

# Supplement: Case Study: Sudoku

For Introduction to Programming Using Python

By Y. Daniel Liang

This case study can be presented along with Chapter 11, "Multidimensional Lists."

This book teaches you how to program using a wide variety of problems with various levels of difficulty. We use simple, short, and stimulating examples to introduce programming and problem-solving techniques and use interesting and challenging examples to motivate students in programming. This section presents an interesting problem of a sort that appears in the newspaper every day. It is a number-placement puzzle, commonly known as *Sudoku*.

## 1 Problem Description

Sudoku is a  $9 \times 9$  grid divided into smaller  $3 \times 3$  boxes (also called regions or blocks), as shown in Figure 1(a). Some cells, called *fixed cells*, are populated with numbers from 1 to 9. The objective is to fill the empty cells, also called *free cells*, with numbers 1 to 9 so that every row, every column, and every  $3 \times 3$  box contains the numbers 1 to 9, as shown in Figure 1(b).

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

(a) Input



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

(b) Output

**Figure 1**

(b) is the solution to the Sudoku puzzle in (a).

For convenience, we use value 0 to indicate a free cell, as shown in Figure 2(a). The grid can be naturally represented using a two-dimensional list, as shown in Figure 2(b).

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	0	0	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

(a)

```
grid =
[[5, 3, 0, 0, 7, 0, 0, 0, 0],
 [6, 0, 0, 1, 9, 5, 0, 0, 0],
 [0, 9, 8, 0, 0, 0, 0, 6, 0],
 [8, 0, 0, 0, 6, 0, 0, 0, 3],
 [4, 0, 0, 8, 0, 3, 0, 0, 1],
 [7, 0, 0, 0, 2, 0, 0, 0, 6],
 [0, 6, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 4, 1, 9, 0, 0, 5],
 [0, 0, 0, 0, 8, 0, 0, 7, 9]]
```

(b)

**Figure 2**

A grid can be represented using a two-dimensional list.

### 2 Problem-Solving Strategy

How do you solve this problem? An intuitive approach is to employ the following three rules:

Rule 1: Fill in free cells from the first to the last.

Rule 2: Fill in a smallest number possible.

Rule 3: If no number can fill in a free cell, backtrack.

For example, you can fill 1 into grid[0][2], 2 into grid[0][3], 4 into grid[0][5], 8 into grid[0][6], and 9 into grid[0][7], as shown in Figure 3(a).

5	3	<u>1</u>	<u>2</u>	7	<u>4</u>	<u>8</u>	<u>9</u>	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6							
			4	1	9			5
				8			7	9

(a)

5	3	<u>1</u>	<u>2</u>	7	<u>4</u>	<u>9</u>	<u>8</u>	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6							
			4	1	9			5
				8			7	9

(b)

**Figure 3**

The program attempts to fill in free cells.

Now look at grid[0][8]. There is no possible value to fill in this cell. You need to backtrack to the previous free cell at grid[0][7] and reset its value. Since grid[0][7] is already 9, no new value is possible. So you have to backtrack to its previous free cell at grid[0][6] and change its value to 9. Continue to move forward to set grid[0][7] to 8, as shown in Figure 3(b). Now there is still no possible value for

