```
'''
Licensing Information: Please do not distribute or publish solutions to this
project. You are free to use and extend Driverless Car for educational
purposes. The Driverless Car project was developed at Stanford, primarily by
Chris Piech (piech@cs.stanford.edu). It was inspired by the Pacman projects.
'''
from engine.const import Const
import util, math, random, collections

# Class: ExactInference
# ---------------------
# Maintain and update a belief distribution over the probability of a car
# being in a tile using exact updates (correct, but slow times).
class ExactInference(object):

    # Function: Init
    # --------------
    # Constructor that initializes an ExactInference object which has
    # numRows x numCols number of tiles.
    def __init__(self, numRows, numCols):
        self.skipElapse = False ### ONLY USED BY GRADER.PY in case problem 3 has
not been completed
        # util.Belief is a class (constructor) that represents the belief for a
single
        # inference state of a single car (see util.py).
        self.belief = util.Belief(numRows, numCols)
        self.transProb = util.loadTransProb()



################################################################################
##
    # Problem 2:
    # Function: Observe (update the probabilities based on an observation)
    # -----------------
    # Takes |self.belief| -- an object of class Belief, defined in util.py --
    # and updates it in place based on the distance observation $d_t$ and
    # your position $a_t$.
    #
    # - agentX: x location of your car (not the one you are tracking)
    # - agentY: y location of your car (not the one you are tracking)
    # - observedDist: true distance plus a mean-zero Gaussian with standard
    #                 deviation Const.SONAR_STD
    #
    # Notes:
    # - Convert row and col indices into locations using util.rowToY and
util.colToX.
    # - util.pdf: computes the probability density function for a Gaussian
    # - Don't forget to normalize self.belief after you update its
probabilities!

################################################################################
##

    def observe(self, agentX, agentY, observedDist):
        # BEGIN_YOUR_CODE (our solution is 6 lines of code, but don't worry if
you deviate from this)
        for col in range(self.belief.getNumCols()):
            for row in range(self.belief.getNumRows()):
                distance = math.sqrt((agentX - util.colToX(col))**2 + (agentY -
util.rowToY(row))**2)
                pdf = util.pdf(distance, Const.SONAR_STD, observedDist)
                prob = self.belief.getProb(row, col)
                self.belief.setProb(row, col, prob * pdf)
        self.belief.normalize()
        # END_YOUR_CODE


################################################################################
##
    # Problem 3:
    # Function: Elapse Time (propose a new belief distribution based on a
learned transition model)
    # ---------------------
    # Takes |self.belief| and updates it based on the passing of one time step.
    # Notes:
    # - Use the transition probabilities in self.transProb, which is a
dictionary
    #   containing all the ((oldTile, newTile), transProb) key-val pairs that
you
    #   must consider.
    # - If there are ((oldTile, newTile), transProb) pairs not in
self.transProb,
    #   they are assumed to have zero probability, and you can safely ignore
them.
    # - Use the addProb and getProb methods of the Belief class to access and
modify
    #   the probabilities associated with a belief.  (See util.py.)
    # - Be careful that you are using only the CURRENT self.belief distribution
to compute
    #   updated beliefs.  Don't incrementally update self.belief and use the
updated value
    #   for one grid square to compute the update for another square.
    # - Don't forget to normalize self.belief after all probabilities have been
updated!

################################################################################
##
    def elapseTime(self):
        if self.skipElapse: return ### ONLY FOR THE GRADER TO USE IN Problem 2
        # BEGIN_YOUR_CODE (our solution is 6 lines of code, but don't worry if
you deviate from this)
        updatedBelief = util.Belief(self.belief.getNumRows(),
self.belief.getNumCols(), 0)
        for oldTile, newTile in self.transProb:
            val = self.transProb[(oldTile, newTile)]
            if val > 0:
                delta = val * self.belief.getProb(oldTile[0], oldTile[1])
                updatedBelief.addProb(newTile[0], newTile[1], delta)
        updatedBelief.normalize()
        self.belief = updatedBelief

        # END_YOUR_CODE

    # Function: Get Belief
    # ---------------------
    # Returns your belief of the probability that the car is in each tile. Your
    # belief probabilities should sum to 1.
    def getBelief(self):
        return self.belief


# Class: Particle Filter
# ----------------------
# Maintain and update a belief distribution over the probability of a car
# being in a tile using a set of particles.
class ParticleFilter(object):
    NUM_PARTICLES = 200

    # Function: Init
    # --------------
    # Constructor that initializes an ParticleFilter object which has
    # (numRows x numCols) number of tiles.
    def __init__(self, numRows, numCols):
        self.belief = util.Belief(numRows, numCols)

        # Load the transition probabilities and store them in an integer-valued
defaultdict.
