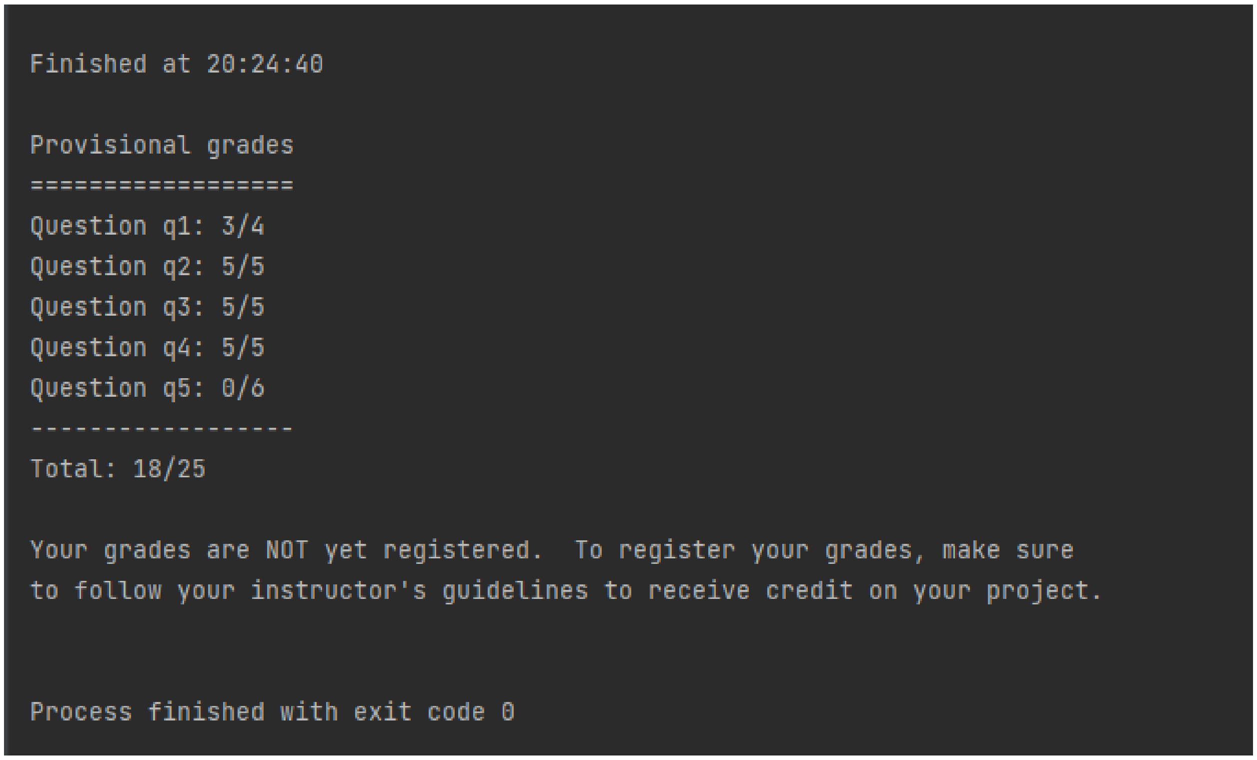
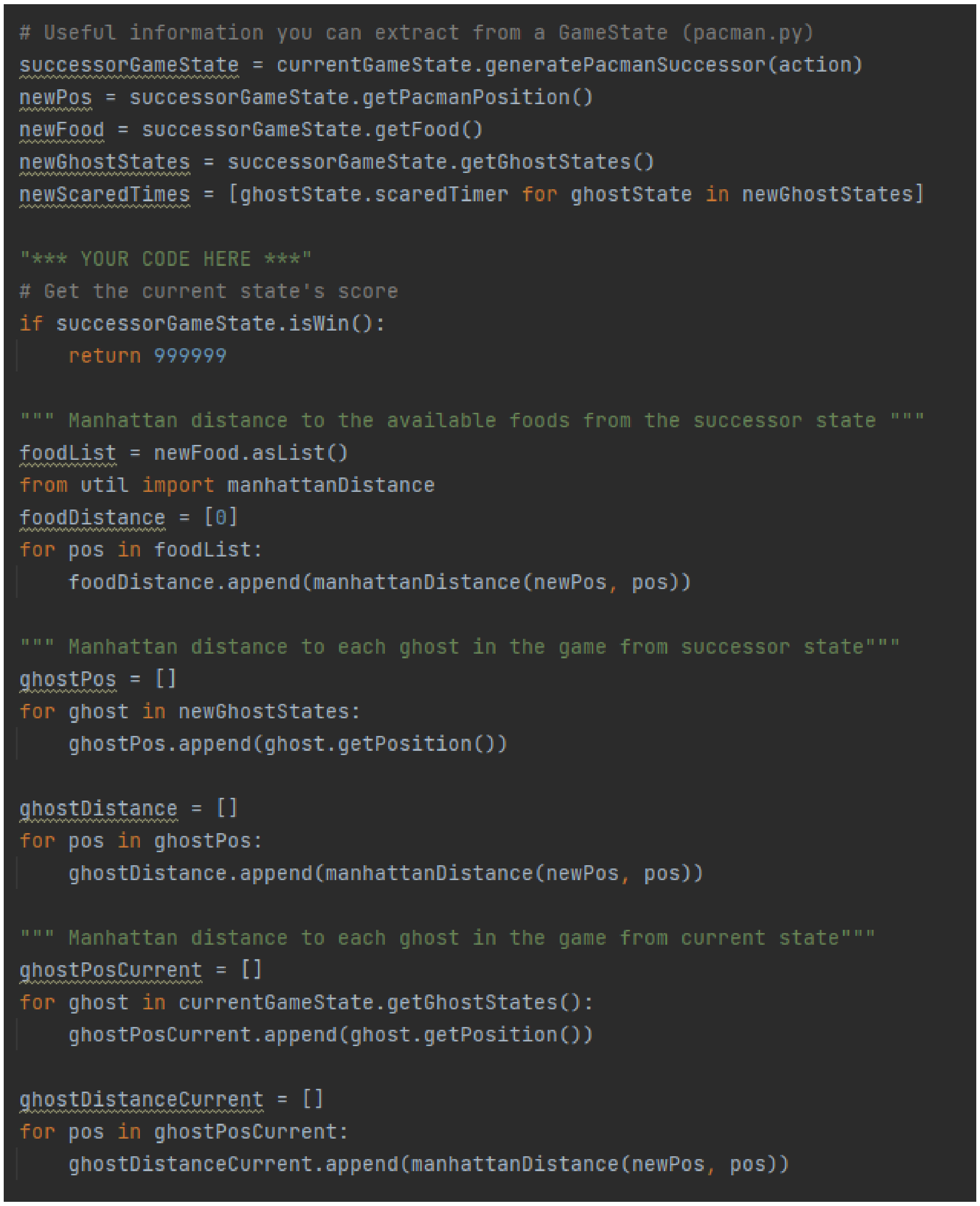
This is the result of the autograder.

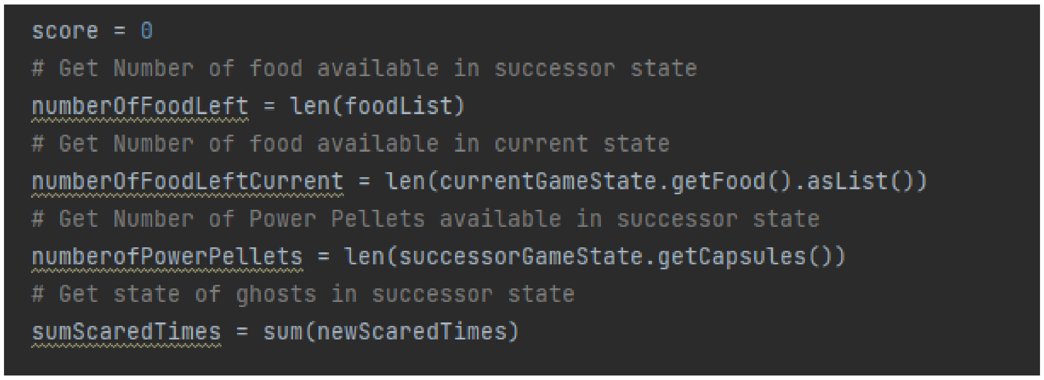


```
Finished at 20:24:40

Provisional grades
==================
Question q1: 3/4
Question q2: 5/5
Question q3: 5/5
Question q4: 5/5
Question q5: 0/6
------------------
Total: 18/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.


