import tkinter as tk

import numpy as np

## # Grid size

GRID\_SIZE = 5

CELL\_SIZE = 100  # Each cell is 100x100 pixels

## # Initial positions

player\_pos = [0, 0]  # Player starts at top-left

random\_int1 = np.random.randint(1, 5)

random\_int2 = np.random.randint(1, 5)

enemy\_pos = [random\_int1, random\_int2]

goal\_pos = [4, 4]    # Goal is at bottom-right

## # Movement directions (row, col)

'''

This dictionary is usually used in grid-based games or simulations (like a maze, chessboard, or snake game)

to represent movements in a 2D grid. Each key is a direction, and each value is a tuple that represents how coordinates

change when you move in that direction.

'''

MOVES = {

    "Up": (-1, 0), # Row decrement as we go up ie., Move 1 row up, column stays the same

    "Down": (1, 0), # Row increment as we go down ie., Move 1 row down, column stays the same

    "Left": (0, -1), # Column decrement as we go left ie., Move 1 column left, row stays the same

    "Right": (0, 1), # Column increment as we go right ie., Move 1 column right, row stays the same

    }

## # Create GUI window

root = tk.Tk()

root.title("Grid Game: Reach the Goal!")

## # Canvas setup

canvas = tk.Canvas(root, width=GRID\_SIZE \* CELL\_SIZE, height=GRID\_SIZE \* CELL\_SIZE, bg="white")

canvas.pack()

## # Draw grid lines

for i in range(GRID\_SIZE + 1):

    canvas.create\_line(i \* CELL\_SIZE, 0, i \* CELL\_SIZE, GRID\_SIZE \* CELL\_SIZE, fill="black")

    canvas.create\_line(0, i \* CELL\_SIZE, GRID\_SIZE \* CELL\_SIZE, i \* CELL\_SIZE, fill="black")

'''

1. canvas.create\_rectangle(...)

This is a Tkinter method that draws a rectangle on the canvas. It takes coordinates of the top-left and bottom-right corners:

canvas.create\_rectangle(x1, y1, x2, y2, fill="color")

(x1, y1) → top-left corner

(x2, y2) → bottom-right corner

fill → color of the rectangle

2. goal\_pos[1] \* CELL\_SIZE and goal\_pos[0] \* CELL\_SIZE

Here, goal\_pos is likely a tuple (row, column) representing the goal's position on the grid.

goal\_pos[0] → row number

goal\_pos[1] → column number

CELL\_SIZE is the size of each grid square in pixels.

So multiplying goal\_pos by CELL\_SIZE converts grid coordinates → pixel coordinates.

3. Why (goal\_pos[1] + 1) \* CELL\_SIZE and (goal\_pos[0] + 1) \* CELL\_SIZE?

We want the rectangle to fill one cell, not just be a point.

Adding +1 moves from the top-left corner to the bottom-right corner of that cell.

Example:

If goal\_pos = (2, 3) and CELL\_SIZE = 50:

Coordinate  Calculation     Result (pixels)

x1          3 \* 50          150

y1          2 \* 50          100

x2          (3+1) \* 50      200

y2          (2+1) \* 50      150

So the rectangle covers pixels (150, 100) to (200, 150) → exactly one cell.

4. fill="green"

The rectangle will be green, representing the “goal” visually on the grid.

'''

## # Draw goal (green square)

goal\_square = canvas.create\_rectangle(

    goal\_pos[1] \* CELL\_SIZE, goal\_pos[0] \* CELL\_SIZE,

    (goal\_pos[1]+1) \* CELL\_SIZE, (goal\_pos[0]+1) \* CELL\_SIZE,

    fill="green")

## # Draw enemy (red square)

enemy\_square = canvas.create\_rectangle(

    enemy\_pos[1] \* CELL\_SIZE, enemy\_pos[0] \* CELL\_SIZE,

    (enemy\_pos[1] + 1) \* CELL\_SIZE, (enemy\_pos[0] + 1) \* CELL\_SIZE,

    fill="red"

)

## # Draw player (blue square)

player\_square = canvas.create\_rectangle(

    player\_pos[1] \* CELL\_SIZE, player\_pos[0] \* CELL\_SIZE,

    (player\_pos[1] + 1) \* CELL\_SIZE, (player\_pos[0] + 1) \* CELL\_SIZE,

    fill="blue"

)

## # Function to display message and stop game

'''

Let’s break this step by step and explain everything, including why `//2` is used.

---

### The code:

```python

def show\_message(message, color):

    canvas.create\_text(

        GRID\_SIZE \* CELL\_SIZE // 2, GRID\_SIZE \* CELL\_SIZE // 2,

        text=message, font=("Arial", 24, "bold"), fill=color

    )

    root.unbind("<KeyPress>")  # Stop player movement

    root.after\_cancel(enemy\_movement)  # Stop enemy movement

```

---

### 1. \*\*Purpose of the function\*\*

`show\_message` is a \*\*helper function to display a message on the canvas\*\* (like “Game Over” or “You Win”) and then \*\*stop the game\*\* by preventing further movement.

