# Project Documentation

This documentation provides an in-depth overview of each module in the Sudoku application:

display.py

,

logic.py

,

solver.py

, and

**init**.py

. Each section details the purpose, functionalities, and integration of the respective module within the project.

## Table of Contents

* display.py Documentation
  + Overview
  + Functions
    - draw\_grid(screen)
    - draw\_numbers(screen, board, highlight=None, initial\_board=None)
    - draw\_button(screen, rect, text, mouse\_pos)
    - display\_sudoku(board, initial\_board)
  + Usage Example
  + Module Integration
* logic.py Documentation
  + Overview
  + Functions
    - find\_empty(board)
    - is\_valid(board, num, pos)
    - generate\_full\_board()
    - fill\_board(board)
    - count\_solutions(board, solutions, limit=2)
    - generate\_puzzle(board, difficulty=40)
  + Usage Example
  + Module Integration
* solver.py Documentation
  + Overview
  + Functions
    - find\_empty(board)
    - is\_valid(board, num, pos)
    - solve\_generator(board)
  + Usage Example
  + Module Integration
* **init**.py Documentation
  + Overview
  + Contents
  + Module Integration

## 

display.py

Documentation

### Overview

The

display.py

module is responsible for the graphical user interface (GUI) of the Sudoku application. Utilizing Pygame, it renders the Sudoku grid, displays numbers, handles user interactions through buttons, and visualizes the solving process step-by-step.

### Functions

#### draw\_grid(screen)

**Description:** Draws the Sudoku grid on the provided Pygame screen, distinguishing between regular and thicker lines for 3x3 subgrids.

**Parameters:** - screen (pygame.Surface): The Pygame screen surface where the grid will be rendered.

**Usage:**

draw\_grid(screen)

#### draw\_numbers(screen, board, highlight=None, initial\_board=None)

**Description:** Renders the numbers on the Sudoku grid. It differentiates between pre-filled cells and those filled by the solver, and highlights the current cell being processed.

**Parameters:** - screen (pygame.Surface): The Pygame screen surface. -

board

(

list

of

list

of int): The current state of the Sudoku board. - highlight (tuple, optional): The (row, col) of the cell to highlight. - initial\_board (

list

of

list

of int, optional): The original puzzle board to differentiate pre-filled cells.

**Usage:**

draw\_numbers(screen, board, highlight=(2, 3), initial\_board=initial\_puzzle)

#### draw\_button(screen, rect, text, mouse\_pos)

**Description:** Draws an interactive button with hover effects on the Pygame screen.

**Parameters:** - screen (pygame.Surface): The Pygame screen surface. - rect (pygame.Rect): The rectangle defining the button’s position and size. - text (str): The label text on the button. - mouse\_pos (tuple): The current position of the mouse, used to detect hover state.

**Usage:**

draw\_button(screen, random\_button\_rect, "Randomize", mouse\_pos)

#### display\_sudoku(board, initial\_board)

**Description:** Main function that initializes the Pygame window, handles the rendering loop, manages user interactions, and animates the solving process.

## **Parameters:**

board

(

list

of

list

of int): The current state of the Sudoku board. - initial\_board (

list

of

list

of int): The original puzzle board.

**Usage:**

display\_sudoku(sudoku\_board, initial\_puzzle)

### Usage Example

if \_\_name\_\_ == "\_\_main\_\_":  
 import copy  
 from logic import generate\_full\_board, generate\_puzzle  
  
 # Generate a solvable puzzle  
 full\_board = generate\_full\_board()  
 sudoku\_board = generate\_puzzle(full\_board, difficulty=40)  
 initial\_puzzle = copy.deepcopy(sudoku\_board)  
 print("Initial puzzle generated.")  
  
 # Display the Sudoku puzzle  
 display\_sudoku(sudoku\_board, initial\_puzzle)

### Module Integration

## **Imports:**

solver.py

Uses solve\_generator to fetch solving steps. -

logic.py

Utilizes

generate\_full\_board

and generate\_puzzle for puzzle creation.

* **Interaction Flow:**
  1. **Puzzle Generation:** Calls

generate\_full\_board

and generate\_puzzle from

logic.py

to create a new Sudoku puzzle. 2. **Displaying Puzzle:** Uses draw\_grid, draw\_numbers, and draw\_button to render the puzzle and buttons. 3. **Solving Animation:** When the “Solve” button is clicked, initializes solve\_generator from

solver.py

and updates the board step-by-step based on the generator’s output.

## 

logic.py

Documentation

### Overview

The

logic.py

module forms the backbone of the Sudoku application, handling the generation of complete Sudoku boards, creating playable puzzles with unique solutions, and ensuring the validity of number placements. It employs backtracking algorithms to fill and validate Sudoku boards, ensuring a diverse and reliable puzzle generation process.

### Functions

#### 

find\_empty(board)

**Description:** Finds the next empty cell in the Sudoku board.

## **Parameters:**

board

(

list

of

list

of int): The current state of the Sudoku board represented as a 9x9 grid. Empty cells are denoted by 0.

**Returns:** - tuple (int, int): The (row, column) indices of the first empty cell found. - None: If the board is completely filled with no empty cells.

**Example:**

board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
empty = find\_empty(board)  
print(empty) # Output might be (0, 2)

#### is\_valid(board, num, pos)

**Description:** Checks whether placing a specific number at a given position is valid according to Sudoku rules. It ensures that the number does not duplicate in the corresponding row, column, or 3x3 subgrid.

## **Parameters:**

board

(

list

of

list

## of int): The current state of the Sudoku board.

num

## (int): The number to be placed on the board (1-9).

pos

(tuple of int): The (row, column) position where the number is to be placed.

**Returns:** - bool: True if the placement is valid, False otherwise.

**Example:**

board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
valid = is\_valid(board, 4, (0, 2))  
print(valid) # Output: True or False

#### 

generate\_full\_board()

**Description:** Generates a complete and valid Sudoku board by filling in all cells using a backtracking algorithm.

**Parameters:** - None

## **Returns:**

list

of

list

of int: A fully solved 9x9 Sudoku board.

**Example:**

full\_board = generate\_full\_board()  
for row in full\_board:  
 print(row)

#### 

fill\_board(board)

**Description:** Recursively fills the Sudoku board to create a complete puzzle. It employs a backtracking approach, randomly shuffling numbers to ensure varied board generation.

## **Parameters:**

board

(

list

of

list

of int): The Sudoku board to be filled. Initially, it should contain zeros (0) in empty positions.

**Returns:** - bool: True if the board is successfully filled, False otherwise.

**Example:**

board = [[0 for \_ in range(9)] for \_ in range(9)]  
success = fill\_board(board)  
if success:  
 for row in board:  
 print(row)

#### count\_solutions(board, solutions, limit=2)

**Description:** Counts the number of possible solutions for a given Sudoku board. The counting stops once the number of solutions reaches the specified limit. This function is crucial for ensuring that generated puzzles have unique solutions.

## **Parameters:**

board

(

list

of

list

of int): The Sudoku board to evaluate. - solutions (

list

of int): A single-item list acting as a mutable counter for the number of solutions found. - limit (int, optional): The maximum number of solutions to count before stopping. Defaults to 2.

**Returns:** - None: The function updates the solutions list in place.

**Example:**

board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
solutions = [0]  
count\_solutions(board, solutions, limit=2)  
print(solutions[0]) # Output: Number of solutions found (1 or more)

#### generate\_puzzle(board, difficulty=40)

**Description:** Removes numbers from a fully solved Sudoku board to create a playable puzzle. The difficulty parameter determines how many cells are removed. The function ensures that the resulting puzzle has only one unique solution.

## **Parameters:**

board

(

list

of

list

of int): A fully solved Sudoku board. - difficulty (int, optional): The number of cells to remove. Higher values increase difficulty by removing more cells. Defaults to 40.

## **Returns:**

list

of

list

of int: A Sudoku puzzle board with numbers removed to match the specified difficulty.

**Example:**

full\_board = generate\_full\_board()  
puzzle = generate\_puzzle(full\_board, difficulty=40)  
for row in puzzle:  
 print(row)

### Usage Example

import copy  
from logic import generate\_full\_board, generate\_puzzle  
  
# Generate a complete Sudoku board  
full\_board = generate\_full\_board()  
print("Full Board:")  
for row in full\_board:  
 print(row)  
  
# Create a puzzle with a difficulty of 40 cells removed  
puzzle = generate\_puzzle(full\_board, difficulty=40)  
print("\nSudoku Puzzle:")  
for row in puzzle:  
 print(row)

**Output:**

Full Board:  
[5, 3, 4, 6, 7, 8, 9, 1, 2]  
[6, 7, 2, 1, 9, 5, 3, 4, 8]  
...  
[2, 8, 7, 4, 1, 9, 6, 3, 5]  
  
Sudoku Puzzle:  
[5, 3, 0, 6, 7, 8, 9, 1, 2]  
[6, 0, 2, 1, 9, 5, 3, 0, 8]  
...  
[2, 8, 0, 4, 1, 9, 6, 3, 5]

### Module Integration

The

logic.py

module interacts closely with other modules in the Sudoku application to provide a seamless experience:

* **solver.py**:
  + Utilizes is\_valid and

find\_empty

from

logic.py

to implement the solving algorithm. - Leverages solve\_generator to yield board states step-by-step for animation purposes.

* **display.py**:
  + Uses

generate\_full\_board

and generate\_puzzle to create and present puzzles to the user. - Integrates with the solver to visualize the solving process in the Pygame interface.

* **\_\_init\_\_.py**:
  + Facilitates easy imports of functions from

logic.py

into other modules.

\*\*Example Integration in

display.py

:\*\*

from logic import generate\_full\_board, generate\_puzzle  
from solver import solve\_generator  
  
# Generate and display a puzzle  
full\_board = generate\_full\_board()  
puzzle = generate\_puzzle(full\_board, difficulty=40)  
  
# Initialize the solver generator  
solver = solve\_generator(puzzle)  
  
# Iterate through solver steps for animation  
for step in solver:  
 # Update the display with the current board state  
 display.update(step)  
 pygame.display.flip()  
 pygame.time.delay(100) # Control animation speed

## 

solver.py

Documentation

### Overview

The

solver.py

module implements a generator-based Sudoku solver utilizing the backtracking algorithm. It efficiently finds solutions by recursively trying valid numbers in empty cells and backtracking upon encountering conflicts. The generator approach allows the solving process to be animated step-by-step within the GUI.

### Functions

#### 

find\_empty(board)

**Description:** Finds the next empty cell in the Sudoku board.

## **Parameters:**

board

(

list

of

list

of int): The current state of the Sudoku board represented as a 9x9 grid. Empty cells are denoted by 0.

**Returns:** - tuple (int, int): The (row, column) indices of the first empty cell found. - None: If the board is completely filled with no empty cells.

**Example:**

board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
empty = find\_empty(board)  
print(empty) # Output might be (0, 2)

#### is\_valid(board, num, pos)

**Description:** Checks whether placing a specific number at a given position is valid according to Sudoku rules. It ensures that the number does not duplicate in the corresponding row, column, or 3x3 subgrid.

## **Parameters:**

board

(

list

of

list

## of int): The current state of the Sudoku board.

num

## (int): The number to be placed on the board (1-9).

pos

(tuple of int): The (row, column) position where the number is to be placed.

**Returns:** - bool: True if the placement is valid, False otherwise.

**Example:**

board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
valid = is\_valid(board, 4, (0, 2))  
print(valid) # Output: True or False

#### solve\_generator(board)

**Description:** A generator-based Sudoku solver using the backtracking algorithm. It yields the state of the board after each number placement or removal, facilitating step-by-step animation.

## **Parameters:**

board

(

list

of

list

of int): The current state of the Sudoku board.

## **Yields:**

list

of

list

of int: The board state after each number placement or removal.

**Example:**

from solver import solve\_generator  
  
board = [  
 [5, 3, 0, 0, 7, 0, 0, 0, 0],  
 # ... rest of the board  
]  
  
generator = solve\_generator(board)  
for step in generator:  
 print(step) # Each printed step represents a board state

### Usage Example

from solver import solve\_generator  
from logic import generate\_full\_board, generate\_puzzle  
import copy  
  
# Generate a complete Sudoku board  
full\_board = generate\_full\_board()  
  
# Create a puzzle by removing numbers  
puzzle = generate\_puzzle(full\_board, difficulty=40)  
initial\_puzzle = copy.deepcopy(puzzle)  
  
print("Starting Sudoku Solver...")  
solver = solve\_generator(puzzle)  
  
# Iterate through solver steps  
for step\_num, step\_board in enumerate(solver, start=1):  
 print(f"Step {step\_num}:")  
 for row in step\_board:  
 print(row)  
 print("\n")

**Output:**

Starting Sudoku Solver...  
Step 1:  
[5, 3, 4, 6, 7, 8, 9, 1, 2]  
[6, 7, 2, 1, 9, 5, 3, 4, 8]  
...  
Step N:  
[...]  
...

### Module Integration

The

solver.py

module is integral for the solving animation within the GUI:

* \*\*Integration with

display.py

:\*\* - The

display.py

module initializes the solve\_generator with the current puzzle state. - It iterates through the generator, updating the display after each yielded board state to animate the solving process.

* \*\*Dependency on

logic.py

:\*\* - Utilizes is\_valid and

find\_empty

functions from

logic.py

to validate number placements and locate empty cells.

\*\*Example Integration in

display.py

:\*\*

from solver import solve\_generator  
  
def display\_sudoku(board, initial\_board):  
 # ... [initial setup code]  
   
 solver = solve\_generator([row[:] for row in board])  
 solving = True  
 last\_solve\_step = pygame.time.get\_ticks()  
   
 while running:  
 # ... [event handling code]  
   
 if solving and solver:  
 current\_time = pygame.time.get\_ticks()  
 if current\_time - last\_solve\_step >= SOLVE\_DELAY:  
 try:  
 current\_board = next(solver)  
 # Update the display with current\_board  
 last\_solve\_step = current\_time  
 except StopIteration:  
 solving = False  
 # ... [rendering code]

## 

**init**.py

Documentation

### Overview

The

**init**.py

file designates the directory as a Python package, enabling the importation of modules within the package more conveniently. It serves as the initialization script for the package, allowing for cleaner and more organized imports.

### Contents

# \_\_init\_\_.py  
  
from .display import display\_sudoku  
from .solver import solve\_generator  
from .logic import generate\_full\_board, generate\_puzzle

### Module Integration

* **Facilitates Imports:**
  + By importing specific functions from

display.py

,

solver.py

, and

logic.py

,

**init**.py

allows users to access these functions directly from the package without needing to reference individual modules.

**Example Usage:**

from sudoku import display\_sudoku, solve\_generator, generate\_full\_board, generate\_puzzle  
  
# Generate a puzzle  
full\_board = generate\_full\_board()  
puzzle = generate\_puzzle(full\_board, difficulty=40)  
  
# Display the puzzle  
display\_sudoku(puzzle, full\_board)

* **Enhances Readability and Maintenance:**
  + Centralizes the import statements, making it easier to manage and understand the package’s interface.
* **Scalability:**
  + As the project grows, additional modules and functions can be imported and exposed through

**init**.py

, maintaining a clean namespace.

**Note:** Ensure that all modules (display.py,

solver.py

,

logic.py

) are located in the same directory as

**init**.py

to facilitate proper imports.

# Additional Notes

* **Consistency:** Ensure that the documentation for each module is kept up-to-date with the code changes to maintain accuracy.
* **Enhancements:** Consider adding examples of common use cases, error handling, and edge case management to provide comprehensive guidance for future users or contributors.
* **Formatting:** Utilize consistent formatting across all documentation sections for readability and professionalism.

If you have any further questions or need additional assistance with your project documentation, feel free to ask!