DASH MASTER

INTRODUCTION:

A simple multiplayer game using the pygame library, combined with elements of reinforcement learning (Q-learning) for agent behaviour. The game involves players represented by cars moving on a grid. The objective is to accumulate points by dashing into other players while avoiding collisions. The reinforcement learning aspect is used to train the players to make optimal movements over multiple episodes.

Key components of the code include:

1. **Game Setup:** Players are initialized on a grid, each with its own starting position and image. pygame is used to create a graphical window for displaying the game.
2. **Player Class:** Represents a player in the game with attributes such as position, score, and image. Supports movement, dashing into other players, and resetting to initial state.
3. **Q-learning Functions:** Functions for updating Q-values based on state-action-reward transitions. Exploration-exploitation strategy for choosing actions.
4. **Game Loop:** The main loop runs for a specified number of episodes. Within each episode, players move according to the Q-learning strategy and accumulate scores. Collision events are detected, and scores are updated accordingly.

5. **Visualization:** pygame is used for visualizing the game grid, player positions, scores, and remaining time. The game is slowed down for visualization purposes.
6. **Training Loop:** Q-values are updated during the game loop to train the players. Winner and scores are printed after each episode.
7. **Results Plotting:** Matplotlib is used to plot the winner's score against the episode number.
8. **Game Over Display:** After all episodes, the winners and their scores are displayed for a short period.
9. **Final Plot Display:** A plot of winner scores against episode numbers is displayed using Matplotlib.

ENVIRONMENT DETAILS:

This can be explained by understanding the environment code:

(i) Importing Libraries:

pygame, sys, random, time, numpy:

(ii) It initializes the pygame library, which is used for creating the game window and handling graphical elements. Here, some constants are defined, such as the size of the grid, cell size, screen size, and colour values.

(iii) Player Class and Reset function:

```python
# Player class
class Player:
    def __init__(self, x, y, image_path):
        self.x = x
        self.y = y
        self.score = 0
        self.image = pygame.image.load(image_path)
        self.initial_x = x  # Store the initial x position
        self.initial_y = y  # Store the initial y position
    def reset(self):
        # Reset the player's state to the initial position and score
        self.x = self.initial_x
        self.y = self.initial_y
        self.score = 0

    def move(self, dx, dy):
        new_x = (self.x + dx) % GRID_SIZE
        new_y = (self.y + dy) % GRID_SIZE

        # Check if the new position is within the boundaries
        if 0 <= new_x < GRID_SIZE and 0 <= new_y < GRID_SIZE:
            self.x = new_x
            self.y = new_y
        else:
            # Handle the case when the new position is outside the
boundaries
            pass
```

The Player class constructor initializes a new instance with specified initial coordinates and an image file path. It stores current positions, sets the score to 0, and loads the player's image using pygame's image.load() function. The initial positions are retained for state resets. The move method adjusts the player's position within a grid, calculating new positions based on specified coordinate changes. It ensures the player wraps around the grid and updates positions within boundaries, preventing the player from exceeding grid limits.

(iv) <u>Reward Function</u>:

```
def dash(self, other_player):
        self.score += 1
        other_player.score -= 0.5
```

The **dash** method represents a player dashing into another player and takes another player (**other_player**) as a parameter. When this method is invoked, it increases the score of the player performing the dash (**self.score**) by 1. Simultaneously, it decreases the score of the other player passed as a parameter (**other_player**) by 0.5, simulating a penalty for being dashed. The use of a penalty value of 0.5 implies that each dash has a cost associated with it for the player being dashed into.

(v) Visualization Function:

```
# Draw the grid and players
    screen.fill(WHITE)

    # Draw the scores above the grid
    font = pygame.font.Font(None, 20)
    for i, player in enumerate(players):
        score_text = font.render(f"Player {i + 1} Score: {player.score}",
True, RED)
        screen.blit(score_text, (10 + i * 200, 550))

    # Draw the time remaining on the right side
    remaining_time = max(0, game_duration - (time.time() - start_time))
    time_text = font.render(f"Time: {int(remaining_time)} seconds", True, RED)
    screen.blit(time_text, (SCREEN_SIZE[0] - time_text.get_width() - 10, 10))

    # Draw the grid
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            pygame.draw.rect(screen, (200, 200, 200), (i * CELL_SIZE,  *
CELL_SIZE, CELL_SIZE, CELL_SIZE), 1)

    # Draw the car images
    for i, player in enumerate(players):
        car_image = pygame.transform.scale(car_images[i], (CELL_SIZE,
CELL_SIZE))
        screen.blit(car_image, (player.x * CELL_SIZE, player.y CELL_SIZE))

    pygame.display.flip()
    pygame.time.delay(500)  # Delay to slow down the game for visualization
purposes
```

