ChatGPT-4o

|  |  |  |
| --- | --- | --- |
| Question text | Result | Accuracy |
| # The "Random" strategy # This agent randomly chooses between ATTACK and FARM with equal probability. | # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability.  class RandomAgent(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color)  def next\_move(self, other\_id):  return random.choice([ATTACK, FARM]) | Correct |
|  | # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability.  class RandomAgent(Group):  color = "tab:blue" # Assigning a unique color for Random agents  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  return random.choice([ATTACK, FARM]) | Correct |
|  | class RandomAgent(Group):  color = "tab:blue" # Assign a unique color for RandomAgent visualization    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color) # Assign a moderate speed and unique color    def next\_move(self, other\_id):  return random.choice([ATTACK, FARM]) # Randomly choose between ATTACK and FARM | Correct |
| # The "Pavlovian" strategy (Win-Stay, Lose-Shift)  # Cooperates if it and its opponent moved alike in previous move and defects if they moved differently.  # https://plato.stanford.edu/entries/prisoner-dilemma/strategy-table.html | # The "Pavlovian" strategy (Win-Stay, Lose-Shift)  # Cooperates if it and its opponent moved alike in previous move and defects if they moved differently.  class Pavlovian(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  # If this is the first interaction, default to cooperate (FARM)  if other\_id not in self.memory.keys():  return FARM  # Retrieve the last moves of self and opponent  last\_opponent\_move = self.memory[other\_id][-1]  last\_self\_move = self.my\_actions\_memory[other\_id][-1]  # Win-Stay, Lose-Shift logic  if last\_self\_move == last\_opponent\_move:  return FARM # Cooperate if moves were alike  else:  return ATTACK # Defect if moves were different | Correct |
|  | # The "Pavlovian" strategy (Win-Stay, Lose-Shift)  # Cooperates if it and its opponent moved alike in previous move and defects if they moved differently.  class Pavlov(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  return FARM # Default to cooperation initially  else:  # Check the last move of self and opponent  my\_last\_move = self.my\_actions\_memory[other\_id][-1]  opponent\_last\_move = self.memory[other\_id][-1]  # Win-Stay: If moves are alike, continue the same move  if my\_last\_move == opponent\_last\_move:  return FARM  else:  # Lose-Shift: If moves are different, defect  return ATTACK | Correct |
|  | class Pavlovian(Group):  color = "tab:blue" # Assigning a distinct color to identify Pavlovian agents  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # Default to cooperating (FARM) if no memory of the opponent's actions  return FARM  else:  # Get the last moves of self and the opponent  last\_opponent\_move = self.memory[other\_id][-1]  last\_own\_move = self.my\_actions\_memory[other\_id][-1]    # Win-Stay, Lose-Shift logic  if last\_own\_move == last\_opponent\_move:  return FARM # Stay (Cooperate) if moves were alike  else:  return ATTACK # Shift (Defect) if moves were different | Correct |
| # Reactive (with parameters y,p,q)  # Cooperates with probability y in first round and with probabilities p or q after opponent cooperates or defects.  # https://plato.stanford.edu/entries/prisoner-dilemma/strategy-table.html | # Reactive (with parameters y, p, q)  # Cooperates with probability y in first round and with probabilities p or q after opponent cooperates or defects.  class Reactive(Group):  color = "tab:blue" # Assign a unique color for Reactive agents    def \_\_init\_\_(self, x, y, initial\_coop\_prob, coop\_after\_coop\_prob, coop\_after\_defect\_prob):  """  Initialize a Reactive agent with specific probabilities.    :param x: Initial x-coordinate of the agent  :param y: Initial y-coordinate of the agent  :param initial\_coop\_prob: Probability of cooperating in the first round  :param coop\_after\_coop\_prob: Probability of cooperating after opponent cooperates  :param coop\_after\_defect\_prob: Probability of cooperating after opponent defects  """  super().\_\_init\_\_(x, y, 0.01, \_\_class\_\_.color)  self.y = initial\_coop\_prob  self.p = coop\_after\_coop\_prob  self.q = coop\_after\_defect\_prob  def next\_move(self, other\_id):  """  Decide the next move based on the opponent's previous move and the agent's strategy probabilities.    :param other\_id: The ID of the opponent  :return: FARM (0) if cooperating, ATTACK (1) if defecting  """  if other\_id not in self.memory:  # First round: decide based on initial cooperation probability  return FARM if random.random() < self.y else ATTACK    # Check the opponent's last move  last\_move = self.memory[other\_id][-1]  if last\_move == FARM:  # Opponent cooperated: decide based on p  return FARM if random.random() < self.p else ATTACK  else:  # Opponent defected: decide based on q  return FARM if random.random() < self.q else ATTACK | Incorrect. Confused by the fact that ‘y’ was used in the description as a parameter and in the existing code as a coordinate. |
