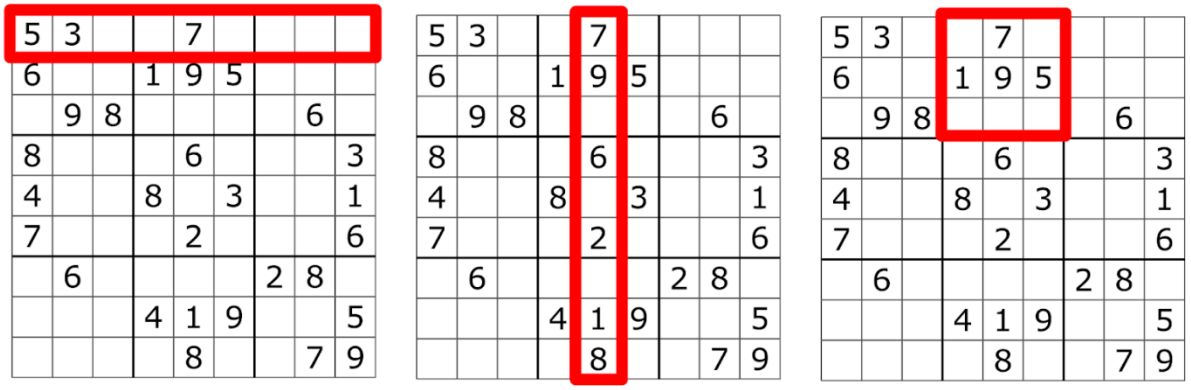
Goals of the Assignment

The goal of this assignment is to practice using Python sets. You will write an algorithm that determines whether or not a Sudoku puzzle solution is valid.

Sudoku is a puzzle game that challenges players to fill each of the squares in a 9x9 grid with the digits between 1 and 9. The rules state that the same digit cannot appear twice in any *row* or *column*. In addition, the board is divided into 3x3 *regions*, each of which must not include the same digit twice.



A partially completed Sudoku puzzle. The boxes indicate examples of a row, a column, and one of the 3x3 regions on the board, each of which must contain only unique integers from 1-9. In this example, the number 7 in the center of the top row is in each of the highlighted areas.

Every square on the board exists in exactly one row, one column, and one region. The number written in each square must be unique to all three. The *set* data structure stores unique values, and can therefore be used to determine whether or not a number has already been used.

Activities

This activity deals with the Sudoku of an `NxN` grid with the digits between `1` and `N`, where `N = 9, 16, 25, 36, ….` In this extended version of the game, the board is divided into `nxn` *regions*, where `n = sqrt(N)`, each of which must not include the same digit twice. As usual, the same digit cannot appear twice in any *row* or *column*.

A Sudoku puzzle comprises four components: a two-dimensional board and three lists of sets storing values at each row, column, and region. Those sets will be used to verify the board in an efficient manner. As you attempt to place a digit between `1` and `N` on the board, only a valid move should update the puzzle.

Open the provided file named "`sudoku.py`" and examine the code. Write the following functions:

1. `make_puzzle(N)` – builds an initial puzzle data structure and returns it.
    a. Create a board as a two-dimensional list filled with zeros indicating empty squares. Then add `N` digits between `1` and `N` to the board at randomly chosen locations such that the puzzle is valid.

a. Create a list of sets named `row_sets`, where `row_sets[k], 0<=k<N`, is the set containing non-zero digits in the kth row `board[k]`. Similarly, make `col_sets` and `reg_sets`.
b. You should represent the puzzle as a Python dictionary containing four string keys: `'board'`, `'row_sets'`, `col_sets'`, and `'reg_sets'`.

For example, `make_puzzle(9)` returns:

```
{'board': [[0, 0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 2, 0,
0, 0, 0, 9], [0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0,
0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0], [6, 0, 0,
0, 0, 0, 0, 0, 3], [0, 0, 0, 0, 0, 7, 0, 0, 0], [8, 0,
0, 0, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 5, 0]],
'row_sets': [{1}, {9, 2}, set(), set(), set(), {3, 6},
{7}, {8}, {4, 5}], 'col_sets': [{8, 4, 6}, set(), set(),
{2}, set(), {7}, set(), {5}, {1, 3, 9}], 'reg_sets':
[[set(), {2}, {1, 9}], [{6}, set(), {3}], [{8, 4}, {7},
{5}]]}
```

0. `get_square(puzzle, row, col)` – returns information about the square at the `row` and `col` position in the board. The information must include the value in the square and the three sets relevant to it.

The square information will be represented as a dictionary again. Using the puzzle in the above example, `get_square(puzzle, 0, 8)` must return:

```
{'value': 1, 'row_set': {1}, 'col_set': {1, 3, 9},
'reg_set': {1, 9}}
```

0. `move(puzzle, row, col, new_value)` – uses the `get_square` function to check if the square at the `row` and `col` position in the board is empty and `new_value` can be placed there without breaking the rules. If so, add `new_value` to the board and to the related sets, and return `True`. Otherwise, return `False` without modifying the `puzzle`.

0. `fill_puzzle(puzzle)` – uses the `move` function to fill the `puzzle` with random digits (between `1` and `N`) at random positions. You may stop filling when the number of attempts is at least `N**4` or 75% of the board is filled.

0. Your `main` function should create an initial puzzle and print it. Print the board using the provided `print_board` function. Then it should fill the board and print the final puzzle and board. It should also print elapsed time required to fill the board.

According to faculty tests, it took 0.81 seconds to fill 75% of a 100x100 board and 15.11 seconds for a 400 x 400 board.

0. What are the expected time complexities of `move` and `fill_puzzle`? Include your answer as a comment in the `sudoku.py` file.