Process finished with exit code 0
```

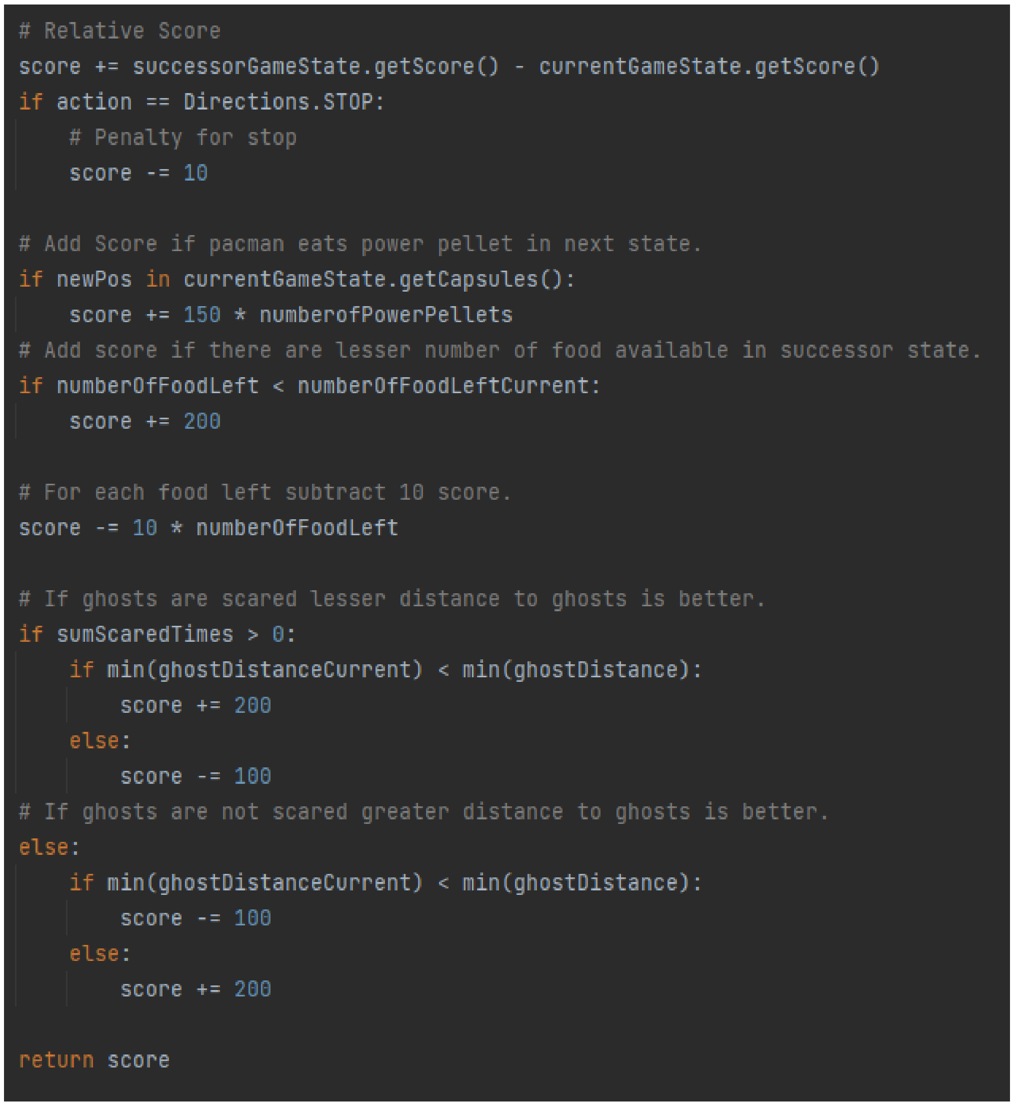
For question 1, In my implementation, I improved the ReflexAgent by modifying the evaluation function. I prioritized states closer to food by giving higher scores based on manhattan distance. I also penalized states close to ghosts to avoid them.
Finding the right weights for the evaluation function required trial and error method.