|  | class Reactive(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y, y\_prob, p\_prob, q\_prob):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color)  self.y\_prob = y\_prob # Probability to cooperate in the first round  self.p\_prob = p\_prob # Probability to cooperate after opponent cooperates  self.q\_prob = q\_prob # Probability to cooperate after opponent defects  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # First interaction: cooperate with probability y  return FARM if random.random() < self.y\_prob else ATTACK  else:  # Subsequent interactions: choose based on the opponent's last move  last\_move = self.memory[other\_id][-1]  if last\_move == FARM: # Opponent cooperated  return FARM if random.random() < self.p\_prob else ATTACK  elif last\_move == ATTACK: # Opponent defected  return FARM if random.random() < self.q\_prob else ATTACK | Correct |
|  | # The "Reactive" strategy  # Cooperates with probability y in the first round and with probabilities p or q after opponent cooperates or defects.  class Reactive(Group):  color = "tab:blue" # Assign a unique color for Reactive agents    def \_\_init\_\_(self, x, y, y\_prob, p\_prob, q\_prob):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color) # Use a speed of 0.015 for Reactive agents  self.y\_prob = y\_prob # Probability of cooperation in the first round  self.p\_prob = p\_prob # Probability of cooperation after opponent cooperates  self.q\_prob = q\_prob # Probability of cooperation after opponent defects  def next\_move(self, other\_id):  if other\_id not in self.memory.keys(): # First interaction  return FARM if random.random() < self.y\_prob else ATTACK  else:  # Check the last move of the opponent  last\_move = self.memory[other\_id][-1]  if last\_move == FARM: # Opponent cooperated  return FARM if random.random() < self.p\_prob else ATTACK  else: # Opponent defected  return FARM if random.random() < self.q\_prob else ATTACK | Correct |
| #Random TitTat  # On the first turn, it randomly chooses to farm or attack, then does whatever the other agent did last time. | # Random TitTat  # On the first turn, it randomly chooses to farm or attack, then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:blue" # Assign a unique color for visualization    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.initial\_move = random.choice([FARM, ATTACK]) # Choose FARM or ATTACK randomly  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # If it's the first interaction with this agent, return the initial random move  return self.initial\_move  else:  # Otherwise, mimic the last move of the other agent  return self.memory[other\_id][-1] | Incorrect(bad description) |
|  | #Random TitTat  # On the first turn, it randomly chooses to farm or attack, then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = random.choice([FARM, ATTACK]) # Choose a random move for the first turn  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  return self.first\_move  else:  return self.memory[other\_id][-1] | Incorrect (bad description) |
|  | # Random TitTat  # On the first turn, it randomly chooses to farm or attack, then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = random.choice([FARM, ATTACK])    def next\_move(self, other\_id):  # On the first interaction, it uses the randomly chosen first move  if other\_id not in self.memory.keys():  return self.first\_move  # Otherwise, it mimics the last move of the other agent  else:  return self.memory[other\_id][-1] | Incorrect (bad description) |
| # The "Grudger" strategy  # Initially cooperates but if the other agent has ever attacked, this agent will always ATTACK in response. | # The "Grudger" strategy  # Initially cooperates but if the other agent has ever attacked, this agent will always ATTACK in response.  class Grudger(Group):  color = "tab:blue" # Assigning a unique color for Grudger agents  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.hold\_grudge = set() # Set to track agents who have attacked this agent  def next\_move(self, other\_id):  # Check if the agent is holding a grudge against the other agent  if other\_id in self.hold\_grudge:  return ATTACK  # Otherwise, cooperate (Farm)  return FARM  def record\_moves(self, my\_move, id\_num, move):  super().record\_moves(my\_move, id\_num, move)  # If the other agent attacks, hold a grudge  if move == ATTACK:  self.hold\_grudge.add(id\_num) | Correct, uses an unnecessary set to track grudges. |
|  | # The "Grudger" strategy  # Initially cooperates but if the other agent has ever attacked, this agent will always ATTACK in response.  class Grudger(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.grudges = set() # A set to track agents that have attacked this agent.  def next\_move(self, other\_id):  if other\_id in self.grudges:  return ATTACK # Attack if the other agent has attacked before.  elif other\_id in self.memory.keys() and ATTACK in self.memory[other\_id]:  self.grudges.add(other\_id) # Add to grudges if the other has attacked.  return ATTACK  else:  return FARM # Cooperate by default. | Correct, uses an unnecessary set to track grudges. |
