School of Mechanical & Manufacturing Engineering (SMME),
National University of Sciences and Technology (NUST),
Sector H-12, Islamabad.

Name: **Bashayer Amjad**

CMS ID: **466710**

AE-01

Project Report

Fundamentals of Programming II (CS-223)

Submitted to: Lab Engr Laiba Waheed

Objective

The objective of this project was to design anything using Python, showcasing the use of libraries.

Abstract

This project is about creating a Sudoku game using Python and Pygame. Sudoku is a logic-based puzzle, and the game brings it to life with an intuitive graphical interface. Players can choose a difficulty level, solve puzzles, and receive feedback on their inputs. The game includes features like a timer, error highlighting, and a hint system to make the experience enjoyable and interactive. This report explains the objectives, implementation process, and programming concepts used in building the game.

Introduction

Sudoku is a well-known logic puzzle where the goal is to fill a 9x9 grid so that every row, column, and 3x3 sub-grid contains the numbers 1 to 9 without repetition. It is entirely based on logic. The difficulty depends on how many numbers are already filled in and their positions.

This project brings Sudoku into a digital format using Python. It combines gameplay with an easy-to-use graphical interface. Along with the fun of solving puzzles, this project helped explore practical programming concepts like event handling, UI design, and problem-solving logic. The game provides features like difficulty settings, hints, and error highlighting to enhance the player's experience.

Game Description

The Sudoku game includes several features to make it enjoyable and interactive:

- **Difficulty Levels**: Players can choose from Easy, Medium, or Hard. Each level has a different number of pre-filled cells.
- **Timer**: Tracks how long the player takes to solve the puzzle.
- **Hints**: Players can reveal the correct number for a selected cell if they are stuck.

- **Error Highlighting**: Incorrect entries are marked in red to help players identify mistakes.
- **End Game Option**: Players can quit anytime to start a new puzzle.
- **New Game Functionality**: The game can be restarted with a fresh puzzle.

The interface includes a clean Sudoku grid, buttons for hints and game controls, and a visually appealing look.

Concepts Used

This project applied several important programming concepts and techniques:

1. **Libraries and Frameworks**:
    - *Pygame*: Used for graphics, handling user input, and building the interactive interface.
    - *Math and Random*: Used for puzzle generation and removing numbers for difficulty settings.
2. **Game Logic**:
    - The Sudoku puzzle is represented as a grid in the code.
    - A *backtracking algorithm* generates the puzzle and ensures it has a valid solution.
    - Difficulty is adjusted by removing numbers from the grid based on the chosen level.
3. **User Interface Design**:
    - The grid provides clear visual feedback using colors to guide the player.
    - Buttons provide easy access to hints, quitting, or starting a new game.
4. **Input Handling**:
    - Mouse clicks allow players to select grid cells.
    - Keyboard input enables players to type numbers into the cells.
5. **Error Handling and Feedback**:

- Incorrect entries are flagged with a distinct color.
- Hints are available to guide players to the correct solution if needed.

Code Explanation

```
 8  import pygame
 9  import sys
10  import time
11  from pygame.locals import QUIT, MOUSEBUTTONDOWN, KEYDOWN
12  import math
13  import random
```

Figure 1: Libraries used

- pygame: This is the main library used for creating graphical games in Python. We use it to create the game window, handle events, and draw on the screen.
- pygame.locals: This imports specific constants (like QUIT, MOUSEBUTTONDOWN, and KEYDOWN) used to handle specific events in Pygame.
- math: This library is used for mathematical operations like (math.sqrt) for grid generation.
- random: Used to generate random numbers for puzzles and shuffling numbers in Sudoku.

```
18  # Screen dimensions and constants
19  SCREEN_WIDTH, SCREEN_HEIGHT = 600, 600
20  GRID_SIZE = 540
21  CELL_SIZE = GRID_SIZE // 9
22  WHITE = (255, 255, 255)
23  BLACK = (0, 0, 0)
24  GRAY = (200, 200, 200)
25  RED = (255, 0, 0)
26  GREEN = (0, 255, 0)
27  # Define aesthetic color palette
28  WHITE = (255, 255, 255)
29  LIGHT_BLUE = (173, 216, 230)  # Soft light blue
30  DARK_BLUE = (70, 130, 180)    # For grid borders
31  LIGHT_GRAY = (245, 245, 245)  # For the background color of the grid
32  HIGHLIGHT_COLOR = (255, 204, 0)  # Golden yellow for highlighting cells
33  RED = (255, 77, 77)  # Soft red for incorrect entries
34  GREEN = (0, 204, 102)  # Vibrant green for correct entries
35  BUTTON_COLOR = (70, 130, 180)  # Soft blue for buttons
36  BUTTON_HOVER_COLOR = (100, 150, 200)  # Lighter blue for hover effect
```