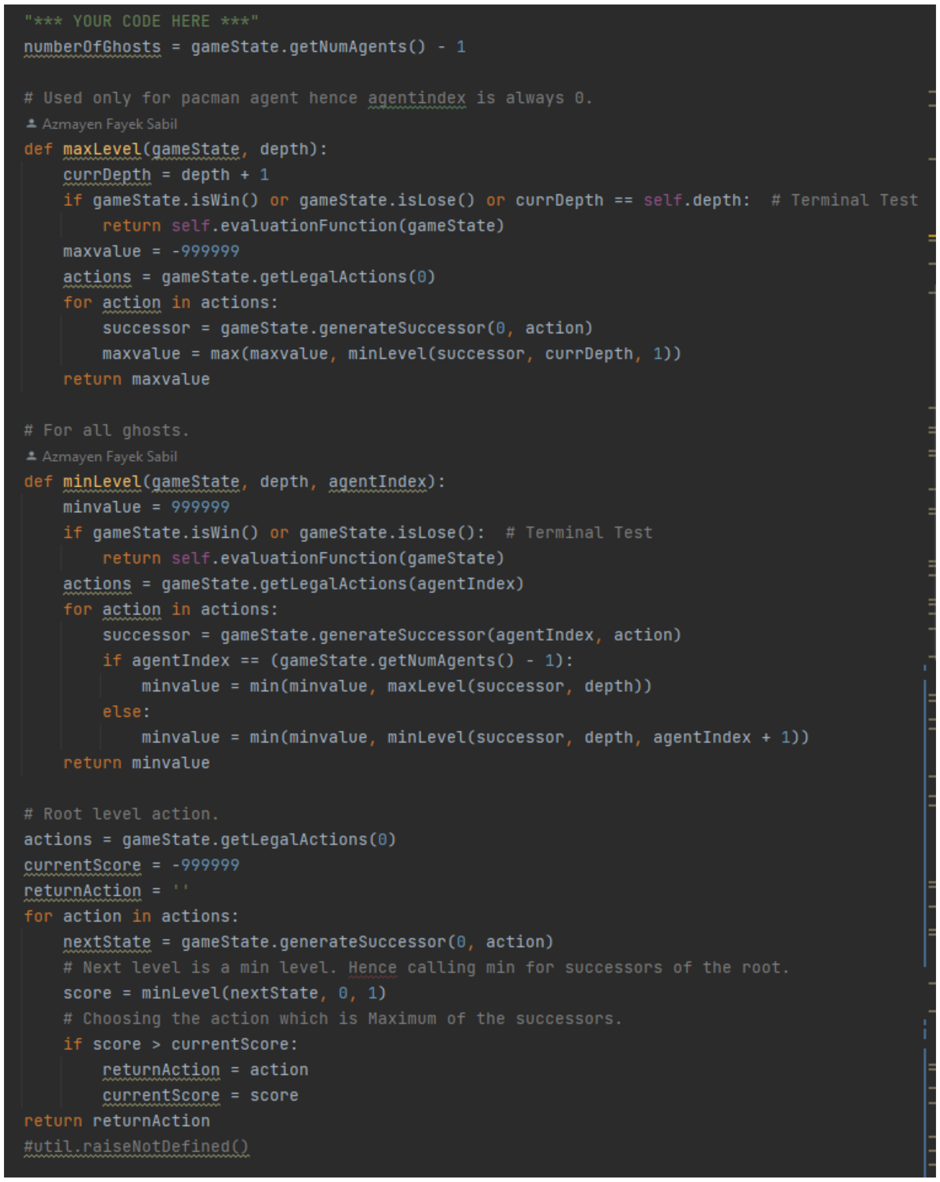
```python
# Useful information you can extract from a GameState (pacman.py)
successorGameState = currentGameState.generatePacmanSuccessor(action)
newPos = successorGameState.getPacmanPosition()
newFood = successorGameState.getFood()
newGhostStates = successorGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

"*** YOUR CODE HERE ***"
# Get the current state's score
if successorGameState.isWin():
    return 999999

""" Manhattan distance to the available foods from the successor state """
foodList = newFood.asList()
from util import manhattanDistance
foodDistance = [0]
for pos in foodList:
    foodDistance.append(manhattanDistance(newPos, pos))

""" Manhattan distance to each ghost in the game from successor state"""
ghostPos = []
for ghost in newGhostStates:
    ghostPos.append(ghost.getPosition())

ghostDistance = []
for pos in ghostPos:
    ghostDistance.append(manhattanDistance(newPos, pos))

""" Manhattan distance to each ghost in the game from current state"""
ghostPosCurrent = []
for ghost in currentGameState.getGhostStates():
    ghostPosCurrent.append(ghost.getPosition())

ghostDistanceCurrent = []
for pos in ghostPosCurrent:
    ghostDistanceCurrent.append(manhattanDistance(newPos, pos))
```



```python
score = 0
# Get Number of food available in successor state
numberOfFoodLeft = len(foodList)
# Get Number of food available in current state
numberOfFoodLeftCurrent = len(currentGameState.getFood().asList())
# Get Number of Power Pellets available in successor state
numberofPowerPellets = len(successorGameState.getCapsules())
# Get state of ghosts in successor state
sumScaredTimes = sum(newScaredTimes)
```


```python
# Relative Score
score += successorGameState.getScore() - currentGameState.getScore()
if action == Directions.STOP:
    # Penalty for stop
    score -= 10