|  | # The "Grudger" strategy  # Initially cooperates but if the other agent has ever attacked, this agent will always ATTACK in response.  class Grudger(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.grudge\_list = set() # Keeps track of agents who have attacked this agent  def next\_move(self, other\_id):  # If the other agent has attacked before, always ATTACK  if other\_id in self.grudge\_list:  return ATTACK  # If not, cooperate (FARM)  return FARM  def record\_moves(self, my\_move, id\_num, move):  super().record\_moves(my\_move, id\_num, move)  # Record grudge if attacked  if move == ATTACK:  self.grudge\_list.add(id\_num) | Correct, uses an unnecessary set to track grudges. |
| #The "Exploiter" strategy  # It starts by cooperating, then  # 1. If the other agent has never attacked, it attacks.  # 2. If the other agent also attacks on its first move, it keeps attacking.  # 3. If the other agent farmed on its first move, it attacks until the other agent attacks. Then:  # 3a. It farms twice.  # 3b. From then on it copies the other agent's last move. | # The "Exploiter" strategy  # It starts by cooperating, then  # 1. If the other agent has never attacked, it attacks.  # 2. If the other agent also attacks on its first move, it keeps attacking.  # 3. If the other agent farmed on its first move, it attacks until the other agent attacks. Then:  # 3a. It farms twice.  # 3b. From then on it copies the other agent's last move.  class Exploiter(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.action\_history = {} # Keeps track of other agents' first move  self.temp\_farm\_count = {} # Tracks farming after the first attack  def next\_move(self, other\_id):  if other\_id not in self.action\_history:  # If encountering the other agent for the first time, start by farming  self.action\_history[other\_id] = []  self.temp\_farm\_count[other\_id] = 0  return FARM    if not self.action\_history[other\_id]:  # Record the other agent's first move  self.action\_history[other\_id].append(self.memory[other\_id][-1] if other\_id in self.memory else FARM)  first\_move = self.action\_history[other\_id][0]  if first\_move == ATTACK:  # If the other agent attacked on its first move, keep attacking  return ATTACK    if first\_move == FARM:  # If the other agent farmed on its first move  if ATTACK not in self.memory[other\_id]:  # Keep attacking until the other agent attacks  return ATTACK  else:  # The other agent has attacked at least once  if self.temp\_farm\_count[other\_id] < 2:  # Farm twice after the first attack  self.temp\_farm\_count[other\_id] += 1  return FARM  else:  # Copy the other agent's last move after farming twice  return self.memory[other\_id][-1]  return FARM # Default to farming (should not reach here logically) | Correct behavior with many unneeded variables. |
|  | #The "Exploiter" strategy  # It starts by cooperating, then  # 1. If the other agent has never attacked, it attacks.  # 2. If the other agent also attacks on its first move, it keeps attacking.  # 3. If the other agent farmed on its first move, it attacks until the other agent attacks. Then:  # 3a. It farms twice.  # 3b. From then on it copies the other agent's last move.  class Exploiter(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.farm\_count = 0 # To track the two farm moves after an attack.  def next\_move(self, other\_id):  # Case 1: If no memory of the other agent, start by cooperating.  if other\_id not in self.memory.keys():  return FARM    # Retrieve the other agent's history.  other\_moves = self.memory[other\_id]  if len(other\_moves) == 0:  return ATTACK # Should not happen but a safeguard.    # Case 2: If the other agent has never attacked, attack.  if ATTACK not in other\_moves:  return ATTACK    # Case 3: If the other agent attacked on its first move, keep attacking.  if other\_moves[0] == ATTACK:  return ATTACK    # Case 4: If the other agent farmed on its first move, but then attacked:  if self.farm\_count > 0:  # 3a: Farm twice after the first attack.  self.farm\_count -= 1  return FARM  if other\_moves[-1] == ATTACK and self.farm\_count == 0:  # After being attacked, switch to farm twice.  self.farm\_count = 2  return FARM    # 3b: Copy the other agent's last move.  return other\_moves[-1] | Incorrect, uses a single variable to track its concession, gets stuck in a cycle. |
