**COSE361(01)\_hw1**

2021150449 통계학과 신진섭

**클래스: Omok**

**Omok** 클래스는 오목 게임의 주요 로직을 처리.

**\_\_init\_\_**: 게임 보드를 초기화. 19x19 크기의 보드를 사용하며, 모든 칸은 빈 상태(0)로 시작.

**make\_move**: 특정 위치에 돌을 놓는 move. 해당 위치가 비어 있으면 돌을 놓고 **True**를 반환.

**get\_legal\_moves**: 현재 돌이 놓인 위치 주변 1칸 이내의 빈 칸을 리스트로 제공.

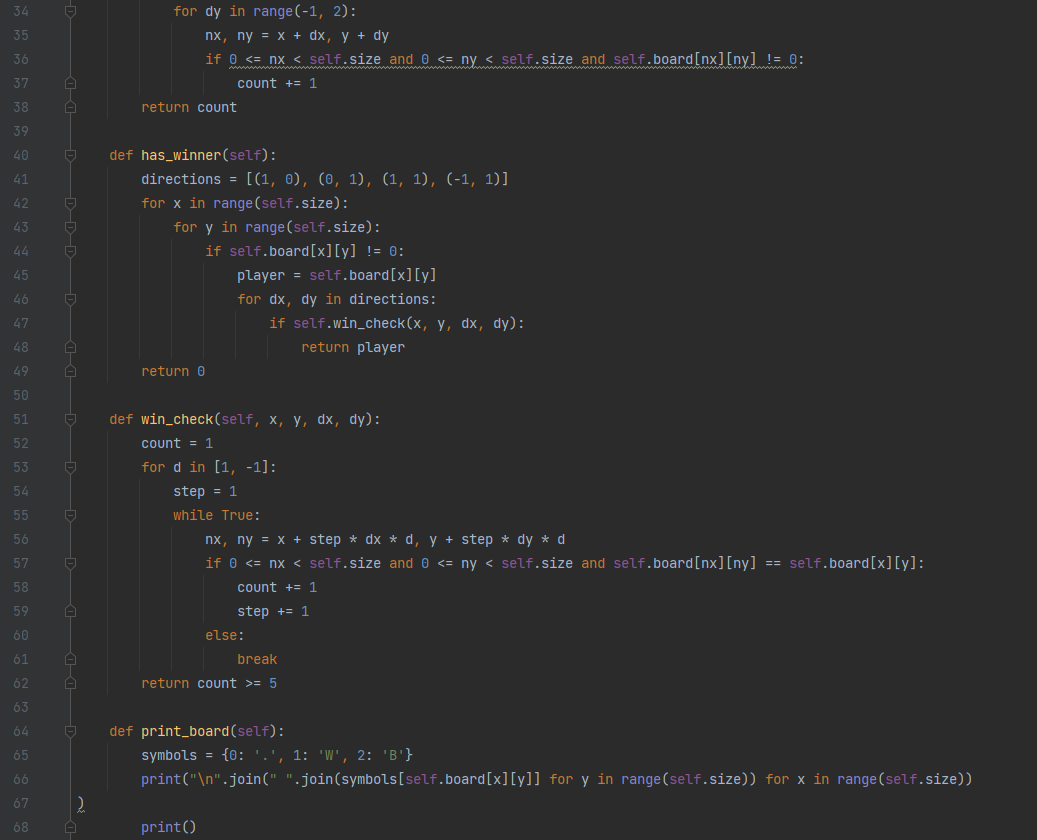
**evaluate\_move**: 특정 수 주변에 이미 놓인 돌의 수를 세어 move의 가치를 평가. **Move odering** 과정에서 사용.