Figure 2: Setting up constants and colors

- SCREEN_WIDTH and SCREEN_HEIGHT: These define the size of the window where the game will be displayed.
- GRID_SIZE: This will be where the actual Sudoku board is drawn.
- CELL_SIZE: Each cell in the 9x9 Sudoku grid will be GRID_SIZE divided by 9.

- Color Palette: These are color definitions in RGB format.

```
# Fonts
FONT = pygame.font.SysFont("Poppins", 40)  # Title font
SMALL_FONT = pygame.font.SysFont("Poppins", 20)

# Global variables
selected_cell = None
timer_start = None
difficulty = None
incorrect_cells = set()  # To keep track of incorrect cells
```

Figure 3: Variables and Fonts

- selected_cell: This stores the currently selected cell in the Sudoku grid, represented as a tuple (row, column). Initially, it's set to None.
- timer_start: This holds the starting time for the game (for tracking the time the player has taken).
- difficulty: This will store the selected difficulty level (easy, medium, or hard).
- incorrect_cells: A set that tracks the coordinates of the cells that the player has entered incorrectly. We use a set because it automatically handles duplicates.

Main Drawing Functions

```
def draw_grid(screen, puzzle, solution, selected_cell):
    """Draw the Sudoku grid, numbers, and highlights."""
    screen.fill(LIGHT_GRAY)  # Light background for the grid

    # Draw grid lines with soft borders
    for i in range(10):
        line_width = 3 if i % 3 == 0 else 1
        pygame.draw.line(screen, DARK_BLUE, (0, i * CELL_SIZE), (GRID_SIZE, i * CELL_SIZE), line_width)
        pygame.draw.line(screen, DARK_BLUE, (i * CELL_SIZE, 0), (i * CELL_SIZE, GRID_SIZE), line_width)

    # Draw numbers with appropriate colors
    for row in range(9):
        for col in range(9):
            num = puzzle[row][col]
            if num is not None:  # Only render if num is not None
                if (row, col) in incorrect_cells:
                    color = RED  # Highlight incorrect cells
                elif (row, col) not in incorrect_cells and num == solution[row][col]:
                    color = GREEN  # Highlight correct cells
                else:
                    color = BLACK  # Default color for normal cells
                text = FONT.render(str(abs(num)), True, color)  # Display absolute value for negative entries
                screen.blit(text, (col * CELL_SIZE + 20, row * CELL_SIZE + 15))

    # Highlight selected cell with a yellow border
    if selected_cell:
        pygame.draw.rect(screen, HIGHLIGHT_COLOR, (selected_cell[1] * CELL_SIZE, selected_cell[0] * CELL_SIZE, CELL_SIZE, CELL
```

Figure 4: draw_grid()

