othello.py

Othello is a turn-based two-player strategy board game. The players take turns placing pieces--one player white and the other player black--on an 8x8 board in such a way that captures some of the opponent's pieces, with the goal of finishing the game with more pieces of their color on the board.

Every move must capture one more more of the opponent's pieces. To capture, player A places a piece adjacent to one of player B's pieces so that there is a straight line (horizontal, vertical, or diagonal) of adjacent pieces that begins with one of player A's pieces, continues with one more more of player B's pieces, and ends with one of player A's pieces.

For example, if Black places a piece on square (5, 1), he will capture all of Black's pieces between (5, 1) and (5, 6):

```
  1 2 3 4 5 6 7 8     1 2 3 4 5 6 7 8
1 . . . . . . . .   1 . . . . . . . .
2 . . . . . . . .   2 . . . . . . . .
3 . . o @ . o . .   3 . . o @ . o . .
4 . . o o @ @ . .   4 . . o o @ @ . .
5 . o o o o @ . .   5 @ @ @ @ @ @ . .
6 . . . @ o . . .   6 . . . @ o . . .
7 . . . . . . . .   7 . . . . . . . .
8 . . . . . . . .   8 . . . . . . . .
```

For more more information about the game (which is also known as Reversi) including detailed rules, see the entry on Wikipedia. Additionally, this implementation doesn't take into account some tournament-style Othello details, such as game time limits and a different indexing scheme.

We will implement representations for the board and pieces and the mechanics of playing a game. We will then explore several game-playing strategies. There is a simple command-line program provided for playing against the computer or comparing two strategies.

Written by Daniel Connelly. This implementation follows chapter 18 of Peter Norvig's "Paradigms of Artificial Intelligence".

Table of contents

Board representation

We represent the board as a 100-element list, which includes each square on the board as well as the outside edge. Each consecutive sublist of ten elements represents a single row, and

each list element stores a piece. An initial board contains four pieces in the center:

```
? ? ? ? ? ? ? ? ? ?
? . . . . . . . . ?
? . . . . . . . . ?
? . . . . . . . . ?
? . . . o @ . . . ?
? . . . @ o . . . ?
? . . . . . . . . ?
? . . . . . . . . ?
? . . . . . . . . ?
? ? ? ? ? ? ? ? ? ?
```

This representation has two useful properties:

1. Square (m,n) can be accessed as `board[mn]`. This avoids the need to write functions that convert between square locations and list indexes.
2. Operations involving bounds checking are slightly simpler.

The outside edge is marked ?, empty squares are ., black is @, and white is o. The black and white pieces represent the two players.

```
EMPTY, BLACK, WHITE, OUTER = '.', '@', 'o', '?'
PIECES = (EMPTY, BLACK, WHITE, OUTER)
PLAYERS = {BLACK: 'Black', WHITE: 'White'}
```

To refer to neighbor squares we can add a direction to a square.

```
UP, DOWN, LEFT, RIGHT = -10, 10, -1, 1
UP_RIGHT, DOWN_RIGHT, DOWN_LEFT, UP_LEFT = -9, 11, 9, -11
DIRECTIONS = (UP, UP_RIGHT, RIGHT, DOWN_RIGHT, DOWN, DOWN_LEFT,
```

List all the valid squares on the board.

```
def squares():

    return [i for i in xrange(11, 89) if 1 <= (i % 10) <= 8]
```

Create a new board with the initial black and white positions filled.

```
def initial_board():


    board = [OUTER] * 100
    for i in squares():
        board[i] = EMPTY


    board[44], board[45] = WHITE, BLACK
    board[54], board[55] = BLACK, WHITE
    return board
```

The middle four squares should hold the initial piece positions.

Get a string representation of the board.

```
def print_board(board):


    rep = ''
    rep += '  %s\n' % ' '.join(map(str, range(1, 9)))
    for row in xrange(1, 9):
        begin, end = 10*row + 1, 10*row + 9
        rep += '%d %s\n' % (row, ' '.join(board[begin:end]))
    return rep
```

Playing the game

We need functions to get moves from players, check to make sure that the moves are legal, apply the moves to the board, and detect when the game is over.

Checking moves

A move must be both valid and legal: it must refer to a real square, and it must form a bracket with another piece of the same color with pieces of the opposite color in between.

Is move a square on the board?

```
def is_valid(move):
```

Get player's opponent piece.

```
    return isinstance(move, int) and move in squares()

def opponent(player):

    return BLACK if player is WHITE else WHITE
```

Find a square that forms a bracket with `square` for `player` in the given `direction`. Returns None if no such square exists.

```
def find_bracket(square, player, board, direction):



    bracket = square + direction
    if board[bracket] == player:
        return None
    opp = opponent(player)
    while board[bracket] == opp:
        bracket += direction
    return None if board[bracket] in (OUTER, EMPTY) else bracket
```

Is this a legal move for the player?

```
def is_legal(move, player, board):

    hasbracket = lambda direction: find_bracket(move, player, bo
    return board[move] == EMPTY and any(map(hasbracket, DIRECTIO
```

Making moves

When the player makes a move, we need to update the board and flip all the bracketed pieces.

