Daily Coding Problem #219

Problem

This problem was asked by Salesforce.

Connect 4 is a game where opponents take turns dropping red or black discs into a 7 x 6 vertically suspended grid. The game ends either when one player creates a line of four consecutive discs of their color (horizontally, vertically, or diagonally), or when there are no more spots left in the grid.

Design and implement Connect 4.

Solution

For any design question, it is helpful to spend some time thinking about the structure of our solution before jumping into coding. What are some of the core features of Connect 4, as described above?

- We should represent the board in a way that allows it to change state with each move.

- We should be able to display the board to the user.
- Players should be able to input valid moves.
- We should be able to check whether a win condition has been met.
- To play our game, we should repeatedly display the board, make a move, and check for a win condition.

Here is a skeleton of how our code could look, then:

```
class Game:
    def __init__(self):
        self.board = [['.'  for _ in range(7)] for _ in range(6)]
        self.game_over = False
        self.winner = None
        self.last_move = None
        self.players = ['x', 'o']
        self.turn = 0

    def play(self):
        while not self.game_over:
            self.print_board()
            self.move(self.players[self.turn])
            self.check_win()

        self.print_outcome()

    def print_board(self):
        # Display the board to the user.
        pass

    def move(self, player):
        # Get and validate input from the user, and update the board's state.
        pass

    def is_valid(self, move):
        # Make sure the move can be made.
        pass
```

```
    def check_win(self):
        # Check for four in a row, and set `self.game_over` and `self.winner` if
found.
        pass

    def print_outcome(self):
        # Congratulate the user on winning, or declare that there was a tie.
        pass
```

Let's go through each of these methods in turn.

Printing the board

We will represent our board as a `7 x 6` matrix, where `board[i][j]` represents the `i`th row and `j`th column. Each location in the board can be filled in by an `x` or an `o`, representing opposing players. To display the board, we will print out each row one at a time.

```
def print_board(self):
    for row in self.board:
        print "".join(row)
```

Making a move

To make a move, we should first ask the user for a column to place a disc in. An input is valid as long as it is a number between `0` and `6`, and as long as that column is not already full. Given a valid input, we update the board state and change the `turn` variable. Note that since discs are dropped vertically into the board, each column will be filled from bottom to top.

Additionally, we will track `last_move` as a class variable, since that will allow us to more efficiently check the win condition.

```
def move(self, player):
    col = raw_input("{0}'s turn to move: ".format(player))
    while not self.is_valid(col):
```

```
            col = raw_input("Move not valid. Please try again: ")

        row, col = 5, int(col)
        while self.board[row][col] != '.':
            row -= 1

        self.board[row][col] = player
        self.turn = 1 - self.turn
        self.last_move = (row, col)

    def is_valid(self, col):
        try:
            col = int(col)
        except ValueError:
            return False
        if 0 <= col <= 6 and self.board[0][col] == '.':
            return True
        else:
            return False
```

Checking for a win

A naive way of checking for four in a row would be to enumerate all possible lines of four, and for each of these, check to see if all the values are either `x` or `o`. We can improve this by noting that if a player has just placed a disc in `board[row][col]`, the only possible wins are those that include that particular row and column.

To make use of this, we will first locate the row, column, positive diagonal, and negative diagonal corresponding to the location of the last played move. For each of these, we will check whether any length-four slice all consists of the same player's value, and if so, update the variables `game_over` and `winner`.

Finally, we must check if the board is completely full, in which case the game is over regardless of whether or not someone has won.

```
    def check_win(self):
        row, col = self.last_move
```

```
        horizontal = self.board[row]
        vertical = [self.board[i][col] for i in range(6)]

        neg_offset, pos_offset = col - row, col + row
        neg_diagonal = [row[i + neg_offset] for i, row in enumerate(self.board) if 0
<= i + neg_offset <= 6]
        pos_diagonal = [row[-i + pos_offset] for i, row in enumerate(self.board) if
0 <= -i + pos_offset <= 6]

        possible_wins = [horizontal, vertical, pos_diagonal, neg_diagonal]
        for p in possible_wins:
            for i in range(len(p) - 3):
                if len(set(p[i : i + 4])) == 1 and p[i] != '.':
                    self.game_over = True
                    self.winner = p[i]
                    break

        if all(self.board[0][col] != '.' for col in range(7)):
            self.game_over = True
```

The full game

Putting it all together, our code should look something like this:

```
class Game:
    def __init__(self):
        self.board = [['.'  for _ in range(7)] for _ in range(6)]
        self.game_over = False
        self.winner = None
        self.last_move = None
        self.players = ['x', 'o']
        self.turn = 0

    def play(self):
        while not self.game_over:
            self.print_board()
```

```
            self.move(self.players[self.turn])
            self.check_win()

        self.print_outcome()

    def print_board(self):
        for row in self.board:
            print "".join(row)

    def move(self, player):
        col = raw_input("{0}'s turn to move: ".format(player))
        while not self.is_valid(col):
            col = raw_input("Move not valid. Please try again: ")

        row, col = 5, int(col)
        while self.board[row][col] != '.':
            row -= 1

        self.board[row][col] = player
        self.turn = 1 - self.turn
        self.last_move = (row, col)

    def is_valid(self, col):
        try:
            col = int(col)
        except ValueError:
            return False
        if 0 <= col <= 6 and self.board[0][col] == '.':
            return True
        else:
            return False

    def check_win(self):
        row, col = self.last_move

        horizontal = self.board[row]
        vertical = [self.board[i][col] for i in range(6)]
```

```
        neg_offset, pos_offset = col - row, col + row
        neg_diagonal = [row[i + neg_offset] for i, row in enumerate(self.board)
if 0 <= i + neg_offset <= 6]
        pos_diagonal = [row[-i + pos_offset] for i, row in enumerate(self.board)
if 0 <= -i + pos_offset <= 6]

        possible_wins = [horizontal, vertical, pos_diagonal, neg_diagonal]
        for p in possible_wins:
            for i in range(len(p) - 3):
                if len(set(p[i : i + 4])) == 1 and p[i] != '.':
                    self.game_over = True
                    self.winner = p[i]
                    break

        if all(self.board[0][col] != '.' for col in range(7)):
            self.game_over = True

    def print_outcome(self):
        self.print_board()
        if not self.winner:
            print "Game over: it was a draw!"
        else:
            print "Game over: {0} won!".format(self.winner)
```