

Assignment: Gekitai

Instructions

- You can use anything in the [Python Standard Library](#); but don't use any third-party library.
- The underlined blue texts in the Sample Runs section refers to the user inputs. It does NOT mean that your program needs to underline them.
- Please follow closely the format of the sample output. Your program should produce **exactly** the same output as the sample (same text, symbols, letter case, spacing, etc.). The output should end with a single newline character, which has been provided by the `print()` function by default.
- Please follow exactly the specified name, parameter list and return value(s) for the functions to be written if stated (for our easier grading). You may define additional functions for specific tasks or further task decomposition if you see fit.

Introduction

Write a Python module `gekитай.py` to implement a two-player board game called [Gekitai](#) (a Japanese word which means “repel” or “push away”). Here is the game description from the designer (we rephrased a little bit): Gekitai is a 3-in-a-row game played on a 6x6 grid. Each player has eight colored pieces and takes turns placing them anywhere on any open space on the board. When placed, a piece pushes all adjacent pieces outwards one space if there is an open space for it to move to (or off the board). Pieces shoved off the board are returned to the player. If there is not an open space on the opposite side of the pushed piece, it does not push (a newly played piece cannot push two or more other lined-up pieces). The first player who either (1) lines up three of their color in a row (horizontal or vertical or diagonal) at the end of their turn (after pushing), or (2) has all eight of their pieces on the board (also after pushing), is declared the winner. To quickly understand how to play this game, you may also watch this game review [video](#) or play this [online game](#) implementing Gekitai.

The key idea of this game is the “repel” effect when placing a piece onto the board. For example, refer to Figure 1 below. Suppose that the player (Red) of the current turn is to put a piece at location B2. Then all the three pieces (both the player's and opponent's) adjacent to B2 will be pushed away by one square outward. So, the black piece originally at A1 is shoved off the board and recycled to the player (Black) for making future turns whereas the pieces at C2 and B3 will be repelled to new positions D2 and B4 respectively.

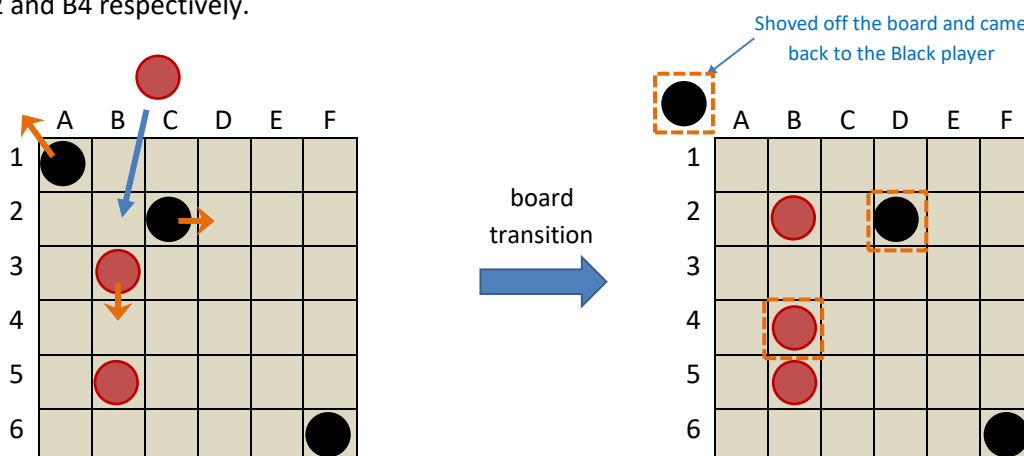


Figure 1: An example move on the game board and its effect on the adjacent pieces

However, remember (from the above description) that a move cannot push away a piece which has another adjacent piece (both the player's and opponent's) occupying the square it is being pushed

onto. For example, look at Figure 2 below. If the player (Black) puts a piece at B3, it won't repel the piece at B4 downward because there is no open space (B5 is already occupied), and only the piece at B2 will be pushed upward to position B1.

