Due Date: Sunday March 27th, 11:59pm.

- The purpose of this assignment is to practice building, inspecting, and modifying 2D lists effectively.
- This often requires nested for-loops, but not always. It involves thinking of the structure like an N x M matrix of labeled spots, each with a row- and column-index. As before, we are restricting ourselves to the most basic functionalities, and any others that you want, you should implement yourself.
- You will turn in a single python file following our naming convention: `netID_Lab#_P#.py`
- Include the following in comments at the top of your file: Name, G#, lecture and lab sections, and any references or comments we ought to know.
- Use the Piazza discussion forums (and professor/TA office hours) to obtain assistance. Any post with project code in it must be made private, visible to "Instructors" and you alone. Have a specific question ready, and both show what you're thinking and show what you've tried independently before you got stuck. We will prod for more details if needed.

Background:

Two-dimensional lists aren't conceptually more difficult than single-dimension lists, but in practice the nested loops and more complex traversals and interactions merit some extra practice. We will create grids of cells and simulate the n-queens problem.

Procedure

- Implement the functions described later in this document, using the following testing file as you go.
  - http://cs.gmu.edu/~yzhong/112/p4/tester4p.py
  - Invoke it as with prior assignments: `python3 tester4p.py yourcode.py`
  - You can name functions to test (e.g. just `get_coords` and `get_size`):
    `python3 tester4p.py yourcode.py get_coords get_size`
- Download this samples file, which has typed out many example boards and some extra functionality that convert your code into a mini game of solving N-queens puzzle. You can use the provided examples to help testing your code. You will be able to play the game when your implementation is ready!
  - http://cs.gmu.edu/~yzhong/112/p4/p4provided.py

- **Remember: your grade is significantly based on passing test cases – try to completely finish individual functions before moving on. The easiest way to implement many functions is to call the earlier/easier functions, so it will pay off twice to complete functions before moving on. Don't let yourself be "almost done" with a function, but be missing all the test cases!**

Allowed/Disallowed Things

You may only use the following things. If you use something disallowed, you won't get points for using it.

- no modules may be imported.
- basic statements, variables, operators, del, indexing, slicing, are all allowed
- any form of control flow we've covered is allowed (if/else, loops, break, continue)
- data types: int, float, string, bool, list and tuple
- functions: range(), len(), int(), str()
- methods: xs.append(), xs.extend(), xs.insert(), xs.pop(),xs.remove()
- calling other functions of the project (and your own helper functions). ***Please do this!*** ☺

N-Queens Problem

The **N-queens puzzle** is the problem of placing N queens on an NxN chessboard so that no queen can attack another queen. Two queens can attack each other if they are on the same row, the same column, or the same diagonal. The figures below show examples of two queens that can attack each other on a 4x4 board. N-queens problem is a generalized version of the famous eight queens puzzle, for which the wiki page is a decent introduction, too: http://en.wikipedia.org/wiki/Eight_queens_puzzle.

```
--Q-        ----        Q---        --Q-
----        -Q-Q        ----        ----
--Q-        ----        --Q-        Q---
----        ----        ----        ----
```

same column | **same row** | **same (major) diagonal** | **same (minor) diagonal**

Data Structure

Indexing a board

When we have a board, we have two dimensions. The first dimension indicates the row (top row to bottom row), and the second dimension indicates the column (left column to right column). A chess board is always square with the same number of rows and columns, and every row must have the same number of cells. Thus with N rows and N columns, we have a board with indexes as shown to the right.

