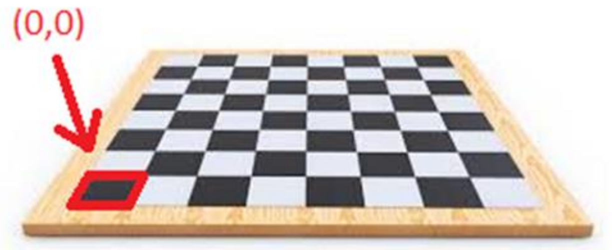
CISC 160
Lab 01

Background

We will be creating a chess game. For this game, you will need to create classes for some of the pieces "on" the board. Each class will track the current state and some of the rules for the chess piece, while the rest of the rules will be handled by an overseeing application.

User Requirements

To do this, we will need you to create different types of chess pieces to populate the board. All `Chess_Pieces` will have some of the same properties: a unique ID (represented by a string), a position on a board represented by an (x, y) coordinate, a color (represented by a string), a direction ("UP" or "DOWN"), and a board size (expressed as a tuple of (width, height)). The default size of the chess board should be 8×8. However, this should be configurable by parameter. Regardless of the size of the board, (0,0) should be considered to be in the lower, left hand corner. Keep in mind that the orientation of the board does matter, but assume the coordinate system's origin is the lower left hand corner from where you are viewing the chess board (See image below of standard 8x8 chess board). If a piece is not on the chess board yet, its position should be `(None, None)`.



All **`Chess_Pieces`** will also have some universally similar behaviors: If a `Chess_Piece` is not already on the board, it should be able to be placed on any valid position on the chess board. You do not need to worry about whether that position is already occupied, but this action should not change the position of the `Chess_Piece` if its already on the board. The `Chess_Piece` should be able to move to a position that is a valid movement position for the piece (**although the list of valid moves is not specified within the `Chess_Piece` class itself, since each `Chess_Piece` has its own rules for movement**). Any `Chess_Piece` should be able to be removed from the board at any point. Finally, a `Chess_Piece` can try to take another piece. This is done (generally; note the exception for `Pawns`, below) by checking to see if the other `Chess_Piece` is on a position that this `Chess_Piece` can move onto. If the target `Chess_Piece` is in a valid position for this `Chess_Piece`, then this `Chess_Piece` is moved to the position of the target `Chess_Piece` and the target `Chess_Piece` is removed from the board.

With the general behavior of all `Chess_Pieces` defined, we need to define the behaviors of individual pieces. A **`Pawn`** is a `Chess_Piece` that can only move one or two spaces in the direction that it is facing. The `Pawn` also has special rules for taking another piece: The target `Chess_Piece` must be one diagonal space in the direction that it is facing. For example, if the `Pawn` is on (1, 3) and is facing "UP," then the only positions that the `Pawn` can take are (0, 4), and (2, 4). Finally, the `Pawn` can be replaced by another piece at certain points. The piece replacing the Pawn cannot already be on the board when replacing the Pawn. When a Pawn is replaced, it is removed and another Chess_Piece is placed in the Pawn's former position.

A **Knight** is a Chess_Piece that moves by leaping in an "L-shape," meaning that it moves one space ahead and two spaces to a side OR two spaces ahead and one space to a side. These spaces are the only spaces that the Knight can land on to move or take another piece and any Chess_Pieces in any intermediate space are safe. For example, if the Knight started on (1, 3), then the list of potential valid moves would contain (2, 1), (2, 5), (3, 2), (3, 4), (0, 5), and (0, 1) and these would be the only positions where the Knight could take another Chess_Piece. The moves to positions (-1, 4) and (-1, 2) would be illegal because they would place the Knight off of the board.

A **King** is a Chess_Piece that can only move one space in any of the 8 directions (straight line or diagonal). There are other rules restricting how a King can move but we will leave overseeing application to deal with those specific rules.

Finally, there are three pieces that have a strong similarity in behavior. A **Bishop** is a Chess_Piece that can move diagonally for 1 or more positions. A **Rook** is a Chess_Piece that can move laterally (horizontally or vertically) for 1 or more positions. Finally, a **Queen** is a Rook that can move diagonally OR laterally (horizontally or vertically) for 1 or more positions.

FINAL NOTES: The proper way to represent that there are no valid moves within any specific implementation is to return an empty list, not None. No Chess_Piece, when attempting to determine the valid moves, needs to check for obstructions from other Chess_Piece. This behavior and condition will be checked by the overseeing game program at run time.

Lab Requirements

For this lab, create 7 different classes using object-oriented principles and design to implement the above behaviors. These objects are to be called Chess_Piece, Pawn, Knight, King, Bishop, Rook, and Queen. Any Chess_Piece will have its ID, initial position, color, and direction declared at instantiation. Each Chess_Piece can also have the size of the chess board it is on declared at instantiation but, if it is not, then the board size should default to the standard 8×8 board. Every other object is a Chess_Piece; however, we are going to consider the Queen to also be a more specific type of Rook. **The Chess_Piece class should be in its own code file named chess_piece.py. All other classes should be included into a single file called pieces.py**.

All objects will have pure accessor methods for ALL state variables. Additionally, all objects will have an accessor method that checks if piece is on the chess board. If it is, it should return True and False otherwise. Finally, the method get_valid_moves should return a list of tuples representing all of the valid moves for the given piece. All other methods are mutator methods which behave as described above.

Remember: These requirements are not all encompassing. Use your brain, your knowledge of the system, your knowledge of coding conventions, and the names and descriptions of certain features to make logical leaps about feature behaviors. For instance, is there a modular way to check if a given position is on the chess board? Also, remember that you may add private helper methods or add return values to methods if one is not otherwise specified as these will not affect the modularity of separately designed code.

<u>Submission</u>: Submit your chess_piece.py and pieces.py files.

<u>Incomplete Class Diagrams</u>

Class:	Chess_Piece, Knight, King, Bishop, Rook, or Queen
Fields:	Up to you as long as you follow the conventions
Construct:	__init__(ID, initial position, color, direction, board size)
Behaviors:	• get_ID() • get_position() • get_color() • get_direction() • get_board_size() • is_piece_on_board() • get_valid_moves() • place((int, int)) • move((int, int)) • remove() • take(Chess_Piece)

Class:	Pawn
Fields:	Up to you as long as you follow the conventions
Construct:	__init__(ID, initial position, color, direction, board size)
Behaviors:	• get_ID() • get_position() • get_color() • get_direction() • get_board_size() • is_piece_on_board() • get_valid_moves() • place((int, int)) • move((int, int)) • remove() • take(Chess_Piece) • replace(Chess_Piece)