- Background Setup
- **Draw Grid Lines:** Draws 10 lines (horizontal & vertical), thicker for 3rd lines, in blue.
- **Render Numbers:** Iterates through grid, coloring numbers red (incorrect) or green (correct).
- **Highlight Selected Cell:** Draws a yellow rectangle around the selected cell.

```python
def draw_background(screen):
    """Draw background."""
    for i in range(SCREEN_HEIGHT):
        color = (
            int(LIGHT_BLUE[0] + (255 - LIGHT_BLUE[0]) * i / SCREEN_HEIGHT),
            int(LIGHT_BLUE[1] + (255 - LIGHT_BLUE[1]) * i / SCREEN_HEIGHT),
            int(LIGHT_BLUE[2] + (255 - LIGHT_BLUE[2]) * i / SCREEN_HEIGHT))
        pygame.draw.line(screen, color, (0, i), (SCREEN_WIDTH, i))

def draw_buttons(screen):
    """Draw the timer, hint, end game, and new game buttons."""
    elapsed_time = time.time() - timer_start
    timer_text = SMALL_FONT.render(f"Time: {int(elapsed_time)}s", True, DARK_BLUE)
    screen.blit(timer_text, (10, GRID_SIZE + 10))

    # Draw Hint button with rounded corners and hover effect
    hint_button = pygame.Rect(400, GRID_SIZE + 10, 80, 30)
    pygame.draw.rect(screen, BUTTON_COLOR, hint_button, border_radius=15)  # Rounded corners
    if hint_button.collidepoint(pygame.mouse.get_pos()):
        pygame.draw.rect(screen, BUTTON_HOVER_COLOR, hint_button, border_radius=15)
    hint_text = SMALL_FONT.render("Hint", True, WHITE)
    screen.blit(hint_text, (410, GRID_SIZE + 15))
    # Draw End Game button with rounded corners and hover effect
    solve_button = pygame.Rect(500, GRID_SIZE + 10, 80, 30)
    pygame.draw.rect(screen, BUTTON_COLOR, solve_button, border_radius=15)  # Rounded corners
    if solve_button.collidepoint(pygame.mouse.get_pos()):
        pygame.draw.rect(screen, BUTTON_HOVER_COLOR, solve_button, border_radius=15)
    solve_text = SMALL_FONT.render("End Game", True, WHITE)
    screen.blit(solve_text, (510, GRID_SIZE + 15))

    return hint_button, solve_button
```

Figure 5: draw_background and buttons()

- **Timer Display**: Shows elapsed time in blue below the grid.
- **Hint Button**: Creates a rounded button at (400, GRID_SIZE + 10) with hover effects and "Hint" text.
- **End Game Button**: Similar to the hint button, positioned at (500, GRID_SIZE + 10) with "End Game" text.
- **Return Values**: Returns hint_button and solve_button for interaction handling

```python
def draw_difficulty_selection(screen):
    """Draw difficulty selection buttons."""
    screen.fill(WHITE)
    title_text = FONT.render("Select Difficulty", True, BLACK)
    screen.blit(title_text, (150, 200))

    difficulties = ["Easy", "Medium", "Hard"]
    buttons = []
    for i, diff in enumerate(difficulties):
        button = pygame.Rect(200, 300 + i * 60, 200, 50)
        pygame.draw.rect(screen, GRAY, button)
        text = FONT.render(diff, True, BLACK)
        screen.blit(text, (button.x + 50, button.y + 10))
        buttons.append((button, diff.lower()))

    pygame.display.flip()
    return buttons
```

Figure 6: draw_difficulty_selection()

- **Background & Title**: Fills screen white and shows "Select Difficulty" title.
- **Buttons**: Draws three difficulty buttons ("Easy", "Medium", "Hard") in gray.
- **Return**: Returns a list of buttons with difficulty labels for interaction.

