**REPORTE FINAL BOT – “NACHO”**

Luis Ángel Reyes Frausto

Samuel Iván Sánchez Salazar

**Introducción.**

Para el desarrollo de este bot decidimos implementar el algoritmo minimax ya que es ideal para este tipo de juegos y ya teníamos conocimiento de él por haber sido nuestro tema de exposición. Las distintas implementaciones del bot pueden ser consultadas en el github: <https://github.com/Tankel/Tic-Tac-Chec-Bot>, donde la versión definitiva de la que estaremos hablando es la de playerNacho.py

**Evolución del bot utilizando minimax y otras estrategias.**

Antes de implementar el minimax, consideramos que algo crucial del bot, sería el orden y posición en que pondríamos las primeras piezas, así que en los primeros 4 movimientos tratamos de alinear las piezas en horizontalente en la fila de abajo y si esto no era posible la poníamos de manera random. Haciendo esto notamos que le llegaba a ganar hasta el 70% de las partidas al bot random.

Al implementar el minimax, el bot llegaba a ganar hasta el 90% de las partidas al random, al implementar como heursitica claramente el si alguno ganaba (+16 si yo ganaba, -16 si el oponente ganaba), y si ninguno se cumplía calculamos: número máximo de piezas alineadas – número máximo de piezas alineadas del oponente, de tal manera que beneficia al que tenga más.

Después notamos que, si el random por pura suerte empezaba la partida y trataba también de alinear en los primeros 4 movimientos, nuestro bot no hacía nada por bloquearlo, al parecer es efectivo para tratar de alinear las 4 nuestras pero no al evitar que el oponente alinee las otras 4. Es ahí que notamos que la función minimax no funciona en cuanto a profundidad, siendo esta “1”, funciona muy rápido y correctamente, pero al aumentarla no presenta ningún cambio e incluso llega a perder más partidas. Este error no pudimos resolverlo, pero aun así la profundidad de 1 resulta muy efectiva.

Por este problema decidimos implementar otro método llamado blockOpponent el cual es llamado cuando el oponente ya tiene 3 piezas alineadas y yo no, el cual primero coloca una de mis piezas (si es que no tiene en el tablero) en la celda faltante del oponente, sino checa mis piezas en el tablero y se posiciona en cualquier lugar de línea (comiendo una pieza enemiga o poniéndose en la celda vacía).

Otro problema común que presentamos comúnmente es que después de todos estos intentos de movimientos, se me retornaba un tablero vacío (ya que no existía un mejor movimiento dada la heurística), por lo que, si pasa esto, el bot hará un movimiento random.

Finalmente experimentamos un poco con el orden en que ponemos las piezas iniciales (alineadas horizontalmente) peleando unas alineaciones contra otras, donde no había mucha variación, pero nos quedamos con el siguiente orden: Bishop, Pawn, Rook, Knight.

Al final de todo este recorrido, nuestro bot fue capaz de ganarle aproximadamente el 98% de las partidas al bot random.

**Documentación.**

Atributos**.**

La clase cuenta en general con los siguiente atributos inicializados y distribuidos en las clases \_\_init\_\_ y setColor:

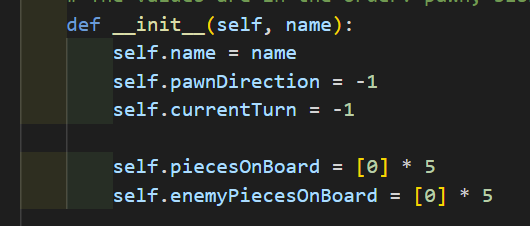
* **name**: El nombre del jugador.
* **pawnDirection**: La dirección de movimiento del peón del jugador (-1 para ir hacia adelante, 1 para ir hacia atrás).
* **currentTurn**: El turno actual del jugador.
* **piecesOnBoard**: Una lista que indica qué piezas del jugador están en el tablero.
* **enemyPiecesOnBoard**: Una lista que indica qué piezas del oponente están en el tablero.
* **piecesCode**: Una lista que contiene los códigos de valor de las piezas del jugador de tal manera que al pasarle una pieza del jugador como índice, devuelve su verdadero color.

Librerías: copy, random, time.

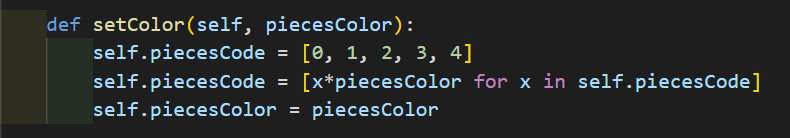
Funciones.

La clase TTCPlayer del bot random nos resultó muy útil usándolo de referencia, por lo que los métodos setColor, updatePawnDirection, sameSign, isInsideBoard, wasPieceMovement, getPawnValidMovements, getBishopValidMovements, getKnightValidMovements, getRookValidMovements y getValidMovements son exactamente los mismos.

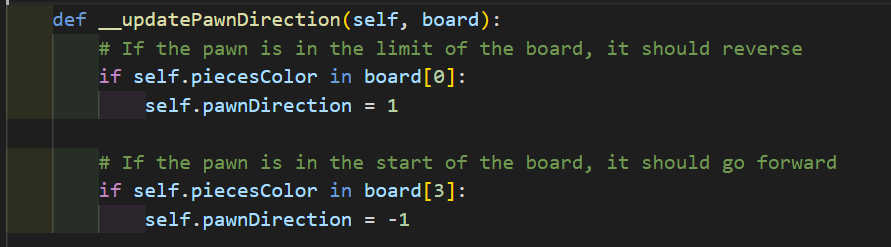
* **\_\_init\_\_(self, name**): inicializa los atributos de la clase.



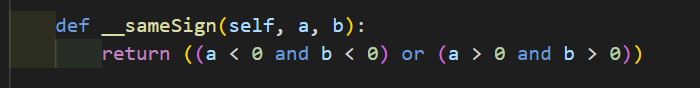
* **setColor(self, piecesColor):** establece el color de las piezas del jugador (-1 para negro, 1 para blanco).



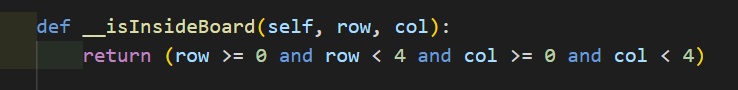
* **\_\_updatePawnDirection(self, board):** actualiza la dirección del peón del jugador en función de su posición en el tablero.



* **\_\_sameSign(self, a, b):** verifica si dos piezas tienen el mismo color.