(0,0)	(0,1)	…	**(0,N-1)**
(1,0)	…	…	(1,N-1)
…	…	…	…
(N-1,0)	(N-1,1)	…	**(N-1,N-1)**

		Q	
Q			
			Q

Representing a board

We will use a list of lists of Booleans to represent a chessboard with queens. The outer (first) dimension represents the rows (from top to bottom), and the inner lists represent the items in that row, from left to right. All inner lists should have the same length. If a cell has a queen, its boolean value is `True`; otherwise `False`. The 4x4 board to the left can be represented as:

```
board_four = [[False, False, True, False], [True, False, False, False],
              [False, False, False, True], [False, False, False, False]]
board_four_empty = [[False, False, False, False], [False, False, False, False],
                    [False, False, False, False], [False, False, False, False]]
```

Given a board, we can also use a list of lengh-2 tuples to record the locations of all queens on the board. Every tuple in the list is an integer pair (row, column) that represents the coordinate of a queen. For example, the 4x4 boards above corresponds to the lists of coordinates below:

```
board_four_coords = [(0,2),(1,0),(2,3)]
board_four_empty_coords = []
```

Printing a board

Given a board, we can display it on screen as an NxN table, with one row on each line and one cell represented by a character. For example, if we use `'Q'` to represent a cell with a queen and `'-'` to represent an empty cell, the string representations of the 4x4 boards above would be:

```
board_four_string = '--Q-\nQ---\n---Q\n----\n'
board_four_empty_string = '----\n----\n----\n----\n'
```

When print() is called for these strings, the corresponding boards are displayed as the left. Note: there is an empty line at the very end.

```
--Q-
Q---
---Q
----
```

```
----
----
----
----
```

Goal

We will implement a series of functions that build up boards and check whether the board presents a solution of the N-queens problem. Along the way, we will be creating boards, inspecting boards, and modifying boards.

Functions

Note: for most functions, you should NOT modify the incoming board represented as a 2D list. The only exceptions are place_one and remove_one on Page 5.

- **`build_empty_board(size):`** Given positive **`int`** values for the **`size`** of a board, create a **`list of lists of Booleans`** that has **`False`** values at each location (representing empty cells).
  - **Assume: `size`** is a non-negative integer. Zero size creates an empty list.
  - `build_empty_board(2)` → `[[False, False], [False, False]]`
  - `build_empty_board(1)` → `[[False]]`
  - `build_empty_board(4)` → `[[False, False, False, False], [False, False, False, False], [False, False, False, False], [False, False, False, False]]` *#board_four_empty*

- **`build_board(coords,size):`** Given a **`list of coordinates`** for the locations of queens and a non-negative **`int size`**, build a board (a list of lists of Booleans) of that size, and mark cells with a queen **`True`**. A valid coordinate must be a pair of non-negative integers within range. If any coordinate is invalid, **`None`** should be returned.
  - **Assume: `coords`** is a **`list`** of tuples, each tuple is a pair of integers; **`size`** is a non-negative integer.
  - `build_board([(0,0)],1)` → `[[True]]`
  - `build_board([(0,0)],2)` → `[[True, False], [False, False]]`
  - `build_board([(0,0),(1,1)],2)` → `[[True, False], [False, True]]`
  - `build_board([(0,0),(-1,3)],2)` → `None` *#invalid coordinate*

- **`is_square(board):`** Given a **`board`** as a **`list of lists of Booleans`**, check whether it represents a square board with the same number of rows and columns. If the rows don't all have the same number of items in them, or if the number of rows does not match the number of columns, this function returns **`False`**. Empty board is square.
  - **Assume: `board`** is a **`list of lists of Booleans`**.
  - `is_square([[True,False,False],[False,False,False],[True,False,False]])` → `True`
  - `is_square([[True,False,False],[False,False,False]])` → `False` *#2 rows x 3 columns*
  - `is_square([[True],[False,False],[True,False,False]])` → `False` *#rows not same length*

- **`get_size(board):`** Given a **`board`** as a **`list of lists of Booleans`**, find out the size of the board. If the board is not square, this function returns **`None`**.
  - **Assume: `board`** is a **`list of lists of Booleans`**.
  - **Hint**: Use your **`is_square`** definition!
  - `get_size([[True,False,False],[False,False,False],[True,False,False]])` → `3`
  - `get_size([[True,False,False],[False,False,False]])` → `None` *# not square*
  - `get_size([[True],[False,False],[True,False,False]])` → `None` *# not square*