# Add Score if pacman eats power pellet in next state.
if newPos in currentGameState.getCapsules():
    score += 150 * numberofPowerPellets
# Add score if there are lesser number of food available in successor state.
if numberOfFoodLeft < numberOfFoodLeftCurrent:
    score += 200

# For each food left subtract 10 score.
score -= 10 * numberOfFoodLeft

# If ghosts are scared lesser distance to ghosts is better.
if sumScaredTimes > 0:
    if min(ghostDistanceCurrent) < min(ghostDistance):
        score += 200
    else:
        score -= 100
# If ghosts are not scared greater distance to ghosts is better.
else:
    if min(ghostDistanceCurrent) < min(ghostDistance):
        score -= 100
    else:
        score += 200

return score
```

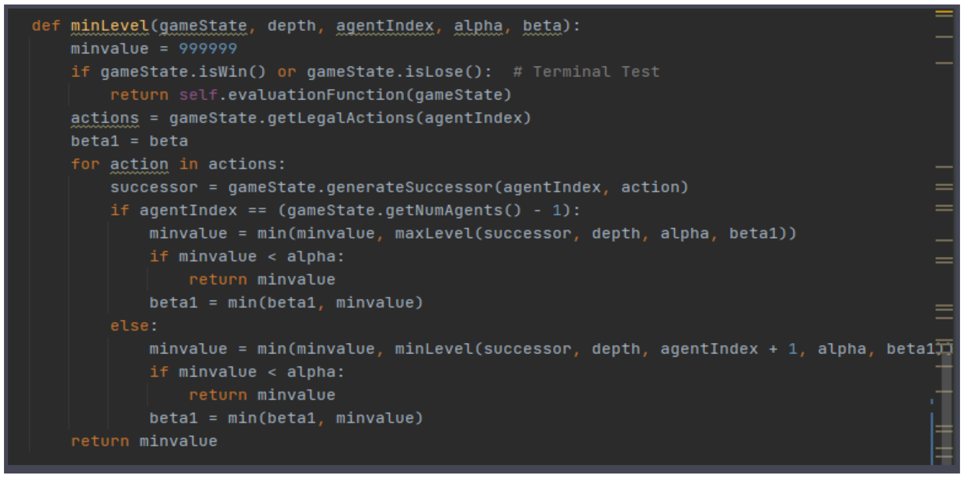
For question 2, I extended the MinimaxAgent class in multiAgents.py to create an adversarial search agent. I implemented the minimax algorithm to search through the game tree and determine the optimal moves for the agent. The agent considers both its own and the opponent's possible moves to choose the best action at each state. The minimax algorithm involves recursively evaluating the possible outcomes of each move and selecting the one that maximizes the agent's utility while minimizing the opponent's utility. The agent performed well in adversarial scenarios, making strategic decisions to maximize its chances of winning. Further optimizations such as alpha-beta pruning can be implemented to improve efficiency. Overall, the adversarial search agent demonstrates the ability to make informed decisions in competitive game scenarios.



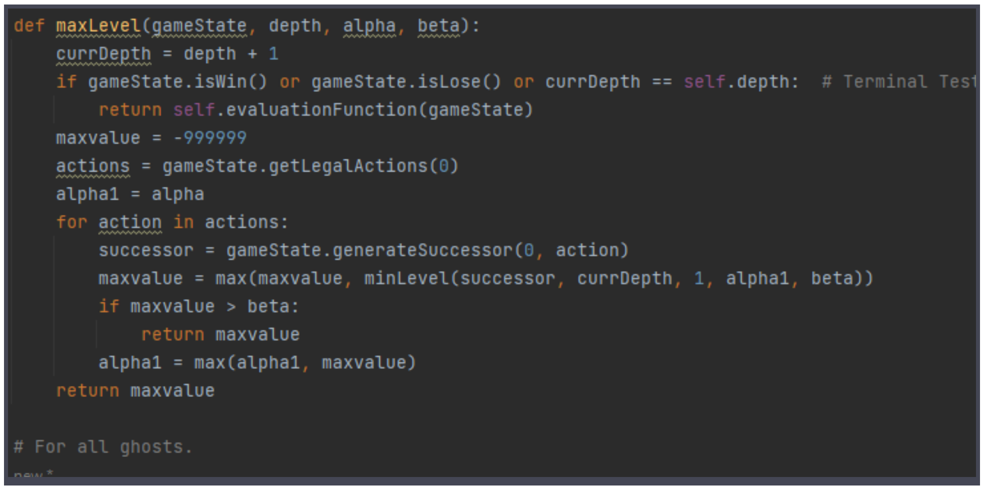
```
"*** YOUR CODE HERE ***"
numberOfGhosts = gameState.getNumAgents() - 1