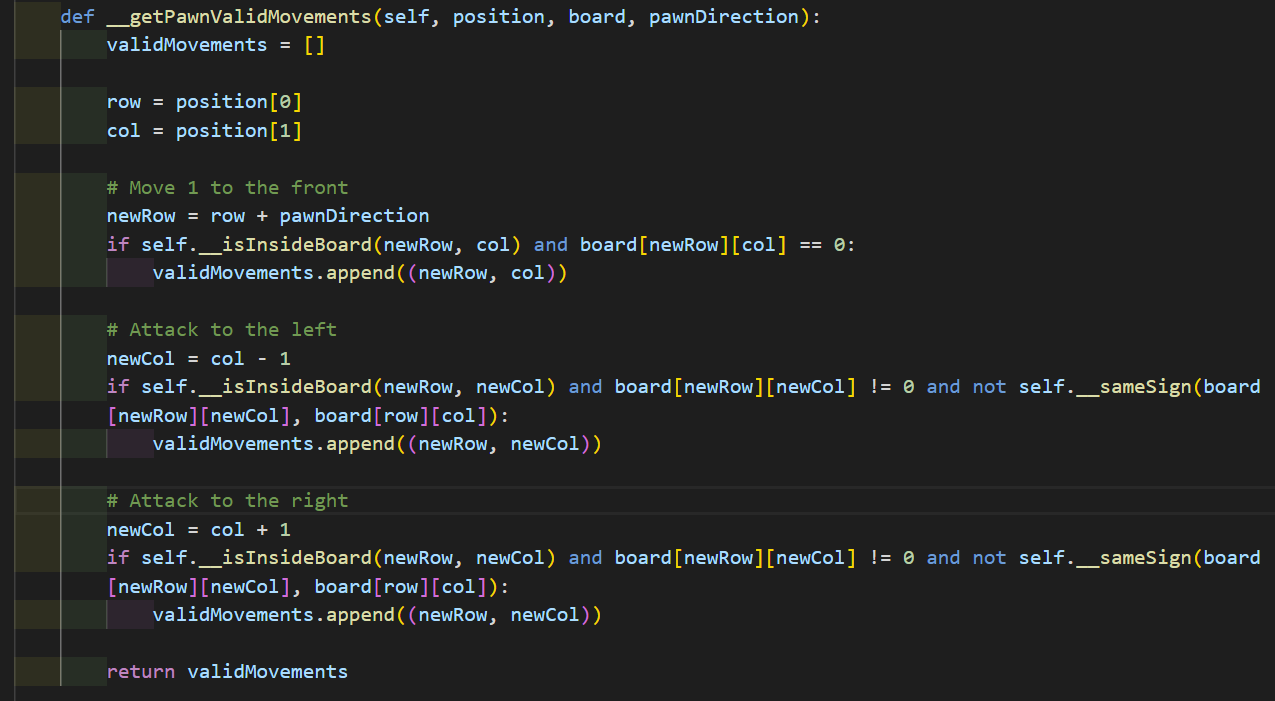
* **\_\_isInsideBoard(self, row, col):** verifica si una posición está dentro del tablero.



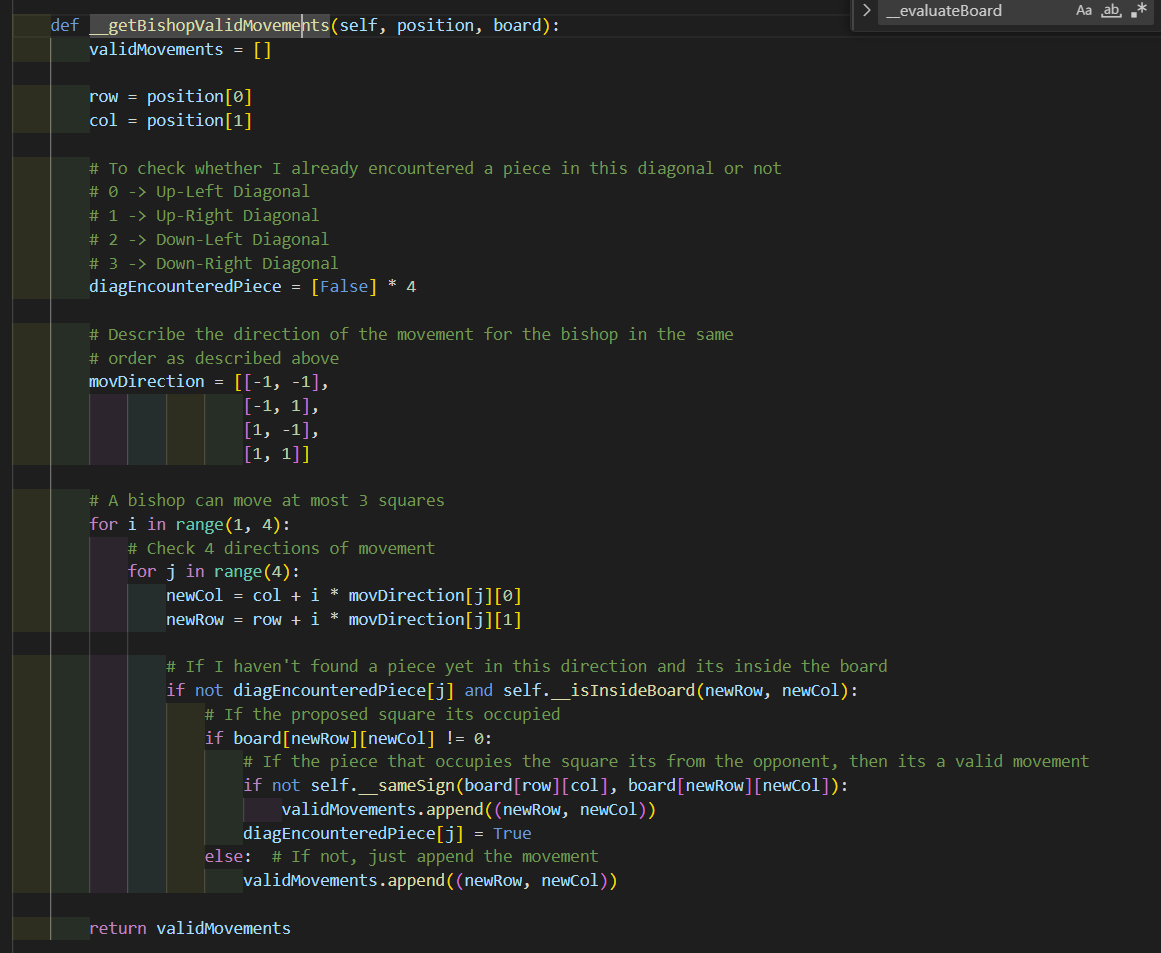
* **\_\_wasPieceMovement(self, oldBoard, newBoard):** verifica si un movimiento fue un movimiento válido o una captura.



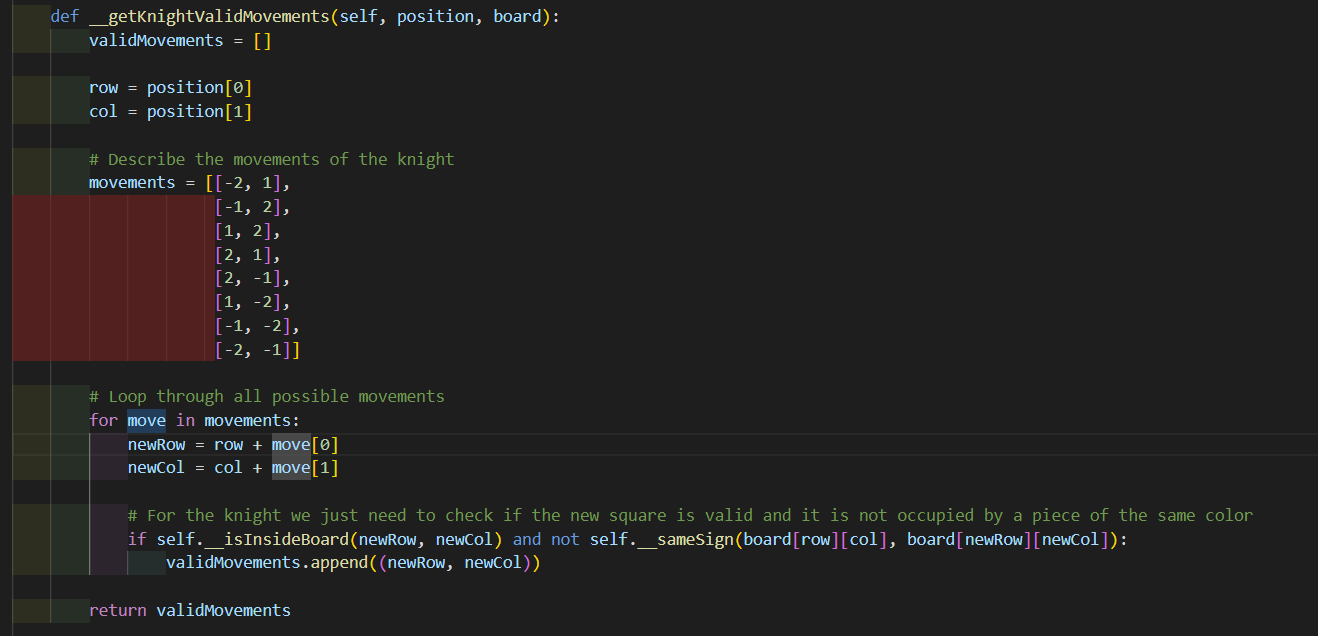
* **\_\_getPawnValidMovements(self, position, board, pawnDirection):** da los movimientos válidos para un peón en una posición dada.