Sample output 16 x 16 puzzle:
Board size: 16 x 16
Initial puzzle:
{'board': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 13, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 3, 0, 15, 0, 0, 0], [0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 11, 0, 9, 12], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], 'row_sets': [set(), {10}, set(), {7}, {14}, set(), {2, 13}, set(), set(), {1, 6}, set(), {3, 4, 15}, {5}, {8}, {16, 9, 11, 12}, set()], 'col_sets': [set(), {13}, set(), {8, 5}, {4}, set(), {10, 14}, set(), set(), {2, 7}, {3}, {16}, {11, 15}, {1, 6}, {9}, {12}], 'reg_sets': [[set(), {10}, {7}, set()], [{13}, {14}, {2}, set()], [set(), {4}, {3}, {1, 6, 15}], [{8, 5}, set(), {16}, {9, 11, 12}]]}

```
Initial board:
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][10][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][07][00][00]  [00][00][00][00]

[00][00][00][00]  [00][00][14][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][13][00][00]  [00][00][00][00]  [00][02][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]

[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][06][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [04][00][00][00]  [00][00][03][00]  [15][00][00][00]

[00][00][00][05]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][08]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
[00][00][00][00]  [00][00][00][00]  [00][00][00][16]  [11][00][09][12]
[00][00][00][00]  [00][00][00][00]  [00][00][00][00]  [00][00][00][00]
```

Final puzzle:
{'board': [[4, 0, 7, 15, 0, 12, 0, 16, 1, 6, 2, 8, 14, 11, 0, 9], [11, 16, 2, 0, 14, 6, 10, 7, 4, 9, 0, 15, 3, 5, 13, 8], [3, 8, 9, 0, 0, 5, 4, 13, 16, 14, 10, 0, 7, 12, 2, 15], [1, 0, 10, 13, 3, 8, 9, 0, 5, 7, 0, 12, 4, 0, 6, 16], [10, 9, 12, 7, 0, 0, 14, 15, 6, 16, 4, 3, 13, 0, 11, 5], [0, 0, 3, 11, 1, 13, 2, 8, 15, 0, 0, 7, 12, 4, 10, 14], [6, 13, 5, 16, 0, 0, 7, 11, 12, 2, 1, 10, 8, 9, 3, 0], [0, 14, 0, 4, 9, 3, 16, 10, 8, 13, 11, 5, 0, 15, 1, 2], [8, 11, 0, 6, 13, 2, 0, 1, 14, 4, 15, 0, 5, 10, 12, 7], [12, 5, 15, 14, 16, 9, 11, 3, 2, 8, 7, 0, 0, 6, 4, 13], [7, 1, 4, 10, 15, 14, 12, 0, 11, 0, 16, 9, 2, 3, 8, 0], [13, 2, 0, 9, 4, 0, 5, 6, 10, 12, 3, 0, 15, 14, 16, 11], [9, 0, 1, 5, 12, 7, 15, 2, 0, 10, 0, 4, 6, 13, 0, 3], [0, 15, 16, 8, 11, 4, 6, 0, 3, 5, 9, 1, 0, 0, 14, 0], [0, 4, 6, 2, 5, 10, 13, 0, 7, 15, 14, 16, 11, 8, 9, 12], [14, 3, 13, 12, 0, 16, 0, 9, 0, 0, 8, 11, 10, 2, 7, 1]], 'row_sets': [{1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 16}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16}, {2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16}, {1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 16}, {3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16}, {1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 14, 15}, {1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16}, {1, 2, 3, 4, 5, 8, 9, 10, 11, 13, 14, 15, 16}, {1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15}, {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16}, {1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 14, 15, 16}, {2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16}, {1, 2, 3, 4, 5, 6, 7, 9, 10, 12, 13, 15}, {1, 3, 4, 5, 6, 8, 9, 11, 14, 15, 16}, {2, 4, 5, 6, 7,

8, 9, 10, 11, 12, 13, 14, 15, 16}, {1, 2, 3, 7, 8, 9, 10, 11, 12, 13, 14, 16}], 'col_sets': [{1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14}, {1, 2, 3, 4, 5, 8, 9, 11, 13, 14, 15, 16}, {1, 2, 3, 4, 5, 6, 7, 9, 10, 12, 13, 15, 16}, {2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}, {1, 3, 4, 5, 9, 11, 12, 13, 14, 15, 16}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16}, {2, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16}, {1, 2, 3, 6, 7, 8, 9, 10, 11, 13, 15, 16}, {1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 15, 16}, {2, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16}, {1, 2, 3, 4, 7, 8, 9, 10, 11, 14, 15, 16}, {1, 3, 4, 5, 7, 8, 9, 10, 11, 12, 15, 16}, {2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15}, {1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15}, {1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16}, {1, 2, 3, 5, 7, 8, 9, 11, 12, 13, 14, 15, 16}], 'reg_sets': [[{1, 2, 3, 4, 7, 8, 9, 10, 11, 13, 15, 16}, {3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16}, {1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16}, {2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16}], [{3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 16}, {1, 2, 3, 7, 8, 9, 10, 11, 13, 14, 15, 16}, {1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 15, 16}, {1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15}], [{1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, {1, 2, 3, 4, 5, 6, 9, 11, 12, 13, 14, 15, 16}, {2, 3, 4, 7, 8, 9, 10, 11, 12, 14, 15, 16}, {1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16}], [{1, 2, 3, 4, 5, 6, 8, 9, 12, 13, 14, 15, 16}, {2, 4, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16}, {1, 3, 4, 5, 7, 8, 9, 10, 11, 14, 15, 16}, {1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14}]]}