explanation of alpha-beta pruning code in relation to the pseudocode of the algorithm.

```python
def alphabeta(self, board, maximizing, alpha, beta):
    # terminal case
    case = board.final_state()

    # player 1 wins
    if case == 1:
        return 1, None  # eval, move

    # player 2 wins
    if case == 2:
        return -1, None

    # draw
    elif board.isfull():
        return 0, None
```

In the pseudocode, the terminal conditions are checked first. If the game has reached a terminal state (win, loss, or draw), the evaluation value is returned along with the move (which is None in this case). This corresponds to the base case in the pseudocode.

The remaining code implements the recursive part of the algorithm:

```python
    if maximizing:
        max_eval = -100
        best_move = None
        empty_sqrs = board.get_empty_sqrs()

        for (row, col) in empty_sqrs:
            temp_board = copy.deepcopy(board)
            temp_board.mark_sqr(row, col, 1)
            eval, _ = self.alphabeta(temp_board, False, alpha, beta)
            if eval > max_eval:
                max_eval = eval
                best_move = (row, col)
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break

        return max_eval, best_move
```

In the pseudocode, this corresponds to the MAX player's turn. The code initializes the maximum evaluation value (`max_eval`) to a very low value and the best move (`best_move`) to None. It retrieves the empty squares on the board and iterates over them.

For each empty square, it creates a copy of the board, marks the square with the MAX player's move, and recursively calls the `alphabeta` function with the `maximizing` flag set to False. The returned evaluation value is compared with the current maximum evaluation value (`max_eval`), and if it is greater, it updates `max_eval` and `best_move` .

The code also updates the alpha value with the maximum evaluation value and checks if beta is less than or equal to alpha. If it is, it breaks out of the loop, as the remaining moves will not affect the final decision.

```
    elif not maximizing:
        min_eval = 100
        best_move = None
        empty_sqrs = board.get_empty_sqrs()

        for (row, col) in empty_sqrs:
            temp_board = copy.deepcopy(board)
            temp_board.mark_sqr(row, col, self.player)
            eval, _ = self.alphabeta(temp_board, True, alpha, beta)
            if eval < min_eval:
                min_eval = eval
                best_move = (row, col)
            beta = min(beta, min_eval)
            if beta <= alpha:
                break

        return min_eval, best_move
```

In the pseudocode, this corresponds to the MIN player's turn. The code is similar to the MAX player's turn, but with the roles of minimum and maximum evaluations reversed. It initializes the minimum evaluation value (`min_eval`) to a very high value and the best move (`best_move`) to None. It performs the same steps as the MAX player's turn but with the considerations for the minimum evaluation value (`min_eval`) and the beta value.

The code updates the beta value with the minimum evaluation value and checks if beta is less than or equal to alpha. If it is, it breaks out of the loop.

The code then returns the evaluation value and the best move for the current player's turn.