Update the board to reflect the move by the specified player.

```
def make_move(move, player, board):

    board[move] = player
    for d in DIRECTIONS:
        make_flips(move, player, board, d)
    return board
```

Flip pieces in the given direction as a result of the move by player.

```
def make_flips(move, player, board, direction):



    bracket = find_bracket(move, player, board, direction)
    if not bracket:
        return
    square = move + direction
    while square != bracket:
        board[square] = player
        square += direction
```

Monitoring players

```
class IllegalMoveError(Exception):

    def __init__(self, player, move, board):
        self.player = player
        self.move = move
        self.board = board

    def __str__(self):
        return '%s cannot move to square %d' % (PLAYERS[self.pla
```

Get a list of all legal moves for player.

```
def legal_moves(player, board):

    return [sq for sq in squares() if is_legal(sq, player, board
```

Can player make any moves?

```
def any_legal_move(player, board):

    return any(is_legal(sq, player, board) for sq in squares())
```

Putting it all together

Each round consists of:

- Get a move from the current player.
- Apply it to the board.
- Switch players. If the game is over, get the final score.

Play a game of Othello and return the final board and score.

```
def play(black_strategy, white_strategy):


    board = initial_board()
    player = BLACK
    strategy = lambda who: black_strategy if who == BLACK else w
    while player is not None:
        move = get_move(strategy(player), player, board)
        make_move(move, player, board)
        player = next_player(board, player)
    return board, score(BLACK, board)
```

Which player should move next? Returns None if no legal moves exist.

```
def next_player(board, prev_player):



    opp = opponent(prev_player)
    if any_legal_move(opp, board):
        return opp
    elif any_legal_move(prev_player, board):
        return prev_player
    return None
```

Call strategy(player, board) to get a move.

```
def get_move(strategy, player, board):


    copy = list(board) # copy the board to prevent cheating
    move = strategy(player, copy)
    if not is_valid(move) or not is_legal(move, player, board):
        raise IllegalMoveError(player, move, copy)
    return move
```

Compute player's score (number of player's pieces minus opponent's).

```
def score(player, board):



    mine, theirs = 0, 0
    opp = opponent(player)
    for sq in squares():
        piece = board[sq]
        if piece == player: mine += 1
        elif piece == opp: theirs += 1
    return mine - theirs
```

Play strategies

Random

The easiest strategy to implement simply picks a move at random.

```
import random
```

A strategy that always chooses a random legal move.

```
def random_strategy(player, board):


    return random.choice(legal_moves(player, board))
```

Local maximization

A more sophisticated strategy could look at every available move and evaluate them in some way. This consists of getting a list of legal moves, applying each one to a copy of the board, and choosing the move that results in the "best" board.

Construct a strategy that chooses the best move by maximizing evaluate(player, board) over all boards resulting from legal moves.

```
def maximizer(evaluate):
```

One possible evaluation function is `score`. A strategy constructed with `maximizer(score)` will always make the move that results in the largest immediate gain in pieces.

A more advanced evaluation function might consider the relative worth of each square on the board and weight the score by the value of the pieces held by each player. Since corners and (most) edge squares are very valuable, we could weight those more heavily, and add negative weights to the squares that, if acquired, could lead to the opponent capturing the corners or edges.

A strategy constructed as `maximizer(weighted_score)`, then, will always return the move that results in the largest immediate *weighted* gain in pieces.

Compute the difference between the sum of the weights of player's squares and the sum of the weights of opponent's squares.

Minimax search

The maximizer strategies are very short-sighted, and a player who can consider the implications of a move several turns in advance could have a significant advantage. The minimax algorithm does just that.

Find the best legal move for player, searching to the specified depth. Returns a tuple (move, min_score), where min_score is the guaranteed minimum score achievable for player if the move is made.

We define the value of a board to be the opposite of its value to our opponent, computed by recursively applying `minimax` for our opponent.

When depth is zero, don't examine possible moves--just determine the value of this board to the player.

We want to evaluate all the legal moves by considering their implications `depth` turns in advance. First, find all the legal moves.

