```
from util import manhattanDistance
from game import Directions
import random, util

from game import Agent

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 __init__(self):
    self.lastPositions = []
    self.dc = None


  def getAction(self, gameState):
    """
    getAction chooses among the best options according to the evaluation
function.

    getAction takes a GameState and returns some Directions.X for some X in the
set {North, South, West, East, Stop}

------------------------------------------------------------------------------
    Description of GameState and helper functions:

    A GameState specifies the full game state, including the food, capsules,
    agent configurations and score changes. In this function, the |gameState|
argument
    is an object of GameState class. Following are a few of the helper methods
that you
    can use to query a GameState object to gather information about the present
state
    of Pac-Man, the ghosts and the maze.

    gameState.getLegalActions():
        Returns the legal actions for the agent specified. Returns Pac-Man's
legal moves by default.

    gameState.generateSuccessor(agentIndex, action):
        Returns the successor state after the specified agent takes the action.
        Pac-Man is always agent 0.

    gameState.getPacmanState():
        Returns an AgentState object for pacman (in game.py)
        state.configuration.pos gives the current position
        state.direction gives the travel vector

    gameState.getGhostStates():
        Returns list of AgentState objects for the ghosts

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

    gameState.getScore():
        Returns the score corresponding to the current state of the game


    The GameState class is defined in pacman.py and you might want to look into
```

```
that for
    other helper methods, though you don't need to.
    """
    # 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


    return legalMoves[chosenIndex]

  def evaluationFunction(self, currentGameState, action):
    """
    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 (oldFood) 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.
    """
    # Useful information you can extract from a GameState (pacman.py)
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    oldFood = currentGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    return successorGameState.getScore()


def scoreEvaluationFunction(currentGameState):
  """
    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)

################################################################################
######
# Problem 1b: implementing minimax

class MinimaxAgent(MultiAgentSearchAgent):
  """
    Your minimax agent (problem 1)
  """

  def getAction(self, gameState):
    """
      Returns the minimax action from the current gameState using self.depth
      and self.evaluationFunction. Terminal states can be found by one of the
following:
      pacman won, pacman lost or there are no legal moves.

      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.getScore():
        Returns the score corresponding to the current state of the game

      gameState.isWin():
        Returns True if it's a winning state

      gameState.isLose():
        Returns True if it's a losing state

      self.depth:
        The depth to which search should continue

    """

    # BEGIN_YOUR_CODE (our solution is 20 lines of code, but don't worry if you
deviate from this)
    def getMin(gameState, index, d):
      valAction = (float("inf"), Directions.STOP)
      nextAgent = index + 1
      depth = d
      if nextAgent == gameState.getNumAgents():
        nextAgent = 0
        depth -= 1
      for action in gameState.getLegalActions(index):
        minimaxVal = Vminimax(gameState.generateSuccessor(index, action),
nextAgent, depth)
        if minimaxVal < valAction[0]:   # update expected utility
          valAction = (minimaxVal, action)
      return valAction

    def getMax(gameState, index, d):
      valAction = (float("-inf"), Directions.STOP)
```

```
      for action in gameState.getLegalActions(index):
        minimaxVal = Vminimax(gameState.generateSuccessor(index, action), index
+ 1, d)
        if minimaxVal > valAction[0]:   # update expected utility
          valAction = (minimaxVal, action)
      return valAction

    def Vminimax(gameState, index, d):
      if d == 0 or gameState.isLose() or gameState.isWin():
        return self.evaluationFunction(gameState)
      if index == 0:  # pacman
        return getMax(gameState, index, d)[0]
      else:           # ghost
        return getMin(gameState, index, d)[0]

    return getMax(gameState, self.index, self.depth)[1]
    # END_YOUR_CODE

################################################################################
######
# Problem 2a: implementing alpha-beta

class AlphaBetaAgent(MultiAgentSearchAgent):
  """
    Your minimax agent with alpha-beta pruning (problem 2)
  """

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

    # BEGIN_YOUR_CODE (our solution is 36 lines of code, but don't worry if you
deviate from this)
    def getMin(gameState, agentIndex, d, alpha, beta):
      valAction = (float("inf"), Directions.STOP)
      nextAgent = agentIndex + 1
      depth = d
      if nextAgent == gameState.getNumAgents():
        nextAgent = 0
        depth -= 1
      for action in gameState.getLegalActions(agentIndex):
        minimaxVal = Vminimax(gameState.generateSuccessor(agentIndex, action),
nextAgent, depth, alpha, beta)
        if minimaxVal < valAction[0]:   # update expected utility
          valAction = (minimaxVal, action)
        beta = min(beta, valAction[0])
        if valAction[0] < alpha:
          return valAction
      return valAction

    def getMax(gameState, agentIndex, d, alpha, beta):
      valAction = (float("-inf"), Directions.STOP)
      for action in gameState.getLegalActions(agentIndex):
        minimaxVal = Vminimax(gameState.generateSuccessor(agentIndex, action),
agentIndex + 1, d, alpha, beta)
        if minimaxVal > valAction[0]:   # update expected utility
          valAction = (minimaxVal, action)
        alpha = max(alpha, valAction[0])
        if valAction[0] > beta:
          return valAction
      return valAction

    def Vminimax(gameState, agentIndex, d, alpha, beta):
```

```
      if d == 0 or gameState.isLose() or gameState.isWin():
        return self.evaluationFunction(gameState)
      if agentIndex == 0:   # pacman
        return getMax(gameState, agentIndex, d, alpha, beta)[0]
      else:                 # ghost
        return getMin(gameState, agentIndex, d, alpha, beta)[0]

    return getMax(gameState, self.index, self.depth, float("-inf"),
float("inf"))[1]
    # END_YOUR_CODE

################################################################################
######
# Problem 3b: implementing expectimax

class ExpectimaxAgent(MultiAgentSearchAgent):
  """
    Your expectimax agent (problem 3)
  """

  def getAction(self, 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.