Generating the Puzzle

```python
def generate_puzzle(difficulty):
    """Generate a Sudoku puzzle based on difficulty."""
    N = 9  # Size of the board
    SRN = int(math.sqrt(N))

    # Difficulty map: percentage of elements to remove
    difficulty_map = {"easy": 0.35, "medium": 0.45, "hard": 0.55}
    removal_percentage = difficulty_map[difficulty]
    K = int(N * N * removal_percentage)  # Number of elements to remove

    # Initialize empty board
    mat = [[0 for _ in range(N)] for _ in range(N)]

    def fill_box(row, col):
        """Fill a 3x3 box with unique numbers."""
        nums = list(range(1, N + 1))
        random.shuffle(nums)
        for i in range(SRN):
            for j in range(SRN):
                mat[row + i][col + j] = nums.pop()

    def un_used_in_box(row_start, col_start, num):
        """Check if a number is unused in a 3x3 box."""
        for i in range(SRN):
            for j in range(SRN):
                if mat[row_start + i][col_start + j] == num:
                    return False
        return True
```

```python
    def check_if_safe(i, j, num):
        """Check if a number can be placed safely in a cell."""
        return (
            num not in mat[i]  # Check row
            and all(row[j] != num for row in mat)  # Check column
            and un_used_in_box(i - i % SRN, j - j % SRN, num)  # Check box
        )

    def fill_remaining(i, j):
        """Recursively fill the remaining cells of the board."""
        if i == N - 1 and j == N:
            return True
        if j == N:
            i += 1
            j = 0
        if mat[i][j] != 0:
            return fill_remaining(i, j + 1)
        for num in range(1, N + 1):
            if check_if_safe(i, j, num):
                mat[i][j] = num
                if fill_remaining(i, j + 1):
                    return True
                mat[i][j] = 0
        return False
```

```
183     def remove_k_digits():
184         """Remove K digits to create the puzzle."""
185         count = K
186         while count > 0:
187             i = random.randint(0, N - 1)
188             j = random.randint(0, N - 1)
189             if mat[i][j] != 0:
190                 mat[i][j] = 0
191                 count -= 1
192
193     # Step 1: Fill diagonal 3x3 matrices
194     for i in range(0, N, SRN):
195         fill_box(i, i)
196
197     # Step 2: Fill remaining cells
198     fill_remaining(0, 0)
199
200     # Step 3: Save the solution (completed board)
201     solved_sudoku = [row[:] for row in mat]
202
203     # Step 4: Remove K digits to create the puzzle
204     remove_k_digits()
205
206     # Prepare puzzle and solution in the required format
207     puzzle = [[cell if cell != 0 else None for cell in row] for row in mat]
208     solution = [[solved_sudoku[i][j] for j in range(N)] for i in range(N)]
209
210     return puzzle, solution
211
```

Figure 7: Generating Sudoku Board

Generates a Sudoku puzzle and its solution based on the specified difficulty.

- **Difficulty**: A percentage of cells to be removed (difficulty_map) decides how many elements are removed (K).
- **Board Initialization**: Creates a 9x9 grid initialized with zeroes.
- **Functions:**
- fill_box(row, col): Fills a 3x3 subgrid starting at (row, col) with unique random numbers.
- un_used_in_box(row_start, col_start, num): Checks if num is unused in the 3x3 box.
- check_if_safe(i, j, num): Validates whether placing num in (i, j) is safe: Not present in the same row, column, or subgrid.
- fill_remaining(i, j) (Backtracking): Recursively fills the remaining cells, checking for valid placements and undoing invalid ones.
- remove_k_digits(): Randomly removes K cells to create the puzzle by setting their value to 0.

- **Steps:**
    - Fill Diagonal Subgrids: Ensures the diagonal 3x3 boxes have valid random numbers using fill_box().
    - Fill Remaining Cells: Uses fill_remaining() and backtracking to complete the board.
    - Store Solution: Saves a copy of the solved Sudoku before removing digits.
    - Remove Digits: Removes K random digits to generate the puzzle.
- **Output:** Returns two 9x9 grids:
    - **Puzzle**: Incomplete Sudoku board with removed digits.
    - **Solution**: Full, completed Sudoku board.

```
212  def handle_input(puzzle, solution, pos, key):
213      """Handle user input to modify the puzzle."""
214      row, col = pos
215      if puzzle[row][col] is None or puzzle[row][col] < 0:  # Allow input for empty or incorrect cells
216          if solution[row][col] == int(key):
217              puzzle[row][col] = int(key)  # Set correct value
218              incorrect_cells.discard((row, col))  # Remove from incorrect set if it was there
219          else:
220              puzzle[row][col] = -int(key)  # Mark as incorrect (negative for visual indication)
221              incorrect_cells.add((row, col))  # Add to incorrect cells
222
223  def game_over_screen(screen):
224      """Display 'Game Over' screen with a New Game button."""
225      screen.fill(WHITE)
226      game_over_text = FONT.render("Game Over!", True, RED)
227      screen.blit(game_over_text, (200, 250))
228
229      new_game_button = pygame.Rect(200, 350, 200, 50)
230      pygame.draw.rect(screen, GRAY, new_game_button)
231      new_game_text = FONT.render("New Game", True, BLACK)
232      screen.blit(new_game_text, (new_game_button.x + 20, new_game_button.y + 10))
233
234      pygame.display.flip()
235      while True:
236          for event in pygame.event.get():
237              if event.type == QUIT:
238                  pygame.quit()
239                  sys.exit()
240              if event.type == MOUSEBUTTONDOWN:
241                  if new_game_button.collidepoint(event.pos):
242                      return True  # Start new game when button is pressed
```