# Used only for pacman agent hence agentindex is always 0.
Azmayen Fayek Sabil
def maxLevel(gameState, depth):
    currDepth = depth + 1
    if gameState.isWin() or gameState.isLose() or currDepth == self.depth:  # Terminal Test
        return self.evaluationFunction(gameState)
    maxvalue = -999999
    actions = gameState.getLegalActions(0)
    for action in actions:
        successor = gameState.generateSuccessor(0, action)
        maxvalue = max(maxvalue, minLevel(successor, currDepth, 1))
    return maxvalue

# For all ghosts.
Azmayen Fayek Sabil
def minLevel(gameState, depth, agentIndex):
    minvalue = 999999
    if gameState.isWin() or gameState.isLose():  # Terminal Test
        return self.evaluationFunction(gameState)
    actions = gameState.getLegalActions(agentIndex)
    for action in actions:
        successor = gameState.generateSuccessor(agentIndex, action)
        if agentIndex == (gameState.getNumAgents() - 1):
            minvalue = min(minvalue, maxLevel(successor, depth))
        else:
            minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1))
    return minvalue

# Root level action.
actions = gameState.getLegalActions(0)
currentScore = -999999
returnAction = ''
for action in actions:
    nextState = gameState.generateSuccessor(0, action)
    # Next level is a min level. Hence calling min for successors of the root.
    score = minLevel(nextState, 0, 1)
    # Choosing the action which is Maximum of the successors.
    if score > currentScore:
        returnAction = action
        currentScore = score
return returnAction
#util.raiseNotDefined()
```

For question 3,

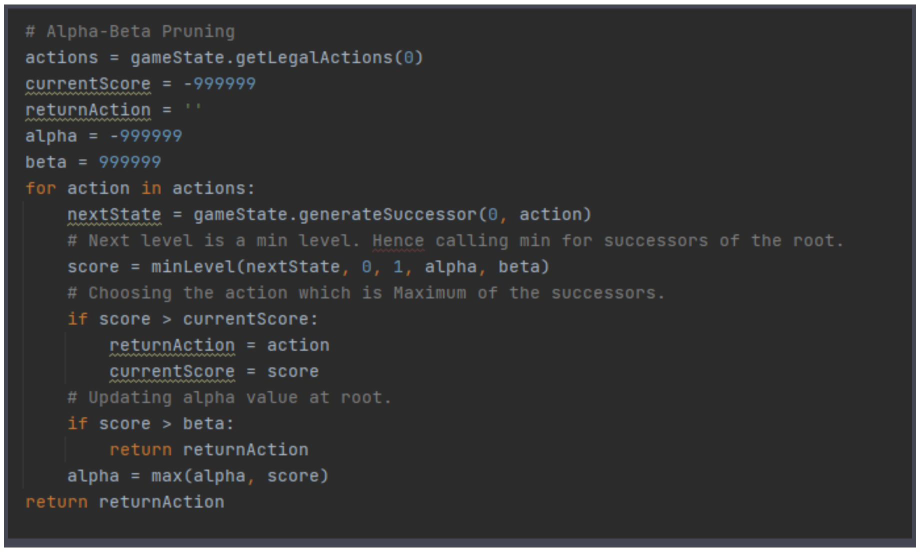
I implemented the AlphaBetaAgent class in multiagents.py by extending the MinimaxAgent class and incorporating the alpha-beta pruning technique. Alpha-beta pruning helps reduce the number of unnecessary nodes explored in the minimax tree by pruning branches that are known to be inferior. This optimization improves the agent's efficiency in searching for the optimal move. The agent maintains two values, alpha and beta, which represent the lower and upper bounds of the best possible utility values. During the search, if the agent finds a node that exceeds the beta value (for a maximizing player) or falls below the alpha value (for a minimizing player), it prunes the remaining branches. This allows the agent to explore fewer nodes and still arrive at the optimal solution. The AlphaBetaAgent performs significantly faster than the basic MinimaxAgent, making it more suitable for larger game trees and time-constrained scenarios.