The pygame screen is updated to visually represent the game environment. The screen is first filled with a white colour. Then, scores for each player are drawn above the grid, displaying the player number and their respective scores in red text. The remaining time of the game is displayed on the right side of the screen in red text. The grid is drawn by creating rectangles for each cell, forming a grid structure with a light gray colour. Finally, the car images representing each player are drawn on the grid. These images are scaled to match the size of the grid cells. The pygame display is updated, and a delay of 500 milliseconds is introduced for visualization purposes, slowing down the game for better observation.

(vi) Result Display Function:

```
# Find the winners (players with the highest score)
max_score = max(player.score for player in players)
winners = [player for player in players if player.score == max_score]

# Display the winners and their scores for 2 seconds
screen.fill(WHITE)
if winners:
    if len(winners) == 1:
        winner_text = font.render(f"Winner: Player {players.index(winners[0])
+ 1} - Score: {winners[0].score}", True, RED)
        screen.blit(winner_text, (SCREEN_SIZE[0] // 2 -
winner_text.get_width() // 2, SCREEN_SIZE[1] // 2 - winner_text.get_height()
// 2))
    else:
        winners_text = font.render("Winners:", True, RED)
        screen.blit(winners_text, (SCREEN_SIZE[0] // 2 -
winners_text.get_width() // 2, SCREEN_SIZE[1] // 2 - winners_text.get_height()
// 2))

        y_offset = SCREEN_SIZE[1] // 2
        for winner in winners:
            winner_text = font.render(f"Player {players.index(winner) + 1} -
Score: {winner.score}", True, RED)
            screen.blit(winner_text, (SCREEN_SIZE[0] // 2
winner_text.get_width() // 2, y_offset))
            y_offset += winner_text.get_height()
```

RL COMPONENT DETAILS (Training the agent using Q-Learning Algorithm):

(i) State Function:

```
# State Function: Defines a function (get_state) that returns the current
state of the game based on player positions.
```

```
def get_state(players):
    return frozenset((player.x, player.y) for player in players)
```

The **get_state** function is a state function that takes a list of players as input and returns the current state of the game. It does so by creating a **frozenset** containing the unique x and y positions of each player. This frozen set represents a snapshot of the game state based on the current positions of all players. The use of a **frozenset** ensures immutability and makes the state easily change, suitable for tasks like state comparison or storage in data structures that require changing elements.

(ii) <u>Q-Learning Update Function</u>:

```
#Q-learning Update Function: Defines a function (q_learning_update) for
updating Q-values based on the Q-learning algorithm.
def q_learning_update(q_values, state, action, reward, next_state, alpha,
gamma):
    flat_state = tuple(sorted(tuple(coord for pos in state for coord in pos)))
    flat_next_state = tuple(sorted(tuple(coord for pos in next_state for coord
in pos)))
    action_tuple = (action,)  # Convert action to a tuple

    if flat_state not in q_values:
        q_values[flat_state] = {action_tuple: 0.0}
    elif action_tuple not in q_values[flat_state]:
        q_values[flat_state][action_tuple] = 0.0

    q_values[flat_state][action_tuple] += alpha * (reward + gamma *
max(q_values.get(flat_next_state, {}).values(), default=0.0) -
q_values[flat_state][action_tuple])
```

The **q_learning_update** function updates Q-values based on the Q-learning algorithm:

Inputs: current Q-values (**q_values**), current state (**state**), performed action (**action**), received reward (**reward**), next state (**next_state**), and learning parameters (**alpha** and **gamma**). Flattens state and next_state into sorted tuples for consistency. Converts the action to a tuple for dictionary use. Updates Q-value using the Q-learning formula, considering learning rate (**alpha**), received reward, and the maximum Q-value for the next state. Initializes dictionary entries if state or action is not present.

(iii) Action Function:

```
def choose_action(q_values, state, epsilon):
    if state not in q_values or np.random.uniform(0, 1) < epsilon:
        return np.random.choice(4)  # Assuming 4 possible actions (four
directions to move)
    else:
        return np.argmax(q_values[state])
```

The **choose_action** function selects actions in a Q-learning scenario: Inputs: current Q-values (**q_values**), current state (**state**), and exploration-exploitation parameter (**epsilon**). If state not in Q-values or a random value is below **epsilon**, it returns a random action. If the state is in Q-values and the random value is above **epsilon**, it returns the action with the maximum Q-value, favoring exploitation for the highest expected cumulative reward.