|  | class Exploiter(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color)  self.has\_other\_attacked = {} # Track if the other agent has attacked  self.other\_first\_move = {} # Track the other agent's first move  self.attack\_until\_attacked = {} # Track if we're in the "attack until attacked" phase  self.farm\_counter = {} # Counter for farming twice after being attacked  def next\_move(self, other\_id):  # If this is the first interaction with the other agent, initialize data  if other\_id not in self.memory:  self.has\_other\_attacked[other\_id] = False  self.other\_first\_move[other\_id] = None  self.attack\_until\_attacked[other\_id] = True  self.farm\_counter[other\_id] = 0  return FARM # Start by cooperating    # Check if the other agent has attacked  if ATTACK in self.memory[other\_id]:  self.has\_other\_attacked[other\_id] = True    # Record the first move of the other agent if not already done  if self.other\_first\_move[other\_id] is None:  self.other\_first\_move[other\_id] = self.memory[other\_id][0]  # Determine behavior based on the first move and attack history  if not self.has\_other\_attacked[other\_id]:  return ATTACK # Attack if the other agent has never attacked    if self.other\_first\_move[other\_id] == ATTACK:  return ATTACK # Keep attacking if the other agent attacked first    if self.attack\_until\_attacked[other\_id]:  if ATTACK not in self.memory[other\_id]:  return ATTACK # Attack until the other agent attacks  else:  self.attack\_until\_attacked[other\_id] = False # Stop attacking once the other agent attacks  self.farm\_counter[other\_id] = 2 # Start the farming phase    if self.farm\_counter[other\_id] > 0:  self.farm\_counter[other\_id] -= 1  return FARM # Farm twice after being attacked    # Copy the other agent's last move from then on  return self.memory[other\_id][-1] | Correct |
| #Once-burned, twice-shy strategy  # On the first move, if the most recent move of the most recently-added key in the memory is an attack, it will attack. On subsequent moves copy the other agent's last move. | # Once-burned, twice-shy strategy  # On the first move, if the most recent move of the most recently-added key in the memory is an attack, it will attack.  # On subsequent moves, copy the other agent's last move.  class OnceBurnedTwiceShy(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id in self.memory.keys():  # Check the last move of the other agent  return self.memory[other\_id][-1]  elif self.memory: # If there's memory but this ID isn't present  # Check the most recent move of the most recently-added key in the memory  last\_added\_key = list(self.memory.keys())[-1]  if self.memory[last\_added\_key][-1] == ATTACK:  return ATTACK  return FARM | Correct |
|  | # Once-burned, twice-shy strategy  # On the first move, if the most recent move of the most recently-added key in the memory is an attack,  # it will attack. On subsequent moves, copy the other agent's last move.  class OnceBurnedTwiceShy(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  # Check if this is the first interaction with this other\_id  if other\_id not in self.memory.keys():  # Look at the most recently added key in memory  if self.memory:  last\_key = list(self.memory.keys())[-1]  if self.memory[last\_key][-1] == ATTACK:  return ATTACK  return FARM  else:  # Copy the last move of the other agent  return self.memory[other\_id][-1] | Correct |
|  | #Once-burned, twice-shy strategy  # On the first move, if the most recent move of the most recently-added key in the memory is an attack, it will attack. On subsequent moves copy the other agent's last move.  class OnceBurned(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # No interaction with this agent yet  if self.memory: # Check if memory is not empty  most\_recent\_id = max(self.memory.keys(), key=lambda k: len(self.memory[k]))  if self.memory[most\_recent\_id][-1] == ATTACK:  return ATTACK  return FARM  else:  # Copy the other agent's last move  return self.memory[other\_id][-1] | Incorrect: Uses highest-id instead of most-recently-added |
| #Random TitTat  # When first meeting a new agent, it randomly chooses to farm or attack, then does whatever the other agent did last time. | # The "Random TitTat" strategy  # When first meeting a new agent, it randomly chooses to farm or attack,  # then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:blue" # Assigning a unique color for Random TitTat agents    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # Randomly choose FARM or ATTACK when meeting a new agent  return random.choice([FARM, ATTACK])  else:  # Follow the last move of the other agent  return self.memory[other\_id][-1] | Correct |
|  | # Random TitTat  # When first meeting a new agent, it randomly chooses to farm or attack, then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  # If meeting for the first time, randomly choose FARM or ATTACK  if other\_id not in self.memory.keys():  return random.choice([FARM, ATTACK])  else:  # Otherwise, mimic the last move of the other agent  return self.memory[other\_id][-1] | Correct |
|  | #Random TitTat  # When first meeting a new agent, it randomly chooses to farm or attack,  # then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:purple" # Assign a unique color for this agent type  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # Randomly choose to FARM or ATTACK when first meeting an agent  return random.choice([FARM, ATTACK])  else:  # Repeat the move the other agent made last time  return self.memory[other\_id][-1] | Correct |