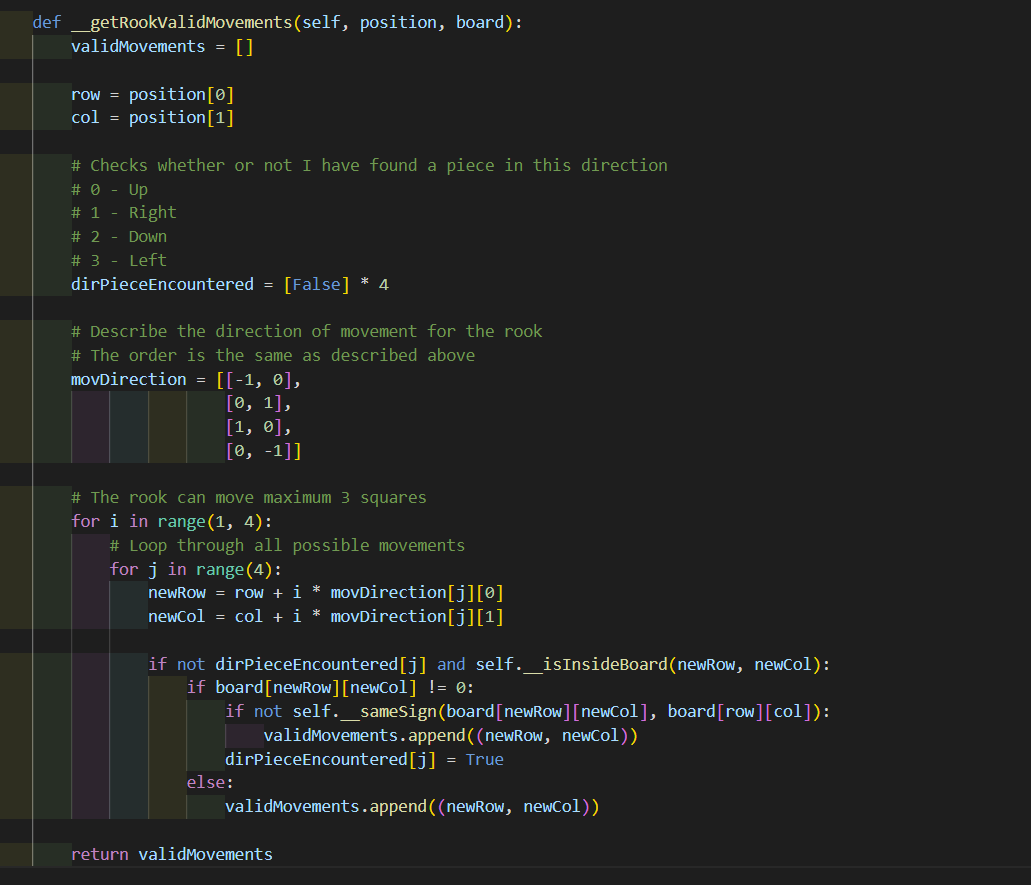
* **\_\_getBishopValidMovements(self, position, board):** da los movimientos válidos para un alfil en una posición dada.



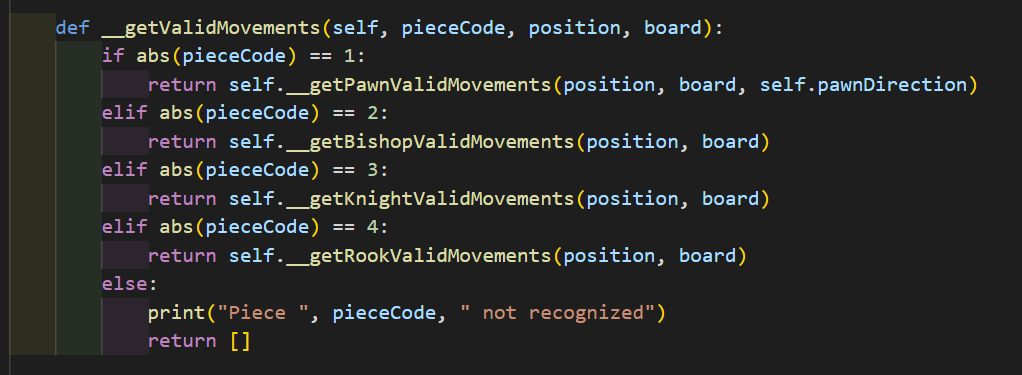
* **\_\_getKnightValidMovements(self, position, board):** da los movimientos válidos para un caballo en una posición dada.



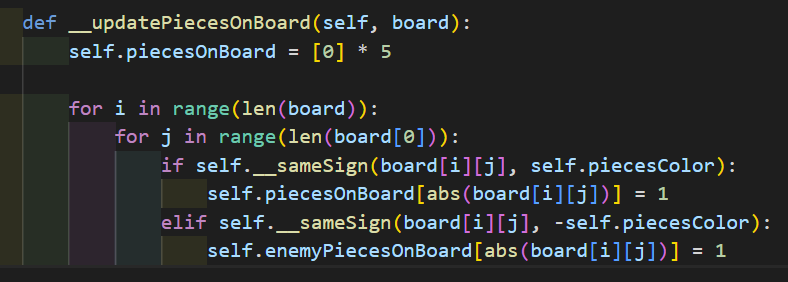
* **\_\_getRookValidMovements(self, position, board):** da los movimientos válidos para una torre en una posición dada.



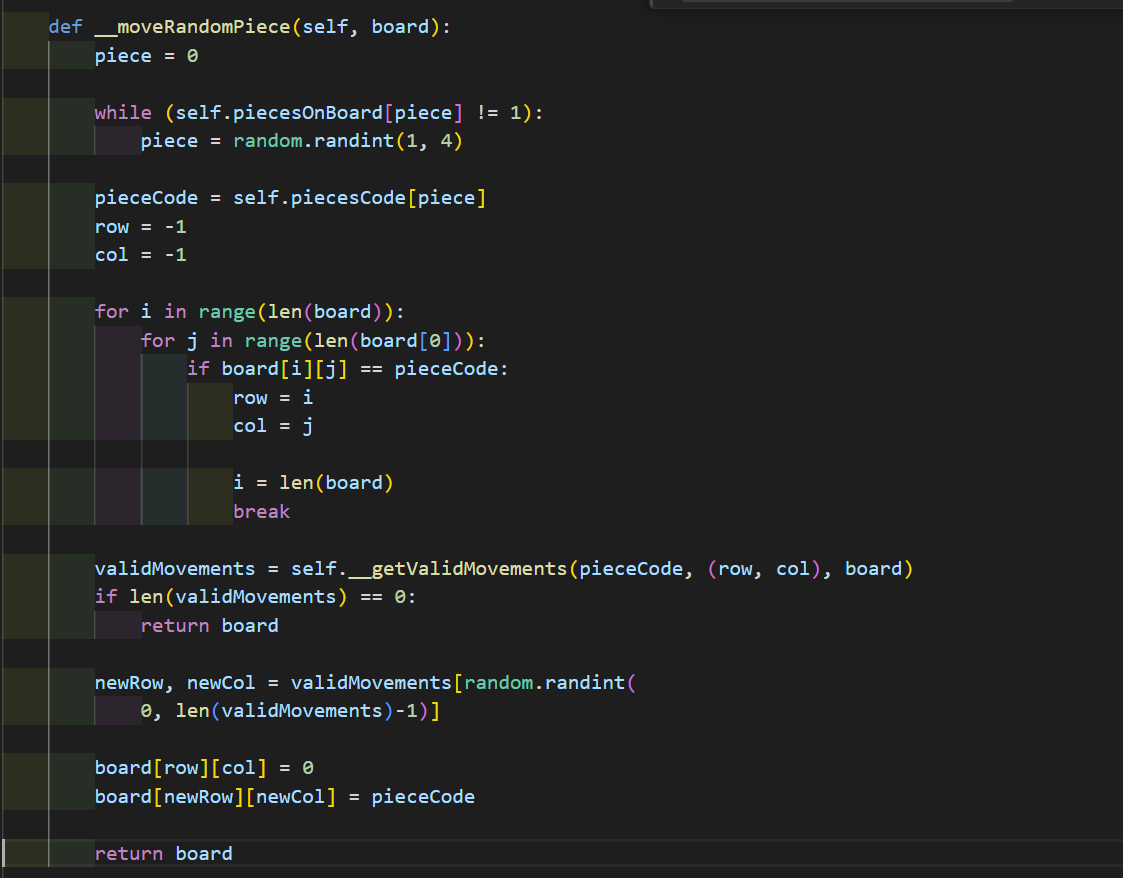
* **\_\_getValidMovements(self, pieceCode, position, board):** da los movimientos válidos para una pieza en una posición dada.



