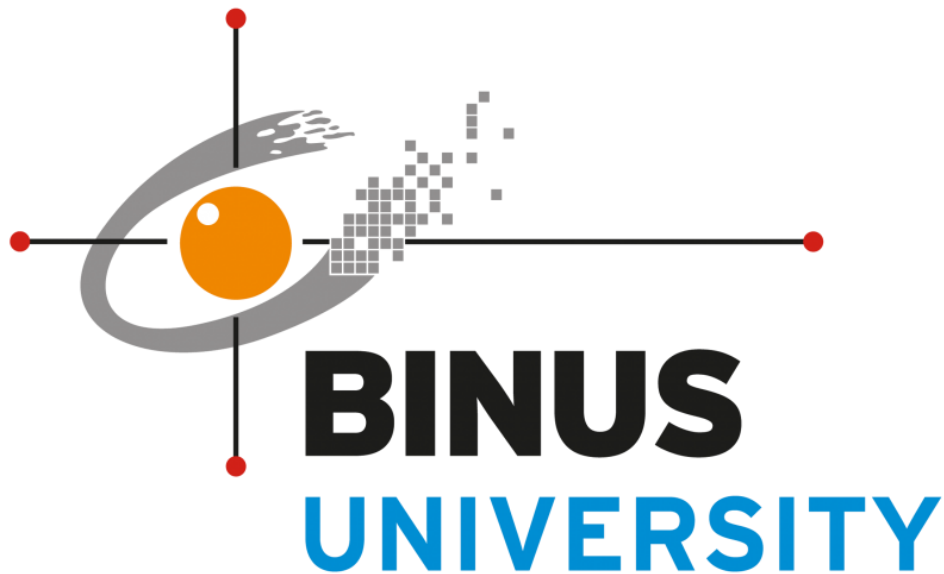


SUDOKU

Creating a Sudoku game using Python and Pygame



BINUS University International

Algorithm and Programming

Final Project 2024

Iglesias Sidharta Handojo – 2902530621

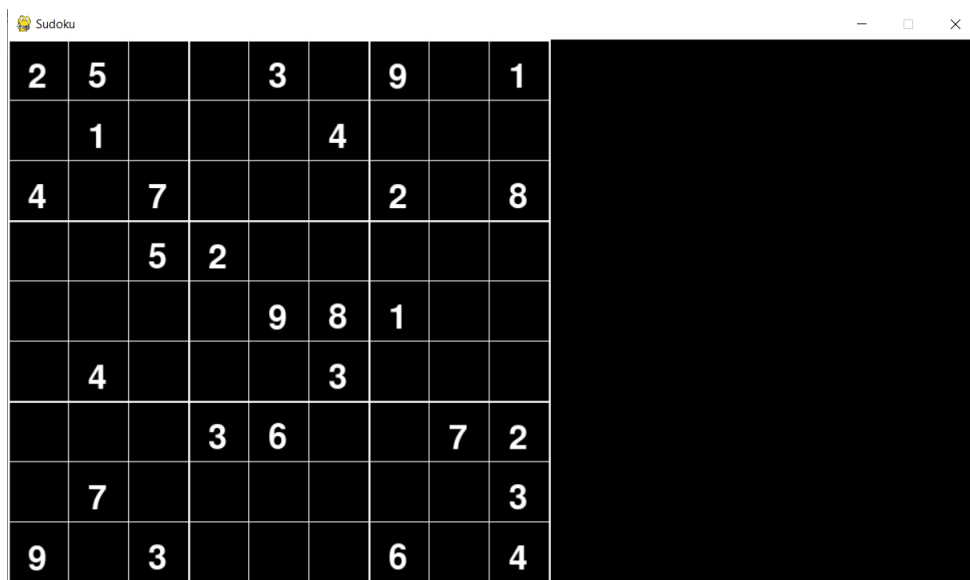
Introduction to Sudoku

Sudoku is a popular logic-based number-placement puzzle that challenges players to fill a 9x9 grid so that each row, column, and 3x3 subgrid contains the digits from 1 to 9, without repetition. This project focuses on creating a digital version of the game using the Pygame library. The aim is to provide an interactive and user-friendly interface where players can engage in solving a predefined Sudoku puzzle. The project incorporates essential game mechanics, such as grid navigation, number placement, and rule validation, to ensure a seamless gameplay experience. In addition, features like dynamic cell highlighting and real-time feedback add to the usability of the application. This report outlines the design, development, and implementation of the game, emphasizing its core functionalities and how they come together to form a fully functional Sudoku game.