ChatGPT-4o mini

|  |  |  |
| --- | --- | --- |
| # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability. | import random  # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability.  class Random(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color)    def next\_move(self, other\_id):  return random.choice([ATTACK, FARM]) | Correct |
|  | import random  # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability.  class RandomAgent(Group):  color = "tab:blue" # You can choose any color for this agent  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  # Randomly choose ATTACK or FARM with equal probability  return random.choice([ATTACK, FARM]) | Correct |
|  | import random  # The "Random" strategy  # This agent randomly chooses between ATTACK and FARM with equal probability.  class RandomAgent(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color) # Slightly different speed for variety    def next\_move(self, other\_id):  return random.choice([ATTACK, FARM]) | Correct |
| # The "Pavlovian" strategy (Win-Stay, Lose-Shift)  # Cooperates if it and its opponent moved alike in previous move and defects if they moved differently. | class Pavlov(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  # If this agent has interacted with the other agent before  if other\_id in self.memory.keys():  # Check the last moves of both agents  last\_move = self.my\_actions\_memory[other\_id][-1]  last\_other\_move = self.memory[other\_id][-1]    # Win-Stay, Lose-Shift strategy:  if last\_move == last\_other\_move:  return FARM # Cooperate (stay)  else:  return ATTACK # Defect (shift)  else:  # If no prior interaction, cooperate by default  return FARM | Correct |
|  | class Pavlovian(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .02, \_\_class\_\_.color)  def next\_move(self, other\_id):  # If no memory exists, cooperate (FARM)  if other\_id not in self.memory:  return FARM    # Get the last moves of both agents  my\_last\_move = self.my\_actions\_memory[other\_id][-1] if other\_id in self.my\_actions\_memory else None  opponent\_last\_move = self.memory[other\_id][-1] if other\_id in self.memory else None    # If both the current agent and the opponent moved similarly, cooperate (FARM)  if my\_last\_move is not None and opponent\_last\_move is not None:  if my\_last\_move == opponent\_last\_move:  return FARM  else:  return ATTACK  else:  return FARM # Default to cooperating if no prior interaction  def record\_moves(self, my\_move, id\_num, move):  # Similar to other agents, record the moves and update memory  super().record\_moves(my\_move, id\_num, move) | Correct |
|  | # The "Pavlovian" strategy (Win-Stay, Lose-Shift)  # Cooperates if it and its opponent moved alike in previous move and defects if they moved differently.  class Pavlovian(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color)    def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  return FARM # If it's the first interaction, cooperate by default.    # Check if both the current move and the opponent's previous move were the same.  last\_my\_move = self.my\_actions\_memory[other\_id][-1]  last\_opponent\_move = self.memory[other\_id][-1]    if last\_my\_move == last\_opponent\_move:  return FARM # "Win-Stay" strategy: continue cooperating if both cooperated or both defected  else:  return ATTACK # "Lose-Shift" strategy: defect if the opponent's last move was different | Correct |
| # Reactive (with parameters y,p,q)  # Cooperates with probability y in first round and with probabilities p or q after opponent cooperates or defects. | class Reactive(Group):  color = "tab:blue"  def \_\_init\_\_(self, x, y, p, q):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.y = y # Probability of cooperation in the first round  self.p = p # Probability of cooperation if opponent cooperates  self.q = q # Probability of cooperation if opponent defects  self.first\_move = True # Flag to track if it's the first round  def next\_move(self, other\_id):  if self.first\_move:  self.first\_move = False  # Cooperate with probability y in the first round  return FARM if random.random() < self.y else ATTACK  else:  # After the first round, cooperate with probability p if opponent cooperates  # or probability q if opponent defects  last\_opponent\_move = self.memory[other\_id][-1] if other\_id in self.memory else FARM # Default to cooperate if no memory  if last\_opponent\_move == FARM:  return FARM if random.random() < self.p else ATTACK  else:  return FARM if random.random() < self.q else ATTACK | Incorrect: confuses parameter y with y coordinate |