- **`get_coords(s):`** Given a **`board`** as a **`list of lists of Booleans`**, read through it and create a list of **`int`** pairs for all cells with a queen. Each pair is a **`(row, column)`** coordinate. The pairs must be ordered by lowest row, and lowest column when rows match.
  - **Assume: `board`** is a **`list of lists of Booleans`**.
  - `get_coords([[True,False],[False,True]])` → `[(0,0), (1,1)]`
  - `get_coords([[True,False],[True,True]])` → `[(0,0), (1,0), (1,1)]`
  - `get_coords([[False,False],[False,False]])` → `[]`

- **`show_board(board, queen='Q', empty='-'):`** Given a **`board`**, and the options indicate what character to use for cells with and without a queen, create a string that represents the indicated board. When we print the returned string, it should show the board on the screen, one line for each row and no spaces between cells. The last character should be a newline. Check Page 2 for additional examples.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**; **`queen`** and **`empty`** are single characters.
  - **Hint**: You'll likely want to call **`print(show_board(someBoard))`** when trying it out.
  - `show_board([[True, False], [False, True]])` → `"Q-\n-Q\n"`
  - `show_board([[False, False, True, False], [False, True, False, False], [False, False, False, True], [False, False, False, False]])` → `"--Q-\n-Q--\n---Q\n----\n"`
  - `show_board([[False,False],[True,True]],'A','.')` → `"..\nAA\n"` `# changed characters`
  - `show_board([])` → `""`

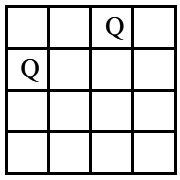
- **`row_conflict(board,r):`** Given a **`board`** and an integer **`r`** indicating a row index, check and return **`True`** if a queen is already placed in row **`r`**; return **`False`** otherwise.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**; **`board`** is square; **`r`** is an integer.
  - **Hint**: Remember, the board is zero-indexed, and negative indexes are ***not*** to be used.
  - `row_conflict([[True,False],[False,False]],0)` → `True`
  - `row_conflict([[True,False],[False,False]],1)` → `False`
  - `row_conflict([[True,False],[False,False]],-2)` → `False` `# negatives not allowed`
  - `row_conflict([[True,False],[False,False]],5)` → `False` `# index out of bound`

- **`column_conflict(board,c):`** Given a **`board`** and an integer **`c`** indicating a column index, check and return **`True`** if a queen is already placed in column **`c`**; return **`False`** otherwise.
  - **Assume:** **`board`** is **`list of lists of Booleans`**; **`board`** is square; **`c`** is an integer.
  - **Hint**: Remember, the board is zero-indexed, and negative indexes are ***not*** to be used.
  - `column_conflict([[True,False],[False,False]],0)` → `True`
  - `column_conflict([[True,False],[False,False]],1)` → `False`

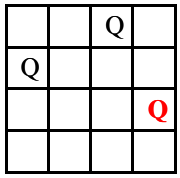
- **`diagonal_conflict(board, r, c):`** Given a **`board`** and two integers indicating row/column positions, determine if a queen is already placed at the same diagonal as coordinate **`(r,c)`**; return the answer as a boolean.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**; **`board`** is square; **`r`** and **`c`** are integers.
  - **Hint**: Remember, the board is zero-indexed, and negative indexes are ***not*** to be used.
  - `diagonal_conflict([[True,False],[False,False]],1,1)` → `True` `# major diagonal`
  - `diagonal_conflict([[False,True],[False,False]],1,0)` → `True` `# minor diagonal`
  - `diagonal_conflict([[True,False],[False,False]],1,0)` → `False` `# column conflict but no diagonal conflict`
  - `diagonal_conflict([[False, True, False, False], [False, False, False, True], [False, False, False, False], [False, False, True, False]], 2, 0)` → `False` `# no conflict`