Figure 8: Handling User Input and Game Over Screen

handle_input()

Processes user inputs to modify the Sudoku puzzle.

- **Steps:**
    - Gets the row and column (pos) of the cell where the input is made.

- Checks if the cell is empty (None) or marked incorrect (negative value):
  - If the input matches the solution, it updates the cell with the correct value and removes it from incorrect_cells.
  - Otherwise, marks the cell as incorrect (stores a negative value) and adds it to incorrect_cells.

game_over_screen()

Displays the "Game Over" screen and allows starting a new game.

➢ **Steps:**

- Fills the screen with a white background and displays the text "Game Over!" in red.
- Draws a "New Game" button with a rectangular button filled with gray color and text ("New Game") rendered in black and centered on the button.
- **Event Handling Loop:** Waits for user actions:
  - Quit Event: Closes the game if the user exits.
  - Mouse Click Event: Starts a new game if the user clicks the button.

Main Function

```
253  def main():
254      global selected_cell, timer_start, difficulty, incorrect_cells
255
256      while True:  # Keep the game running indefinitely unless the user quits
257          screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
258          pygame.display.set_caption("Sudoku")
259
260          # Step 1: Difficulty selection screen
261          difficulty = None
262          while not difficulty:
263              buttons = draw_difficulty_selection(screen)
264              for event in pygame.event.get():
265                  if event.type == QUIT:
266                      pygame.quit()
267                      sys.exit()
268                  if event.type == MOUSEBUTTONDOWN:
269                      for button, diff in buttons:
270                          if button.collidepoint(event.pos):
271                              difficulty = diff
272
273          # Step 2: Generate puzzle and initialize game variables
274          puzzle, solution = generate_puzzle(difficulty)
275          timer_start = time.time()
276          incorrect_cells.clear()
277          selected_cell = None
```

```
279         # Step 3: Main game loop
280         game_active = True
281         while game_active:
282             for event in pygame.event.get():
283                 if event.type == QUIT:
284                     pygame.quit()
285                     sys.exit()
286                 if event.type == MOUSEBUTTONDOWN:
287                     x, y = event.pos
288                     if x < GRID_SIZE and y < GRID_SIZE:
289                         selected_cell = (y // CELL_SIZE, x // CELL_SIZE)
290                 if event.type == KEYDOWN and selected_cell:
291                     if event.unicode.isdigit() and 1 <= int(event.unicode) <= 9:
292                         handle_input(puzzle, solution, selected_cell, event.unicode)
```

```
295             # Call the background function first to set the background
296             draw_background(screen)
297             # Draw grid, buttons, and other UI elements
298             draw_grid(screen, puzzle, solution, selected_cell)
299             hint_button, solve_button = draw_buttons(screen)
300
301             # Check button clicks
302             if event.type == MOUSEBUTTONDOWN:
303                 if hint_button.collidepoint(event.pos):
304                     if selected_cell:
305                         row, col = selected_cell
306                         puzzle[row][col] = solution[row][col]
307                         incorrect_cells.discard((row, col))
308                 if solve_button.collidepoint(event.pos):
309                     puzzle = [row[:] for row in solution]
310                     break  # Exit to trigger game over screen
311
312             # Check if the game is over (all cells filled correctly)
313             if all(puzzle[row][col] == solution[row][col] for row in range(9) for col in range(9)):
314                 game_active = False  # Break out of game loop
315
316             pygame.display.flip()
317
318         # Step 4: Game over screen
319         if game_over_screen(screen):
320             continue  # Restart from the difficulty selection screen
321
```

Figure 9: Main Function

The main function manages the entire flow of the Sudoku game. It handles everything from choosing the difficulty level to what happens when the game ends. It keeps running in a loop so that players can restart the game if they want.