* **\_\_updatePiecesOnBoard(self, board):** actualiza las lista de piezas de ambos jugadores de acuerdo al tablero.

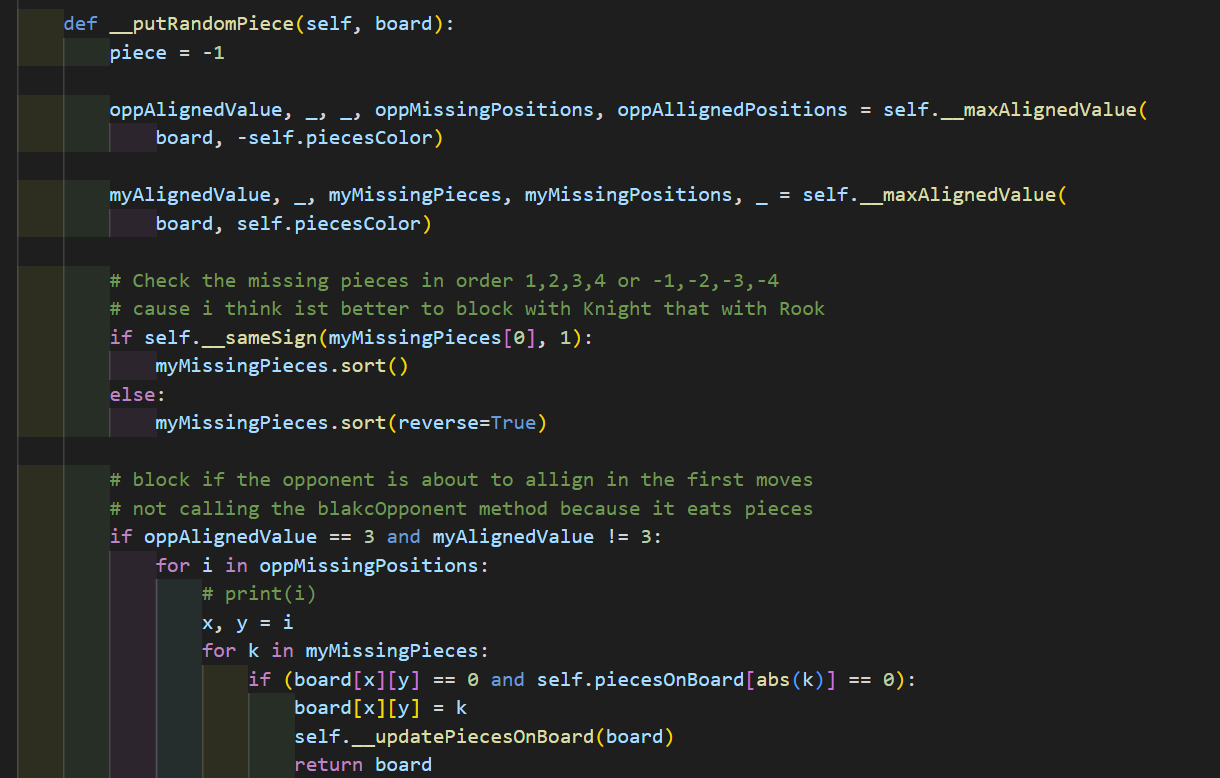


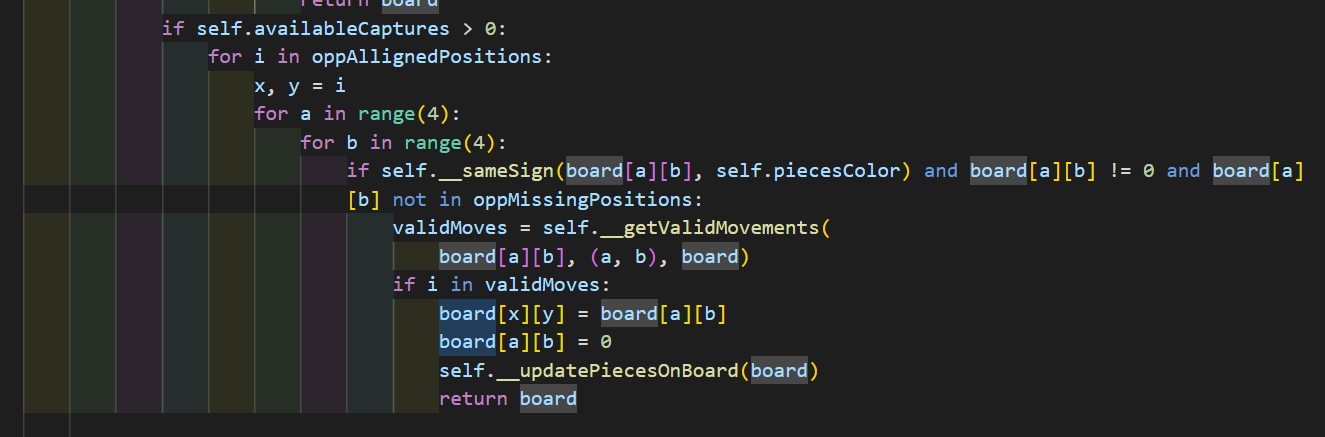
* **\_\_moveRandomPiece(self, board):** mueve una pieza aleatoria del jugador en el tablero.