---

### 2. \*\*`canvas.create\_text(...)`\*\*

This draws text on the canvas. Its parameters:

```python

canvas.create\_text(x, y, text=message, font=("Arial", 24, "bold"), fill=color)

```

\* `x` → horizontal position (in pixels)

\* `y` → vertical position (in pixels)

\* `text` → the message to show

\* `font` → font style, size, weight

\* `fill` → text color

---

### 3. \*\*`GRID\_SIZE \* CELL\_SIZE // 2`\*\*

Here’s why `//2` is used:

\* `GRID\_SIZE \* CELL\_SIZE` → \*\*total width or height of the canvas in pixels\*\*

  \* `GRID\_SIZE` = number of cells in the grid

  \* `CELL\_SIZE` = size of one cell in pixels

  \* Example: 10 cells × 50 pixels = 500 pixels

\* `// 2` → \*\*integer division by 2\*\* to find the \*\*center of the canvas\*\*

  \* `500 // 2 = 250` → place text at pixel 250

  \* `//` ensures it’s an \*\*integer\*\*, because Tkinter expects pixel positions as integers

✅ So this centers the text in the middle of the canvas.

---

### 4. \*\*`root.unbind("<KeyPress>")`\*\*

\* This \*\*disables all key press events\*\*, stopping the player from moving after the game ends.

---

### 5. \*\*`root.after\_cancel(enemy\_movement)`\*\*

\* `enemy\_movement` is likely a \*\*scheduled repeating function\*\* (like `root.after(1000, move\_enemy)`), moving the enemy automatically.

\* `after\_cancel` \*\*stops that scheduled movement\*\*, effectively freezing the game.

---

### ✅ Summary

\* `canvas.create\_text(... // 2, ... // 2)` → Draws the message \*\*in the center of the canvas\*\*.

\* `root.unbind("<KeyPress>")` → Stops player control.

\* `root.after\_cancel(enemy\_movement)` → Stops enemy movement.

'''

## # Function to display message and stop game

def show\_message(message, color):

    canvas.create\_text(

        GRID\_SIZE \* CELL\_SIZE // 2, GRID\_SIZE \* CELL\_SIZE // 2,

        text=message, font=("Arial", 24, "bold"), fill=color

    )

    root.unbind("<KeyPress>")  # Stop player movement

    root.after\_cancel(enemy\_movement)  # Stop enemy movement

## # Function to move enemy randomly and automatically

def move\_enemy():

    global enemy\_pos, enemy\_movement

    if player\_pos == enemy\_pos:  # If enemy catches player

        show\_message("Game Over!", "red")

        return

## # Choose a random move

    move = np.random.choice(list(MOVES.keys()))

    new\_pos = [enemy\_pos[0] + MOVES[move][0], enemy\_pos[1] + MOVES[move][1]]

'''

Let’s carefully break this down. This is part of \*\*enemy movement logic\*\* in a grid-based game.

---

### The code:

```python

# Choose a random move

move = np.random.choice(list(MOVES.keys()))

new\_pos = [enemy\_pos[0] + MOVES[move][0], enemy\_pos[1] + MOVES[move][1]]

```

---

### 1. \*\*`np.random.choice(list(MOVES.keys()))`\*\*

\* `MOVES` is a dictionary of possible moves (like you showed earlier):

```python

MOVES = {

    "Up": (-1, 0),

    "Down": (1, 0),

    "Left": (0, -1),

    "Right": (0, 1),

}

```

\* `MOVES.keys()` → `["Up", "Down", "Left", "Right"]`

\* `list(MOVES.keys())` → converts it to a Python list

\* `np.random.choice(...)` → randomly selects \*\*one direction\*\* from the list

✅ So `move` will be `"Up"`, `"Down"`, `"Left"`, or `"Right"` randomly.

---

### 2. \*\*`new\_pos = [enemy\_pos[0] + MOVES[move][0], enemy\_pos[1] + MOVES[move][1]]`\*\*

Here’s what’s happening:

\* `enemy\_pos` = current position of the enemy as `[row, col]`

\* `MOVES[move]` = the tuple corresponding to that direction, e.g., `"Up"` → `(-1, 0)`

So we are \*\*calculating the new position\*\* of the enemy after moving one step:

```python

new\_row = enemy\_pos[0] + MOVES[move][0]

new\_col = enemy\_pos[1] + MOVES[move][1]

new\_pos = [new\_row, new\_col]

```

\*\*Example:\*\*

If `enemy\_pos = [3, 2]` and `move = "Up"`:

\* `MOVES["Up"] = (-1, 0)`

\* `new\_pos = [3 + (-1), 2 + 0] = [2, 2]`

So the enemy moves \*\*up by 1 cell\*\*.