|  | import random  class Reactive(Group):  color = "tab:blue"    def \_\_init\_\_(self, x, y, y\_param, p\_param, q\_param):  """  Initialize the Reactive agent with specific probabilities.    :param x: Initial x-coordinate.  :param y: Initial y-coordinate.  :param y\_param: Probability of cooperation in the first round.  :param p\_param: Probability of cooperation if the opponent cooperates.  :param q\_param: Probability of cooperation if the opponent defects.  """  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.y = y\_param # Cooperation probability in the first round  self.p = p\_param # Cooperation probability after opponent cooperates  self.q = q\_param # Cooperation probability after opponent defects  self.first\_round = True # Flag to check if it's the first round of interaction    def next\_move(self, other\_id):  """  Determine the next move based on the Reactive strategy.    :param other\_id: The ID of the other agent.  :return: The move (FARM or ATTACK) based on the strategy.  """  if self.first\_round:  # First round, cooperate with probability y  self.first\_round = False  return FARM if random.random() < self.y else ATTACK  else:  # Subsequent rounds, cooperate based on opponent's previous move  if other\_id in self.memory.keys() and len(self.memory[other\_id]) > 0:  opponent\_last\_move = self.memory[other\_id][-1]  if opponent\_last\_move == FARM:  return FARM if random.random() < self.p else ATTACK  else:  return FARM if random.random() < self.q else ATTACK  else:  # If no previous interaction, act like the first round  return FARM if random.random() < self.y else ATTACK | Incorrect: confuses y parameter with y coordinate |
|  | import random  class Reactive(Group):  color = "tab:blue" # The color of the agent    def \_\_init\_\_(self, x, y, p, q):  """  Initialize the Reactive agent with given parameters.  y: Probability of cooperating in the first round  p: Probability of cooperating after the opponent cooperates  q: Probability of cooperating after the opponent defects  """  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color) # Call the parent constructor  self.y = y # Cooperation probability in the first round  self.p = p # Cooperation probability after opponent cooperates  self.q = q # Cooperation probability after opponent defects  self.last\_opponent\_move = None # Track the last move of the opponent    def next\_move(self, other\_id):  """  Determine the agent's move based on the last move of the opponent.  The agent cooperates with probability y in the first round, and with probability p or q  depending on the opponent's last move.  """  if other\_id not in self.memory:  # First round of interaction  return ATTACK if random.random() > self.y else FARM  else:  # After the first round, base move on the opponent's last move  last\_opponent\_move = self.memory[other\_id][-1]  if last\_opponent\_move == FARM:  return ATTACK if random.random() > self.p else FARM  else:  return ATTACK if random.random() > self.q else FARM | Incorrect: Confuses y coordinate with y parameter |
| #Random TitTat  # On the first turn, it randomly chooses to farm or attack, then does whatever the other agent did last time. | import random  class RandomTitTat(Group):  color = "tab:purple" # You can choose a distinct color  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_turn = True # Flag to track the first interaction  def next\_move(self, other\_id):  if self.first\_turn:  # On the first turn, randomly choose between ATTACK (1) or FARM (0)  self.first\_turn = False  return random.choice([ATTACK, FARM])  else:  # After the first turn, behave like TitTat (repeat the other agent's last move)  if other\_id not in self.memory.keys():  return FARM # If no previous memory, start by farming  else:  return self.memory[other\_id][-1] # Return the last move of the other agent | Incorrect |
|  | class RandomTitTat(Group):  color = "tab:purple" # You can choose any color for this agent  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = True # To track if it's the first move  def next\_move(self, other\_id):  if self.first\_move:  self.first\_move = False  # Randomly choose to attack or farm on the first move  return random.choice([ATTACK, FARM])  else:  # On subsequent moves, do whatever the other agent did last  if other\_id not in self.memory.keys():  return FARM # If there's no memory, assume the other agent farms  else:  return self.memory[other\_id][-1] | Incorrect |
|  | import random  class RandomTitTat(Group):  color = "tab:purple" # Choosing a distinct color for visualization  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = True # Flag to indicate if it's the first move  def next\_move(self, other\_id):  if self.first\_move:  # On the first move, randomly choose either FARM or ATTACK  self.first\_move = False  return random.choice([FARM, ATTACK])  else:  # After the first move, mimic the other agent's last move  if other\_id not in self.memory.keys():  return FARM # If no memory of interaction, cooperate initially  else:  return self.memory[other\_id][-1] # Mimic the other agent's last move | Incorrect |