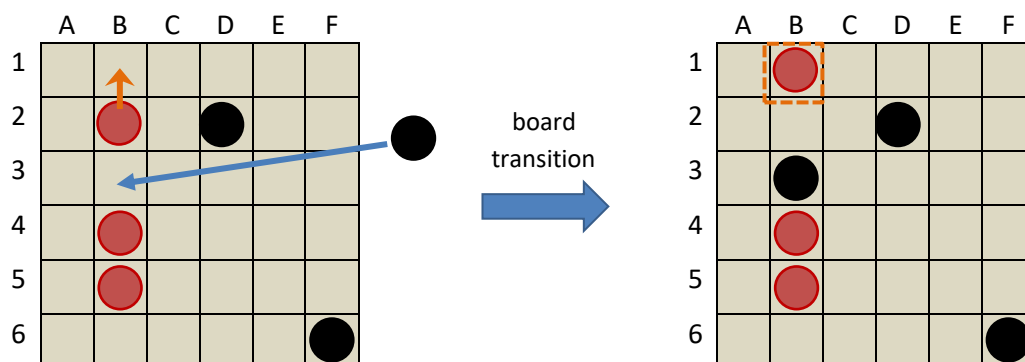


Figure 2: Another example move on the game board and its effect on the adjacent pieces

The last move made by Black was indeed a bad one – if Red puts a piece at B6 now, Red will win the game because three red pieces have formed a vertical line (see Figure 3).

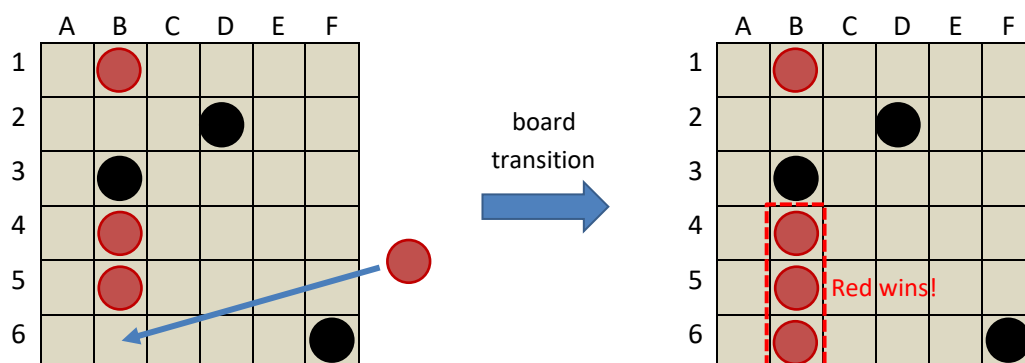


Figure 3: A move that lets Red win the game

To achieve the repelling effect, your program may need to scan all the eight directions (N, NE, E, SE, S, SW, W, NW) around the target position for a piece placement. For example, to put a piece at D3 in Figure 4, it should check if there exist any piece(s) at all the blue cells, and if any, it should check if the corresponding purple cell(s) are empty before moving the pieces outward into the purple cells.

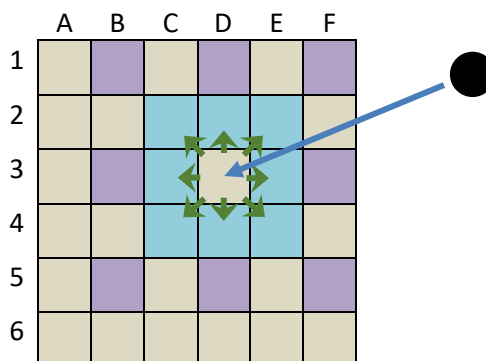


Figure 4: Positions to check to repel pieces adjacent to a target cell

Figure 5 below shows other possible winning conditions. Recall that a line can be formed horizontally, vertically, and diagonally, and that there is another winning condition: if all the eight pieces of the same color are placed on the board, the player of that color will win.