- **Initialize Game Window:**

  Creates the Pygame screen with dimensions SCREEN_WIDTH and HEIGHT. Sets the game window title "Sudoku."

- **Difficulty Selection:**

  Continuously displays difficulty selection buttons until the user chooses a difficulty (easy, medium, hard). Uses mouse clicks to detect and assign the selected difficulty.

- **Puzzle Initialization:**

  Generates the puzzle and solution based on the selected difficulty. Resets game variables:

  - Starts the timer.
  - Clears incorrect_cells.
  - Sets selected_cell to None.

- **Main Game Loop:**

  Handles User Input:

  - Mouse Input: Selects a cell based on the click location within the grid.
  - Keyboard Input: Updates the puzzle if a valid digit (1-9) is entered.

  Draws UI Elements:

  - Calls functions to render the background, grid, and buttons (hint and solve).

  Button Clicks:

  - Hint Button: Fills the selected cell with the correct value from the solution.
  - Solve Button: Reveals the complete solution and ends the game loop.

  Game Completion Check:

  - Ends the game if all cells match the solution.

- **Game Over Screen:**

  Displays the "Game Over" screen. Allows restarting the game by returning to the difficulty selection screen.

- **Infinite Loop for Replay:**

  Restarts the game when "New Game" is clicked, repeating the process from start.

Entry Point for Sudoku Game

```
321
322  if __name__ == "__main__":
323      main()
324
```

Figure 10: Sudoku Game Entry Point

- The if __name__ == "__main__": block is the standard entry point for Python scripts.
- It ensures that the main() function is executed only when the script is run directly, not when it is imported as a module in another script.