(iv) RL and Training Parameters:

```
# RL parameters
q_values = dict()
epsilon = 0.05
learning_rate = 0.05
discount_factor = 0.6

# Training parameters
num_episodes = 1000
game_duration_per_episode = 15  # seconds
```

RL parameters and training parameters for the Q-learning algorithm are initialized:

q_values: A dictionary for Q-values, initialized as empty.

epsilon: Exploration-exploitation parameter set to 0.05 for action selection balance.

learning_rate: Set to 0.05, determining the influence of new information on Q-value updates.

discount_factor: Set to 0.6, representing the importance of future rewards in updates.

num_episodes: Number of training episodes set to 1000 for agent-environment interactions.

game_duration_per_episode: Duration per episode set to 15 seconds, defining the time limit for interactions with the environment.

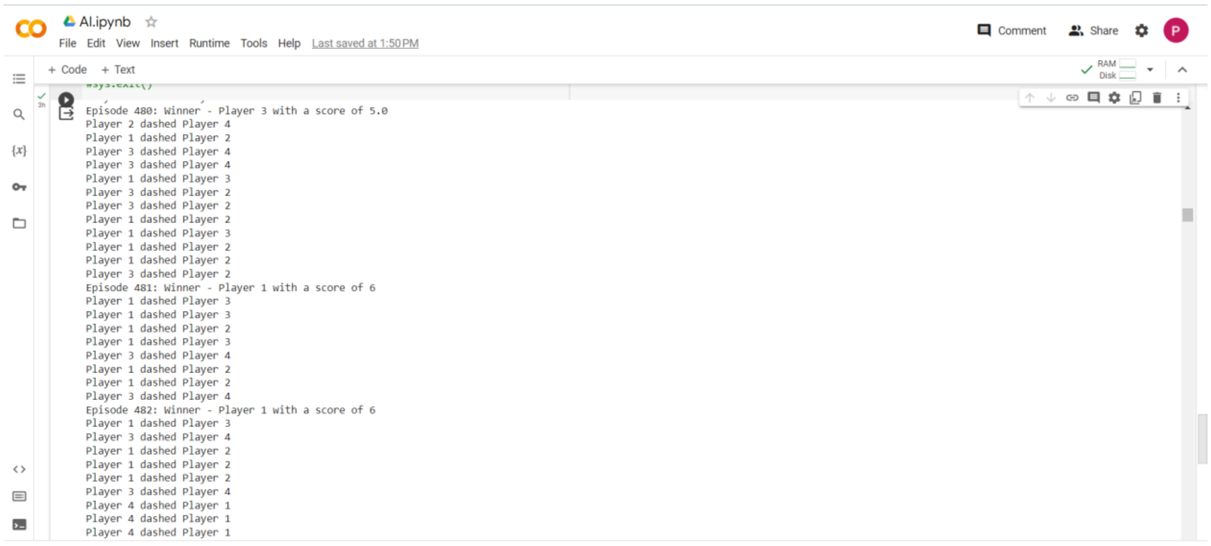
(v) <u>Multiple Episodes Training using RL</u>:

```python
# Game loop for training multiple episodes
for episode in range(num_episodes):
    # Reset players and Q-values for a new episode
    for player in players:
        player.reset()

    q_values.clear()  # Reset Q-values for a new episode

    start_time = time.time()
    while time.time() - start_time < game_duration_per_episode:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        # Handle player movements using RL
        for i, player in enumerate(players):
            state = get_state(players)
            action = choose_action(q_values, state, epsilon)
            dx, dy = [(0, 1), (1, 0), (0, -1), (-1, 0)][action]
            player.move(dx, dy)

            for other_player in players:
                if player != other_player and player.x == other_player.x and
player.y == other_player.y:
                    player.dash(other_player)
                    print(f"Player {players.index(player) + 1} dashed Player
{players.index(other_player) + 1}")

            next_state = get_state(players)
            reward = player.score

            # Convert state and next_state to flat tuples
            state_key = tuple(state)
            next_state_key = tuple(next_state)

            q_learning_update(q_values, state_key, action, reward,
next_state_key, learning_rate, discount_factor)
```

The code implements a game loop for training RL agents using a Q-learning algorithm. It iterates over a specified number of episodes, resetting player positions and Q-values. The loop runs for a set duration, handling player movements, interactions, and updates Q-values accordingly. The algorithm selects actions based on the current state and employs an exploration-exploitation strategy. A dash action is performed when players occupy the same position, and Q-values are updated. The loop also handles quitting the game window. Overall, the code simulates RL agent interactions, refining Q-values based on observed states and rewards.