**has\_winner**: 게임 보드를 검사하여 승리 조건을 만족하는지 확인.

**win\_check**: 지정된 방향으로 연속된 돌이 다섯 개 있는지 확인.

**print\_board**: 현재 게임 보드의 상태를 출력.





### 함수: iterative\_deepening

**best\_move**: 최선의 수를 저장.

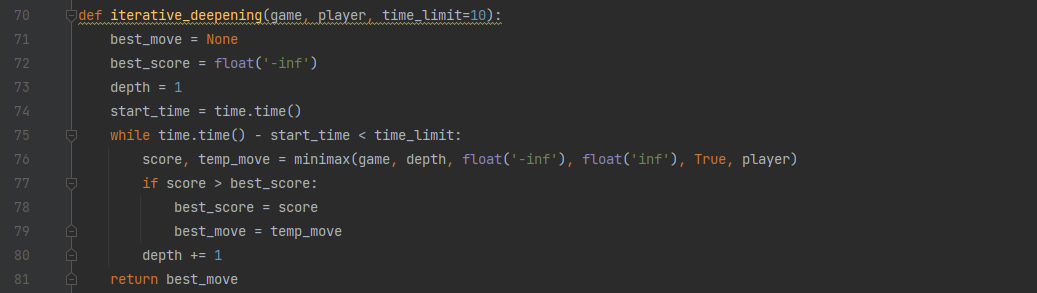
**best\_score**: 최고 점수를 저장.

**depth**: 탐색의 시작 깊이.

**while time.time() - start\_time < time\_limit**: 함수가 주어진 **시간 제한(10초)** 내에서 실행.

**score, temp\_move = minimax(game, depth, float('-inf'), float('inf'), True, player)**: 현재 깊이에서 minmax 알고리즘을 호출하여 최선의 점수와 수를 계산. True는 현재 플레이어가 최대화 플레이어임을 나타냄.

**depth += 1**: 다음 반복을 위해 탐색 깊이를 1씩 증가. 점차적으로 더 깊은 수준의 탐색이 가능.



### 함수: minimax

**depth == 0**: 더 이상 깊이 탐색할 수 없을 때, 현재 노드의 평가값을 계산.

**node.has\_winner()**: 승리 조건이 충족되면, 해당 노드의 평가값을 계산.

**evaluate(node, player)**: 현재 게임 상태를 평가하여 점수를 반환.

**legal\_moves**: 현재 플레이어가 수를 둘 수 있는 모든 위치를 반환.

**Maximizing\_player==True** 경우: 각 자식 노드에서 반환된 최소값 중 큰 값을 탐색.

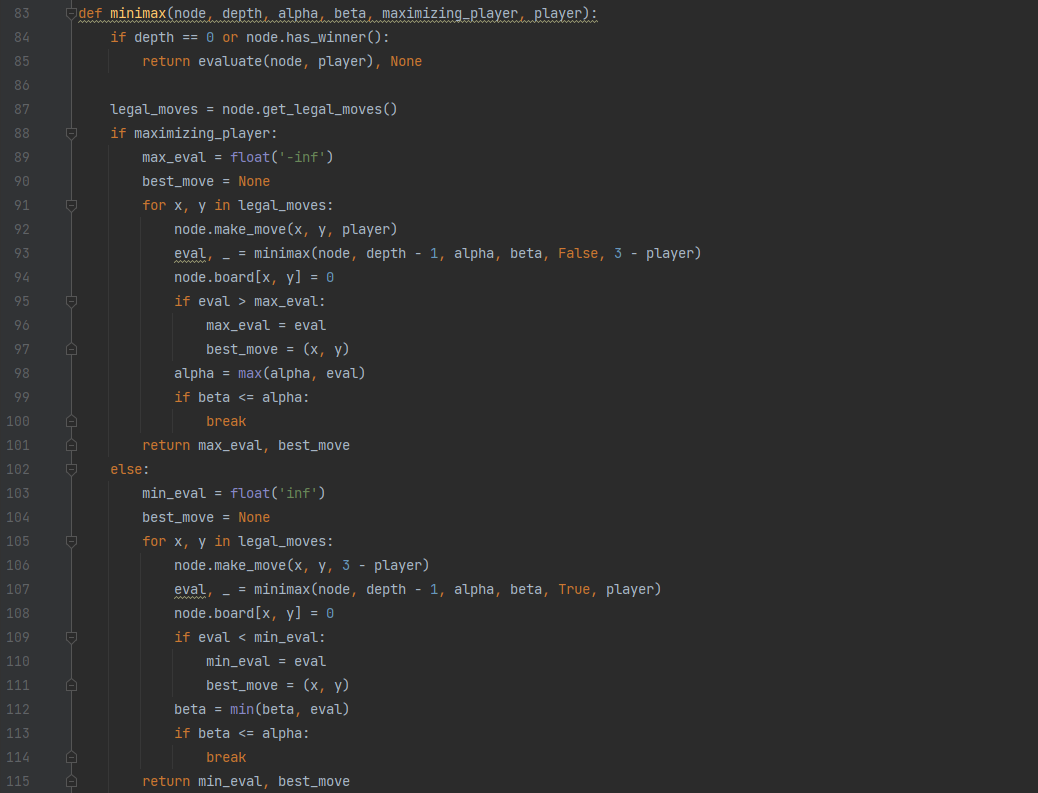
**node.make\_move(x, y, player)**: 현재 플레이어가 (x, y) 위치에 돌을 두어 게임 보드 상태를 임시로 변경.