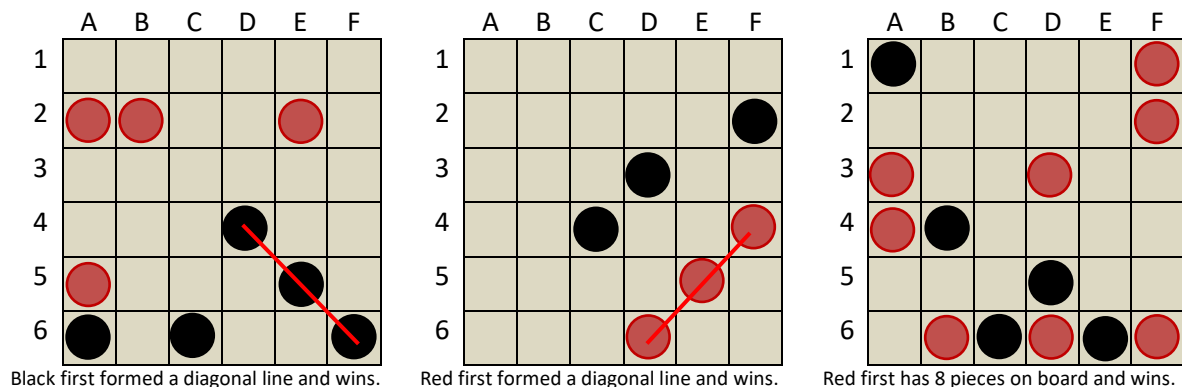


Figure 5: Other winning conditions

Game Board Representation

We are going to implement this game as a console-based program and the game board will be represented by characters arranged in Figure 6 below:

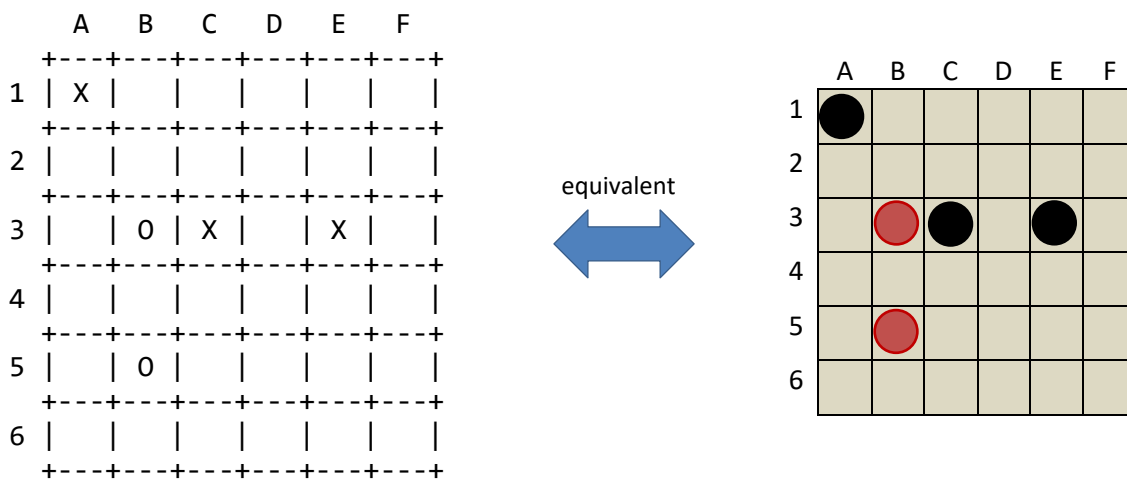


Figure 6: An example game board in the console

In the beginning, all cells on the board are empty. Pieces of player 1 (Black) and player 2 (Red) are denoted by 'X' and 'O' symbols, respectively.

Program Flow

1. The program starts the game with an empty board. Player 1 (X) takes the first turn.
2. Then, prompt the current player to enter a cell address to put a piece onto the board.
3. A piece cannot be placed onto the board if the target cell is not empty or if the position is outside the board's boundaries. In case it is not placeable, display a warning message "Invalid move!" and go back to Step 2.
4. Update the board by putting the input piece to the target position and repelling existing adjacent pieces away if there exists open space. (Recycle any off-the-board pieces to their owning players)
5. Swap player to take the next turn.

6. Repeat steps 2–5 until a player wins (forming a line of L pieces or having all P pieces put on the board). If both players satisfy either of the winning conditions concurrently, the moving player (i.e., the one taking the current turn) is regarded the winner.
7. When a game finishes, display the message “Player X wins!” or “Player O wins!” accordingly.