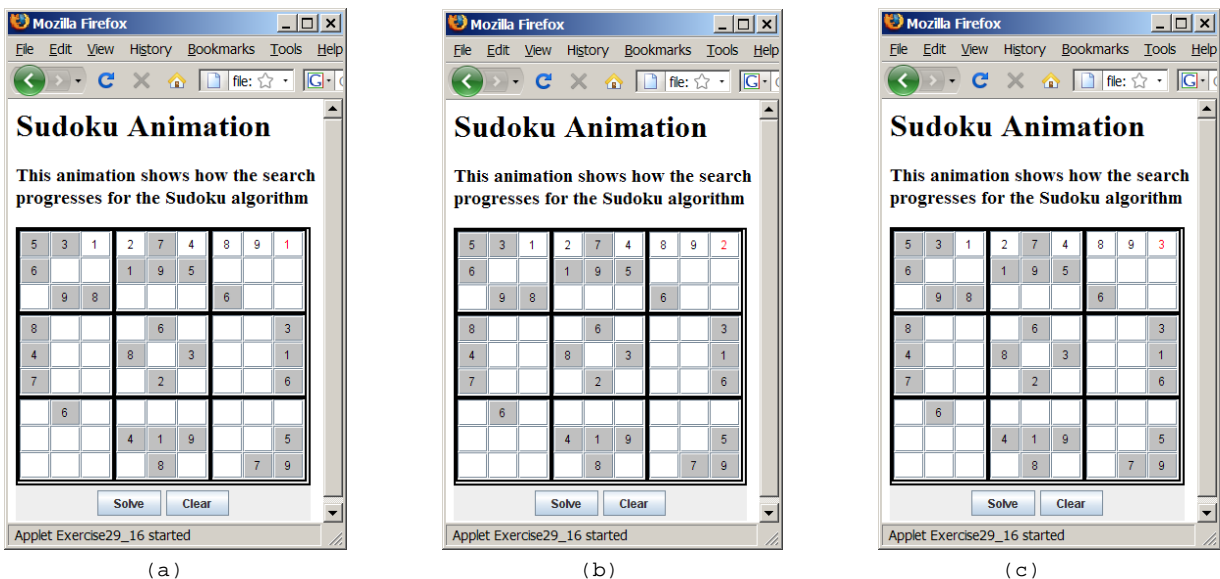
`grid[0][8]`. Backtrack to `grid[0][7]`: no possible new value for this cell. Backtrack to `grid[0][6]`: no possible new value for this cell. Backtrack to `grid[0][5]` and change it to 6. Now continue to move forward.

The search moves forward and backward continuously until one of the following two cases arises:

- All free cells are filled. A solution is found.
- The search is backtracked to the first free cell with no new possible value. The puzzle has no solution.

**Pedagogical NOTE**

Follow the link [www.cs.armstrong.edu/liang/animation/SudokuAnimation.html](http://www.cs.armstrong.edu/liang/animation/SudokuAnimation.html) to see how the search progresses. As shown in Figure 4(a), number 1 is placed in the first row and last column. This number is invalid, so the next value 2 is placed in Figure 4(b). This number is still invalid, so the next value 3 is placed in Figure 4(c). The simulation displays all the search steps.

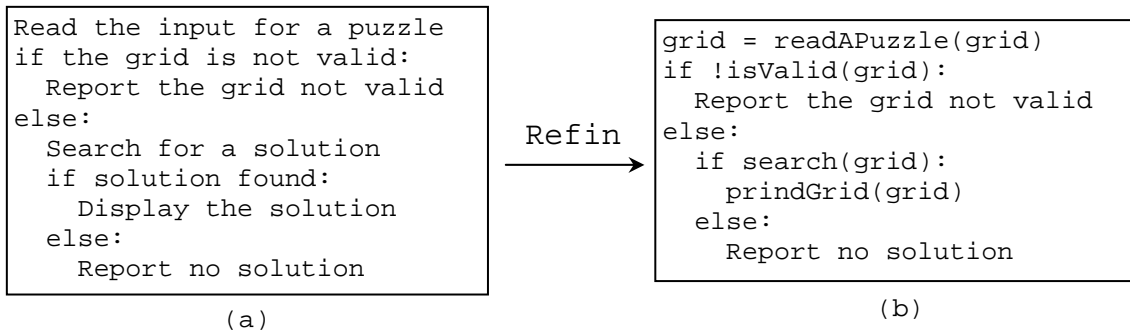


**Figure 4**

*The animation tool enables you to observe how the search works for solving a Sudoku puzzle.*