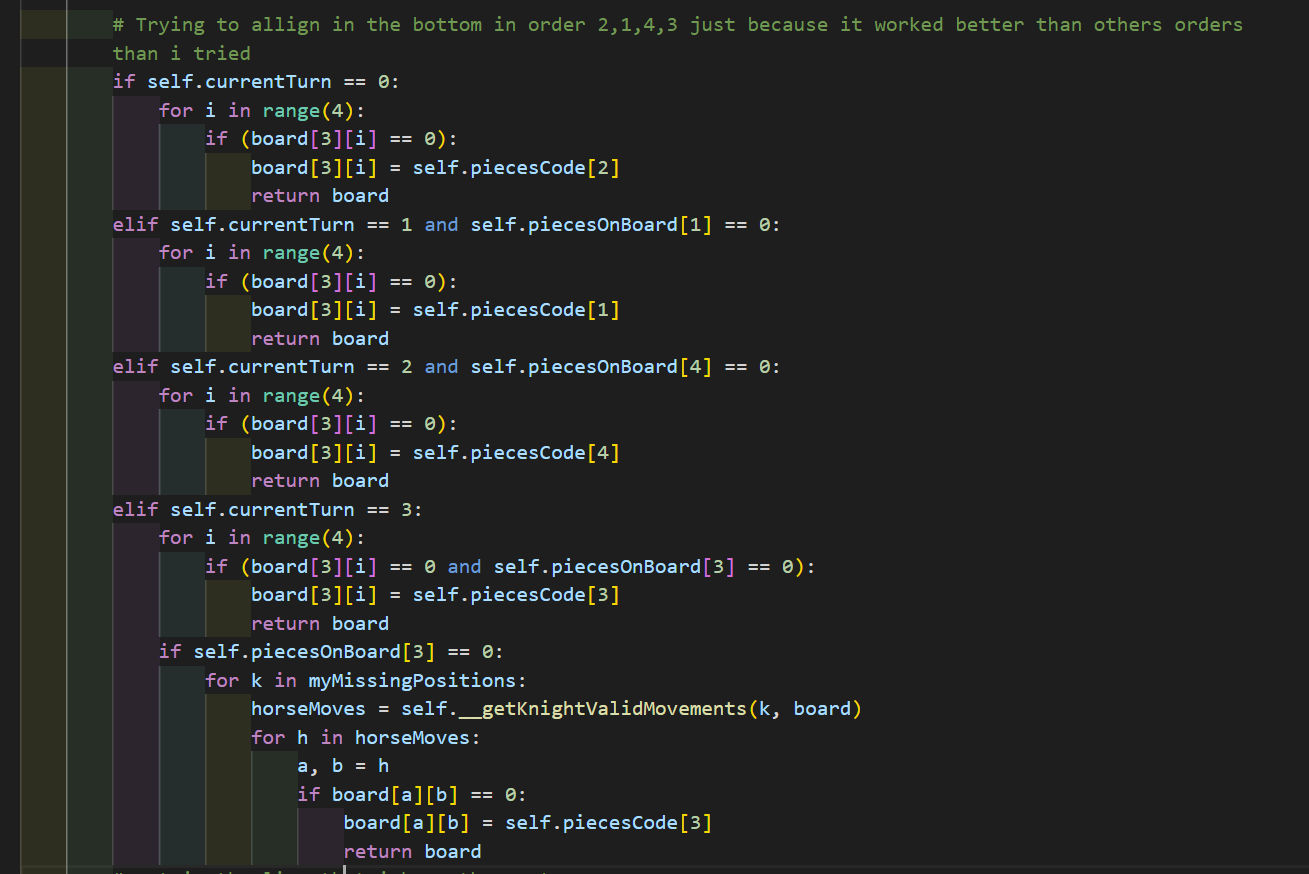
* **\_\_putRandomPiece(self, board):**

Primero bloquea al oponente poniendo una pieza en su casilla faltante si está a punto de ganar.

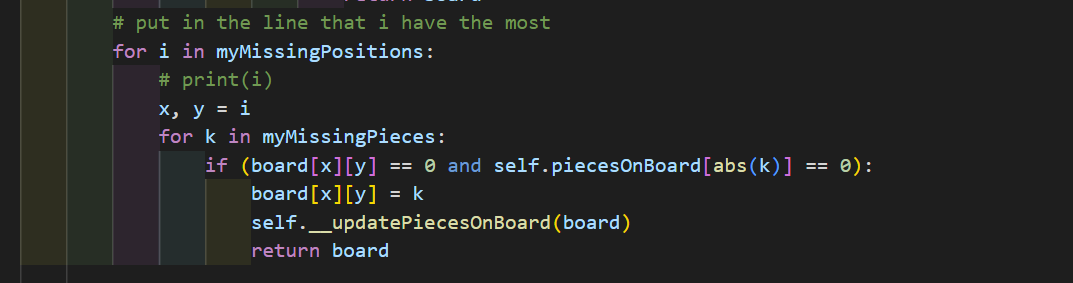




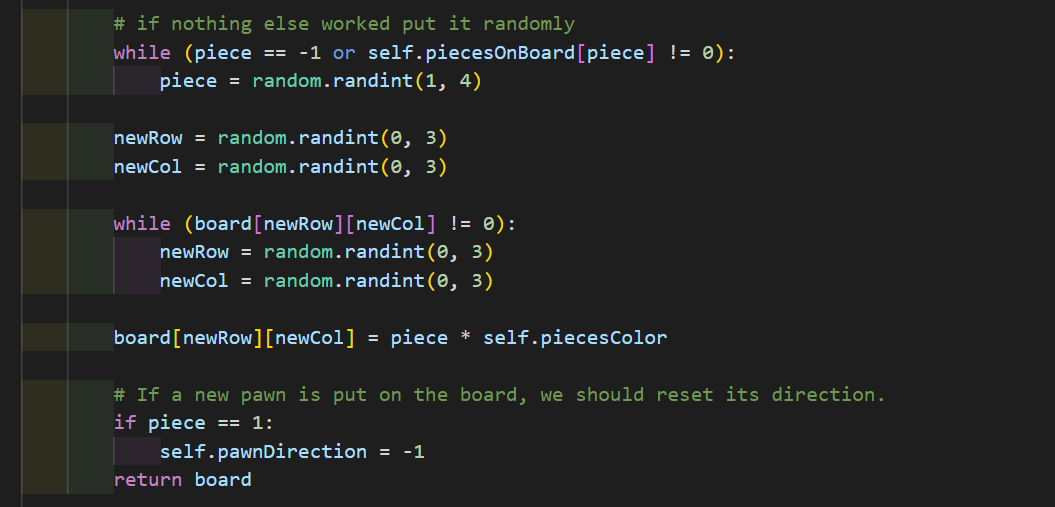
En los primeros 4 turnos coloca las piezas conforme a la estrategia descrita.



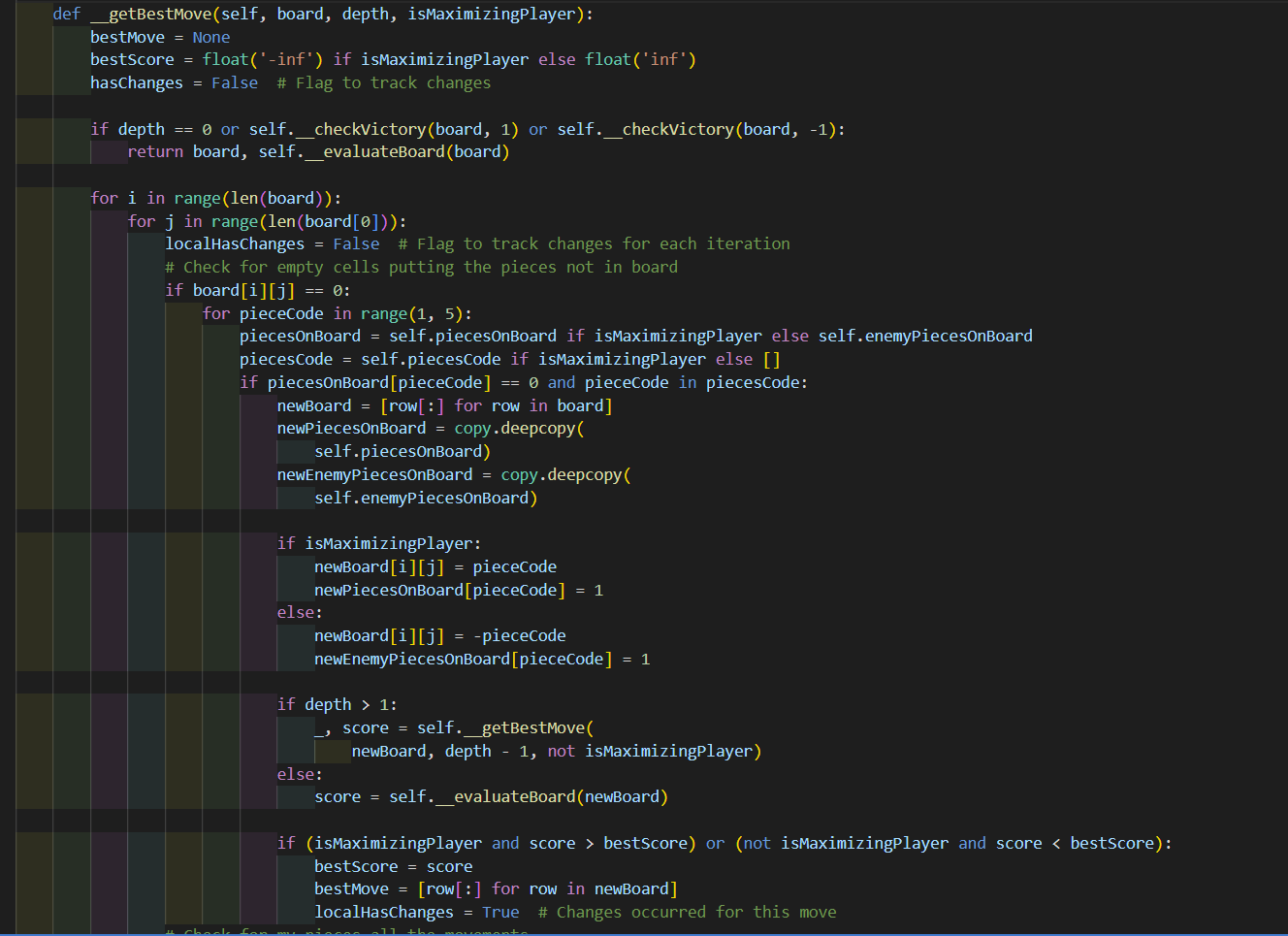
En los demás turnos pone una pieza en la línea que tiene más piezas.