Assumptions

- The board is always a square, i.e., number of rows = number of columns.
- There are three basic configuration parameters for this game:
 - N = 6 is the board size (N^2 = number of cells);
 - P = 8 is the number of pieces per player;
 - L = 3 is the number of pieces required to form a line.
- To check if your program is scalable on these parameters instead of hardcoding, we may have a few test cases that vary their values although this may affect the original game rules. To support our testing, the constructor of your Gekitai class must follow this method signature:

```
__init__(self, N: int = 6, P: int = 8, L: int = 3)
```

Then we can create game objects for different test cases such as:

```
g1 = Gekitai()  
g2 = Gekitai(7, 9, 4)  
g3 = Gekitai(P=6)
```

- You may assume the range for N is between 6 and 8, P between 6 and 12, and L between 3 and the minimum of N and P. Validation on these parameters is not strictly necessary as we will not test your program with values that are out of range, but you are welcome to add the following assertion when constructing the Gekitai object:

```
assert 6 <= N <= 8 and 6 <= P <= 12 and 3 <= L <= min(N, P)
```

- We assume that cell address inputs always follow the Excel-like format of one letter (A-Z or a-z) plus one integer (1-26). Uppercase or lowercase inputs like "A1", "c6", or even having some spaces in the string like " b 5 " will be accepted as normal. Use exception handling techniques to avoid program crash due to weird inputs like "AA1", "A01", "abc", "A3.14", "#a2", ... for cell addresses, which may raise ValueError when getting the user input.
- There is no draw game based on the official game rules. Due to the repelling effect, it is possible that a move may cause both players to form a line of L pieces at the same time. In this case, the official game rules regard the moving player as the winner. The same applies to the case that the moving player putting his/her last piece on the board but causing the opponent to form a line.

OO Design for this Game

You must use OOP to develop this program. Please take reference of the class diagram below which is our suggested OO design for this problem.

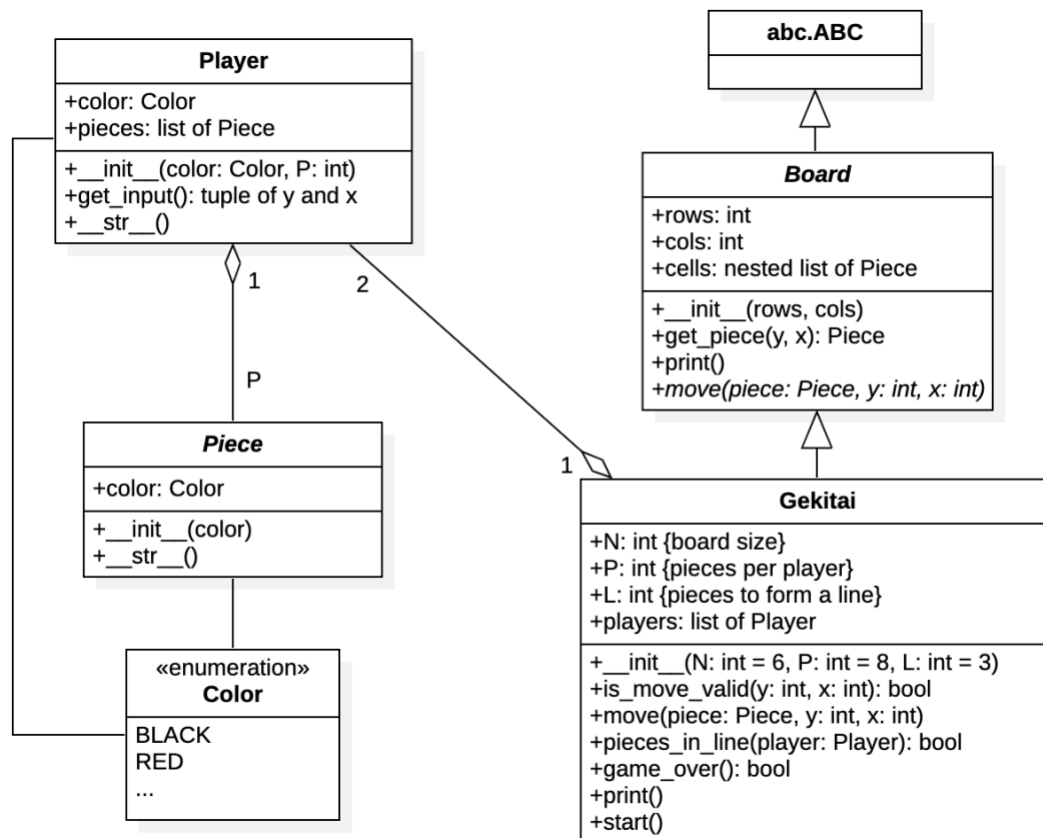


Figure 7: Class diagram for the game program design

Note: For the operations or methods in the diagram, for simplicity, we omit the `self` parameter which is Python-specific and not quite meaningful in the class diagram.

