```python
# multiAgents.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 util import manhattanDistance
from game import Directions
import random, util

from game import Agent
from pacman import GameState

class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.

    The code below is provided as a guide.  You are welcome to change
    it in any way you see fit, so long as you don't touch our method
    headers.
    """


    def getAction(self, gameState: GameState):
        """
        You do not need to change this method, but you're welcome to.

        getAction chooses among the best options according to the evaluation function.

        Just like in the previous project, getAction takes a GameState and returns
        some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
        """
        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"

        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed successor
        GameStates (pacman.py) and returns a number, where higher numbers are better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        fooddist = []
        ghostdist = 0
        score = successorGameState.getScore()
        closestFood = 0
        for food in newFood.asList():
            fooddist.append(manhattanDistance(newPos, food))
            if fooddist:
                closestFood = min(fooddist)
            else:
                closestFood = 0
        score += 5 / (closestFood + 1)
        for ghost in newGhostStates:
            ghostdist = (manhattanDistance(newPos, ghost.getPosition()))
```

```
            if ghost.scaredTimer > 0:
                score += 100 / (ghostdist + 1)
            elif ghostdist < 2:
                score -= 1000
            else:
                score -= 5 / (ghostdist + 1)
        return score

def scoreEvaluationFunction(currentGameState: GameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()

class MultiAgentSearchAgent(Agent):
    """
    This class provides some common elements to all of your
    multi-agent searchers.  Any methods defined here will be available
    to the MinimaxPacmanAgent, AlphaBetaPacmanAgent & ExpectimaxPacmanAgent.

    You *do not* need to make any changes here, but you can if you want to
    add functionality to all your adversarial search agents.  Please do not
    remove anything, however.

    Note: this is an abstract class: one that should not be instantiated.  It's
    only partially specified, and designed to be extended.  Agent (game.py)
    is another abstract class.
    """

    def __init__(self, evalFn = 'scoreEvaluationFunction', depth = '2'):
        self.index = 0 # Pacman is always agent index 0
        self.evaluationFunction = util.lookup(evalFn, globals())
        self.depth = int(depth)

class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.generateSuccessor(agentIndex, action):
        Returns the successor game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        def minimax(agentIndex: int, depth: int, gameState: GameState):
            bestAction = None
            if depth == 0 or gameState.isLose() or gameState.isWin():
                return self.evaluationFunction(gameState), bestAction
            if agentIndex == 0:
                maxEval = float("-inf")
                for action in gameState.getLegalActions(agentIndex):
                    eval = minimax(1, depth, gameState.generateSuccessor(agentIndex, action))
                    maxEval = max(maxEval, eval[0])
                    if maxEval == eval[0]:
                        bestAction = action
                return maxEval, bestAction
            else:
                minEval = float("inf")
                for action in gameState.getLegalActions(agentIndex):
                    if agentIndex == gameState.getNumAgents() - 1:
                        eval = minimax(0, depth - 1, gameState.generateSuccessor(agentIndex, action))
                    else:
                        eval = minimax(agentIndex + 1, depth, gameState.generateSuccessor(agentIndex, action))
                    minEval = min(minEval, eval[0])
                    if minEval == eval[0]:
```

```
                    bestAction = action
            return minEval, bestAction
        minimax_value, minimax_action = minimax(0, self.depth, gameState)
        return minimax_action

class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        def alphabeta(agentIndex: int, depth: int, alpha: float, beta: float, gameState: GameState):
            bestAction = None
            if depth == 0 or gameState.isLose() or gameState.isWin():
                return self.evaluationFunction(gameState), bestAction
            if agentIndex == 0:
                maxEval = float("-inf")
                for action in gameState.getLegalActions(agentIndex):
                    eval = alphabeta(1, depth, alpha, beta, gameState.generateSuccessor(agentIndex, action))
                    maxEval = max(maxEval, eval[0])
                    if maxEval == eval[0]:
                        bestAction = action
                        alpha = max(alpha, eval[0])
                        if beta < alpha:
                            break
                return maxEval, bestAction
            else:
                minEval = float("inf")
                for action in gameState.getLegalActions(agentIndex):
                    if agentIndex == gameState.getNumAgents() - 1:
                        eval = alphabeta(0, depth - 1, alpha, beta, gameState.generateSuccessor(agentIndex, action))
                    else:
                        eval = alphabeta(agentIndex + 1, depth, alpha, beta, gameState.generateSuccessor(agentIndex, action))
                    minEval = min(minEval, eval[0])
                    if minEval == eval[0]:
                        bestAction = action
                        beta = min(beta, eval[0])
                        if beta < alpha:
                            break
                return minEval, bestAction

        alpha = float("-inf")
        beta = float("inf")
        alphabeta_value, alphabeta_action = alphabeta(0, self.depth, alpha, beta, gameState)
        return alphabeta_action


class ExpectimaxAgent(MultiAgentSearchAgent):
    """
      Your expectimax agent (question 4)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the expectimax action using self.depth and self.evaluationFunction

        All ghosts should be modeled as choosing uniformly at random from their
        legal moves.