    """

    # BEGIN_YOUR_CODE (our solution is 20 lines of code, but don't worry if you
deviate from this)
    def getExpectimax(gameState, agentIndex, d):
      valAction = []
      nextAgent = agentIndex + 1
      depth = d
      if nextAgent == gameState.getNumAgents():
        nextAgent = 0
        depth -= 1
      for action in gameState.getLegalActions(agentIndex):
        minimaxVal = Vexpectimax(gameState.generateSuccessor(agentIndex,
action), nextAgent, depth)
        valAction.append(minimaxVal)
      return sum(valAction)/len(valAction)

    def getMax(gameState, agentIndex, d):
      valAction = (float("-inf"), Directions.STOP)
      for action in gameState.getLegalActions(agentIndex):
        minimaxVal = Vexpectimax(gameState.generateSuccessor(agentIndex,
action), agentIndex + 1, d)
        if minimaxVal > valAction[0]:
          valAction = (minimaxVal, action)
      return valAction

    def Vexpectimax(gameState, agentIndex, d):
      if d == 0 or gameState.isLose() or gameState.isWin():
        return self.evaluationFunction(gameState)
      if agentIndex == 0:   # pacman
        return getMax(gameState, agentIndex, d)[0]
      else:                 # ghost
        return getExpectimax(gameState, agentIndex, d)

    return getMax(gameState, self.index, self.depth)[1]
    # END_YOUR_CODE

################################################################################
```

```
######
# Problem 4a (extra credit): creating a better evaluation function

def betterEvaluationFunction(currentGameState):
  """
    Your extreme, unstoppable evaluation function (problem 4). Note that you
can't fix a seed in this function.

    DESCRIPTION: <write something here so we know what you did>
  """

  # BEGIN_YOUR_CODE (our solution is 13 lines of code, but don't worry if you
deviate from this)
  curPos = currentGameState.getPacmanPosition()
  foodList = currentGameState.getFood().asList()
  ghostStates = currentGameState.getGhostStates()
  capsules = currentGameState.getCapsules()
  score = currentGameState.getScore()
  minGhostDistance = float("-inf")
  minCapsuleDistance = float("-inf")
  scaredScore = 0

  # Iterate over ghost states
  distanceScores = []
  ghostDistances = []
  for ghost in ghostStates:
    dist = util.manhattanDistance(ghost.getPosition(), curPos)
    ghostDistances.append(dist)
    if ghost.scaredTimer == 0:
      distanceScores.append((dist + 10) * -1)
      break
    if ghost.scaredTimer > 8:
      if dist <= 4: distanceScores.append(40)
      if dist <= 3: distanceScores.append(60)
      if dist <= 2: distanceScores.append(80)
      if dist <= 1: distanceScores.append(90)
  if len(ghostDistances) > 0: minGhostDistance = min(ghostDistances)
  if len(distanceScores) > 0: scaredScore = max(distanceScores)

  # Iterate over capsules
  capsuleDistances = []
  for caps in capsules: capsuleDistances.append(util.manhattanDistance(caps,
curPos))
  if len(capsuleDistances) > 0: minCapsuleDistance = min(capsuleDistances)

  # Iterate over food list
  foodDistances = []
  for food in foodList: foodDistances.append(util.manhattanDistance(food,
curPos))

  # Update score
  if minCapsuleDistance < minGhostDistance: score += 30
  score += scaredScore
  score -= .35 * sum(foodDistances)

  return score
  # END_YOUR_CODE

# Abbreviation
better = betterEvaluationFunction
```