This assignment aims at testing your implementation skills rather than your OO design skills. Therefore, whenever possible, just follow our given design in your submitted code (while you are encouraged to try other designs in your own separate code versions that need not be submitted). Having that said, we allow some leeway to differ from this suggested design as long as your program generates the correct output. In fact, the above diagram is only a baseline design and some details (at attribute or operation level) are omitted. **During your coding, please feel free to add new attributes or methods when you see it fit. For example, you may want to define extra attributes for keeping the current turn of the game, recording who has won the game, etc.**

Some points to assist your understanding of the class diagram and coding:

- Arrow means *inheritance*, e.g., Gekitai is a subclass of Board.
- Diamond means *aggregation*, in simple terms, the class on the end with the diamond consists of the class on the other end, e.g., a player holds P pieces. In OO terms, one Player object aggregates P Piece objects (in simple words, one player consists of P pieces).
- Gekitai is a two-player board game. So, one Gekitai object aggregates 2 Player objects.
- In coding, we usually use an array or list to realize an aggregation relationship, e.g., for the Player class, define an instance variable which is a list of P Piece objects.
- A simple line without any shape means composition (A has B). For example, a player has a color; a piece also has a color. So, the Player class has a line to the Color class. The Piece class also has a line to the Color class. To implement this relationship, it is just as simple as defining an instance variable in Player class and in Piece class respectively, which is of the Color type.

- The *Board* class is shown in italic in the diagram. This notation means that the class is *abstract*. It is an abstract class (or ABC) as it contains an abstract method *move()*, again in italic, whose implementation is not yet known until we know what game the board represents. The *Gekitai* class (representing a specific board game) should override this abstract method by its own version of how to move a piece to reflect its game rules, e.g., repelling adjacent pieces.
- While the *Board* class is abstract, it does contain some methods that are concrete and reusable. For example, the method to print the board and the method to get a piece from a target position (y, x), which means row y column x on the board, are common for most board games. We prefer implementing these methods at the parent class level for better generalization and code reuse.

Color Enum Class

Pieces and players should have a color attribute for identification purpose. Regarding this, a simple way can be adding an integer 0 or 1 to the objects to distinguish black from red. But we recommend using an *enumeration* class to define a new type for color. Check out the [enum](#) module in Python. For example, you can define the following *Color* class by extending *Enum*:

```
from enum import Enum
class Color(Enum):
    BLACK = 'X'
    RED = 'O'
```

```
c = Color.RED
```

Then you can get the name string 'RED' via `c.name` and the value string 'O' via `c.value`. Using *enum* may ease your implementation. But if you feel it is a new concept and is hard for you, you can skip using *enum* at all and use your own way to achieve the same effect of giving a player or piece a color attribute.

Piece Class

Attributes:

- **color**: the color identifying which player this piece belongs to

Methods:

This class may have no methods. But depending on your implementation choice, you may choose to override the `__str__()` method to print the *Piece* object as some more meaningful string like the 'X' or 'O' symbol than the object's type and memory address. (Doing so or not is optional.)

Player Class

Attributes:

- **color**: the color identifying the player
- **pieces**: a list of *Piece* objects which represent game pieces that are with the player, i.e., pending to be put onto the game board.

Methods:

get_input(self) -> (int, int)

Get the input cell address (Excel-format) from the user (current player) and return a tuple (y, x) where y and x (zero-based indexes) refer to the target cell at row y and column x.

Again, depending on your implementation choice, you may override the `__str__()` method to print the Player object as some string reflecting the 'X' or 'O' symbols it holds if this brings convenience.

Board Class

Attributes:

- `cells`: a nested list (2D array) implementing the game board. Each element (representing a cell) is storing either a Piece object or something such as `None` that represents an empty cell.
- `rows`: the number of rows for the cells array
- `cols`: the number of columns for the cells array

Methods:

`get_piece(self, y, x) -> Piece`

Return the element `cells[y][x]` which is either a Piece object or something like `None` that represents an empty cell. (Note: think about how to handle cases that `y` and `x` are beyond the dimensions of the cells array.)

`print(self)`

Print the game board in the format as shown in Figure 6.

`move(self, piece: Piece, y: int, x: int)`

(An abstract method that is being overridden by a subclass)

Gekitai Class

It is a subclass of the Board abstract class.