**3 Program Design**

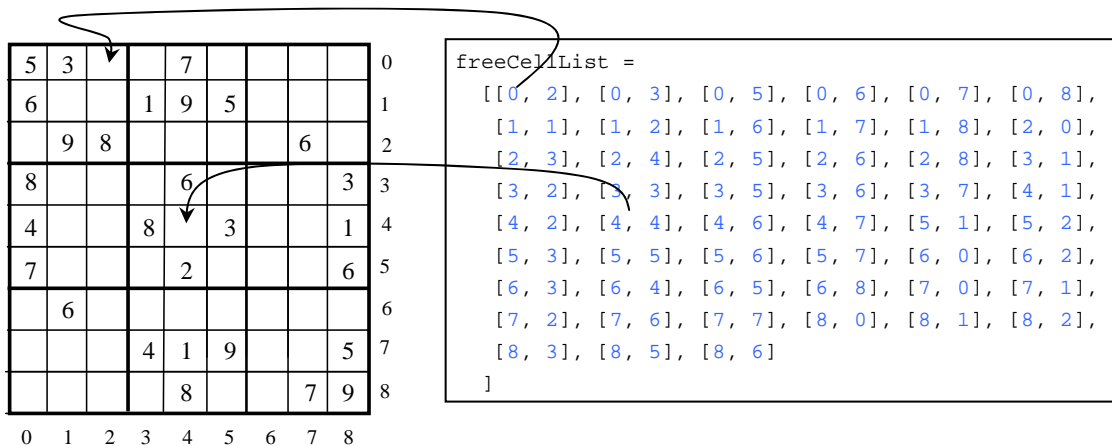
The program can be designed as shown in (a) and further refined with functions as in (b):



The `readAPuzzle` function reads a Sudoku puzzle from the console into `grid`. The `printGrid` function displays the contents in `grid` to the console. The `isValid` function checks whether the grid is valid. These functions are easy to implement. We now turn our attention to the `search` function.

#### 4 Search Algorithm

To better facilitate search on free cells, the program stores free cells in a two-dimensional list, as shown in Figure 5. Each row in the list has two columns, which indicate the subscripts of the free cells in the grid. For example, `{freeCellList[0][0], freeCellList[0][1]}` (i.e., `{0, 2}`) is the subscript for the first free cell `grid[0][2]` in the grid and `{freeCellList[25][0], freeCellList[25][1]}` (i.e., `{4, 4}`) is the subscript for free cell `grid[4][4]` in the grid, as shown in Figure 5.

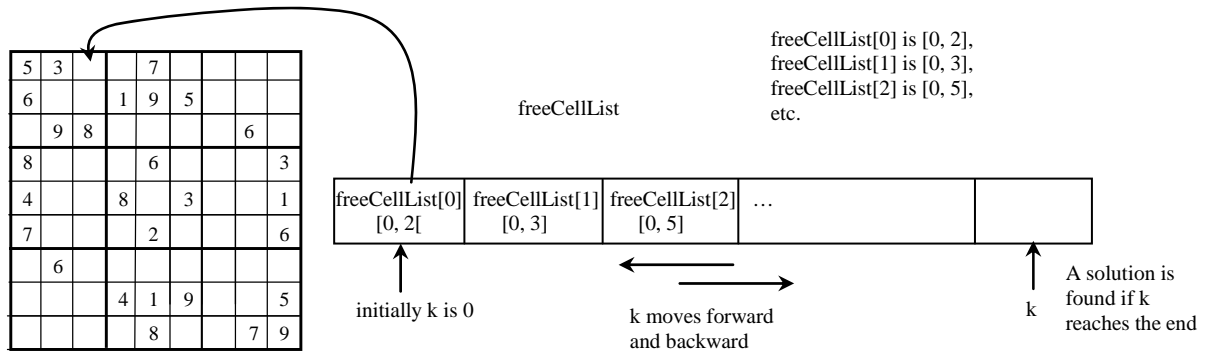


**Figure 5**

`freeCellList` is a two-dimensional list representation for the free cells.

The search starts from the first free cell with `k = 0`, where `k` is the index of the current free cell being considered in the free-cell list, as shown in Figure 6. It fills a valid value in the current free cell and then moves forward to consider the next. If no valid value can be found for the current free cell, the search backtracks to the preceding free cell. This process

continues until all free cells are filled with valid values (a solution is found) or the search backtracks to the first free cell with no solution.



**Figure 6**

The search attempts to fill free cells with appropriate values.