        # Use self.transProbDict[oldTile][newTile] to get the probability of
transitioning from oldTile to newTile.
        self.transProb = util.loadTransProb()
        self.transProbDict = dict()
        for (oldTile, newTile) in self.transProb:
            if not oldTile in self.transProbDict:
                self.transProbDict[oldTile] = collections.defaultdict(int)
            self.transProbDict[oldTile][newTile] = self.transProb[(oldTile,
newTile)]

        # Initialize the particles randomly.
        self.particles = collections.defaultdict(int)
        potentialParticles = list(self.transProbDict.keys())
        for i in range(self.NUM_PARTICLES):
            particleIndex = int(random.random() * len(potentialParticles))
            self.particles[potentialParticles[particleIndex]] += 1

        self.updateBelief()

    # Function: Update Belief
    # ---------------------
    # Updates |self.belief| with the probability that the car is in each tile
    # based on |self.particles|, which is a defaultdict from particle to
    # probability (which should sum to 1).
    def updateBelief(self):
        newBelief = util.Belief(self.belief.getNumRows(),
self.belief.getNumCols(), 0)
        for tile in self.particles:
            newBelief.setProb(tile[0], tile[1], self.particles[tile])
        newBelief.normalize()
        self.belief = newBelief


################################################################################
##
    # Problem 4 (part a):
    # Function: Observe:
    # -----------------
    # Takes |self.particles| and updates them based on the distance observation
    # $d_t$ and your position $a_t$.
    #
    # This algorithm takes two steps:
    # 1. Re-weight the particles based on the observation.
    #    Concept: We had an old distribution of particles, and now we want to
    #             update this particle distribution with the emission
probability
    #             associated with the observed distance.
    #             Think of the particle distribution as the unnormalized
posterior
    #             probability where many tiles would have 0 probability.
    #             Tiles with 0 probabilities (i.e. those with no particles)
    #             do not need to be updated.
    #             This makes particle filtering runtime to be O(|particles|).
    #             By comparison, the exact inference method (used in problem 2 +
3)
    #             assigns non-zero (though often very small) probabilities to
most tiles,
    #             so the entire grid must be updated at each time step.
    # 2. Re-sample the particles.
    #    Concept: Now we have the reweighted (unnormalized) distribution, we can
now
    #             re-sample the particles, choosing a new grid location for each
of
    #             the |self.NUM_PARTICLES| new particles.
    #
    # - agentX: x location of your car (not the one you are tracking)
    # - agentY: y location of your car (not the one you are tracking)
    # - observedDist: true distance plus a mean-zero Gaussian with standard
deviation Const.SONAR_STD
    #
    # Notes:
    # - Remember that |self.particles| is a dictionary with keys in the form of
    #   (row, col) grid locations and values representing the number of
particles at
    #   that grid square.
    # - Create |self.NUM_PARTICLES| new particles during resampling.
    # - To pass the grader, you must call util.weightedRandomChoice() once per
new
    #   particle.  See util.py for the definition of weightedRandomChoice().

################################################################################
##
    def observe(self, agentX, agentY, observedDist):
        # BEGIN_YOUR_CODE (our solution is 10 lines of code, but don't worry if
you deviate from this)
        particleDict = collections.defaultdict(float)
        for row, col in self.particles:
            distance = math.sqrt((util.colToX(col) - agentX)**2 +
(util.rowToY(row) - agentY)**2)
            pdf = util.pdf(distance, Const.SONAR_STD, observedDist)
            particleDict[(row, col)] = pdf * self.particles[(row, col)]

        self.particles = collections.defaultdict(int)
        for _ in range(0, self.NUM_PARTICLES):
            self.particles[util.weightedRandomChoice(particleDict)] += 1
        # END_YOUR_CODE

        self.updateBelief()


################################################################################
##
    # Problem 4 (part b):
    # Function: Elapse Time (propose a new belief distribution based on a
learned transition model)
    # ---------------------
    # Reads |self.particles|, representing particle locations at time $t$, and
    # writes an updated |self.particles| with particle locations at time $t+1$.
    #
    # This algorithm takes one step:
    # 1. Proposal based on the particle distribution at current time $t$.
    #    Concept: We have a particle distribution at current time $t$, and we
want
    #             to propose the particle distribution at time $t+1$. We would
like
    #             to sample again to see where each particle would end up using
    #             the transition model.
    #
    # Notes:
    # - Transition probabilities are stored in |self.transProbDict|.
    # - To pass the grader, you must loop over the particles using a statement
    #   of the form 'for tile in self.particles: <your code>' and call
    #   util.weightedRandomChoice() to sample a new particle location.
    # - Remember that if there are multiple particles at a particular location,
    #   you will need to call util.weightedRandomChoice() once for each of them!
    # - You should NOT call self.updateBelief() at the end of this function.
################################################################################
##
    def elapseTime(self):
        # BEGIN_YOUR_CODE (our solution is 6 lines of code, but don't worry if
you deviate from this)
        newParticles = collections.defaultdict(int)
        for tile in self.particles:
            if tile in self.transProbDict:
                for _ in range(self.particles[tile]):
                    newParticle =
util.weightedRandomChoice(self.transProbDict[tile])
                    newParticles[newParticle] += 1
        self.particles = newParticles
        self.updateBelief()
        # END_YOUR_CODE

    # Function: Get Belief
    # ---------------------
    # Returns your belief of the probability that the car is in each tile. Your
    # belief probabilities should sum to 1.
    def getBelief(self):
        return self.belief
```