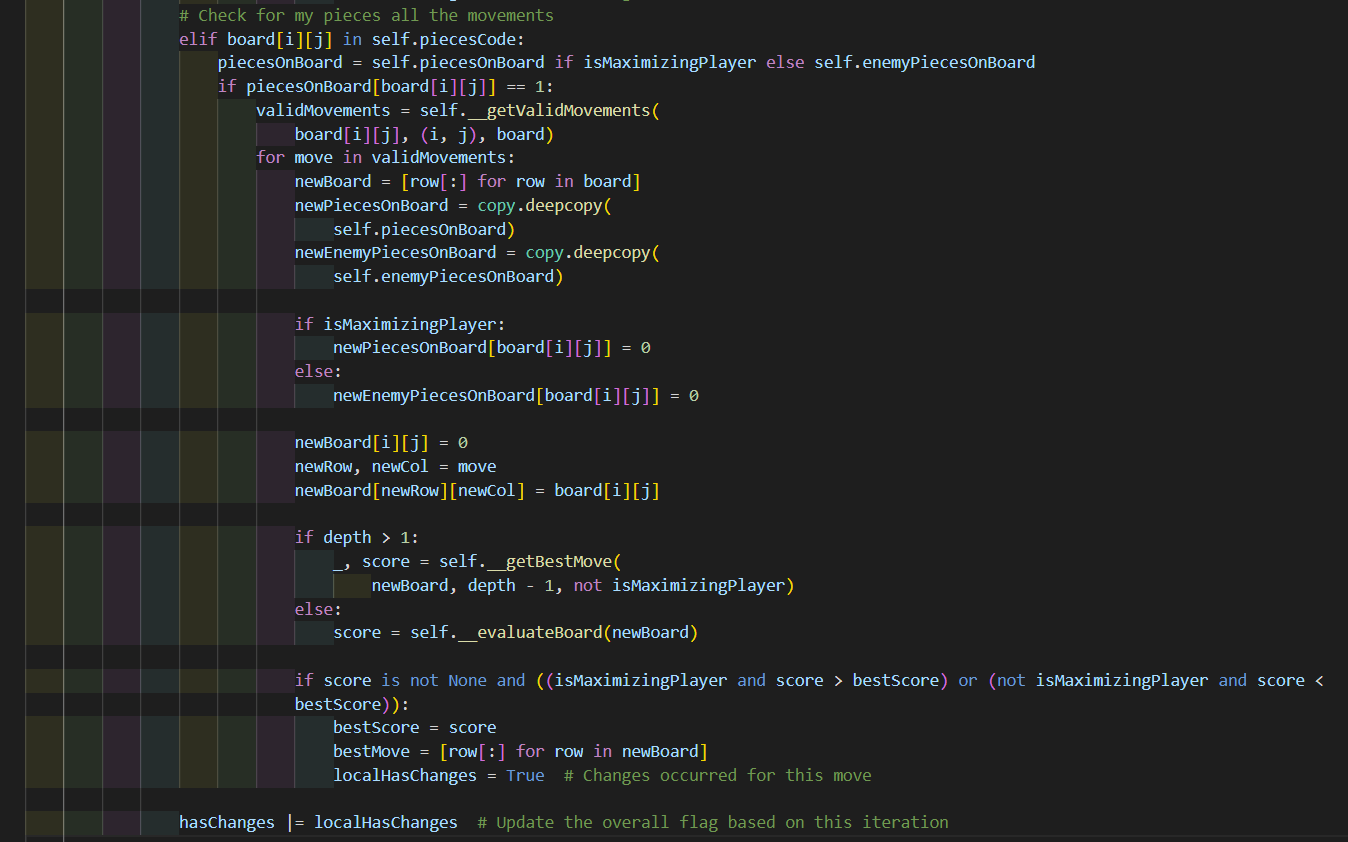


Y si nada de esto funciona, pone una pieza en una posición aleatoria disponible.



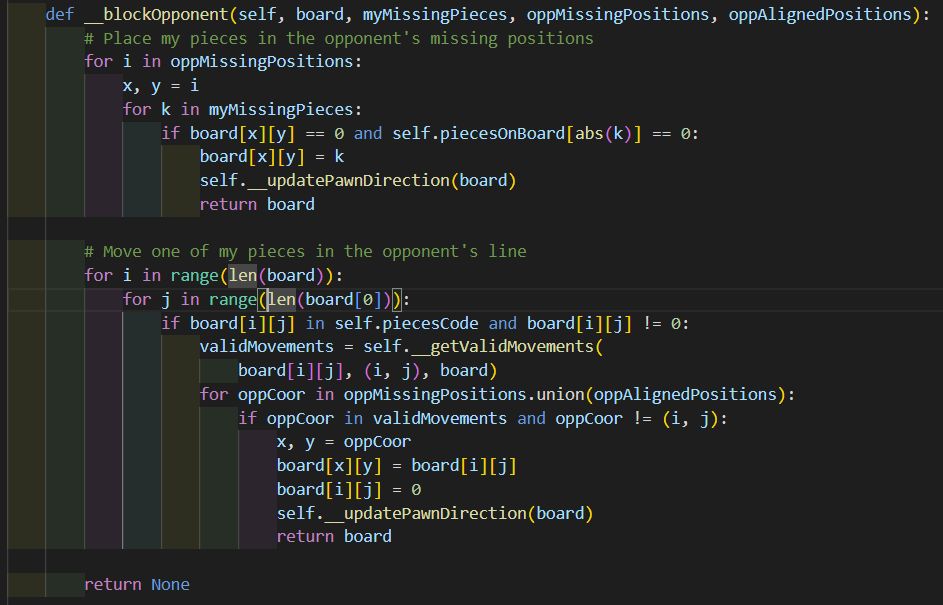
* **\_\_getBestMove(self, board, depth, isMaximizingPlayer):** implementación del algoritmo minimax que hace el mejor movimiento según la heurística de \_\_evaluateBoard.