---

### ✅ Summary

\* `np.random.choice(list(MOVES.keys()))` → randomly pick a direction

\* `enemy\_pos[0] + MOVES[move][0], enemy\_pos[1] + MOVES[move][1]` → calculate the \*\*new position\*\* based on that direction

This is how the enemy moves randomly in the grid.

'''

## # Keep enemy inside the grid

    enemy\_pos[0] = np.clip(new\_pos[0], 0, GRID\_SIZE - 1)

    enemy\_pos[1] = np.clip(new\_pos[1], 0, GRID\_SIZE - 1)

This code is all about **making sure the enemy doesn’t go outside the boundaries of the grid**. Let’s explain it clearly.

**1. np.clip(value, min\_val, max\_val)**

* np.clip is a NumPy function that **limits a value to stay within a range**.
* Rules:
  + If value < min\_val, it becomes min\_val
  + If value > max\_val, it becomes max\_val
  + Otherwise, it stays the same

**Example:**

np.clip(5, 0, 10) # returns 5

np.clip(-2, 0, 10) # returns 0

np.clip(15, 0, 10) # returns 10

**2. Why 0 and GRID\_SIZE - 1?**

* 0 → the first row/column of the grid
* GRID\_SIZE - 1 → the last row/column of the grid
* This ensures that the enemy **stays inside the grid** and doesn’t move off-screen.

**3. How it works**

* enemy\_pos[0] = np.clip(new\_pos[0], 0, GRID\_SIZE - 1)  
  → Makes sure the **row position** stays between 0 and GRID\_SIZE-1
* enemy\_pos[1] = np.clip(new\_pos[1], 0, GRID\_SIZE - 1)  
  → Makes sure the **column position** stays between 0 and GRID\_SIZE-1

## # Update enemy position on canvas

    canvas.coords(

        enemy\_square,

        enemy\_pos[1] \* CELL\_SIZE, enemy\_pos[0] \* CELL\_SIZE,

        (enemy\_pos[1] + 1) \* CELL\_SIZE, (enemy\_pos[0] + 1) \* CELL\_SIZE

    )

‘’’

This code **updates the enemy’s position on the Tkinter canvas**.

**1. canvas.coords()**

* canvas.coords(item, x1, y1, x2, y2) **changes the position or size of a shape** already drawn on the canvas.
* item → the ID of the shape (here enemy\_square)
* (x1, y1) → top-left corner in pixels
* (x2, y2) → bottom-right corner in pixels

✅ So this moves or resizes the rectangle on the canvas without creating a new one.

**2. enemy\_pos[1] \* CELL\_SIZE and enemy\_pos[0] \* CELL\_SIZE**

* enemy\_pos = [row, col] → enemy’s current grid position
* CELL\_SIZE → size of one grid cell in pixels
* Multiply by CELL\_SIZE → convert **grid coordinates to pixels**
* enemy\_pos[1] \* CELL\_SIZE → x-coordinate (column)
* enemy\_pos[0] \* CELL\_SIZE → y-coordinate (row)

**3. (enemy\_pos[1] + 1) \* CELL\_SIZE and (enemy\_pos[0] + 1) \* CELL\_SIZE**

* Add +1 to cover **the full cell**, not just the top-left corner.
* This gives the **bottom-right corner** of the rectangle.

**Example:**

If enemy\_pos = [2, 3] and CELL\_SIZE = 50:

| **Coordinate** | **Calculation** | **Pixel value** |
| --- | --- | --- |
| x1 | 3 \* 50 | 150 |
| y1 | 2 \* 50 | 100 |
| x2 | (3+1) \* 50 | 200 |
| y2 | (2+1) \* 50 | 150 |

✅ The rectangle now perfectly covers **the new cell (2,3)**.

## # Check for collision with player

    if player\_pos == enemy\_pos:

        show\_message("Game Over!", "red")

    else:

        enemy\_movement = root.after(500, move\_enemy)  # Move enemy every 500ms

**1. if player\_pos == enemy\_pos:**

* player\_pos → the current grid coordinates of the player, e.g., [row, col]
* enemy\_pos → the current grid coordinates of the enemy, e.g., [row, col]
* == checks if both lists are **exactly the same**, i.e., the player and enemy are on the same cell

✅ This is how the game **detects a collision**.

**2. show\_message("Game Over!", "red")**

* If the player and enemy are in the same cell, the game **ends**.
* show\_message displays **“Game Over!”** in red on the canvas and **stops all movement** (player and enemy).

**3. else:**

* If the enemy **has not collided with the player**, the game continues.