OUTPUT:

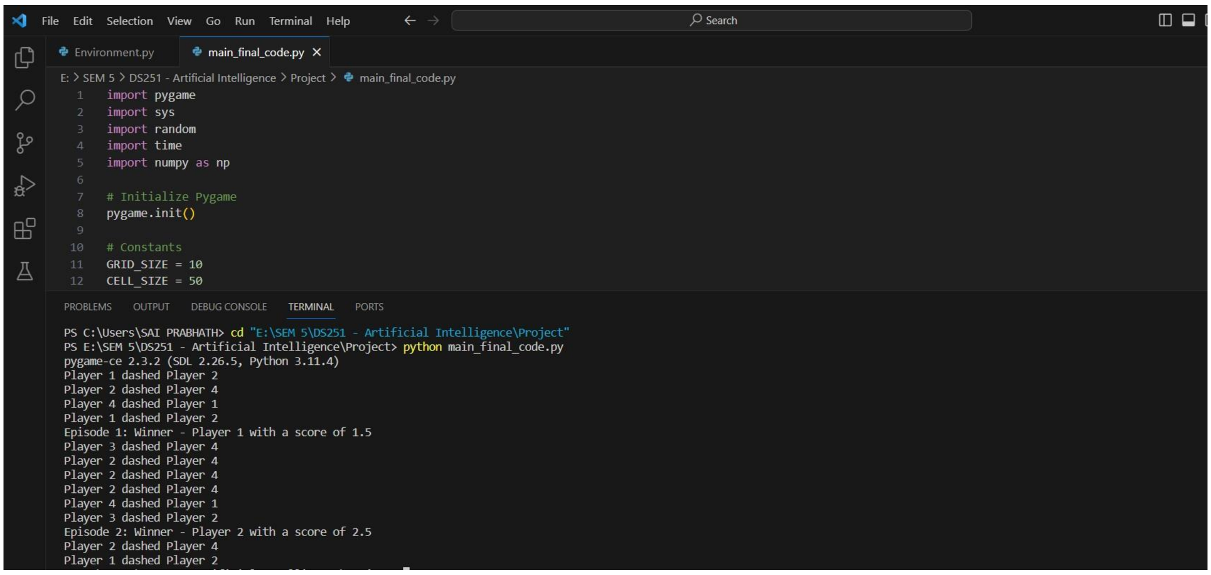
The code simulates a game environment with RL agents using a Q-learning algorithm. In each episode, it prints actions like "Player 1 dashed Player 2." The console output also reveals the winner of each episode along with their score.


AI.ipynb
File Edit View Insert Runtime Tools Help Last saved at 1:50 PM
Comment Share
+ Code + Text

```
#sys.exit()
Episode 480: Winner - Player 3 with a score of 5.0
Player 2 dashed Player 4
Player 1 dashed Player 2
Player 3 dashed Player 4
Player 3 dashed Player 4
Player 1 dashed Player 3
Player 3 dashed Player 2
Player 3 dashed Player 2
Player 1 dashed Player 2
Player 1 dashed Player 3
Player 1 dashed Player 2
Player 1 dashed Player 2
Player 3 dashed Player 2
Episode 481: Winner - Player 1 with a score of 6
Player 1 dashed Player 3
Player 1 dashed Player 3
Player 1 dashed Player 2
Player 1 dashed Player 3
Player 3 dashed Player 4
Player 1 dashed Player 2
Player 1 dashed Player 2
Player 3 dashed Player 4
Episode 482: Winner - Player 1 with a score of 6
Player 1 dashed Player 3
Player 3 dashed Player 4
Player 1 dashed Player 2
Player 1 dashed Player 2
Player 1 dashed Player 2
Player 3 dashed Player 4
Player 4 dashed Player 1
Player 4 dashed Player 1
Player 4 dashed Player 1
```

1. **pygame Window:** Displayed during each episode. Shows the game grid, player positions, scores, and remaining time. Grid drawn with square, player cars move based on RL agent decisions. Player scores displayed above the grid. Remaining time shown on the right side.