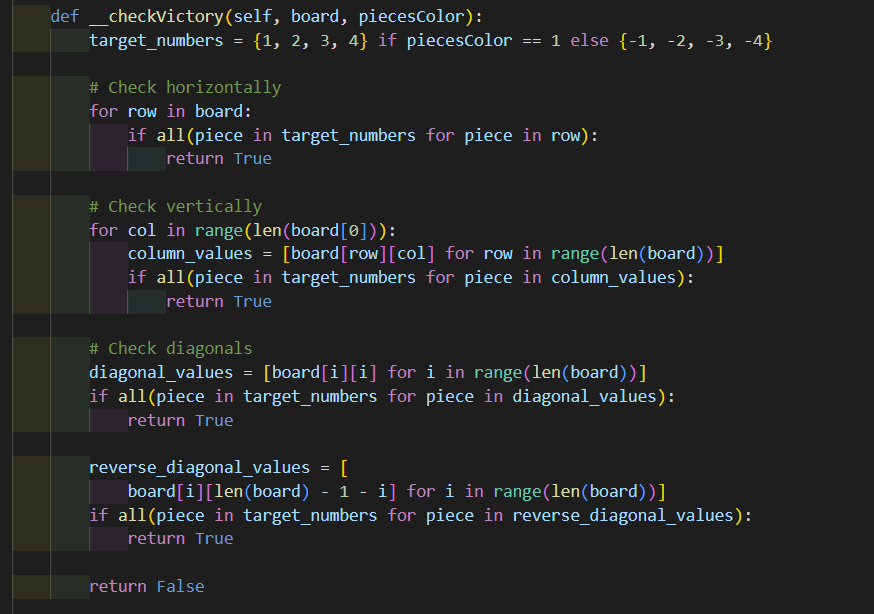




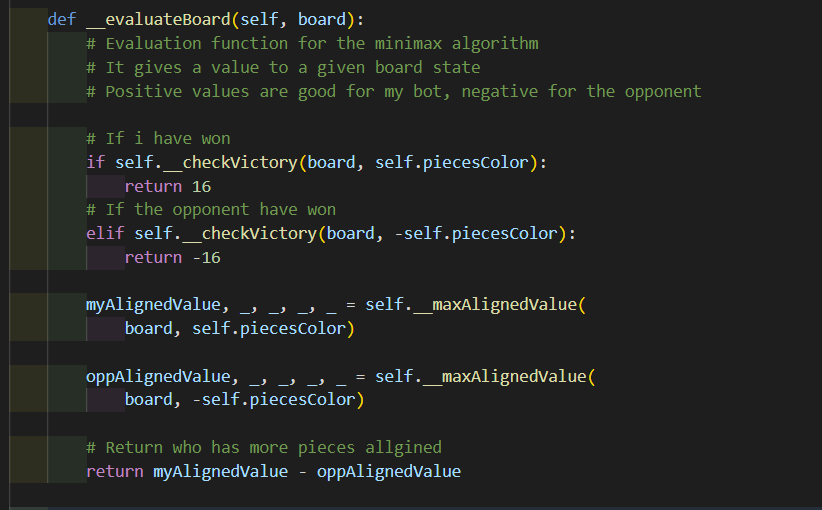
* **\_\_blockOpponent(self, board, myMissingPieces, oppMissingPositions, oppAlignedPositions):** evita que el oponente gane poniendo una de las piezas en la casilla faltante del oponente o comiendo una de sus piezas.



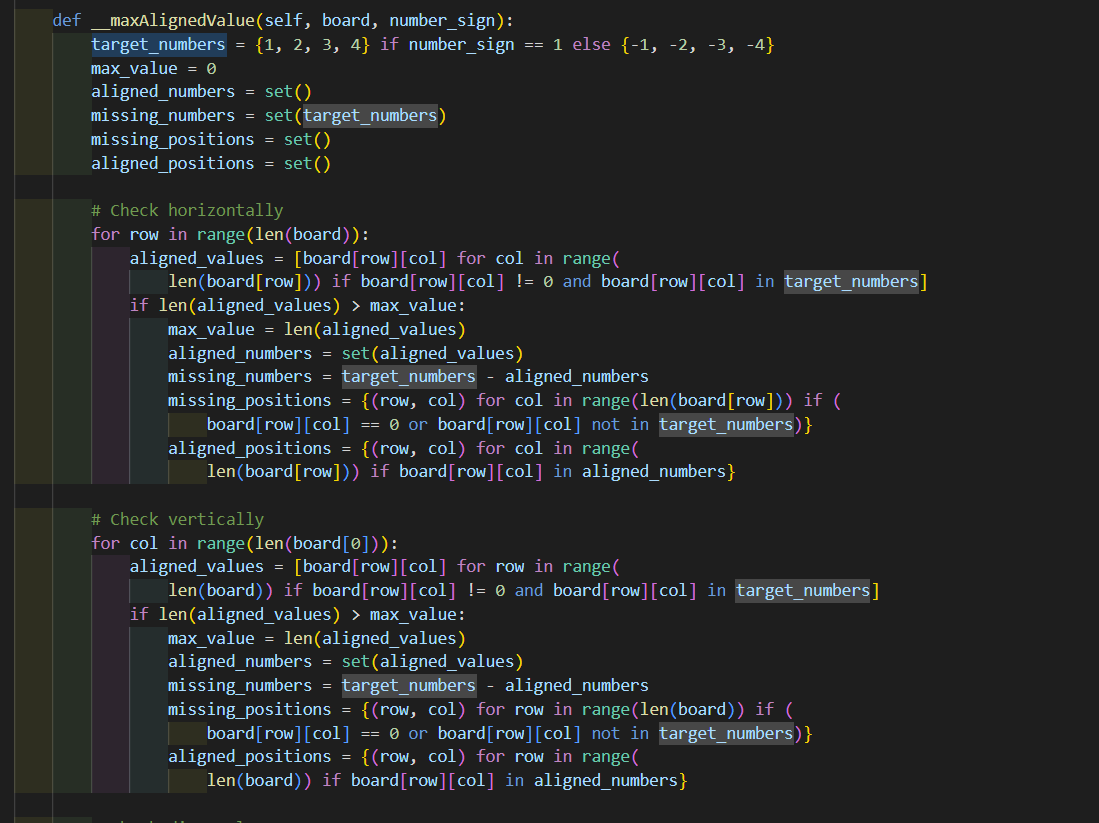
* **\_\_checkVictory(self, board, piecesColor):** verifica si el jugador dado ha ganado.

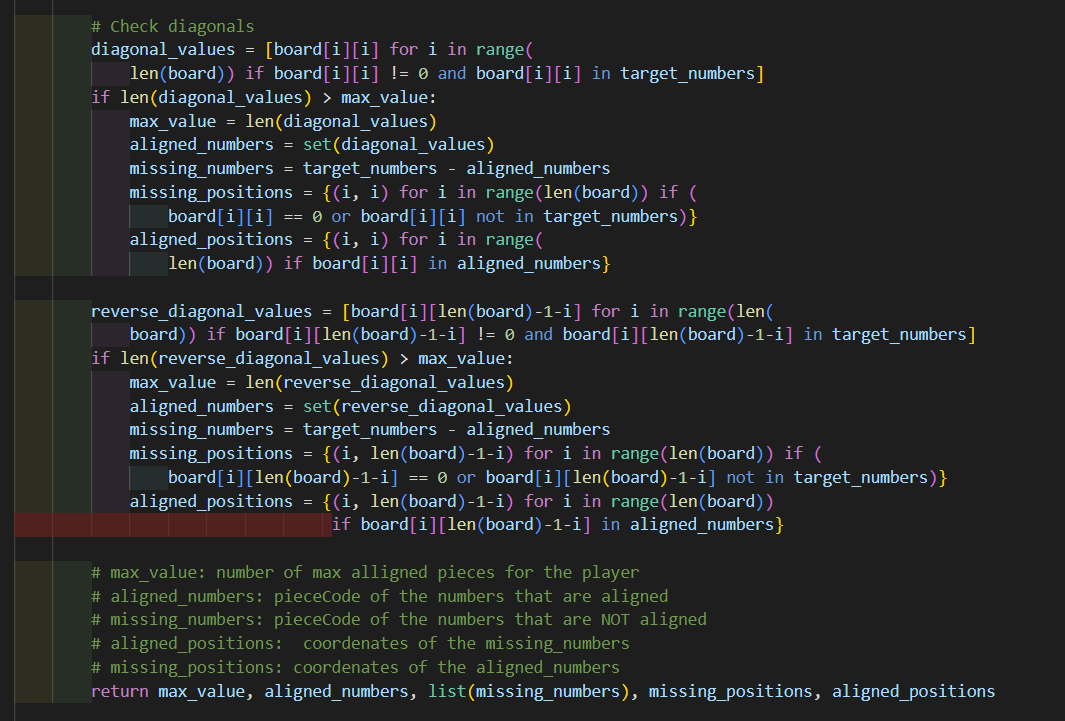


* **\_\_evaluateBoard(self, board):** heurística que evalúa el tablero considerando si alguno de los jugadores ha ganado y el número máximo de piezas que tienen alineadas.