Code Output

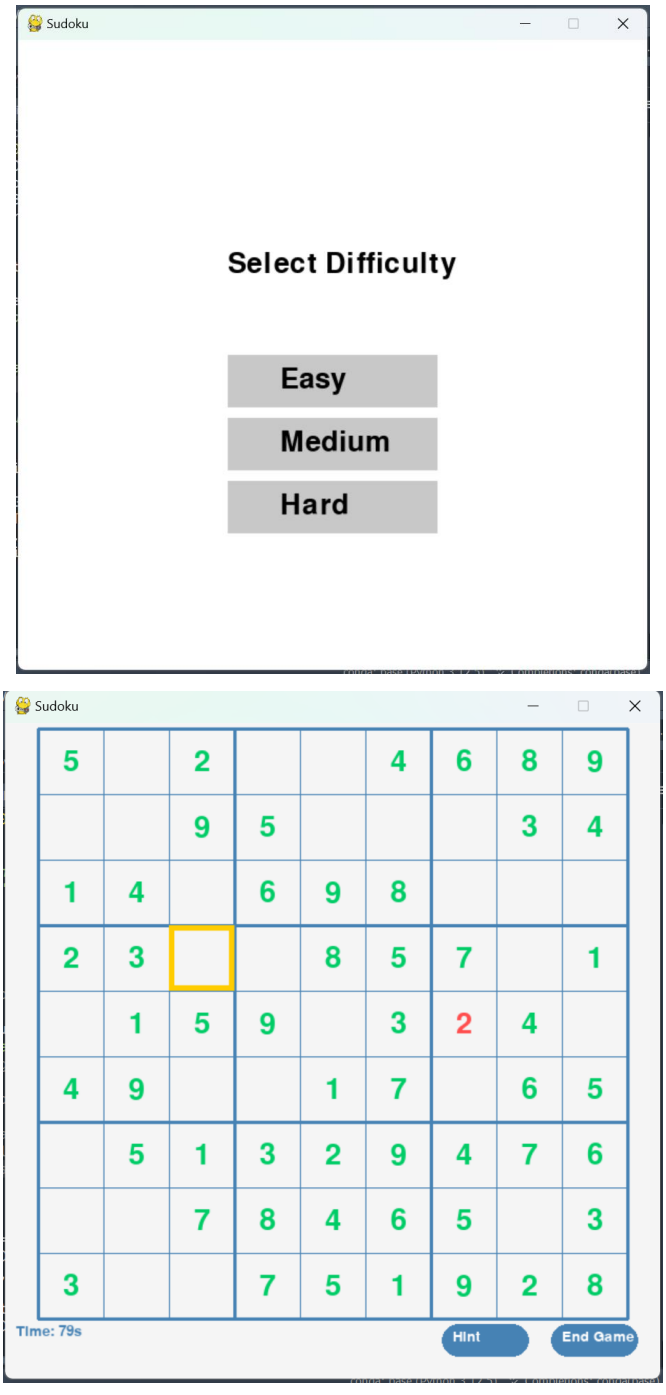

Figure 11: Shows incorrect input, selected cell, timer

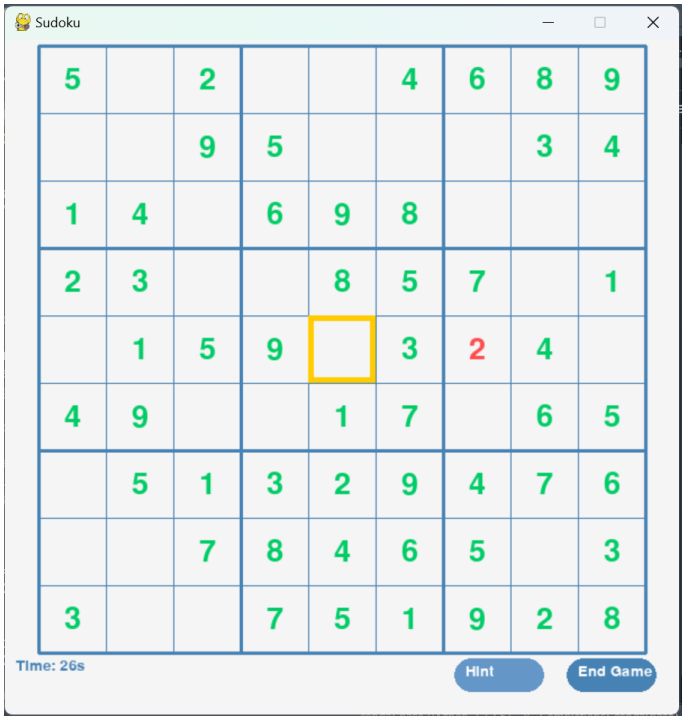

Figure 12: Shows button color change on hover, incorrect number, highlighted cell

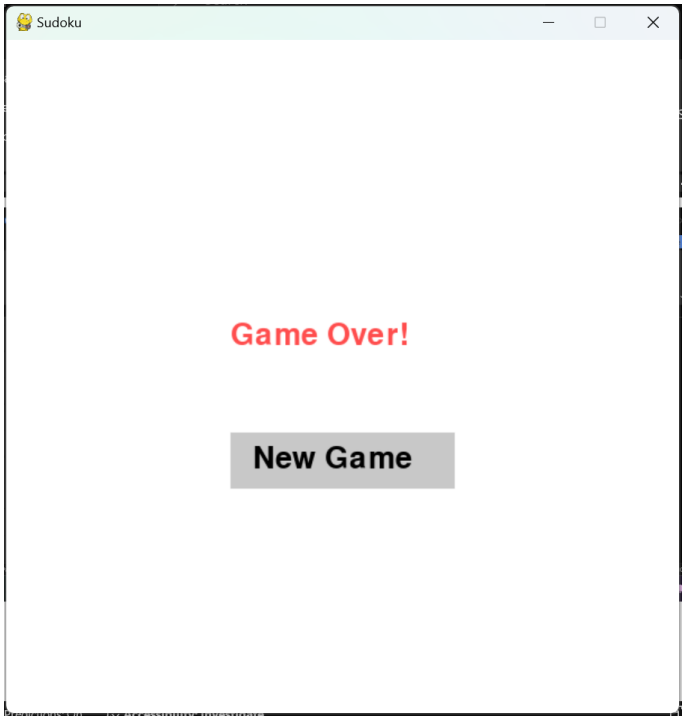

Figure 13: Game Over Screen