- **`is_solved(board):`** Given a **`board`**, verify whether it is a solution of N-queens problem and return **`True`** if it is a complete solution; return **`False`** otherwise.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**.
  - **Hint**: Use **`get_coords`** can help you find the locations of queens.
  - `is_solved([[True]])` → `True`
  - `is_solved([[True,False],[False,False]])` → `False` `# not enough queens`
  - `is_solved([[True,False],[False,True]])` → `False` `# diagonal conflict`
  - `is_solved([[True,False,False],[False,False,False]])` → `False` `# not square board`

- **`place_one(board, r, c):`** Given a **`board`** and two integers indicating row/column positions, check whether another queen can be placed at coordinate **`(r,c)`**. If yes, update board to add the queen at **`(r,c)`** and return **`True`**. Otherwise, return **`False`** and **`board`** should NOT be changed.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**; **`board`** is square; **`r`** and **`c`** are integers.
  - **Hint**: Use your **`row_conflict`**, **`column_conflict`**, and **`diagonal_conflict`** definitions!
  - Examples all use **`board_4`** as drawn to the right.
    - `board_4 = [[False,False,True,False], [True,False, False,False], [False,False,False,False], [False,False, False,False]]`
    - `place_one(board_4, 0, 0)` → `False` *`# row conflict`*
    - `place_one(board_4, 2, 2)` → `False` *`# column conflict`*
    - `place_one(board_4, 2, 1)` → `False` *`# diagonal conflict`*
    - `place_one(board_4, 2, 3)` → `True`
      `# board_4 is updated to [[False,False,True,False],`
      `# [True,False,False,False], [False,False,False,True],`
      `# [False,False,False,False]] after the last function call`
    - `place_one(board_4, -2, -1)` → `False`
      `# negative coordinates not allowed`

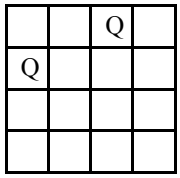


original board_4

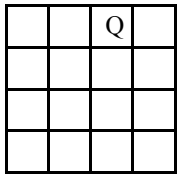


updated board_4 after place_one

- **`remove_one(board, r, c):`** Given a **`board`** and two integers indicating row/column positions, check whether a queen is at coordinate **`(r,c)`**. If yes, update board to remove that queen at **`(r,c)`** and return **`True`**. Otherwise, return **`False`** and **`board`** should NOT be changed.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**; **`board`** is square; **`r`** and **`c`** are integers.
  - Examples all use **`board_4`** as drawn to the right.
    - `board_4 = [[False,False,True,False], [True,False, False,False], [False,False,False,False], [False,False, False,False]]`
    - `remove_one(board_4, 1, 0)` → `True`
      `# board_4 is updated to [[False,False,True,False],`
      `# [False,False,False,False], [False,False,False,False],`
      `# [False,False,False,False]] after the function call`
    - `remove_one(board_4, 1, 1)` → `False` *`#no queen at (1,1)`*
    - `remove_one(board_4, -3, -4)` → `False` *`#negative coordinates`*



original board_4

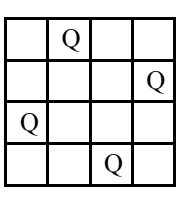


updated board_4 after remove_one

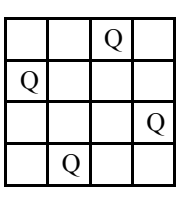
Extra Credit

Solve this problem for extra credit (up to +5%) and good practice.