```python
def minLevel(gameState, depth, agentIndex, alpha, beta):
    minvalue = 999999
    if gameState.isWin() or gameState.isLose():  # Terminal Test
        return self.evaluationFunction(gameState)
    actions = gameState.getLegalActions(agentIndex)
    beta1 = beta
    for action in actions:
        successor = gameState.generateSuccessor(agentIndex, action)
        if agentIndex == (gameState.getNumAgents() - 1):
            minvalue = min(minvalue, maxLevel(successor, depth, alpha, beta1))
            if minvalue < alpha:
                return minvalue
            beta1 = min(beta1, minvalue)
        else:
            minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1, alpha, beta1)
            if minvalue < alpha:
                return minvalue
            beta1 = min(beta1, minvalue)
    return minvalue
```



```python
def maxLevel(gameState, depth, alpha, beta):
    currDepth = depth + 1
    if gameState.isWin() or gameState.isLose() or currDepth == self.depth:  # Terminal Test
        return self.evaluationFunction(gameState)
    maxvalue = -999999
    actions = gameState.getLegalActions(0)
    alpha1 = alpha
    for action in actions:
        successor = gameState.generateSuccessor(0, action)
        maxvalue = max(maxvalue, minLevel(successor, currDepth, 1, alpha1, beta))
        if maxvalue > beta:
            return maxvalue
        alpha1 = max(alpha1, maxvalue)
    return maxvalue

# For all ghosts.
```



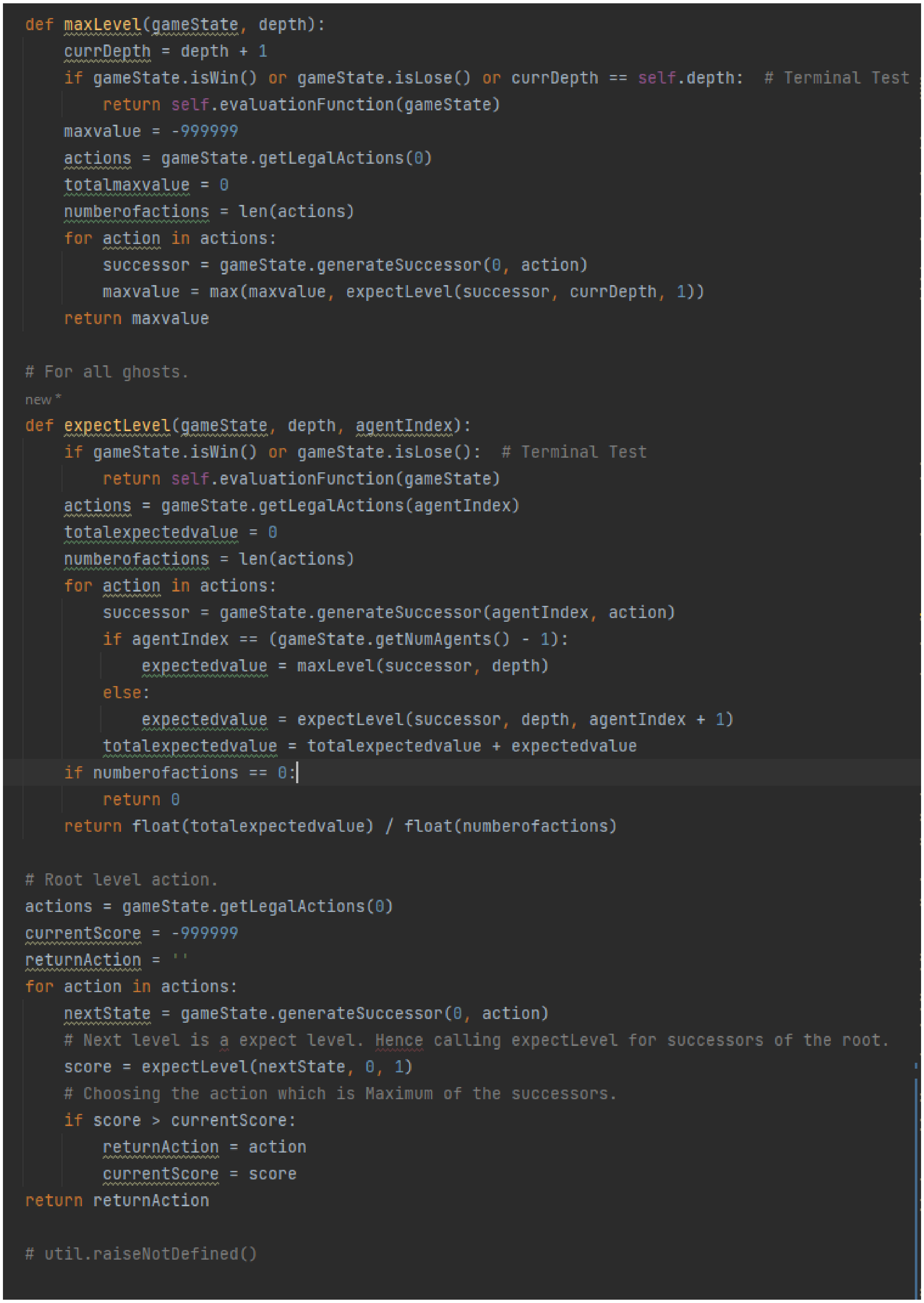
```
# Alpha-Beta Pruning
actions = gameState.getLegalActions(0)
currentScore = -999999
returnAction = ''
alpha = -999999
beta = 999999
for action in actions:
    nextState = gameState.generateSuccessor(0, action)
    # Next level is a min level. Hence calling min for successors of the root.
    score = minLevel(nextState, 0, 1, alpha, beta)
    # Choosing the action which is Maximum of the successors.
    if score > currentScore:
        returnAction = action
        currentScore = score
    # Updating alpha value at root.
    if score > beta:
        return returnAction
    alpha = max(alpha, score)
return returnAction
```

For question 4,

In question 4, we are exploring the concept of Expectimax, which is an extension of the minimax algorithm that takes into account the possibility of facing an opponent who makes suboptimal decisions or introduces uncertainty into the game. Unlike minimax, which assumes the opponent always makes the best moves, Expectimax considers the average value of each possible action the opponent can take.