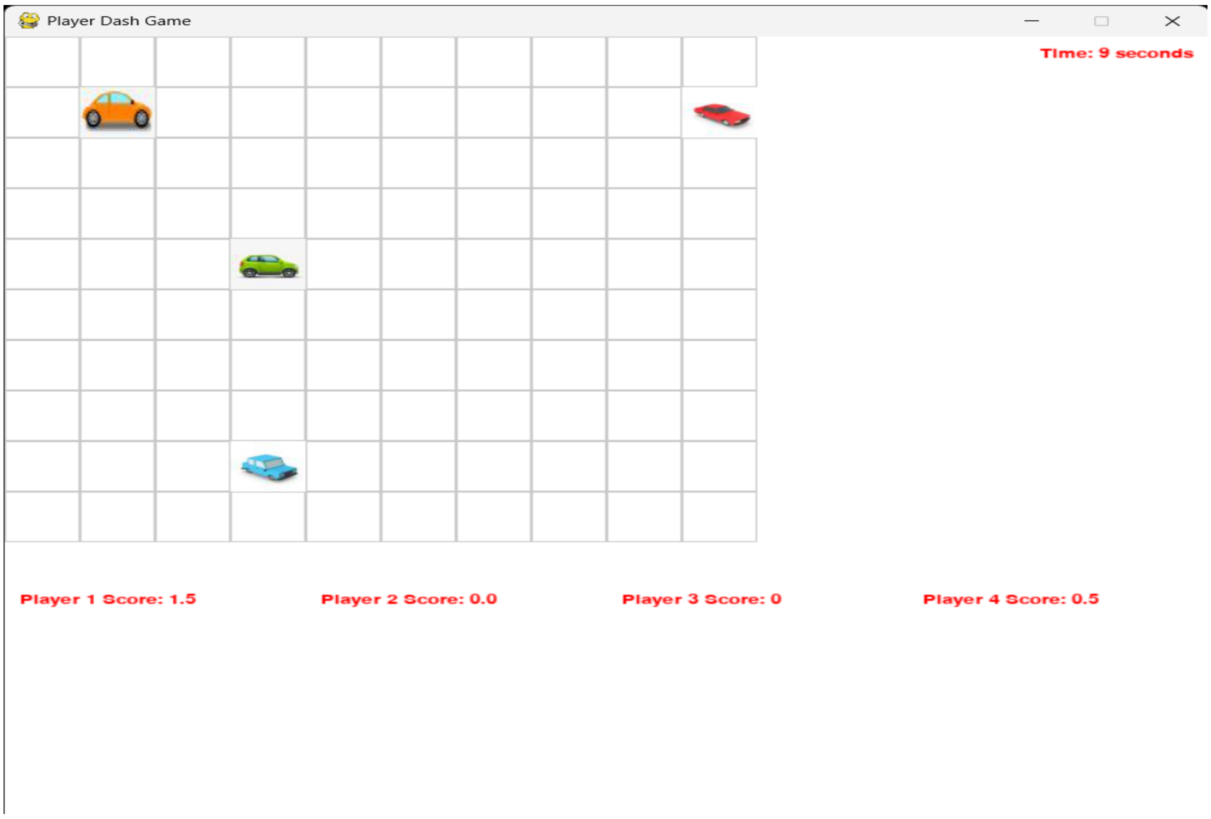
2. **Final Winners Display:** After all episodes, winners determined by highest scores. pygame window updated briefly to show winners and their scores.


File Edit Selection View Go Run Terminal Help
Search
Environment.py
main_final_code.py
E: > SEM 5 > DS251 - Artificial Intelligence > Project > main_final_code.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```python
import pygame
import sys
import random
import time
import numpy as np