Solution Design

1. Graphical User Interface

The game features a 9x9 Sudoku grid displayed on a 970x550 pixel window. The grid is dynamically drawn, with thicker lines separating 3x3 subgrids for better visual clarity.




2	5			3		9		1
	1				4			
4		7			2			8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

2. Game Logic

The logic ensures that the user's inputs adhere to the rules of Sudoku. It checks the validity of moves by ensuring that numbers do not repeat in rows, columns, or 3x3 subgrids.

For example from the picture bellow. This selected grid, we can only fill this grid with number 3, 5, or 6 why? 1, 2, 8, and 9 is in the same grid. 4 is in the same row. Lastly 7 is

in the same column

 Sudoku

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

3. Input Handling:

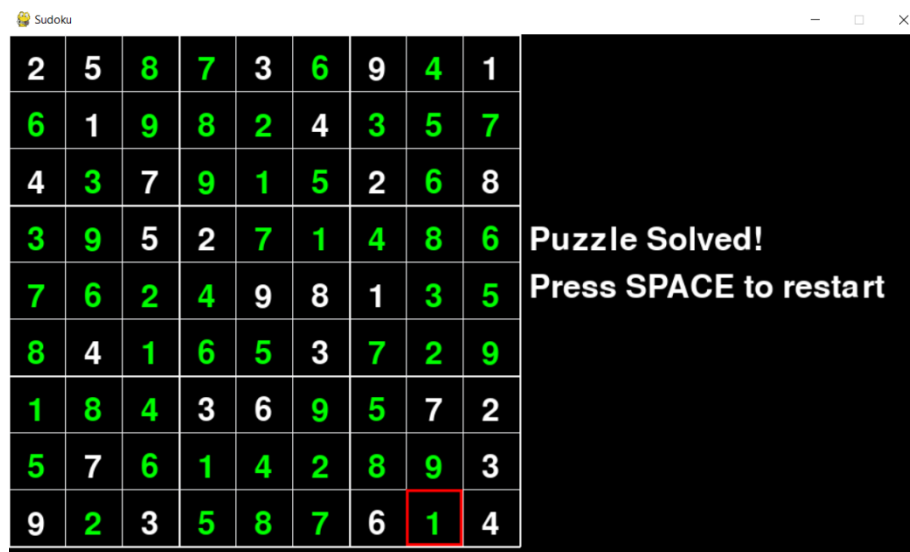
Mouse and keyboard inputs allow users to select grid cells and input numbers. Special keys like BACKSPACE and DELETE enable corrections, while arrow keys allow navigation between cells.

 Sudoku

2	5			3		9	4	1
	1				4	3	5	7
4		7				2	6	8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

4. Game States

The game transitions between "playing" and "solved" states. Upon solving the puzzle correctly, the game displays a congratulatory message and allows users to restart by pressing SPACE.



Discussion of Implementation

1. Game Initialization (__init__ method):

- Pygame is initialized using `pygame.init()`.

```
class SudokuGame:
    def __init__(self):
        pygame.init() # Initialize all Pygame modules
```

- A window with dimensions of 970x550 pixels is created using `pygame.display.set_mode()`.

```
# Create the game window and set its size to 970x550 pixels
self.screen = pygame.display.set_mode((970, 550))
pygame.display.set_caption('Sudoku')
```

- A 9x9 Sudoku grid is set up as a 2D list (`self.grid`), and various constant values (like cell size and colors) are defined.

```
# Define constants
self.cell_size = 540 // 9 # Size of each cell in the Sudoku grid
self.font = pygame.font.Font(None, 50)

self.colors = {
    'red': (255, 0, 0),
    'black': (0, 0, 0),
    'grey': (242, 243, 245),
    'white': (255, 255, 255),
    'green': (0, 255, 0)
}

# Game variables
self.grid = [[2, 5, 0, 0, 3, 0, 9, 0, 1],
              [0, 1, 0, 0, 0, 4, 0, 0, 0],
              [4, 0, 7, 0, 0, 0, 2, 0, 8],
              [0, 0, 5, 2, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 9, 8, 1, 0, 0],
              [0, 4, 0, 0, 0, 3, 0, 0, 0],
              [0, 0, 0, 3, 6, 0, 0, 7, 2],
              [0, 7, 0, 0, 0, 0, 0, 0, 3],
              [9, 0, 3, 0, 0, 0, 6, 0, 4]]
```