Attributes:

- `N`: board size
- `P`: the total number of pieces given to each player
- `L`: the number of pieces required to form a line
- `players`: a list or tuple of the two Player objects

Methods:

`is_move_valid(self, y: int, x: int) -> bool`

Return `True` if it is valid to move a piece onto the board at row `y` and column `x` (zero-based indexes), i.e., if the element `cells[y][x]` is an empty cell. Otherwise, it returns `False`. (Note that depending on your design, if `x` and `y` are beyond the board's boundaries, it can also return `False`.)

`move(self, piece: Piece, y: int, x: int)`

This function carries out updates of the `cells` array to actualize the effects of moving a piece of the current player into the cell at row `y` and column `x` (zero-based indexes). This includes setting the element `cells[y][x]` to `piece` and moving its adjacent pieces (the 8 possible cells surrounding `cells[y][x]`) outward by one square if there is open space for them to move into.

`pieces_in_line(self, player: Player) -> bool`

Return `True` if it can find `L` consecutive pieces in a line on the board that belong to the specified player, and `False` otherwise. Calling this method will be useful to assist determining whether the game is over. When it returns `True`, that means the specified player has won.

`game_over(self) -> bool`

Return `True` if it finds any player has satisfied any of the winning conditions, i.e., L pieces in one line (horizontal, vertical or diagonal) or P pieces arrived on the board.

`print(self)`

This overrides Board's `print()` method. Besides printing the game board, it also prints the list of pieces that are still with each player, i.e., those that are not yet on the board. (See our provided sample output to know more).

`start(self)`

This method implements the main loop of the game, prompting players to make moves in turns by calling the `get_input()` method of each Player object. It calls `is_move_valid()` to check if the input position is valid, and calls `move()` if valid, and removes one piece from the current player's list of pieces, which has been placed onto the board. This method also calls `game_over()` to check if the game-over condition has been reached or not.

Final Remarks

- You should avoid using global variables whenever possible (except for those that are read-only).
- Your program should include suitable comments as documentation. Each class must have a “doc string” to describe what it is (our sample program has purposely omitted them so that you cannot copy and need to write them up in your own words). You are also highly encouraged to do the same at method level. Missing class-level doc strings will invite some mark deduction.

Sample Runs

```
  A  B  C  D  E  F
+---+---+---+---+---+
1 |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |
+---+---+---+---+---+
6 |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player X's turn: H3
Invalid move!
Player X's turn: A-1
Invalid move!
Player X's turn: @# \ 789
Invalid move!
Player X's turn: A 1
```



```

      A B C D E F
+---+---+---+---+---+
1 | X |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |   |
+---+---+---+---+---+
6 |   |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player O's turn: a6

```

```

      A B C D E F
+---+---+---+---+---+
1 | X |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |   |
+---+---+---+---+---+
6 | O |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player X's turn: a6

```

Invalid move!

Player X's turn: c3

```

      A B C D E F
+---+---+---+---+---+
1 | X |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   | X |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |   |
+---+---+---+---+---+
6 | O |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player O's turn: d4

```

Player O's turn: d4

```

      A B C D E F
+---+---+---+---+---+
1 | X |   |   |   |   |   |
+---+---+---+---+---+
2 |   | X |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   | O |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |   |
+---+---+---+---+---+
6 | O |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player X's turn: c3

```

```

Game over:
  A  B  C  D  E  F
+---+---+---+---+---+
1 | X |   |   |   |   |
+---+---+---+---+---+
2 |   | X |   |   |   |
+---+---+---+---+---+
3 |   |   | X |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   | O |
+---+---+---+---+---+
6 | O |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O']
Player X wins!

```

Another example run in which both players formed a line of 3 pieces simultaneously:

```

  A  B  C  D  E  F
+---+---+---+---+---+
1 |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |
+---+---+---+---+---+
6 |   |   |   |   |   |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player X's turn: e6
  A  B  C  D  E  F
+---+---+---+---+---+
1 |   |   |   |   |   |
+---+---+---+---+---+
2 |   |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |
+---+---+---+---+---+
6 |   |   |   |   | X |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
Player O's turn: a2
  A  B  C  D  E  F
+---+---+---+---+---+
1 |   |   |   |   |   |
+---+---+---+---+---+
2 | O |   |   |   |   |
+---+---+---+---+---+
3 |   |   |   |   |   |
+---+---+---+---+---+
4 |   |   |   |   |   |
+---+---+---+---+---+
5 |   |   |   |   |   |
+---+---+---+---+---+
6 |   |   |   |   | X |
+---+---+---+---+---+
X: ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']

