), len(other.coeffs))
        if len(self.coeffs) > len(other.coeffs):
            longerList = self.coeffs
            shorterList = other.coeffs
        else:
            longerList = other.coeffs
            shorterList = self.coeffs
        beginningTerms = len(longerList) - len(shorterList)
        for i in range(beginningTerms):
            result.append(longerList[i])
        for i in range(length):
            result.append(shorterList[i] + longerList[i+beginningTerms])
        return Polynomial(result)

def coefficientsToString(coeffs):
    # This helper function may be useful for your __repr__ method!
    if len(coeffs) == 1:
        return str(coeffs[0])
    else:
        terms = [ ]
        for i in range(len(coeffs)):
            coeff = coeffs[i]
            if (coeff != 0):
                isNegative = (coeff < 0)
                coeff = abs(coeff)
                if (terms != [ ]):
                    terms.append(' - ' if isNegative else ' + ')
                if (terms == [ ]) and isNegative:
                    terms.append('-')
                if ((coeff != 1) or (i == len(coeffs)-1)):
                    terms.append(str(coeff))
                power = len(coeffs)-1-i
                if (power == 1): terms.append('x')
                elif (power > 1): terms.append(f'x**{power}')
        return ''.join(terms)

# @testFunction
def testPolynomialClass():
    p1 = Polynomial([2, 3]) # 2x + 3
    # Write __init__, and store the coefficients in the coeffs property:
    assert(p1.coeffs == [2, 3])

    # Write __repr__, and remember to use coefficientsToString(coeffs):
    assert(str(p1) == '2x + 3')

    # Write evalAt, which evaluates this polynomial at the given x value:
    assert(p1.evalAt(5) == 13)

    p2 = Polynomial([3, 0, -1]) # 3x**2 - 1
    assert(p2.coeffs == [3, 0, -1])
    assert(str(p2) == '3x**2 - 1')
    assert(p2.evalAt(5) == 74)

    L = [p1, p2]
    assert(str(L) == '[2x + 3, 3x**2 - 1]')

    # Write plus, which takes a second Polynomial instance, and returns a
    # third Polynomial, which is the sum of the two polynomials.  To add two
    # polynomials, just add the like terms:
    p3 = p1.plus(p2) # (2x + 3) + (3x**2 - 1) == 3x**2 + 2x + 2

    assert(p3.coeffs == [3, 2, 2])
    assert(str(p3) == '3x**2 + 2x + 2')
    assert(p3.evalAt(10) == 322)
    p4 = p2.plus(p1) # (3x**2 - 1) + (2x + 3) == 3x**2 + 2x + 2
    assert(p4.coeffs == [3, 2, 2])
    assert(str(p4) == '3x**2 + 2x + 2')
    assert(p4.evalAt(10) == 322)

    # Write __eq__ properly:
    assert(p3 != p1)
    assert(p3 == p4)
    assert(p3 != 'ack') # do not crash here!

    # And write __hash__ properly:
    s = set()
    assert(p3 not in s)
    s.add(p3)
    assert(p3 in s)
    assert(p4 in s)

def main():
    testPolynomialClass()

main()
```

WaterPouringPuzzle Classes

In [3]:

```
# from cmu_cs3_utils import testFunction

class Jar:
    def __init__(self, size, holding, goal):
        self.size = size
        self.holding = holding
        self.goal = goal

    def __repr__(self):
        return (f'Jar(size={self.size},' +
                f' holding={self.holding}, goal={self.goal})')

    def __eq__(self, other):
        return (isinstance(other, Jar) and
                self.size == other.size and
                self.holding == other.holding and
                self.goal == other.goal)

class WaterPouringPuzzle:
    def __init__(self, sizes, goals):
        self.sizes = sizes
        self.goals = goals
        self.jars = []
        self.holding = [0] * len(self.sizes)
        for i in range(len(sizes)):
            self.jars.append(Jar(self.sizes[i],self.holding[i],self.goals[i]))

    def isWin(self):
        for i in range(len(self.sizes)):
            if self.jars[i].goal != self.jars[i].holding:
                return False
        return True

    def fillJar(self, jarIndex):
        self.jars[jarIndex].holding = self.sizes[jarIndex]