- **`mirror(board):`** Given a **`board`**, return a new board that is the mirror reflection of the original board. Note: do NOT modify the original board, build up a new board and return the copy.
  - **Assume:** **`board`** is a **`list of lists of Booleans`**.
  - **Hint**: Use your **`get_coords`** definition!
  - `board_four_1 = [[False, True, False, False], [False, False, False, True], [True, False, False, False], [False, False, True, False]]`
  - `mirror(board_four_1)` → `[False, False, True, False], [True, False, False, False], [False, False, False, True], [False, True, False, False]]`



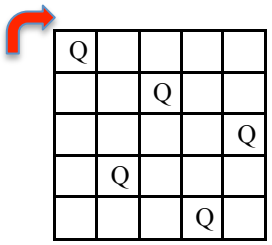
board_four_1

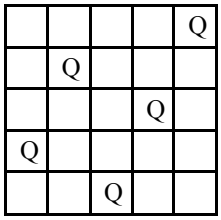


mirror of board_four_1

- **`rotate(board):`** Given a **`board`**, return a new board that can be obtained by rotating the original board clock-wise for 90 degrees. Note: do NOT modify the original board, build up a new board and return the copy.
    - **Assume:** **`board`** is a **`list of lists of Booleans`**.
    - **Hint**: Use your **`get_coords`** definition!
    - `board_five_1 = [[True, False, False, False, False], [False, False, True, False, False], [False, False, False, False, True], [False, True, False, False, False], [False, False, False, True, False]]`
    - `rorate(board_five_1)` → `[[False, False, False, False, True], [False, True, False, False, False], [False, False, False, True, False], [True, False, False, False, False], [False, False, True, False, False]]`

`board_five_1`





`rotate of board_five_1`

Some provided definitions (to play the game of N-queens!)

- **`PROVIDED CODE: solve(size):`** For a given **`size`**, this will try to solve the N-queens problem (N=**`size`**) through a sequence of interactive steps. Starting from an empty board, the user needs to decide whether and where to place or remove a queen from the current board at every step until a solution is found. ***Play with it!***
    - **Assumes:** **`size`** is a non-negative integer.
    - This function calls your implementation of **`build_empty_board`**, **`show_board`**, **`is_solved`**, **`place_one`** and **`remove_one`**. It will not be able to find correct solutions if your implementation of those functions is incomplete or buggy.
    - **How to play**: when your implementation is ready, copy the provided code over into your .py file. Then import the file and enter the interactive mode using **`python3 -i yourcode.py.`** Sample run included in the last page.

Grading Rubric

```
Code passes shared tests:      85  (zero-point for hard-coding)
Well-documented/submitted:     10
Appropriate loop/list usage:    5
TOTAL:                        100  +5 extra credit
```

Reminders on Turning It In:

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available.

You can turn in your code as many times as you want; we only grade the last submission that is <=48 hours late. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment,

you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least for part of the program.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

Use a backup service. Do future you an enormous favor, and just keep all of your code in some automatically synced location, such as a Dropbox or Google Drive folder. Every semester someone's computer is lost/drowned/dead, or their USB drive is lost/destroyed, or their hard drive fails. Don't give these situations the chance to doom your project work!

```
demo$ python3 -i yourcode.py
>>> solve(4)
Welcome to n-queens game! The size you choose to play is 4.
----
----
----
----

Please select: 1--place a queen, 2--remove a queen, 3--quit: (enter to place)1
which row do you want to place the next queen? (0-3) 0
which column do you want to place the next queen? (0-3) 0
Successfully placed a queen at (0,0)! Nice move!
Q---
----
----
----

Please select: 1--place a queen, 2--remove a queen, 3--quit: (enter to place)
which row do you want to place the next queen? (0-3) 1
which column do you want to place the next queen? (0-3) 1
Cannot put a queen at (1,1)! Try again ...
Q---
----
----
----

Please select: 1--place a queen, 2--remove a queen, 3--quit: (enter to place)2
which row do you want to remove a queen? (0-3) 0
which column do you want to remove a queen? (0-3) 0
----
----
----
----

Please select: 1--place a queen, 2--remove a queen, 3--quit: (enter to place)3
Thanks for playing! Good-bye!
>>>
```