Calcudoku

Purpose

To further your understanding of iteration and using multidimensional lists, as well as implementing an exhaustive search algorithm.

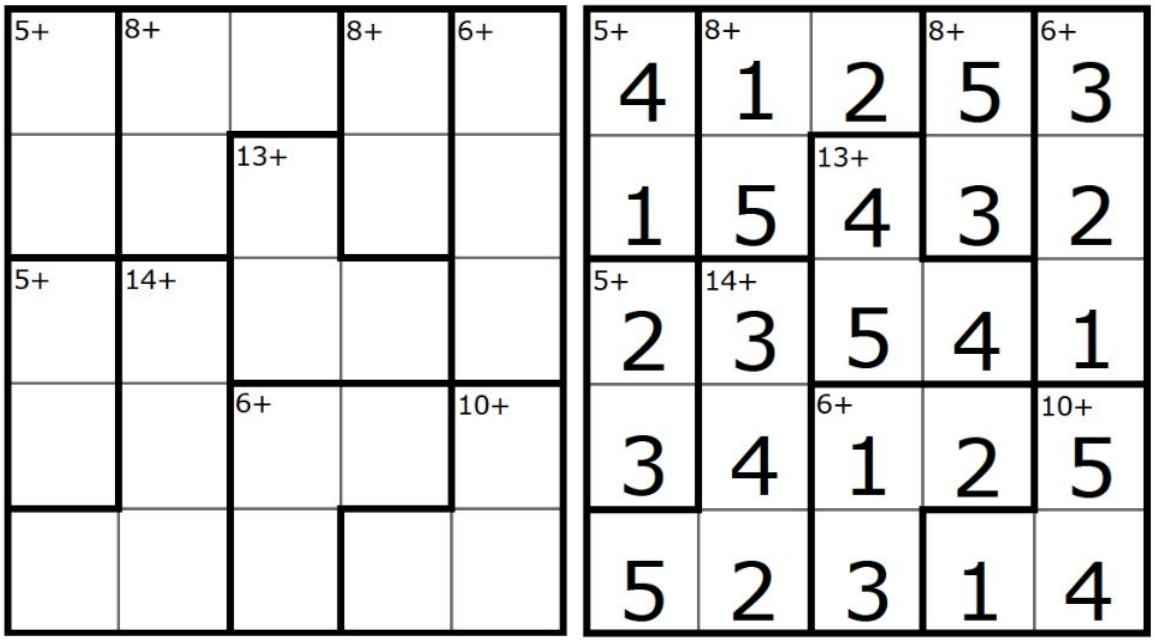
Description

For this program, you will be writing a solver for 5x5 Calcudoku puzzles. A Calcudoku puzzle is an NxN grid where the solution satisfies the following:

- Each row can only have the numbers 1 through N with no duplicates
- Each column can only have the numbers 1 through N with no duplicates
- The sum of the numbers in a cage (areas with a bold border) should equal the number shown in the upper-left portion of the cage

Puzzle input and output files will be made available on Canvas.

Here is a sample puzzle (left) and its solution (right):



Input

The beginning of a sample input file is shown below.

```
9
5 2 0 5
8 3 1 2 6
...
```

The first line contains the number of cages in the puzzle. After the first line, each subsequent line describes a cage. The first number of a line is the sum of the cage, the second is the number of cells in the cage, and all numbers afterward refer to the positions of the cells that make up the cage. In the puzzle, the cell positions are numbered starting with `0` for the upper-left cell and increase from left to right.

A sample run is shown below (user input is in bold):

Number of cages: **9**

Cage number 0: **5 2 0 5**

Cage number 1: **8 3 1 2 6**

Cage number 2: **8 2 3 8**

Cage number 3: **6 3 4 9 14**

Cage number 4: **13 3 7 12 13**

Cage number 5: **5 2 10 15**

Cage number 6: **14 4 11 16 20 21**

Cage number 7: **6 3 17 18 22**

Cage number 8: **10 3 19 23 24**

Output

Your program should display a solution to the puzzle output in the following format:

```
4 1 2 5 3
1 5 4 3 2
2 3 5 4 1
3 4 1 2 5
5 2 3 1 4
```

Implementation

Solving a Puzzle

Your program will solve puzzles using an exhaustive search (or "brute force") approach, in which it tries (potentially) all possible solutions until it finds the correct one. Your algorithm should perform the following:

1. Initialize all cells to `0`
2. Increment the value in the current cell by `1` (starting from the top-left cell)
    a. If the incremented value is greater than the maximum possible value, set the current cell to `0` and move back to the previous cell
    b. Otherwise, check if the number is valid. If so, continue to the next cell to the right (advancing to the next row when necessary)
3. Repeat Step 2 until the puzzle is fully populated and valid

For this algorithm to work, you will need to write functions to test if a puzzle is in a valid state. As you populate the puzzle with numbers, it becomes invalid if:

- Duplicates exist in any row or column
- The sum of values in a fully populated cage does not equal the required sum
- The sum of values in a partially populated cage equals or exceeds the required sum

Minimum Required Program Structure

In the function headers below, the parameter `grid` refers to a 2D list of integers representing the cells of the puzzle and `cages` refer to a 2D list of integers representing the values read from input.