- It tracks the selected row and column (self.selected_row and self.selected_col) along with the game's current state (self.game_state), which is initially set to 'playing'.
- A list of pre-filled cells is created (self.pre_filled), identifying cells that contain non-zero values.

```
self.pre_filled = [(row, col) for row in range(9) for col in range(9) if self.grid[row][col] != 0]
self.selected_row, self.selected_col = None, None
self.game_state = 'playing'
```

2. Drawing the Grid (draw_grid method):

- The grid is drawn with lines to separate cells and subgrids. Each cell is 60 x 60
- Bold white lines mark the borders of 3x3 subgrids, while thinner gray lines mark every individual cell.

```
def draw_grid(self):
    for i in range(10):
        x = i * self.cell_size
        y = i * self.cell_size

        if i % 3 == 0:
            pygame.draw.line(self.screen, self.colors['white'], (x, 0), (x, 540), 2)
            pygame.draw.line(self.screen, self.colors['white'], (0, y), (540, y), 2)
        else:
            pygame.draw.line(self.screen, self.colors['grey'], (x, 0), (x, 540), 1)
            pygame.draw.line(self.screen, self.colors['grey'], (0, y), (540, y), 1)
```

- Each cell in the grid is populated with numbers, with pre-filled numbers displayed in white which can't be deleted so player can't cheat, and user-entered numbers in green.

```
for row in range(9):
    for col in range(9):
        if self.grid[row][col] != 0:
            color = self.colors['white'] if (row, col) in self.pre_filled else self.colors['green']
            text = self.font.render(str(self.grid[row][col]), True, color)
            x_pos = col * self.cell_size + self.cell_size // 3
            y_pos = row * self.cell_size + self.cell_size // 3
            self.screen.blit(text, (x_pos, y_pos))
```

- When a cell is selected, a red rectangle is drawn around it to highlight it.

```
if self.selected_row is not None and self.selected_col is not None:
    pygame.draw.rect(self.screen, self.colors['red'], pygame.Rect(
        self.selected_col * self.cell_size, self.selected_row * self.cell_size, self.cell_size, self.cell_size), 3)
```

3. Event Handling (handle_events method):

- This method handles user input such as:

- Window close events.

```
def handle_events(self):
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            return False
```

- Mouse clicks to select cells.

```
elif event.type == pygame.MOUSEBUTTONDOWN and self.game_state == 'playing':
    if pygame.mouse.get_pressed():
        x_pos, y_pos = pygame.mouse.get_pos()
        self.selected_row = y_pos // self.cell_size
        self.selected_col = x_pos // self.cell_size
```

- Keyboard presses to input numbers (1-9), delete values (backspace), restart the game (spacebar, if solved)

```
elif event.type == pygame.KEYDOWN:
    if event.key in range(pygame.K_1, pygame.K_9 + 1):
        number = event.key - pygame.K_0
        if self.grid[self.selected_row][self.selected_col] == 0 and self.valid_move(self.selected_row, self.selected_col, number):
            self.grid[self.selected_row][self.selected_col] = number
            if self.check_grid():
                self.game_state = 'solved'

    elif event.key in (pygame.K_BACKSPACE, pygame.K_DELETE):
        if (self.selected_row, self.selected_col) not in self.pre_filled:
            self.grid[self.selected_row][self.selected_col] = 0

    elif event.key == pygame.K_SPACE and self.game_state == 'solved':
        self.__init__()
```

,or move between cells (arrow keys)

```
elif event.key == pygame.K_UP:
    if self.selected_row is not None:
        self.selected_row = max(0, self.selected_row - 1)

elif event.key == pygame.K_DOWN:
    if self.selected_row is not None:
        self.selected_row = min(8, self.selected_row + 1)

elif event.key == pygame.K_LEFT:
    if self.selected_col is not None:
        self.selected_col = max(0, self.selected_col - 1)

elif event.key == pygame.K_RIGHT:
    if self.selected_col is not None:
        self.selected_col = min(8, self.selected_col + 1)
```