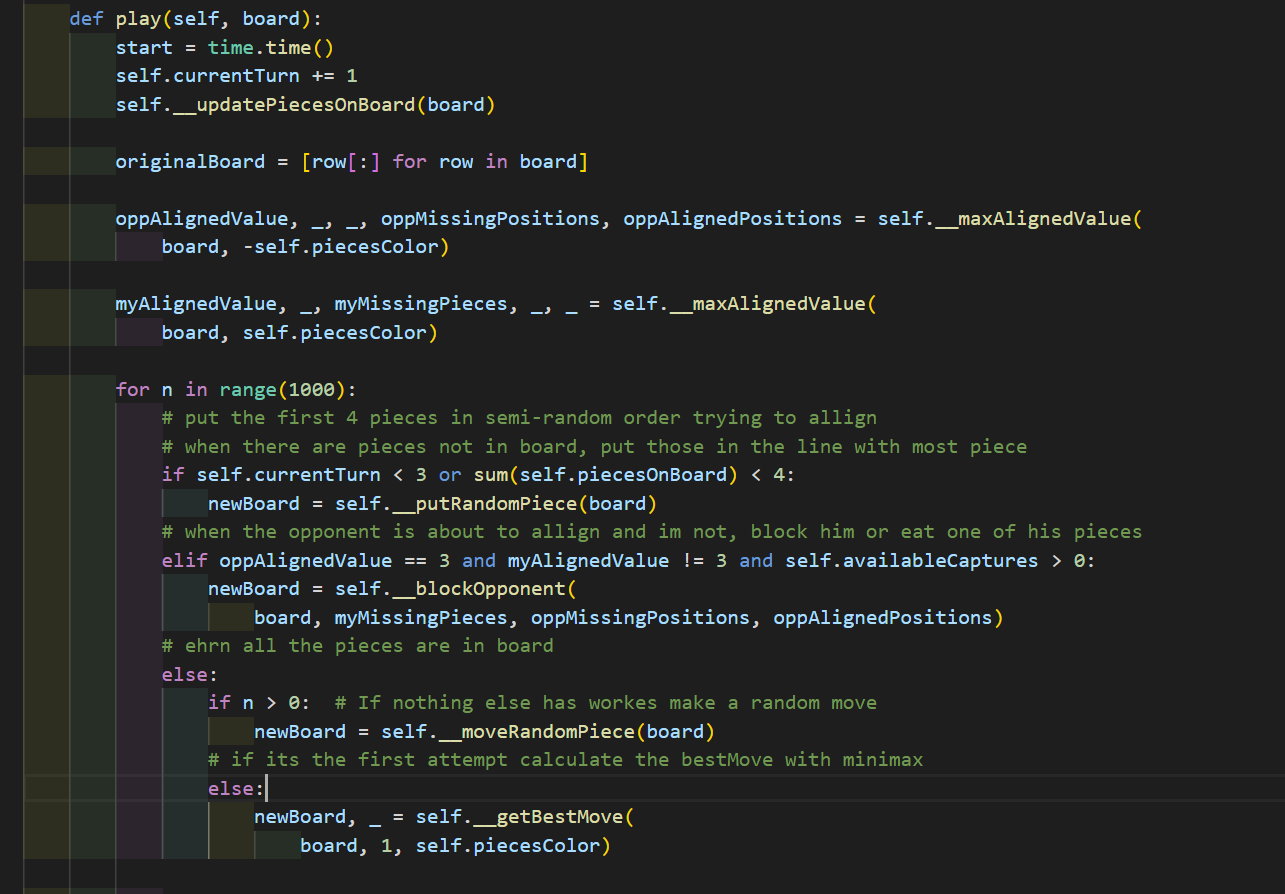


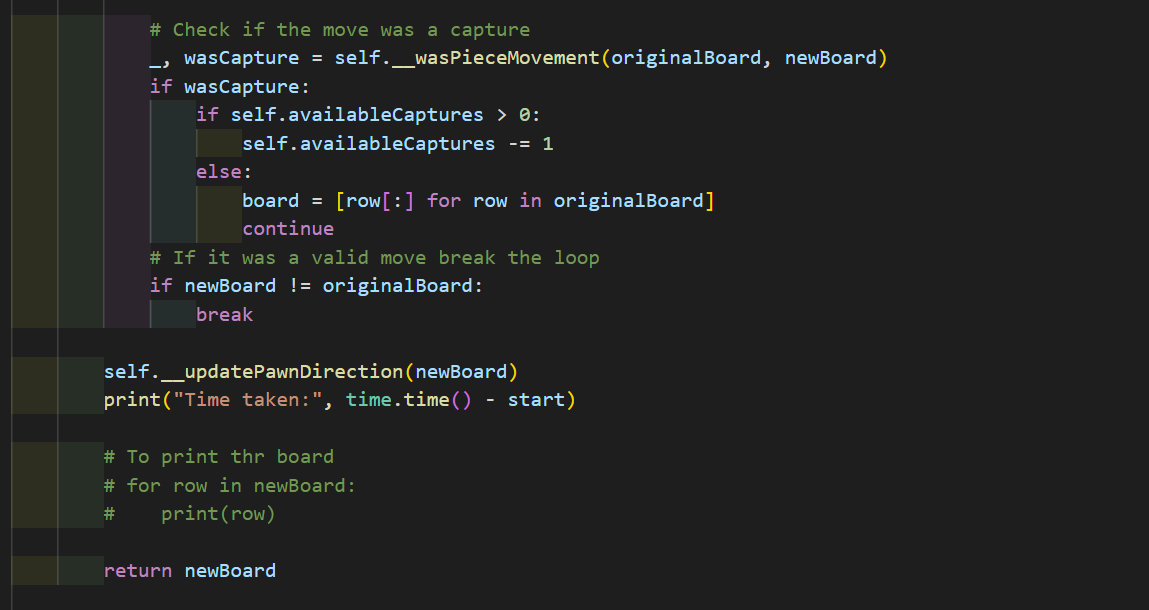
* **\_\_maxAlignedValue(self, board, piecesColor):** función que obtiene:
  + Número máximo de piezas alineadas del jugador.
  + Lista de piezas que están alineadas.
  + Lista de piezas que NO están alineadas.
  + Coordenadas de las celdas faltantes para alinear.
  + Coordenadas de las piezas que ya están alineadas.



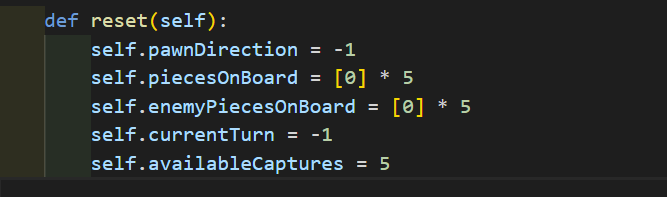


* **play(self, board):** Método principal que representa la estrategia general del bot. Considera el tablero actual, el número de capturas disponibles y el número máximo de turnos permitidos. El método devuelve el tablero actualizado después de que el bot haya realizado su movimiento.





* **reset(self):** método que reinicia los atributos necesarios para iniciar otra partida.



**Conclusión.**

El desarrollar este bot fue muy interesante y frustrante a la vez ya que experimentamos muchas ideas que no funcionaron, las cuales se pueden ver en las otras 5 distintas clases player que implementamos.

Consideramos que algo que podemos mejorar, es sin duda nuestra función minimax, ya que como habíamos comentado no logramos implementarla del todo correcto en cuanto el tema de la profundidad para que pudiera visualizar más jugadas a futuro (quizás por las copias de las matrices y arreglos).

Pero al final logramos implementar otras estrategias para que compensar un poco la falla del minimax y para empezar bien el juego (que quizás no son tan óptimas y nos llevan a consultar demasiadas veces el tablero una y otra vez), pero hemos de decir que el bot funciona bastante decente y logra su cometido de ganar la mayoría de las partidas.