| # The "Grudger" strategy  # Initially cooperates but if the other agent has ever attacked, this agent will always ATTACK in response. | class Grudger(Group):  color = "tab:purple" # Assign a unique color for the Grudger strategy  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.has\_attacked\_before = False # Flag to track if the agent has been attacked before  def next\_move(self, other\_id):  # If the agent has ever been attacked, it will always attack.  if self.has\_attacked\_before:  return ATTACK  else:  # Otherwise, it will start by cooperating.  if other\_id in self.memory and ATTACK in self.memory[other\_id]:  # If the other agent ever attacked, the Grudger will retaliate.  self.has\_attacked\_before = True  return ATTACK  else:  # Cooperate initially  return FARM  def record\_moves(self, my\_move, id\_num, move):  # Override record\_moves to check if the Grudger has been attacked  super().record\_moves(my\_move, id\_num, move)  if move == ATTACK:  self.has\_attacked\_before = True | Incorrect (attacks everyone if it has EVER been attacked by anyone) |
|  | class Grudger(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  # If the other agent has ever attacked, Grudger will attack in response  if other\_id in self.memory and ATTACK in self.memory[other\_id]:  return ATTACK  else:  return FARM | Correct |
|  | class Grudger(Group):  color = "tab:red" # Assigning a unique color for this strategy  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.has\_attacked = False # Flag to track if the agent has been attacked by others  def next\_move(self, other\_id):  # If the agent has ever attacked, it will retaliate by attacking  if self.has\_attacked:  return ATTACK  else:  # Initially, it will cooperate (Farm)  if other\_id in self.memory:  # If the other agent has ever attacked, retaliate  if ATTACK in self.memory[other\_id]:  self.has\_attacked = True  return ATTACK  return FARM # Start by cooperating | Incorrect: Attacks everyone if it has ever been attacked by anyone. |
| #The "Exploiter" strategy  # It starts by cooperating, then  # 1. If the other agent has never attacked, it attacks.  # 2. If the other agent also attacks on its first move, it keeps attacking.  # 3. If the other agent farmed on its first move, it attacks until the other agent attacks. Then:  # 3a. It farms twice.  # 3b. From then on it copies the other agent's last move. | class Exploiter(Group):  color = "tab:purple" # Unique color for Exploiter agents    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .015, \_\_class\_\_.color) # Slightly different speed for the Exploiter    def next\_move(self, other\_id):  # If the agent has no memory of the other agent's previous moves, it starts by farming  if other\_id not in self.memory:  return FARM    # Otherwise, check the other agent's move history and apply the strategy  other\_last\_move = self.memory[other\_id][-1]    # If the other agent has never attacked, Exploiter attacks  if other\_last\_move == FARM and len(self.memory[other\_id]) == 1:  return ATTACK    # If the other agent attacked on the first move, Exploiter keeps attacking  if other\_last\_move == ATTACK and len(self.memory[other\_id]) == 1:  return ATTACK    # If the other agent farmed on the first move, Exploiter attacks until the other agent attacks  if other\_last\_move == FARM and len(self.memory[other\_id]) == 1:  return ATTACK    # After the other agent attacks, Exploiter will farm twice  if len(self.memory[other\_id]) == 2 and other\_last\_move == ATTACK:  return FARM    # After farming twice, Exploiter copies the other agent's last move  if len(self.memory[other\_id]) > 2:  return self.memory[other\_id][-1] | Incorrect |
|  | class Exploiter(Group):  color = "tab:red" # Choosing red for the Exploiter's color  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = True # Track if it's the first interaction  def next\_move(self, other\_id):  if self.first\_move:  self.first\_move = False # Mark that the first move has been made  return FARM # Initially cooperate (farm)  # Now that it's not the first move, we follow the strategy  other\_last\_move = self.memory.get(other\_id, [None])[-1] # Get the last move of the other agent  if other\_last\_move is None:  return ATTACK # If we have no history, attack  if other\_last\_move == ATTACK:  return ATTACK # If the other agent attacked, we continue attacking  if other\_last\_move == FARM:  # If the other agent farmed, we attack until they attack, then farm twice  if self.memory[other\_id][-2:] == [FARM, FARM]:  return FARM # After farming twice, follow their moves  return ATTACK # Otherwise, keep attacking | Incorrect |