# Initialize Pygame
pygame.init()

# Constants
GRID_SIZE = 10
CELL_SIZE = 50
```

```
PS C:\Users\SAI PRABHATH> cd "E:\SEM 5\DS251 - Artificial Intelligence\Project"
PS E:\SEM 5\DS251 - Artificial Intelligence\Project> python main_final_code.py
pygame-ce 2.3.2 (SDL 2.26.5, Python 3.11.4)
Player 1 dashed Player 2
Player 2 dashed Player 4
Player 4 dashed Player 1
Player 1 dashed Player 2
Episode 1: Winner - Player 1 with a score of 1.5
Player 3 dashed Player 4
Player 2 dashed Player 4
Player 2 dashed Player 4
Player 2 dashed Player 4
Player 4 dashed Player 1
Player 3 dashed Player 2
Episode 2: Winner - Player 2 with a score of 2.5
Player 2 dashed Player 4
Player 1 dashed Player 2
```

3. **Visualization Delays**: Delays are introduced in the pygame window for visualization purposes, allowing you to observe the game state and movements.

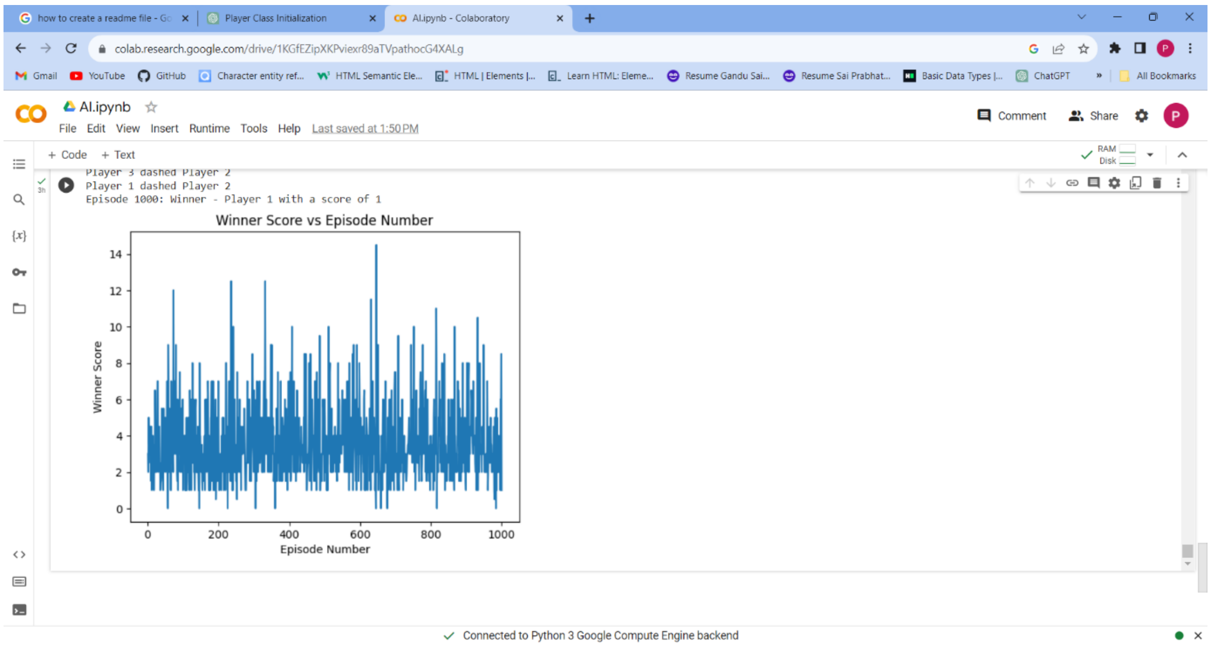


RL GRAPH:

Plotting the graph between episodes and Highest score of the player (Winner in each episode):

```
import matplotlib.pyplot as plt
# Lists to store episode numbers and winner scores
episode_numbers = []
winner_scores = []
# Game loop for training multiple episodes
for episode in range(num_episodes):
    # Reset players and Q-values for a new episode
    for player in players:
        player.reset()
    # Print the winner and scores for each episode

    winner = max(players, key=lambda player: player.score)
    print(f"Episode {episode + 1}: Winner - Player {players.index(winner) + 1}
with a score of {winner.score}")

    # Store episode number and winner score for plotting
    episode_numbers.append(episode + 1)
    winner_scores.append(winner.score)
# Plotting the results
plt.plot(episode_numbers, winner_scores)
plt.title('Winner Score vs Episode Number')
plt.xlabel('Episode Number')
plt.ylabel('Winner Score')
plt.show()
```

The code initializes lists to store episode numbers and corresponding winner scores. It runs a game loop for a specified number of training episodes, determining winners based on scores. Episode and winner score data are collected for later plotting. Matplotlib is used to create a line plot, visualizing winner scores over episodes. The plot is displayed with episode numbers on the x-axis and winner scores on the y-axis.



```
Player 3 dashed Player 2
Player 1 dashed Player 2
Episode 1000: Winner - Player 1 with a score of 1
```

The resulting plot provides a visual representation of how the winner's score changes over the course of the training episodes, offering insights into the learning progress of the RL agents.