# myTeam.py  
# ---------  
# Licensing Information: You are free to use or extend these projects for  
# educational purposes provided that (1) you do not distribute or publish  
# solutions, (2) you retain this notice, and (3) you provide clear  
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.  
#   
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.  
# The core projects and autograders were primarily created by John DeNero  
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).  
# Student side autograding was added by Brad Miller, Nick Hay, and  
# Pieter Abbeel (pabbeel@cs.berkeley.edu).  
  
from captureAgents import CaptureAgent  
import distanceCalculator  
import random, time, util, sys  
from game import Directions  
import game  
from util import nearestPoint  
  
  
#################  
# Team creation #  
#################  
  
def createTeam(firstIndex, secondIndex, isRed,  
 first='OffensiveReflexAgent', second='DefensiveReflexAgent'):  
 *"""  
 This function should return a list of two agents that will form the  
 team, initialized using firstIndex and secondIndex as their agent  
 index numbers. isRed is True if the red team is being created, and  
 will be False if the blue team is being created.  
  
 As a potentially helpful development aid, this function can take  
 additional string-valued keyword arguments ("first" and "second" are  
 such arguments in the case of this function), which will come from  
 the --redOpts and --blueOpts command-line arguments to capture.py.  
 For the nightly contest, however, your team will be created without  
 any extra arguments, so you should make sure that the default  
 behavior is what you want for the nightly contest.  
 """* return [eval(first)(firstIndex), eval(second)(secondIndex)]  
  
  
##########  
# Agents #  
##########  
  
class ReflexCaptureAgent(CaptureAgent):  
 *"""  
 A base class for reflex agents that chooses score-maximizing actions  
 """* def registerInitialState(self, gameState):  
 self.start = gameState.getAgentPosition(self.index)  
 CaptureAgent.registerInitialState(self, gameState)  
  
 def chooseAction(self, gameState):  
 *"""  
 Picks among the actions with the highest Q(s,a).  
 """* actions = gameState.getLegalActions(self.index)  
  
 # You can profile your evaluation time by uncommenting these lines  
 # start = time.time()  
 values = [self.evaluate(gameState, a) for a in actions]  
 # print 'eval time for agent %d: %.4f' % (self.index, time.time() - start)  
  
 maxValue = max(values)  
 bestActions = [a for a, v in zip(actions, values) if v == maxValue]  
  
 foodLeft = len(self.getFood(gameState).asList())  
  
 if gameState.getAgentState(self.index).numCarrying != 0:  
 bestDist = 9999  
 for action in actions:  
 successor = self.getSuccessor(gameState, action)  
 pos2 = successor.getAgentPosition(self.index)  
 dist = self.getMazeDistance(self.start, pos2)  
 if dist < bestDist:  
 bestAction = action  
 bestDist = dist  
 return bestAction  
  
 bestAction = random.choice(bestActions)  
  
 return bestAction  
  
 def getSuccessor(self, gameState, action):  
 *"""  
 Finds the next successor which is a grid position (location tuple).  
 """* successor = gameState.generateSuccessor(self.index, action)  
 pos = successor.getAgentState(self.index).getPosition()  
 if pos != nearestPoint(pos):  
 # Only half a grid position was covered  
 return successor.generateSuccessor(self.index, action)  
 else:  
 return successor  
  
 def evaluate(self, gameState, action):  
 *"""  
 Computes a linear combination of features and feature weights  
 """* features = self.getFeatures(gameState, action)  
 weights = self.getWeights(gameState, action)  
 return features \* weights  
  
 def getFeatures(self, gameState, action):  
 *"""  
 Returns a counter of features for the state  
 """* features = util.Counter()  
 successor = self.getSuccessor(gameState, action)  
 features['successorScore'] = self.getScore(successor)  
 return features  
  
 def getWeights(self, gameState, action):  
 *"""  
 Normally, weights do not depend on the gamestate. They can be either  
 a counter or a dictionary.  
 """* return {'successorScore': 1.0}  
  
  
class OffensiveReflexAgent(ReflexCaptureAgent):  
 *"""  
 A reflex agent that seeks food. This is an agent  
 we give you to get an idea of what an offensive agent might look like,  
 but it is by no means the best or only way to build an offensive agent.  
 """* def getFeatures(self, gameState, action):  
 features = util.Counter()  
 successor = self.getSuccessor(gameState, action)  
 foodList = self.getFood(successor).asList()  
 features['successorScore'] = -len(foodList) # self.getScore(successor)  
  
 # Compute distance to the nearest food  
  
 if len(foodList) > 0: # This should always be True, but better safe than sorry  
 myPos = successor.getAgentState(self.index).getPosition()  
 minDistance = min([self.getMazeDistance(myPos, food) for food in foodList])  
 features['distanceToFood'] = minDistance  
  
 features['enemyClose'] = 0  
 enemies = [successor.getAgentState(i) for i in self.getOpponents(successor)]  
 threats = [a for a in enemies if not a.isPacman and a.getPosition() != None]  
 if len(threats) > 0:  
 dists = [self.getMazeDistance(myPos, a.getPosition()) for a in threats]  
 print dists  
 features['threatDistance'] = min(dists)  
 if (dist <= 3 for dist in dists):  
 features['enemyClose'] = 1  
  
 return features  
  
 def getWeights(self, gameState, action):  
 return {'successorScore': 100, 'distanceToFood': -1, 'threatDistance': -1, 'enemyClose': -100}  
  
  
class DefensiveReflexAgent(ReflexCaptureAgent):  
 *"""  
 A reflex agent that keeps its side Pacman-free. Again,  
 this is to give you an idea of what a defensive agent  
 could be like. It is not the best or only way to make  
 such an agent.  
 """* def getFeatures(self, gameState, action):  
 features = util.Counter()  
 successor = self.getSuccessor(gameState, action)  
  
 myState = successor.getAgentState(self.index)  
 myPos = myState.getPosition()  
  
 # Computes whether we're on defense (1) or offense (0)  
 features['onDefense'] = 1  
 if myState.isPacman: features['onDefense'] = 0  
  
 # Computes distance to invaders we can see  
 enemies = [successor.getAgentState(i) for i in self.getOpponents(successor)]  
 invaders = [a for a in enemies if a.isPacman and a.getPosition() != None]  
 features['numInvaders'] = len(invaders)  
 if len(invaders) > 0:  
 dists = [self.getMazeDistance(myPos, a.getPosition()) for a in invaders]  
 features['invaderDistance'] = min(dists)  
  
 if action == Directions.STOP: features['stop'] = 1  
 rev = Directions.REVERSE[gameState.getAgentState(self.index).configuration.direction]  
 if action == rev: features['reverse'] = 1  
  
 return features  
  
 def getWeights(self, gameState, action):  
 return {'numInvaders': -1000, 'onDefense': 100, 'invaderDistance': -10, 'stop': -100, 'reverse': -2}