**eval, \_ = minimax(...)**: 변경된 게임 상태에 대해 minimax 함수를 재귀적으로 호출.

**node.board[x, y] = 0**: 재귀 호출 후, 임시로 변경했던 게임 보드를 원래 상태로 되돌림.

**alpha = max(alpha, eval)**: 알파 값을 현재까지 발견된 최대 평가값으로 업데이트.

**if beta <= alpha**: **알파-베타 pruning** 수행. 베타가 알파보다 작거나 같아지면, 현재 노드 이후의 노드를 더 이상 탐색할 필요가 없음을 의미하므로 루프를 종료.



### 함수: evaluate [heuristic function]

**score\_line** 한 줄(행, 열, 또는 대각선)에 대해 점수를 계산.

**count**: 현재 연속된 돌의 수.

**open\_ends**: 연속된 돌의 양쪽 끝이 얼마나 열려 있는지를 나타내는 변수.

**last**: 마지막으로 확인한 돌의 색깔.

**If** 같은 색의 돌이 연속되는 경우 count를 증가.

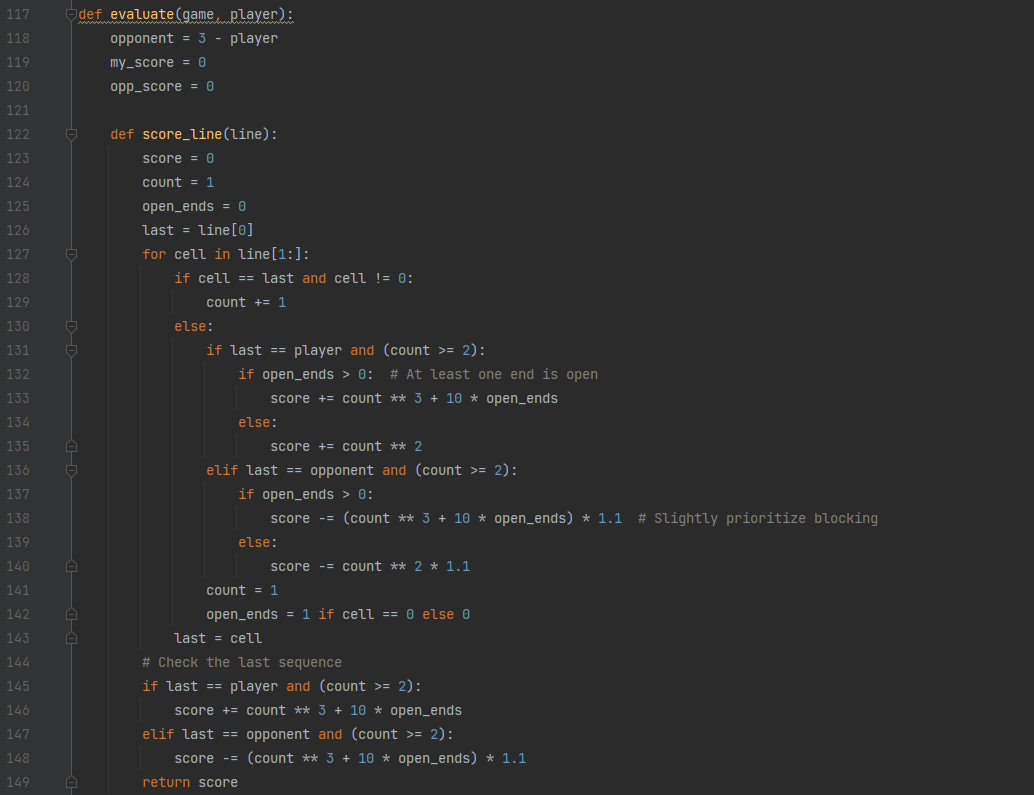
**else** 다른 색의 돌이나 빈 칸을 만나면, 그 시점까지의 연속된 돌에 대해 점수를 계산