4. Move Validation (valid_move method):

- It ensures that the number entered in a selected cell adheres to the rules of Sudoku:
 - The number must not already exist in the same row, column, or 3x3 subgrid.
- The method returns True if the move is valid and False otherwise.

```
def valid_move(self, row, col, num):
    # Check if the number exists in the row
    if num in self.grid[row]:
        return False
    # Check if the number exists in the column
    if num in [self.grid[i][col] for i in range(9)]:
        return False

    # Check if the number exists in the 3x3 subgrid
    subgrid_row_start = (row // 3) * 3
    subgrid_col_start = (col // 3) * 3
    for i in range(subgrid_row_start, subgrid_row_start + 3):
        for j in range(subgrid_col_start, subgrid_col_start + 3):
            if self.grid[i][j] == num:
                return False
    return True
```

5. Checking the Grid (check_grid method):

- This method verifies whether the entire grid is correctly solved:
 - Every row, column, and 3x3 subgrid must contain all numbers from 1 to 9.

- It returns True if the grid is solved and False if there are any errors.

```
def check_grid(self):
    for row in range(9):
        if sorted(self.grid[row]) != list(range(1, 10)):
            return False

    for col in range(9):
        if sorted([self.grid[row][col] for row in range(9)]) != list(range(1, 10)):
            return False

    for box_row in range(0, 9, 3):
        for box_col in range(0, 9, 3):
            subgrid = [
                self.grid[row][col]
                for row in range(box_row, box_row + 3)
                for col in range(box_col, box_col + 3)
            ]
            if sorted(subgrid) != list(range(1, 10)):
                return False

    return True
```

6. Main Game Loop (run method):

- The game runs continuously until the window is closed.
- The screen is cleared, and the grid is re-drawn in each cycle.
- User input is processed, and if the puzzle is solved, a message is displayed along with an instruction to press the spacebar to restart the game.

```
def run(self):
    running = True
    while running:
        self.screen.fill(self.colors['black'])
        self.draw_grid()

        if self.game_state == 'solved':
            text = self.font.render('Puzzle Solved!', True, self.colors['white'])
            text1 = self.font.render('Press SPACE to restart', True, self.colors['white'])
            self.screen.blit(text, (550, 200))
            self.screen.blit(text1, (550, 250))

        running = self.handle_events()
        pygame.display.flip()

    pygame.quit()

if __name__ == "__main__":
    SudokuGame().run()
```

```
elif event.key == pygame.K_SPACE and self.game_state == 'solved':
    self.__init__()
```

7. Restarting the Game:

- If the game is solved and the user presses the spacebar, the grid is reset by re-initializing the game with `self.__init__()`, effectively restarting the puzzle.

Key Features:

- **Interactivity:** Players can select cells by clicking and input numbers using the keyboard.
- **Move Validation:** Each number entered is checked for validity according to Sudoku rules.

- **Game Restart:** After solving the puzzle, the user can restart the game by pressing the spacebar.
- **Game State Tracking:** The program keeps track of whether the puzzle is being played or has been solved.

This simple yet functional Sudoku game allows players to enjoy a classic puzzle, with basic features such as move validation and restart functionality.

Data Structures Used

The grid and its operations rely on nested lists for easy indexing and iteration. Additionally, pre-filled cells are tracked as a list of tuples, simplifying the logic for determining which cells are editable.

Evidence of Working Program

1. **Initial Game Screen:** The game starts with a partially filled Sudoku grid. Cells are separated by thin lines, with thicker borders around 3x3 subgrids.
2. **Cell Selection:** Users can click on any empty cell to select it, with the selected cell outlined in red.

🧠 Sudoku

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

3. **Number Input:** Users can input numbers into the selected cell, provided they are valid moves.

Sudoku

2	5			3		9	4	1
	1				4	3	5	7
4		7				2	6	8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

4. **Puzzle Solved:** Once the puzzle is solved correctly, a message is displayed congratulating the user and providing an option to restart the game.

Sudoku

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Puzzle Solved!
Press SPACE to restart

Conclusion

The Sudoku game demonstrates the use of Python and Pygame for creating interactive applications. The project highlights the importance of integrating graphical interfaces with robust game logic to provide an engaging user experience. With its modular design, the program can be extended to include features like generating random puzzles, providing hints,

and implementing a timer system. This project was a valuable learning experience in developing graphical applications and solving algorithmic challenges associated with puzzle games.

Source: <https://github.com/Iglesias123/Finalproject-algopro.git>