|  | class Exploiter(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  self.first\_move = True  self.other\_first\_move = True  self.other\_first\_move\_action = None  def next\_move(self, other\_id):  if self.first\_move:  # On the first move, it cooperates.  self.first\_move = False  return FARM    # After the first move:  if self.other\_first\_move:  # If the other agent never attacked, it attacks.  if other\_id not in self.memory or (len(self.memory[other\_id]) == 0 and self.memory.get(other\_id, [])[0] != ATTACK):  self.other\_first\_move\_action = ATTACK  self.other\_first\_move = False  return ATTACK  # If the other agent attacked, it continues attacking.  elif self.memory[other\_id][-1] == ATTACK:  return ATTACK  # If the other agent farmed, it attacks until the other attacks, then it farms twice.  else:  self.other\_first\_move\_action = ATTACK  return ATTACK    # After the first move, it follows the strategy based on previous behavior:  if self.other\_first\_move\_action == ATTACK:  # If the other agent farmed on its first move, it attacks until the other attacks, then farms twice.  if self.memory[other\_id][-1] == FARM:  return FARM # First farming turn after attack  elif self.memory[other\_id][-1] == ATTACK:  return ATTACK # Keep attacking if the other is attacking  else:  # If the other agent has attacked, it copies the other agent's last move.  return self.memory[other\_id][-1] if other\_id in self.memory else FARM | Incorrect |
| #Once-burned, twice-shy strategy  # On the first move, if the most recent move of the most recently-added key in the memory is an attack, it will attack. On subsequent moves copy the other agent's last move. | class OnceBurnedTwiceShy(Group):  color = "tab:purple" # Assign a unique color  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  # Check if the agent has interacted with this other agent before  if other\_id not in self.memory:  return FARM # Default to farming if no memory exists  else:  # Get the last move of the other agent  last\_move = self.memory[other\_id][-1]    # First move: Check the last move of the most recently added memory  if len(self.my\_actions\_memory[other\_id]) == 1:  # If the other agent attacked in the last interaction, this agent attacks  if last\_move == ATTACK:  return ATTACK  else:  return FARM  # Subsequent moves: Mimic the other agent's last move  else:  return self.memory[other\_id][-1]    def record\_moves(self, my\_move, id\_num, move):  # Record the moves similarly to other agents, but we need to make sure  # that the once-burned logic is handled correctly (add a move only if it's the first or subsequent move).  super().record\_moves(my\_move, id\_num, move) | Incorrect |
|  | class OnceBurnedTwiceShy(Group):  color = "tab:purple"    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)  def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  return FARM # Default to farming if no memory of the other agent  # If there is a previous record, we will check the last move  last\_move = self.memory[other\_id][-1]  if last\_move == ATTACK:  return ATTACK # Attack if the other agent last attacked  return self.memory[other\_id][-1] # Otherwise, copy the last move of the other agent | Incorrect |
|  | class OnceBurnedTwiceShy(Group):  color = "tab:blue" # Assigning a color for visualization  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  # If this is the first interaction, we check the most recent move of the other agent  if other\_id in self.memory:  # If the other agent attacked last, we attack  if self.memory[other\_id][-1] == ATTACK:  return ATTACK  else:  # If not, we cooperate (FARM)  return FARM  else:  # If no memory, cooperate (FARM)  return FARM | Incorrect |
| #Random TitTat  # When first meeting a new agent, it randomly chooses to farm or attack, then does whatever the other agent did last time. | class RandomTitTat(Group):  color = "tab:purple" # You can choose any color that is not already used    def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  if other\_id not in self.memory:  # If no history with this agent, randomly choose to attack or farm  move = random.choice([ATTACK, FARM])  self.memory[other\_id] = [move] # Record the first move  self.my\_actions\_memory[other\_id] = [move]  return move  else:  # After the first encounter, repeat the move that the other agent did  return self.memory[other\_id][-1] | Correct/Technically incorrect, records the first interaction twice. |
|  | import random  # The "Random TitTat" strategy  # When first meeting a new agent, it randomly chooses to farm or attack,  # then does whatever the other agent did last time.  class RandomTitTat(Group):  color = "tab:purple" # You can choose a color for this strategy  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  if other\_id not in self.memory.keys():  # If it's the first time meeting this agent, choose randomly  return random.choice([ATTACK, FARM])  else:  # Otherwise, do whatever the other agent did last time  return self.memory[other\_id][-1] | Correct |
|  | import random  class RandomTitTat(Group):  color = "tab:purple" # New color for this strategy  def \_\_init\_\_(self, x, y):  super().\_\_init\_\_(x, y, .01, \_\_class\_\_.color)    def next\_move(self, other\_id):  # If we have not encountered this agent before, choose randomly  if other\_id not in self.memory.keys():  return random.choice([ATTACK, FARM])  else:  # Otherwise, do whatever the other agent did last time  return self.memory[other\_id][-1] | Correct |