The search algorithm can be described as follows:

Step 1: (Initialization) Obtain a freeCellList from a grid, as shown in Figure 5. Let k denote the index in freeCellList with k initially 0, as shown in Figure 6.

Repeatedly perform Steps 2-4 until search ends with a solution or no solution

```
{
  Step 2: Let grid[i][j] be the current free cell being considered, where i = freeCellList[k][0] and j = freeCellList[k][1].

  Step 3: If grid[i][j] is 0, fill it with 1.

  Step 4: Consider three cases:
    Case 1: grid[i][j] is valid. If k is the last index in freeCellList, a solution is found. Otherwise, search moves forward with k = k + 1.
    Case 2: grid[i][j] is invalid and grid[i][j] < 9. Set a new value for the free cell with grid[i][j] = grid[i][j] + 1.
    Case 3: grid[i][j] is invalid and grid[i][j] is 9. If k = 0, search ends with no solution. Otherwise backtrack with k = k - 1, reset i = freeCellList[k][0] and j = freeCellList[k][1], and continue to backtrack if grid[i][j] is 9. When grid[i][j] < 9, set grid[i][j] = grid[i][j] + 1.
}
```

## 5 Implementation

Listing 1 gives the source code for the program.

**Listing 1 Sudoku.py**

```
def main():
    # Read a Sudoku puzzle
    grid = readAPuzzle()

    if not isValidGrid(grid):
        print("Invalid input")
    elif search(grid):
        print("The solution is found:")
        printGrid(grid)
    else:
        print("No solution")

    return 0

# Read a Sudoku puzzle from the keyboard
def readAPuzzle():
    print("Enter a Sudoku puzzle:")
    grid = []
    for i in range(9):
        line = input().split()
        grid.append([eval(x) for x in line])
    return grid

# Obtain a list of free cells from the puzzle
def getFreeCellList(grid):
    freeCellList = []
    for i in range(9):
        for j in range(9):
            if grid[i][j] == 0:
                freeCellList.append([i, j])

    return freeCellList

# Display the values in the grid
def printGrid(grid):
    for i in range(9):
        for j in range(9):
            print(grid[i][j], end = " ")
        print()

# Search for a solution
def search(grid):
    freeCellList = getFreeCellList(grid)
    numberOfFreeCells = len(freeCellList)
    if numberOfFreeCells == 0:
        return True # No free cells

    k = 0 # Start from the first free cell
```

```

while True:
    i = freeCellList[k][0]
    j = freeCellList[k][1]
    if grid[i][j] == 0:
        grid[i][j] = 1 # Fill the free cell with number 1

    if isValid(i, j, grid):
        if k + 1 == numberOfFreeCells:
            # No more free cells
            return True # A solution is found
        else:
            # Move to the next free cell
            k += 1
    elif grid[i][j] < 9:
        # Fill the free cell with the next possible value
        grid[i][j] = grid[i][j] + 1
    else:
        # grid[i][j] is 9, backtrack
        while grid[i][j] == 9:
            if k == 0:
                return False # No possible value
            grid[i][j] = 0 # Reset to free cell
            k -= 1 # Backtrack to the preceding free cell
            i = freeCellList[k][0]
            j = freeCellList[k][1]

        # Fill the free cell with the next possible value,
        # search continues from this free cell at k
        grid[i][j] = grid[i][j] + 1

    return True # A solution is found

# Check whether grid[i][j] is valid in the grid
def isValid(i, j, grid):
    # Check whether grid[i][j] is valid at the i's row
    for column in range(9):
        if column != j and grid[i][column] == grid[i][j]:
            return False

    # Check whether grid[i][j] is valid at the j's column
    for row in range(9):
        if row != i and grid[row][j] == grid[i][j]:
            return False

    # Check whether grid[i][j] is valid in the 3-by-3 box
    for row in range((i // 3) * 3, (i // 3) * 3 + 3):
        for col in range((j // 3) * 3, (j // 3) * 3 + 3):
            if row != i and col != j and grid[row][col] == grid[i][j]:
                return False

    return True # The current value at grid[i][j] is valid

# Check whether the fixed cells are valid in the grid
def isValidGrid(grid):
    for i in range(9):