For example, the initial grid shall look like this:
[[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]]

If the list above is referred to as grid, grid[0][0] is Cell #0, grid[0][1] is Cell #1, grid[0][4] is Cell #4, grid[1][0] is Cell #5, and so on. So, the first index is the row number. The second index is the column number. Think about how you can convert a cell number to two indices of the grid using arithmetic operators.

It is a list of 5 lists, where each list contains 5 integers. Use two nested for loops and list.append() to create a 2D list.

`main()`

The `main` function (with optional helper functions) should perform the following steps:

- Store the data from the puzzle input file in a 2D list of integers
- Create a 5x5 grid using a 2D list and fill it with zeros
- Only one `while` loop is recommended to validly populate every cell in the grid

`get_num_cages()`

It asks the user to input the number of cages and returns an int value representing the number of cages. Use input() to ask the user to enter the information. If the input from the user is not a valid number, ask the user to enter the information. Use the built-in isalpha() and isdigit() to check if a string is of letters in alphabets or numbers.

`get_cage_info(num)`

It asks the user to input the information about cages and returns a list of lists, where the number of the lists is num, and each of the lists contains information about a cage. Use input() to ask the user to enter the information. You have to repeat asking the user to input the information the num times. Use Python built-in split() function to split a string into a list of strings at a space: e.g. elements = string.split(). Use the list comprehensions to convert a list of strings to a list of int.

`validate_all(grid, cages)`

It returns True if all 3 validation functions below return `True` and `False` otherwise.

`validate_rows(grid)`

It returns `True` if all rows contain no duplicate positive numbers and `False` otherwise.

`validate_cols(grid)`

It returns `True` if all columns contain no duplicate positive numbers and `False` otherwise. It is recommended that you transpose the grid (convert its rows to columns and columns to rows) and pass this transposition to `validate_rows` to reuse your existing code.

`validate_cages(grid, cages)`

It returns `True` if the sum of values in a fully populated cage equals the required sum or the sum of values in a partially populated cage is less than the required sum and `False` otherwise.

You might want to create these helper functions:

```
def transpose_grid(grid)->list:
    """Transposes a grid.

    Args:
        grid (list): a grid
    Returns:
        list: a list of lists which is a transposed version of the grid.
    """

def create_grid(dim)->list:
    """creates a grid of dim * dim.
    Args:
        dim (int): the dimension
    Returns:
        list: a list of lists of 0s.
    """

def print_grid(grid):
    """Prints the grid.

    Args:
        grid (list): a list of lists
    """
```

Testing

You are required to write at least 3 tests for each function (except `main`) in `calcudoku_tests.py`. Since we are emphasizing test-driven development, you should write tests for each function first. In doing so, you will have a better understanding as to what the functions take as input and produce as output, which makes writing the function definitions easier.

Download puzzles and sample output files from Canvas.

Each puzzle can be found in a separate file:

```
test1.in, test2.in, test3.in, ...
```

Your program should be run using:

```
python3 calcudoku.py < test#.in
```

You should compare your output with the corresponding output files using `diff` (without the use of any flags):

```
test1.out, test2.out, test3.out, ...
```

For example,

```
python3 calcudoku.py < test1.in > my_out1.txt
diff test1.out my_out1.txt
```

If the only difference between the sample output and your output is the messages to prompt the user to enter the number of cages and the cage info, you can consider your output to be correct.

Submission

Submit your calcudoku.py to Gradzilla and get it scored.
There are two parts to the grader for project 3. Part 1 unit-tests your functions. Part 2 tests your overall program: i.e. main(). The score in Part 2 will be your grade for the project.
Submit `calcudoku.py`, `calcudoku_tests.py`, and eight output files (`my_out1.txt`, `my_out2.txt`, `my_out3.txt`, `my_out4.txt`, `my_out5.txt`, `my_out6.txt`, `my_out7.txt`, and `my_out8.txt`) to Canvas.