**4. enemy\_movement = root.after(500, move\_enemy)**

* root.after(delay\_ms, function) → schedules a function to run **after a delay in milliseconds**
* 500 → 500ms = 0.5 seconds
* move\_enemy → the function that moves the enemy

So this line **moves the enemy every 0.5 seconds** repeatedly by scheduling move\_enemy again.

‘’’

## # Move player with arrow keys

def move\_player(event):

    global player\_pos

    move = None

    if event.keysym == "Up":

        move = "Up"

    elif event.keysym == "Down":

        move = "Down"

    elif event.keysym == "Left":

        move = "Left"

    elif event.keysym == "Right":

        move = "Right"

    if move:

        new\_pos = [player\_pos[0] + MOVES[move][0], player\_pos[1] + MOVES[move][1]]

        if 0 <= new\_pos[0] < GRID\_SIZE and 0 <= new\_pos[1] < GRID\_SIZE:

            player\_pos[:] = new\_pos

            canvas.coords(

                player\_square,

                player\_pos[1] \* CELL\_SIZE, player\_pos[0] \* CELL\_SIZE,

                (player\_pos[1] + 1) \* CELL\_SIZE, (player\_pos[0] + 1) \* CELL\_SIZE

            )

**1. def move\_player(event):**

* Defines a function that handles **keyboard input**
* event → contains information about the key pressed (event.keysym)

**2. global player\_pos**

* player\_pos is defined outside the function (probably as [row, col])
* Using global allows the function to **modify the player's position** on the grid

**3. Check which key is pressed**

if event.keysym == "Up":

move = "Up"

elif event.keysym == "Down":

move = "Down"

elif event.keysym == "Left":

move = "Left"

elif event.keysym == "Right":

move = "Right"

* event.keysym → key symbol of the pressed key
* Assigns a direction ("Up", "Down", "Left", "Right") to move

✅ Only arrow keys are considered.

**4. Calculate the new position**

new\_pos = [player\_pos[0] + MOVES[move][0], player\_pos[1] + MOVES[move][1]]

* MOVES[move] gives (row\_change, col\_change) for the chosen direction
* Adds this to the current player\_pos → gets the **new position on the grid**

**Example:**  
If player\_pos = [2, 2] and move = "Up" → MOVES["Up"] = (-1, 0)

* new\_pos = [2 + (-1), 2 + 0] = [1, 2]

**5. Check if new position is inside the grid**

if 0 <= new\_pos[0] < GRID\_SIZE and 0 <= new\_pos[1] < GRID\_SIZE:

* Ensures **player does not move outside the grid**
* new\_pos[0] → row, new\_pos[1] → column

**6. Update player position**

player\_pos[:] = new\_pos

* Updates the player\_pos list **in place** so it reflects the new position

**7. Move the rectangle on the canvas**

canvas.coords(

player\_square,

player\_pos[1] \* CELL\_SIZE, player\_pos[0] \* CELL\_SIZE,

(player\_pos[1] + 1) \* CELL\_SIZE, (player\_pos[0] + 1) \* CELL\_SIZE

)

* canvas.coords updates the **rectangle representing the player**
* Converts **grid coordinates → pixels**
* The player rectangle moves **to the new cell**

**Example:**  
If player\_pos = [1, 2] and CELL\_SIZE = 50 → rectangle covers pixels (2\*50, 1\*50) to (3\*50, 2\*50)

## # Check for win condition

        if player\_pos == goal\_pos:

            show\_message("You Win!", "green")

## # Check for collision with enemy

        elif player\_pos == enemy\_pos:

            show\_message("Game Over!", "red")

## # Bind keyboard input for player movement

root.bind("<KeyPress>", move\_player)

## # Start automatic enemy movement

enemy\_movement = root.after(500, move\_enemy)

**1. root.after(delay, function)**

* root.after(ms, func) schedules a function func to run **after ms milliseconds**.
* ms = 500 → after 500 milliseconds (0.5 seconds)
* func = move\_enemy → the function that moves the enemy

✅ This does **not pause the program**; Tkinter continues running and calls move\_enemy after 500ms.

**2. Why assign to enemy\_movement?**

* root.after returns an **ID for the scheduled task**
* Storing it in enemy\_movement allows you to **cancel it later** if needed:

root.after\_cancel(enemy\_movement) # stops the scheduled enemy movement

**3. Effect in the game**

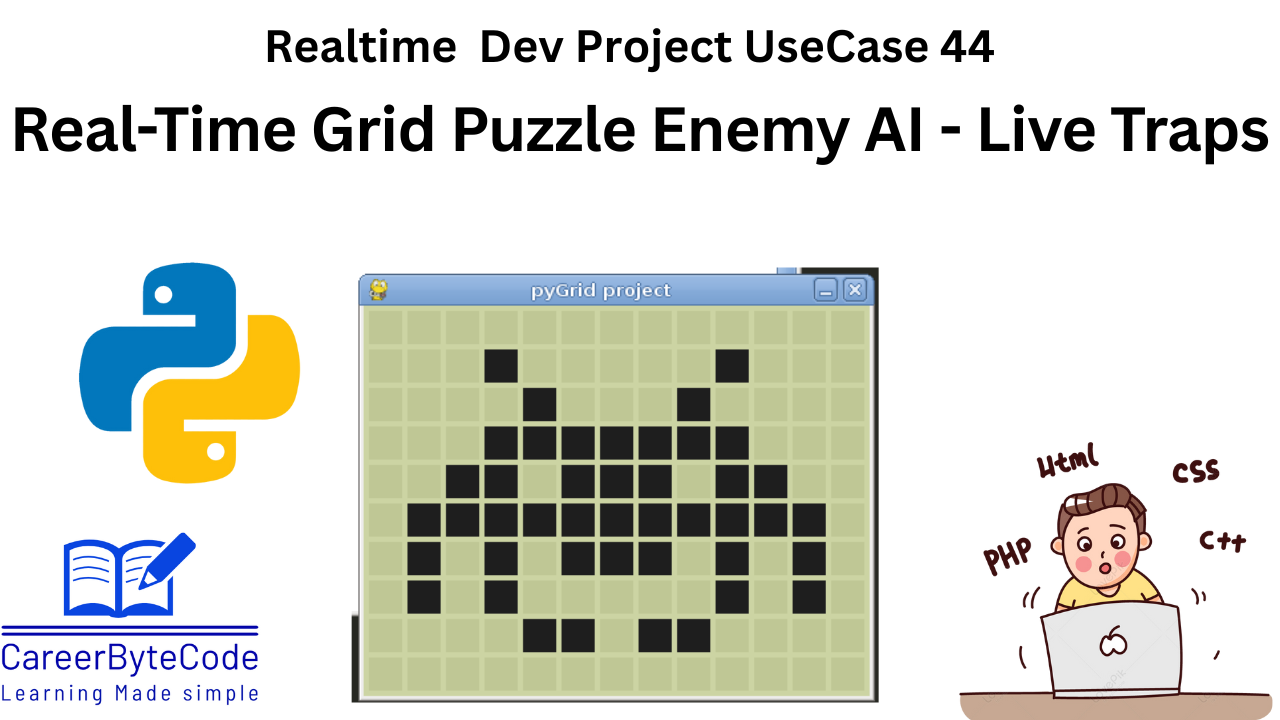
* The enemy will move **every 500ms**
* Each time move\_enemy finishes, it usually schedules itself again using root.after, creating a **loop**
* If the player collides with the enemy, you can stop this loop using after\_cancel

## # Run the GUI

root.mainloop()

**Hands-On Python Fun: Create a Grid-Based Game That Trains Both Your Code and Critical Thinking**

creative, gamified approach to learning GUI programming and logic-building with Python.

[[](https://substackcdn.com/image/fetch/$s_!CxmO!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbaffcff5-c1e9-45d5-8345-dff784849d57_1280x720.png)](https://substackcdn.com/image/fetch/$s_!CxmO!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbaffcff5-c1e9-45d5-8345-dff784849d57_1280x720.png" \t "_blank)

**1. Scenario**

You're trapped in a 5x5 grid world! Your mission is to safely navigate from the top-left corner to the bottom-right goal. But there's a twist—an enemy red square is chasing you down randomly! Can you outsmart the enemy and reach the green goal square before it's too late?

**2. Problem Statement**

Design a fun, interactive grid-based game using Python's Tkinter library. The player starts at the top-left of the grid and must reach the goal at the bottom-right. However, there's a randomly moving enemy. The player must avoid the enemy and reach the goal to win. The game should handle movement, collisions, and game-ending conditions.

**3. Why we need this use case**

This use case provides a creative, gamified approach to learning GUI programming and logic-building with Python. It’s perfect for beginners who want to move beyond basic Python scripts and start building interactive applications. It introduces users to event handling, collision detection, and real-time updates using the Tkinter GUI library, all of which are essential foundations for game development and real-time simulation projects.

This project also builds problem-solving skills by simulating a live decision-making environment where a player must avoid a randomly moving enemy. It serves as a strong bridge between theoretical programming and real-world applications like robotics pathfinding, AI training environments, or algorithm visualization.

**4. When we need this use case**

We need this use case in the following scenarios:

1. **Teaching Event-Driven Programming:** This use case is ideal for introducing students to how events (like key presses) control logic in GUI-based applications.
2. **Beginner-Level Game Development Projects:** Aspiring game developers or hobby coders can use this as a starting point to understand core concepts like player movement, enemy AI, and win/loss conditions.
3. **Python GUI Learning:** If you're transitioning from command-line to graphical applications in Python, this provides a fun and interactive way to get started.
4. **Interview Demonstration Project:** For job seekers, this makes for a great small portfolio project to show your ability to combine logic, UI, and creativity.
5. **STEM Education & Learning Projects:** Great for classroom projects or coding bootcamps where engagement and logic are both important.