If player has no legal moves, then either:

```
    def strategy(player, board):

        def score_move(move):
            return evaluate(player, make_move(move, player, list
        return max(legal_moves(player, board), key=score_move)
    return strategy
```

```
SQUARE_WEIGHTS = [
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0, 120, -20,  20,   5,   5,  20, -20, 120,   0,
    0, -20, -40,  -5,  -5,  -5,  -5, -40, -20,   0,
    0,  20,  -5,  15,   3,   3,  15,  -5,  20,   0,
    0,   5,  -5,   3,   3,   3,   3,  -5,   5,   0,
    0,   5,  -5,   3,   3,   3,   3,  -5,   5,   0,
    0,  20,  -5,  15,   3,   3,  15,  -5,  20,   0,
    0, -20, -40,  -5,  -5,  -5,  -5, -40, -20,   0,
    0, 120, -20,  20,   5,   5,  20, -20, 120,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
]
```

```
def weighted_score(player, board):

    opp = opponent(player)
    total = 0
    for sq in squares():
        if board[sq] == player:
            total += SQUARE_WEIGHTS[sq]
        elif board[sq] == opp:
            total -= SQUARE_WEIGHTS[sq]
    return total
```

```
def minimax(player, board, depth, evaluate):

    def value(board):
        return -minimax(opponent(player), board, depth-1, evalua

    if depth == 0:
        return evaluate(player, board), None

    moves = legal_moves(player, board)

    if not moves:

        if not any_legal_move(opponent(player), board):
            return final_value(player, board), None
```

the game is over, so the best achievable score is victory or defeat;

or we have to pass this turn, so just find the value of this board.

When there are multiple legal moves available, choose the best one by maximizing the value of the resulting boards.

Values for endgame boards are big constants.

The game is over--find the value of this board to player.

```
        return value(board), None

    return max((value(make_move(m, player, list(board))), m) for
```

```
MAX_VALUE = sum(map(abs, SQUARE_WEIGHTS))
MIN_VALUE = -MAX_VALUE

def final_value(player, board):

    diff = score(player, board)
    if diff < 0:
        return MIN_VALUE
    elif diff > 0:
        return MAX_VALUE
    return diff
```

Construct a strategy that uses `minimax` with the specified leaf board evaluation function.

```
def minimax_searcher(depth, evaluate):

    def strategy(player, board):
        return minimax(player, board, depth, evaluate)[1]
    return strategy
```

Alpha-Beta search

Minimax is very effective, but it does too much work: it evaluates many search trees that should be ignored.

Consider what happens when minimax is evaluating two moves, M1 and M2, on one level of a search tree. Suppose minimax determines that M1 can result in a score of S. While evaluating M2, if minimax finds a move in its subtree that could result in a better score than S, the algorithm should immediately quit evaluating M2: the opponent will force us to play M1 to avoid the higher score resulting from M1, so we shouldn't waste time determining just how much better M2 is than M1.

We need to keep track of two values:

- alpha: the maximum score achievable by any of the moves we have encountered.
- beta: the score that the opponent can keep us under by playing other moves.

When the algorithm begins, alpha is the smallest value and beta is the largest value. During evaluation, if we find a move that causes `alpha >= beta`, then we can quit searching this subtree since the opponent can prevent us from playing it.

Find the best legal move for player, searching to the specified depth. Like minimax, but uses the bounds alpha and beta to prune branches.

```
def alphabeta(player, board, alpha, beta, depth, evaluate):

    if depth == 0:
        return evaluate(player, board), None

    def value(board, alpha, beta):

        return -alphabeta(opponent(player), board, -beta, -alpha

    moves = legal_moves(player, board)
    if not moves:
        if not any_legal_move(opponent(player), board):
```

Like in `minimax`, the value of a board is the opposite of its value to the opponent. We pass in `-beta` and `-alpha` as the alpha and beta values, respectively, for the opponent, since `alpha` represents the best score we know we can achieve and is

therefore the worst score achievable by the opponent. Similarly, `beta` is the worst score that our opponent can hold us to, so it is the best score that they can achieve.

If one of the legal moves leads to a better score than beta, then the opponent will avoid this branch, so we can quit looking.

If one of the moves leads to a better score than the current best achievable score, then replace it with this one.

```
            return final_value(player, board), None
        return value(board, alpha, beta), None

    best_move = moves[0]
    for move in moves:
        if alpha >= beta:


            break
        val = value(make_move(move, player, list(board)), alpha,
        if val > alpha:


            alpha = val
            best_move = move
    return alpha, best_move


def alphabeta_searcher(depth, evaluate):


    def strategy(player, board):
        return alphabeta(player, board, MIN_VALUE, MAX_VALUE, de
    return strategy
```

Conclusion

The strategies we've discussed are very general and are applicable to a broad range of strategy games, such as Chess, Checkers, and Go. More advanced strategies for Othello exist that apply various gameplay heuristics; some of these are discussed in "Paradigms of Artificial Intelligence Programming" by Peter Norvig.

See Wikipedia for more details on minimax and alpha-beta search.