1.Collision Detection Calculations

Snake-Prey Collision:

In order to determine when the snake captures the prey we used rectangular bounding boxes. Each game element (snake segments and prey) is treated as an axis-aligned rectangle on the canvas.In our implementation, both the snake's head and the prey are represented as squares of equal dimensions, governed by constants `SNAKE_ICON_WIDTH` and `PREY_ICON_WIDTH`. We ensured they match in size to simplify the collision detection logic.

- **Prey Coordinates:** Defined as `(x, y, x + PREY_ICON_WIDTH, y + PREY_ICON_WIDTH).`
- **Snake Head Coordinate:** A single `(headX, headY)` tuple representing the top-left corner of the snake's head segment.

Coordinate System:

- The game canvas has a top-left origin `(0, 0)` and positive x and y directions extending rightwards and downwards respectively.
- Each segment of the snake, including its head, is represented by a coordinate `(x, y)` corresponding to the segment's center.
- The prey is generated as a square positioned around a chosen `(x, y)` point, forming a bounding box `(x - half, y - half, x + half, y + half)` where `half = PREY_ICON_WIDTH // 2.`

Collision Condition:
To decide if the snake captures the prey, we check for bounding box overlap.

- Let `(sx1, sy1, sx2, sy2)` be the snake's head bounding box and `(px1, py1, px2, py2)` be the prey's bounding box.

A collision occurs if:
`sx1 < px2 and sx2 > px1 and sy1 < py2 and sy2 > py1`

- In other words, the boxes overlap horizontally and vertically.

By using consistent sizes for snake head and prey, we ensure that if the snake's head moves onto the prey's position, a point is awarded immediately. This approach balances realism and simplicity, making the logic easy to adjust if `PREY_ICON_WIDTH` or `SNAKE_ICON_WIDTH` change.

2. Coordinate Updates for Snake Movement

Movement Logic:

- The snake moves in increments of `SNAKE_ICON_WIDTH` in one of four directions: Up, Down, Left, Right.
- Each movement adds a new head coordinate at the front of `snakeCoordinates`.
- If the snake does not eat prey, the tail segment is removed, preserving the snake's length.
- If the snake eats prey, the tail remains, increasing the snake's length by one segment.

Direction Handling:

- The direction is updated via arrow key presses, with checks to prevent immediate reversal (for instance from Left to Right directly).
- To move, we take the current head position `(headX, headY)` and adjust either `headX` or `headY` by ±`SNAKE_ICON_WIDTH` depending on the direction.

3. Prey Creation

Placement Considerations:

- Prey is placed randomly on the canvas with a minimum threshold distance from the walls. This avoids having prey too close to the boundaries where it might be hard to capture.
- The `(x, y)` chosen for prey is then converted into a square bounding box. With the chosen dimensions, `(x, y)` essentially becomes the center of the prey square.

Avoiding Overlap With Snake:

- Before finalizing the prey's position, we ensure it does not overlap with any snake segment. We check that no snake coordinate falls within the new prey's bounding box.
- If overlap occurs, we re-select random coordinates until a valid position is found.

This ensures fairness, preventing prey from appearing inside the snake's body or in unreachable positions.

Wall and self collision:

- **Wall Collision:**
  Compare the snake's head coordinates `(x, y)` to the boundaries of the game area. If `x < 0` or `x >= WINDOW_WIDTH` or `y < 0` or `y >= WINDOW_HEIGHT`, the head has moved outside the playable area, and the game should end.
- **Self-Collision:**
  The snake's body is stored as a list of `(x, y)` tuples. After adding a new head

coordinate, if that head coordinate appears in the rest of the snake's body, it means the snake has collided with itself. Checking `if snakeCoordinates in self.snakeCoordinates[:-1]` ensures we don't consider the newly added head twice.

4. Challenges Faced

1. Collision Precision: Initially, using a single point check (e.g., the snake head's coordinate against the prey's top-left corner) could result in the snake seemingly "passing through" the prey without detection. By switching to bounding box overlap, collisions became more reliable and intuitive.

2. Synchronization and Timing: Since the snake moves in discrete steps, timing the capture event was important. Sometimes, if increments were too large or the prey was not aligned to the snake's movement grid, captures might fail to register. Aligning increments and using equal sizes simplified collision detection.

3. Prey Placement Fairness: Ensuring the prey didn't spawn on the snake or too close to walls required repeated random attempts until a suitable position was found. While simple, this method might be inefficient if the snake gets very large.

5. Potential Improvements (For Future Consideration)

- **Dynamic Difficulty:**
  Gradually increase the snake's speed or introduce obstacles as the player's score rises.
- **Advanced Collision Detection:** For a more complex or visually varied game, consider more sophisticated collision techniques, such as pixel-level checks or using circular hitboxes.
- **Optimized Prey Placement:** Instead of random attempts, use a spatial partitioning or a grid-based approach to efficiently find valid prey positions, especially useful for large snakes or bigger boards.
- **Additional Features:** Consider adding special power-ups, multiple prey types with different scores, or implementing a smoother movement animation for visual appeal.