**5. Challenge questions**

1. **How would you prevent the enemy from moving into the same cell as the player at the start of the game?**
   * *Scenario:* Suppose the random start position of the enemy is generated as [0,0] — same as the player. How will you ensure no collision occurs at game launch?
2. **How can you implement a smarter AI instead of random movement for the red square?**
   * *Scenario:* Imagine the enemy should always try to move closer to the player using Manhattan distance. How would you modify the movement logic?
3. **What changes would you make to increase the difficulty level progressively?**
   * *Scenario:* After every 5 moves, the enemy starts moving faster. How will you implement such dynamic difficulty?
4. **How will you refactor this code for scalability, like converting it into an object-oriented design?**
   * *Scenario:* You are tasked with expanding this game to allow multiple players or enemies. How would an OOP approach make it manageable?
5. **How would you test this game programmatically?**
   * *Scenario:* GUI apps are hard to test. How would you simulate keypresses or validate win/loss conditions during automated testing?
6. **How would you prevent the enemy from spawning in the same cell as the player or goal?**  
   Scenario: During the initial load, the random coordinates of the red square match those of the player or the green goal. This causes an immediate game over or instant win. What logic would you apply to validate enemy position on spawn?
7. **How would you modify the game to allow diagonal movement for the player?**  
   Scenario: A new requirement comes in where users want to move diagonally using keys like 'Q', 'E', 'Z', and 'C'. How would you incorporate diagonal movement without breaking existing functionality?
8. **How would you add a scoring system that rewards faster escapes?**  
   Scenario: A timer starts when the game begins, and the player should get a better score for reaching the goal faster. How would you implement time tracking and score calculation?
9. **How would you modify the game to increase the grid size dynamically based on difficulty level?**  
   Scenario: The player selects a difficulty setting before the game starts—easy (5x5), medium (8x8), hard (10x10). What changes would you make in the grid initialization code?
10. **How would you refactor the code to support multiple enemies?**  
    Scenario: The game needs to support 2–3 red squares moving independently. How would you restructure your code to maintain and update multiple enemy positions?
11. **How would you implement walls or blocked cells in the grid?**  
    Scenario: Certain grid cells are blocked and neither player nor enemy can enter them. How will you represent and enforce those blocked zones during movement checks?
12. **How would you handle screen resize events while preserving gameplay proportions?**  
    Scenario: If a user resizes the game window, the grid and objects should scale accordingly. How would you listen to resize events and dynamically redraw the canvas?
13. **How would you introduce a countdown timer with a fail condition?**  
    Scenario: The player must reach the goal in under 30 seconds. If they don’t, a "Time’s up!" message should display. How would you integrate a countdown and trigger timeout logic?
14. **How would you allow mouse-based movement instead of keyboard input?**  
    Scenario: Instead of arrow keys, a player should be able to click on the adjacent cell to move. What event bindings and coordinate logic would you implement?
15. **How would you implement enemy pathfinding to actively chase the player?**  
    Scenario: Instead of random movements, the enemy now needs to intelligently move one step closer to the player at each turn. How would you compute this shortest path logic?
16. **How would you create a pause and resume feature in the game?**  
    Scenario: A pause button should freeze all movements and a resume button should continue from the same state. What would your pause state management look like?
17. **How would you write test cases for this grid-based game?**  
    Scenario: You need to validate boundary conditions, win/loss logic, and movement. How would you structure unit tests or use mocking to simulate GUI interactions?
18. **How would you save and load game progress using local files?**  
    Scenario: Mid-game, a user wants to save their position and resume later. What file structure or data format (JSON, pickle) would you use to store and retrieve game state?
19. **How would you optimize the game to reduce CPU usage during idle states?**  
    Scenario: The game runs continuously even when no input is made, causing unnecessary CPU cycles. What modifications would you suggest to make it more event-driven?
20. **How would you modify the game so that enemies only move every alternate turn?**  
    Scenario: The enemy should only move every second keypress made by the player. How would you manage turn counters or alternate movement control?
21. **How would you make the game accessible for colorblind users?**  
    Scenario: Red-green colorblind users struggle to differentiate the enemy and goal. What UI changes or accessibility features can be added?
22. **How would you convert this desktop game into a web-based app using Python?**  
    Scenario: The client wants this game available online, but it was built using Tkinter. What frameworks (e.g., Flask with Pygame.js) would you explore for browser deployment?
23. **How would you implement keyboard shortcuts to restart or quit the game?**  
    Scenario: Pressing 'R' should restart and 'Q' should quit. How would you bind these shortcuts and handle game reset?
24. **How would you implement levels where the enemy gets faster after each win?**  
    Scenario: Each time the player wins, the enemy’s speed increases by reducing the movement interval. What mechanism would you use to track levels and adjust timing?
25. **How would you make this game mobile-friendly using Python frameworks?**  
    Scenario: The goal is to make this game playable on Android. Which Python tools (like Kivy or BeeWare) would you use and how would you adapt the GUI logic?
26. **How would you allow multiplayer gameplay where two players control different characters?**  
    Scenario: One player uses arrow keys and the other uses WASD. How would you handle dual movement and collision between players?
27. **How would you enhance the game by adding sound effects for movement and collisions?**  
    Scenario: Add a beep when the player moves and a buzz on collision. Which Python packages like pygame.mixer would you use?
28. **How would you animate the enemy movement instead of instantly jumping cells?**  
    Scenario: You want the red square to slide smoothly to its next position. What animation logic or time-based interpolation would you implement?
29. **How would you implement an undo feature for the last move?**  
    Scenario: The player can press ‘U’ to go back to their previous cell. How will you track and revert the last state?
30. **How would you add support for themes (light mode, dark mode, etc.)?**  
    Scenario: The player can choose between different themes which change background and square colors. What configuration structure and update mechanism would you design?