    def emptyJar(self, jarIndex):
        self.jars[jarIndex].holding = 0
    def pourJar(self, fromJarIndex, toJarIndex):
        sumJar = self.jars[fromJarIndex].holding + self.jars[toJarIndex].holdi
        if sumJar > self.jars[toJarIndex].size:
            self.jars[toJarIndex].holding = self.jars[toJarIndex].size
            self.jars[fromJarIndex].holding=sumJar-self.jars[toJarIndex].holdi
        else:
            self.jars[toJarIndex].holding += self.jars[fromJarIndex].holding
            self.jars[fromJarIndex].holding = 0

# @testFunction
def testJarClass():
    jar1 = Jar(5, 0, 4)
    assert(str(jar1) == 'Jar(size=5, holding=0, goal=4)')
    assert(jar1.size == 5)
    assert(jar1.holding == 0)
    assert(jar1.goal == 4)
    assert(jar1 == Jar(5, 0, 4))

# @testFunction
def testWaterPouringPuzzleClass():
    puzzle1 = WaterPouringPuzzle([5, 3], [4, 0])
    assert(len(puzzle1.jars) == 2)
    jar0 = puzzle1.jars[0]
    jar1 = puzzle1.jars[1]
    assert(jar0 == Jar(5, 0, 4))
    assert(jar1 == Jar(3, 0, 0))
    assert(str(jar0) ==  'Jar(size=5, holding=0, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=0, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.fillJar(1) # fill jar1
    assert(str(jar1) == 'Jar(size=3, holding=3, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.pourJar(1, 0) # pour jar1 into jar0
    assert(str(jar0) ==  'Jar(size=5, holding=3, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=0, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.fillJar(1) # fill jar1 again
    assert(str(jar0) == 'Jar(size=5, holding=3, goal=4)')
    assert(str(jar1) == 'Jar(size=3, holding=3, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.pourJar(1, 0) # pour jar1 into jar0, but only
                          # 2 ounches can be poured
    assert(str(jar0) ==  'Jar(size=5, holding=5, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=1, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.emptyJar(0) # empty jar0
    assert(str(jar0) ==  'Jar(size=5, holding=0, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=1, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.pourJar(1, 0) # pour jar1 into jar0 (1 ounce is poured)
    assert(str(jar0) ==  'Jar(size=5, holding=1, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=0, goal=0)')
    assert(puzzle1.isWin() == False)

    puzzle1.fillJar(1) # fill jar1 once more
    assert(str(jar0) == 'Jar(size=5, holding=1, goal=4)')
    assert(str(jar1) == 'Jar(size=3, holding=3, goal=0)')
    assert(puzzle1.isWin() == False)
    puzzle1.pourJar(1, 0) # pour jar1 into jar0, and we win!
    assert(str(jar0) ==  'Jar(size=5, holding=4, goal=4)')
    assert(str(jar1) ==  'Jar(size=3, holding=0, goal=0)')
    assert(puzzle1.isWin() == True) # We won!!!

    puzzle2 = WaterPouringPuzzle([8, 5, 3], [4, 4, 0])
    assert(len(puzzle2.jars) == 3)
    jar0 = puzzle2.jars[0]
    jar1 = puzzle2.jars[1]
    jar2 = puzzle2.jars[2]
    assert(str(jar0) ==  'Jar(size=8, holding=0, goal=4)')
    assert(str(jar1) ==  'Jar(size=5, holding=0, goal=4)')
    assert(str(jar2) ==  'Jar(size=3, holding=0, goal=0)')
    assert(puzzle2.isWin() == False)

    puzzle2.fillJar(0)
    assert(str(jar0) ==  'Jar(size=8, holding=8, goal=4)')

    puzzle2.pourJar(0, 1)
    assert(str(jar0) ==  'Jar(size=8, holding=3, goal=4)')
    assert(str(jar1) ==  'Jar(size=5, holding=5, goal=4)')
    assert(str(jar2) ==  'Jar(size=3, holding=0, goal=0)')

def main():
    testJarClass()
    testWaterPouringPuzzleClass()

main()
```

In [ ]: