**SudoKing Challenge**

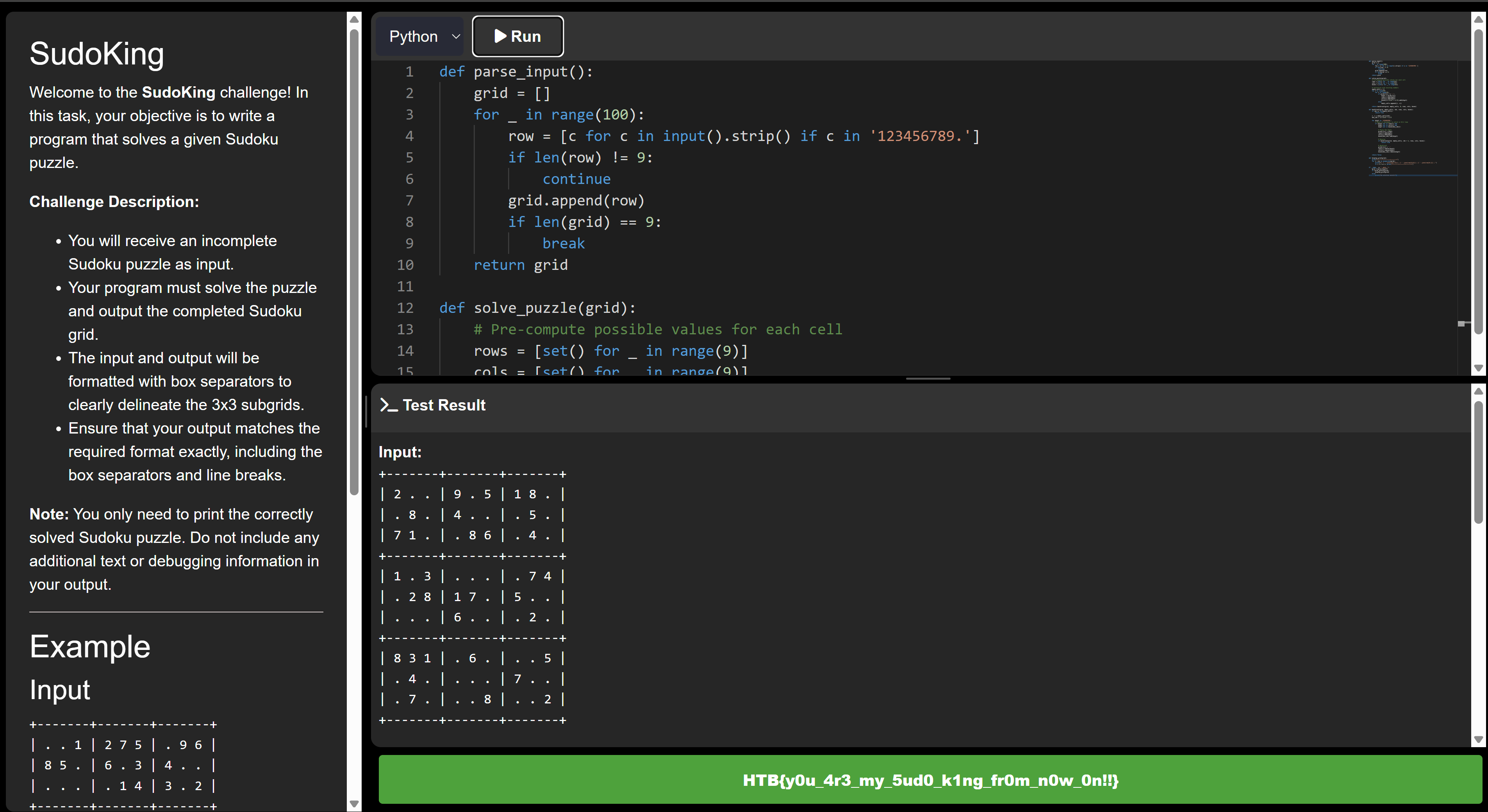
For this challenge, the goal was to implement a program that reads a partially filled Sudoku grid and outputs a fully solved grid, following standard Sudoku rules and a specific display format.

**Approach Overview:**

The solution uses a backtracking algorithm combined with constraint propagation to efficiently solve the puzzle. Here's a breakdown of the approach:

1. **Input Parsing (parse\_input)**:
   * Reads up to 100 lines of input and filters out invalid characters.
   * Extracts only digits 1-9 and . for empty cells.
   * Stops reading once a valid 9x9 grid is built.
2. **Grid Preparation (solve\_puzzle)**:
   * Initializes three lists of sets to track which digits are already used in:
     + Rows
     + Columns
     + 3x3 Boxes
   * Iterates over the grid to fill in these sets.
   * Collects positions of empty cells for focused solving.
3. **Backtracking Solver (backtrack)**:
   * Recursive function that attempts to fill each empty cell.
   * For each cell, it tries placing digits 1-9.
   * A digit is placed only if it is not already present in the corresponding row, column, or 3x3 box (ensuring O(1) validation).
   * If a placement leads to a dead end, it backtracks and tries the next possible digit.
   * The recursion continues until all cells are filled correctly or it concludes that no solution exists.
4. **Output Formatting (display\_grid)**:
   * Prints the solved grid in a structured layout with 3x3 box boundaries.
   * Matches the SudoKing platform's expected visual format exactly.

**Solve**



Flag is hidden



**Code**

def parse\_input():

grid = []

for \_ in range(100):

row = [c for c in input().strip() if c in '123456789.']

if len(row) != 9:

continue

grid.append(row)

if len(grid) == 9:

break

return grid

def solve\_puzzle(grid):

# Pre-compute possible values for each cell

rows = [set() for \_ in range(9)]

cols = [set() for \_ in range(9)]

boxes = [set() for \_ in range(9)]

# Initialize with existing numbers

empty\_cells = []

for r in range(9):

for c in range(9):

if grid[r][c] != '.':

digit = grid[r][c]

rows[r].add(digit)

cols[c].add(digit)

boxes[(r//3)\*3 + c//3].add(digit)

else:

empty\_cells.append((r, c))

return backtrack(grid, empty\_cells, 0, rows, cols, boxes)

def backtrack(grid, empty\_cells, idx, rows, cols, boxes):

if idx == len(empty\_cells):

return True

r, c = empty\_cells[idx]

box\_idx = (r//3)\*3 + c//3

for digit in '123456789':

# Check if placement is valid in O(1) time

if (digit not in rows[r] and

digit not in cols[c] and

digit not in boxes[box\_idx]):

# Place the digit

grid[r][c] = digit

rows[r].add(digit)

cols[c].add(digit)

boxes[box\_idx].add(digit)

# Recurse

if backtrack(grid, empty\_cells, idx + 1, rows, cols, boxes):

return True

# Backtrack

grid[r][c] = '.'

rows[r].remove(digit)

cols[c].remove(digit)

boxes[box\_idx].remove(digit)

return False

def display\_grid(grid):

print("+-------+-------+-------+")

for i, row in enumerate(grid):

print(f"| {' '.join(row[:3])} | {' '.join(row[3:6])} | {' '.join(row[6:])} | ")

if i % 3 == 2: print("+-------+-------+-------+")

if \_\_name\_\_ == "\_\_main\_\_":

grid = parse\_input()

if solve\_puzzle(grid):

display\_grid(grid)

else:

print("No solution exists")