```

```

    for j in range(9):
        if grid[i][j] < 0 or grid[i][j] > 9 or (grid[i][j] != 0 \
            and not isValid(i, j, grid)):
            return False

    return True # The fixed cells are valid

main()

```

### Sample Output

```

Enter a puzzle:
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0

```

The solution is found:

```

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

```

The program invokes the `readAPuzzle()` function (line 3) to read a Sudoku puzzle in a two-dimensional list `grid`. There are three possible outputs from the program:

- The input is invalid (line 5).
- A solution is found (line 7).
- No solution is found (line 10).

The `getFreeCellList` function (lines 25-32) returns a two-dimensional list storing the free-cell positions. `freeCellList[i][j]` indicates a free cell at row index `i` and column index `j`.



The `search` function invokes `getFreeCellList` to find all free cells (line 43). It then starts search from the first free cell with `k = 0` (line 48), where `k` is the position of the current free cell being considered in the free-cell list, as shown in Figure 6.

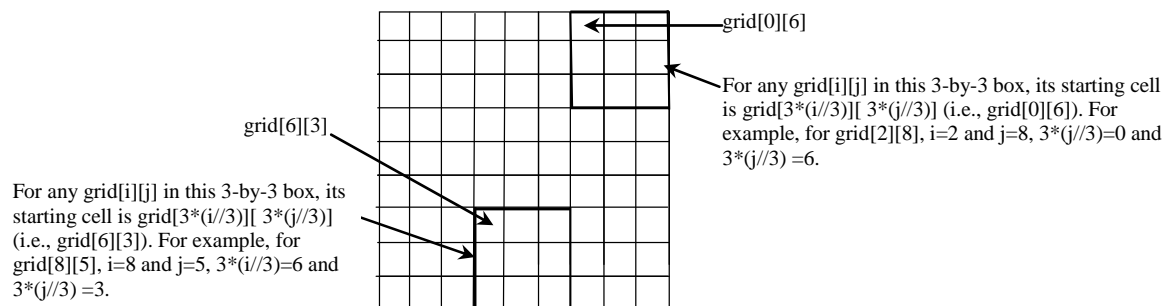
The value in a free cell starts with `1` (line 53). If the value is valid, the next cell is considered (line 61). If the value is not valid, its next value is considered (line 64). If the value is already `9`, the search is backtracked (lines 67-73). All the backtracked cells become free again and their values are reset to `0` (line 70). If the search backtracks to the free-cell list at position `k` and the current free-cell value is not `9`, increase the value by `1` (line 77) and continue the search.

The `search` function returns `True` when the search advances but no more free cells are left (line 58). A solution is found.

The search returns `False` when the search is backtracked to the first cell (line 69) and all possible values are exhausted for the cell. No solution can be found.

The `isValid(i, j, grid)` function checks whether the current value at `grid[i][j]` is valid. It checks whether `grid[i][j]` appears more than once at row `i` (lines 84-86), at column `j` (lines 89-91), and in the  $3 \times 3$  box (lines 94-97).

How do you locate all the cells in the same box? For any `grid[i][j]`, the starting cell of the  $3 \times 3$  box that contains it is `grid[(i // 3) * 3][(j // 3) * 3]`, as illustrated in Figure 7.



**Figure 7**

*The location of the first cell in a  $3 \times 3$  box determines the locations of other cells in the box.*

With this observation, you can easily identify all the cells in the box. Suppose `grid[r][c]` is the starting cell of a  $3 \times 3$  box; the cells in the box can be traversed in a nested loop as follows:

```
# Get all cells in a 3 by 3 box starting at grid[r][c]
for row in range(r, r + 3):
    for col in range(c, c + 3):
        # grid[row][col] is in the box
```

Note that there may be multiple solutions for an input. The program will find one such solution. You may modify the program to find all solutions in Programming Exercise 8.17.

It is cumbersome to enter 81 numbers from the keyboard. You may store the input in a file, say `sudoku.txt`, and run the program using the following command:

```
python Sudoku.py < sudoku.txt
```