To implement Expectimax, we extend the existing MinimaxAgent class and modify the minimax function to handle both maximizing and chance nodes. In chance nodes, we calculate the average value of all possible actions based on their probabilities. This accounts for situations where the opponent's actions are not deterministic.

By incorporating the concept of chance nodes, the Expectimax algorithm provides a more realistic approach to games with uncertain outcomes or opponents who do not always make optimal decisions. It allows us to consider different strategies and evaluate the expected value of each action, rather than relying solely on the assumption of optimal play



```
def maxLevel(gameState, depth):
    currDepth = depth + 1
    if gameState.isWin() or gameState.isLose() or currDepth == self.depth:  # Terminal Test
        return self.evaluationFunction(gameState)
    maxvalue = -999999
    actions = gameState.getLegalActions(0)
    totalmaxvalue = 0
    numberofactions = len(actions)
    for action in actions:
        successor = gameState.generateSuccessor(0, action)
        maxvalue = max(maxvalue, expectLevel(successor, currDepth, 1))
    return maxvalue

# For all ghosts.
new *
def expectLevel(gameState, depth, agentIndex):
    if gameState.isWin() or gameState.isLose():  # Terminal Test
        return self.evaluationFunction(gameState)
    actions = gameState.getLegalActions(agentIndex)
    totalexpectedvalue = 0
    numberofactions = len(actions)
    for action in actions:
        successor = gameState.generateSuccessor(agentIndex, action)
        if agentIndex == (gameState.getNumAgents() - 1):
            expectedvalue = maxLevel(successor, depth)
        else:
            expectedvalue = expectLevel(successor, depth, agentIndex + 1)
        totalexpectedvalue = totalexpectedvalue + expectedvalue
    if numberofactions == 0:
        return 0
    return float(totalexpectedvalue) / float(numberofactions)

# Root level action.
actions = gameState.getLegalActions(0)
currentScore = -999999
returnAction = ''
for action in actions:
    nextState = gameState.generateSuccessor(0, action)
    # Next level is a expect level. Hence calling expectLevel for successors of the root.
    score = expectLevel(nextState, 0, 1)
    # Choosing the action which is Maximum of the successors.
    if score > currentScore:
        returnAction = action
        currentScore = score
return returnAction

# util.raiseNotDefined()
```