```

O: ['O', 'O', 'O', 'O', 'O', 'O', 'O']

Player X's turn: [e4](#)

	A	B	C	D	E	F
1						
2		O				
3						
4					X	
5						
6					X	

X: ['X', 'X', 'X', 'X', 'X', 'X']

O: ['O', 'O', 'O', 'O', 'O', 'O', 'O']

Player O's turn: [c2](#)

	A	B	C	D	E	F
1						
2		O		O		
3						
4					X	
5						
6					X	

X: ['X', 'X', 'X', 'X', 'X', 'X']

O: ['O', 'O', 'O', 'O', 'O', 'O', 'O']

Player X's turn: [e3](#)

	A	B	C	D	E	F
1						
2		O		O		
3					X	
4						
5					X	
6					X	

X: ['X', 'X', 'X', 'X', 'X']

O: ['O', 'O', 'O', 'O', 'O', 'O', 'O']

Player O's turn: [d2](#)

	A	B	C	D	E	F
1						
2		O	O		O	
3						
4						X
5					X	
6					X	

X: ['X', 'X', 'X', 'X', 'X']

O: ['O', 'O', 'O', 'O', 'O']

Player X's turn: [c5](#)

```

      A B C D E F
    +---+---+---+---+---+
  1 |   |   |   |   |   |
    +---+---+---+---+---+
  2 | 0 | 0 |   | 0 |   |
    +---+---+---+---+---+
  3 |   |   |   |   |   |
    +---+---+---+---+---+
  4 |   |   |   |   |   | X |
    +---+---+---+---+---+
  5 |   |   | X |   | X |   |
    +---+---+---+---+---+
  6 |   |   |   |   | X |   |
    +---+---+---+---+---+
X: ['X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O', 'O']
Player O's turn: e1

```

```

      A B C D E F
    +---+---+---+---+---+
  1 |   |   |   |   | 0 |   |
    +---+---+---+---+---+
  2 | 0 | 0 |   |   |   |   |
    +---+---+---+---+---+
  3 |   |   | 0 |   |   |   |
    +---+---+---+---+---+
  4 |   |   |   |   |   | X |
    +---+---+---+---+---+
  5 |   |   | X |   | X |   |
    +---+---+---+---+---+
  6 |   |   |   |   | X |   |
    +---+---+---+---+---+
X: ['X', 'X', 'X', 'X']
O: ['O', 'O', 'O', 'O']
Player X's turn: b6

```

```

      A B C D E F
    +---+---+---+---+---+
  1 |   |   |   |   | 0 |   |
    +---+---+---+---+---+
  2 | 0 | 0 |   |   |   |   |
    +---+---+---+---+---+
  3 |   |   | 0 |   |   |   |
    +---+---+---+---+---+
  4 |   |   |   | X |   | X |
    +---+---+---+---+---+
  5 |   |   |   |   | X |   |
    +---+---+---+---+---+
  6 |   | X |   |   | X |   |
    +---+---+---+---+---+
X: ['X', 'X', 'X']
O: ['O', 'O', 'O', 'O']
Player O's turn: c4

```

```

Game over:
      A B C D E F
    +---+---+---+---+---+
  1 |   |   |   |   | 0 |   |
    +---+---+---+---+---+
  2 | 0 | 0 | 0 |   |   |   |
    +---+---+---+---+---+
  3 |   |   |   |   |   |   |
    +---+---+---+---+---+
  4 |   |   | 0 |   | X | X |
    +---+---+---+---+---+
  5 |   |   |   |   | X |   |
    +---+---+---+---+---+
  6 |   | X |   |   | X |   |
    +---+---+---+---+---+
X: ['X', 'X', 'X']
O: ['O', 'O', 'O']
Player O wins!

```

Sample Executable

For more sample runs, you can execute our provided modules in binary format:

- `gekitai.so` (for macOS)
- `gekitai.pyd` (for Windows)

Put the corresponding binary file in your current directory. Start a Python shell and type the following to start a game using the default configuration for N, P and L.

```
from gekitai import Gekitai
game = Gekitai()
game.start()
```

If you want to customize any of N, P and L, it is just a matter of passing valid arguments when constructing the game object. For example,

```
from gekitai import Gekitai
game = Gekitai(7, 9, 4)
game.start()
```