**6. Prerequisites for the lab**

* Basic understanding of Python syntax and control structures (if-else, functions, loops)
* Familiarity with list indexing and 2D coordinate systems
* Basic knowledge of the Tkinter library for GUI development
* Understanding how events (like keyboard input) trigger functions
* Installed packages:

pip install numpy

To implement this project, you should know:

* Basics of Python (lists, functions, if-else)
* Tkinter for GUI development
* Basic NumPy for randomness and clipping

**6. Advantages and disadvantages of this use case**

**Advantages:**

* Great learning experience for event-driven programming
* Visual feedback enhances debugging and learning
* Builds foundational game dev logic (movement, boundaries, collision)
* Small scope, yet complete enough to be a mini-project
* Easy to extend with AI logic, scorekeeping, or levels

**Disadvantages:**

* Uses Tkinter, which is not suitable for large or commercial games
* GUI logic can become harder to manage without OOP
* Random movement may become predictable or too easy
* Not optimized for mobile or web deployment

**7. Step-by-step implementation instructions**

**Full Python Program**

import tkinter as tk

import numpy as np

# Grid size

GRID\_SIZE = 5

CELL\_SIZE = 100 # Each cell is 100x100 pixels

# Initial positions

player\_pos = [0, 0] # Player starts at top-left

random\_int1 = np.random.randint(1, 5)

random\_int2 = np.random.randint(1, 5)

enemy\_pos = [random\_int1, random\_int2]

goal\_pos = [4, 4] # Goal is at bottom-right

# Movement directions (row, col)

MOVES = {

"Up": (-1, 0),

"Down": (1, 0),

"Left": (0, -1),

"Right": (0, 1),

}

# Create GUI window

root = tk.Tk()

root.title("Grid Game: Reach the Goal!")

# Canvas setup

canvas = tk.Canvas(root, width=GRID\_SIZE \* CELL\_SIZE, height=GRID\_SIZE \* CELL\_SIZE, bg="white")

canvas.pack()

# Draw grid lines

for i in range(GRID\_SIZE + 1):

canvas.create\_line(i \* CELL\_SIZE, 0, i \* CELL\_SIZE, GRID\_SIZE \* CELL\_SIZE, fill="black")

canvas.create\_line(0, i \* CELL\_SIZE, GRID\_SIZE \* CELL\_SIZE, i \* CELL\_SIZE, fill="black")

# Draw goal (green square)

goal\_square = canvas.create\_rectangle(

goal\_pos[1] \* CELL\_SIZE, goal\_pos[0] \* CELL\_SIZE,

(goal\_pos[1] + 1) \* CELL\_SIZE, (goal\_pos[0] + 1) \* CELL\_SIZE,

fill="green"

)

# Draw enemy (red square)

enemy\_square = canvas.create\_rectangle(

enemy\_pos[1] \* CELL\_SIZE, enemy\_pos[0] \* CELL\_SIZE,

(enemy\_pos[1] + 1) \* CELL\_SIZE, (enemy\_pos[0] + 1) \* CELL\_SIZE,

fill="red"

)

# Draw player (blue square)

player\_square = canvas.create\_rectangle(

player\_pos[1] \* CELL\_SIZE, player\_pos[0] \* CELL\_SIZE,

(player\_pos[1] + 1) \* CELL\_SIZE, (player\_pos[0] + 1) \* CELL\_SIZE,

fill="blue"

)