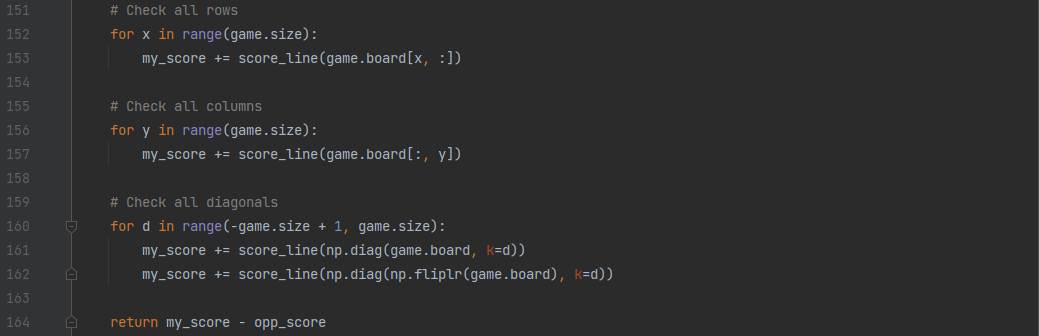
count가 2 이상인 경우 점수를 계산하며, 돌이 열려 있는지 여부에 따라 추가 점수를 부여.

연속된 돌의 수에 따라 지수적으로 점수를 증가.

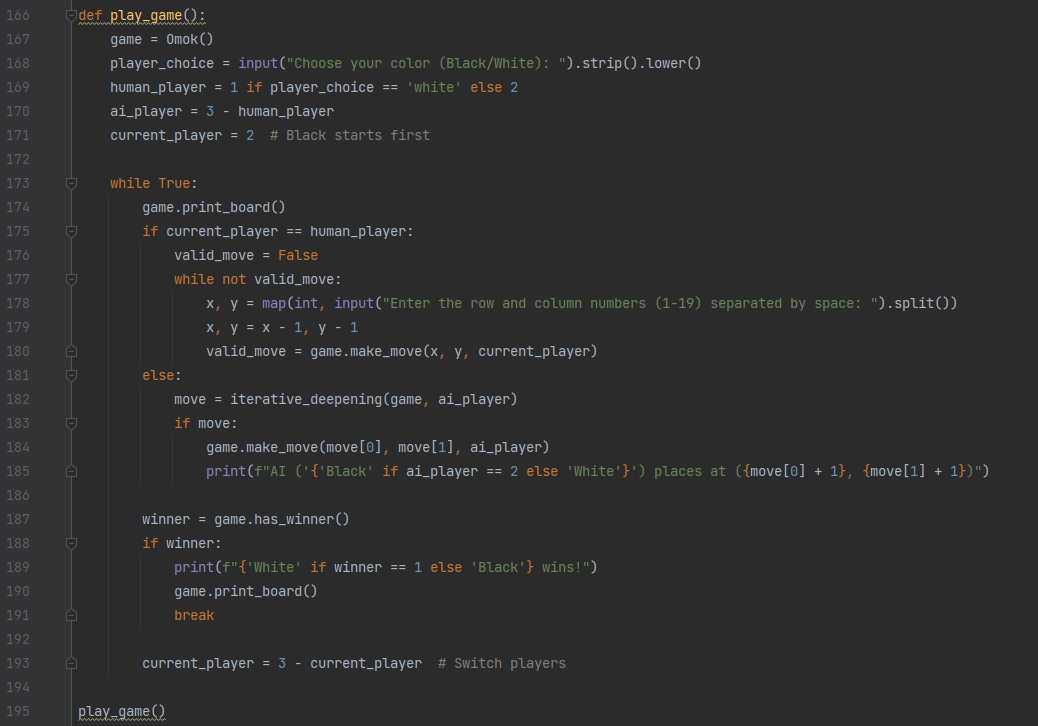
모든 행과 열, 그리고 양 대각선을 순회하면서 **score\_line**을 호출하여 각 줄의 점수를 계산.