        """
        def expectimax(agentIndex: int, depth: int, gameState: GameState):
            bestAction = None
            if depth == 0 or gameState.isLose() or gameState.isWin():
                return self.evaluationFunction(gameState), bestAction
            if agentIndex == 0:
                maxEval = float("-inf")
                for action in gameState.getLegalActions(agentIndex):
                    eval = expectimax(1, depth, gameState.generateSuccessor(agentIndex, action))
                    maxEval = max(maxEval, eval[0])
                    if maxEval == eval[0]:
                        bestAction = action
                return maxEval, bestAction
            else:
                avEval = 0
                sumEval = 0
                i = 0
                for action in gameState.getLegalActions(agentIndex):
                    if agentIndex == gameState.getNumAgents() - 1:
                        eval = expectimax(0, depth - 1, gameState.generateSuccessor(agentIndex, action))
                    else:
                        eval = expectimax(agentIndex + 1, depth, gameState.generateSuccessor(agentIndex, action))
                    sumEval += eval[0]
                    i += 1
                    bestAction = action
                avEval = sumEval / i
```

```
                return avEval, bestAction


        expectimax_value, expectimax_action = expectimax(0, self.depth, gameState)
        return expectimax_action

def betterEvaluationFunction(currentGameState: GameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).

    DESCRIPTION: <write something here so we know what you did>
    I followed the same approach as in Q1: finding the closest food,
    adjusting the score based on the distance to ghosts (prioritizing eating the scared ghosts and avoiding the non-scared ones),
    and I added a bonus for the closest pellet
    Also, because the evaluation function should evaluate states, rather than actions, I used the currentGameState instead of the
successorGameState
    """
    newPos = currentGameState.getPacmanPosition()
    newFood = currentGameState.getFood()
    newGhostStates = currentGameState.getGhostStates()
    fooddist = []
    ghostdist = 0
    score = currentGameState.getScore()
    closestFood = 0
    pelletDist = []
    for food in newFood.asList():
        fooddist.append(manhattanDistance(newPos, food))
        if fooddist:
            closestFood = min(fooddist)
        else:
            closestFood = 0
    score += 5 / (closestFood + 1)
    for ghost in newGhostStates:
        ghostdist = (manhattanDistance(newPos, ghost.getPosition()))
        if ghost.scaredTimer > 0:
            score += 100 / (ghostdist + 1)
        elif ghostdist < 2:
            score -= 1000
        else:
            score -= 5 / (ghostdist + 1)
    pellets = currentGameState.getCapsules()
    if pellets:
        for pellet in pellets:
            pelletDist.append(manhattanDistance(newPos, pellet))
        closestPellet = min(pelletDist)
        score += 100.0 / (closestPellet + 1)
    return score

# Abbreviation
better = betterEvaluationFunction
```