# Function to display message and stop game

def show\_message(message, color):

canvas.create\_text(

GRID\_SIZE \* CELL\_SIZE // 2, GRID\_SIZE \* CELL\_SIZE // 2,

text=message, font=("Arial", 24, "bold"), fill=color

)

root.unbind("<KeyPress>") # Stop player movement

root.after\_cancel(enemy\_movement) # Stop enemy movement

# Function to move enemy randomly and automatically

def move\_enemy():

global enemy\_pos, enemy\_movement

if player\_pos == enemy\_pos: # If enemy catches player

show\_message("Game Over!", "red")

return

# Choose a random move

move = np.random.choice(list(MOVES.keys()))

new\_pos = [enemy\_pos[0] + MOVES[move][0], enemy\_pos[1] + MOVES[move][1]]

# Keep enemy inside the grid

enemy\_pos[0] = np.clip(new\_pos[0], 0, GRID\_SIZE - 1)

enemy\_pos[1] = np.clip(new\_pos[1], 0, GRID\_SIZE - 1)

# Update enemy position on canvas

canvas.coords(

enemy\_square,

enemy\_pos[1] \* CELL\_SIZE, enemy\_pos[0] \* CELL\_SIZE,

(enemy\_pos[1] + 1) \* CELL\_SIZE, (enemy\_pos[0] + 1) \* CELL\_SIZE

)

# Check for collision with player

if player\_pos == enemy\_pos:

show\_message("Game Over!", "red")

else:

enemy\_movement = root.after(500, move\_enemy) # Move enemy every 500ms

# Move player with arrow keys

def move\_player(event):

global player\_pos

move = None

if event.keysym == "Up":

move = "Up"

elif event.keysym == "Down":

move = "Down"

elif event.keysym == "Left":

move = "Left"

elif event.keysym == "Right":

move = "Right"

if move:

new\_pos = [player\_pos[0] + MOVES[move][0], player\_pos[1] + MOVES[move][1]]

if 0 <= new\_pos[0] < GRID\_SIZE and 0 <= new\_pos[1] < GRID\_SIZE:

player\_pos[:] = new\_pos

canvas.coords(

player\_square,

player\_pos[1] \* CELL\_SIZE, player\_pos[0] \* CELL\_SIZE,

(player\_pos[1] + 1) \* CELL\_SIZE, (player\_pos[0] + 1) \* CELL\_SIZE

)

# Check for win condition

if player\_pos == goal\_pos:

show\_message("You Win!", "green")

# Check for collision with enemy

elif player\_pos == enemy\_pos:

show\_message("Game Over!", "red")

# Bind keyboard input for player movement

root.bind("<KeyPress>", move\_player)

# Start automatic enemy movement

enemy\_movement = root.after(500, move\_enemy)

# Run the GUI

root.mainloop()

**🔹 Step 1: Setup**

* Import tkinter for GUI and numpy for random movement.
* Define grid size and cell size.

**🔹 Step 2: Game Elements**

* **Player:** Starts at [0, 0]
* **Goal:** Fixed at [4, 4] (bottom-right)
* **Enemy:** Starts at a random position not overlapping the player

**🔹 Step 3: GUI Setup**

* Create a window and canvas using Tkinter.
* Draw grid lines to create a 5x5 game board.

**🔹 Step 4: Draw Characters**

* Use colored squares to represent:
  + **Player:** Blue
  + **Enemy:** Red
  + **Goal:** Green

**🔹 Step 5: Handle Movement**

* Bind arrow keys to player movement.
* Keep player inside the grid.
* Update player’s position visually on the canvas.

**🔹 Step 6: Enemy Logic**

* Move the enemy every 500 milliseconds using after().
* Move in a random direction using NumPy.
* If the enemy catches the player → Game Over!

**🔹 Step 7: Win/Lose Logic**

* If player reaches goal → “You Win!”
* If enemy and player collide → “Game Over!”
* Stop the game when either condition is met.

**Output**

When you run the game:

* A 5x5 grid appears.
* You control the blue square using arrow keys.
* The red square moves randomly every 0.5 seconds.
* Your goal: Reach the green square without getting caught!

**Possible Messages:**

* ✅ **"You Win!"** (Green text in center)
* ❌ **"Game Over!"** (Red text in center)

**8. Conclusion**

This use case demonstrates how a simple game scenario can be used to teach important programming skills in a highly engaging manner. It encapsulates real-time logic, user interaction, GUI programming, and even a bit of AI thinking (random movement). Beyond just fun, it serves as a meaningful tool to transition from console-based Python programs to event-driven applications. You not only create something visual and interactive but also build a strong foundation in problem-solving and structuring programs for future scalability. The best part? You can expand this project with smarter enemy logic, multiple levels, or even sound effects—making it a great launching pad for personal or professional projects.