최종적으로, **my\_score**에서 **opp\_score**를 빼서 반환. 이 값이 양수면 플레이어가 유리한 상태를 나타내고, 음수면 상대방이 유리한 상태.





### 함수: play\_game



**game = Omok()**: 클래스의 인스턴스를 생성.

**player\_choice**: 사용자에게 Black 또는 White 중 선택하도록 입력을 요청.

**인간 플레이어**의 경우:

**valid\_move = False**: 유효한 수를 두기 전까지 루프를 계속 실행.

**x, y = map(int, input(...).split())** 사용자로부터 좌표 입력 받음.

**AI 플레이어**의 경우:

**move = iterative\_deepening(game, ai\_player)**: **iterative deepening**과 **alpha-beta search** 사용하여 최적의 수 탐색.

**winner = game.has\_winner()**: 승리 조건을 충족하는 플레이어가 있는지 확인.

**current\_player = 3 - current\_player**: 현재 플레이어가 끝난 후 플레이어를 전환.

### Play\_game() 실행 결과

Color 입력



Ai 플레이어 탐색 결과



인간 플레이어 수 입력

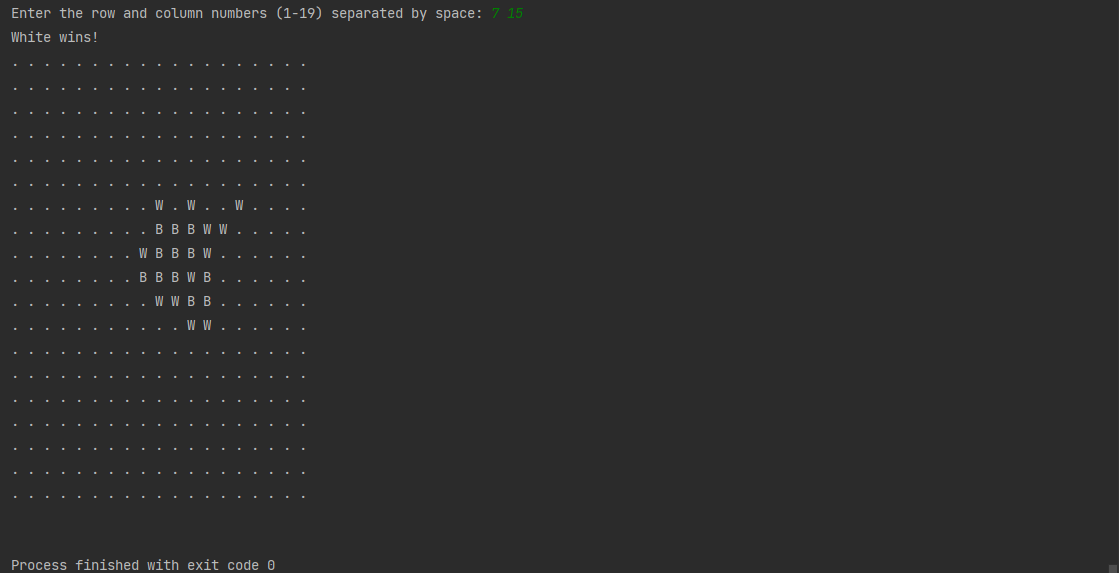


인간 플레이어 입력 결과



최종 결과

초반 시도에서는 아쉬운 성능을 보여줬으며, 4돌의 한 줄을 만드는 시도는 없었고 3돌의 여러 줄을 만드는 시도가 계속해서 이어졌음.



**socre += count \*\* n + 10 \* open\_ends** 의 휴리스틱 함수에서 count의 가중치를 높이고자 지수 파라미터 n을 높임.

이후 더욱 나아진 성능 확인.

