```
# connectfour.py
#
# ICS 32 Winter 2022
# Project #2: Send Me On My Way

'''
This module contains the game logic that underlies a Connect Four
game, implementing such functionality as tracking the state of a game,
updating that state as players make moves, and determining if there is a
winner.  No user interface or network functionality is included; this is
strictly a collection of tools for implementing the game logic.
'''

from collections import namedtuple


# These constants specify the concepts of "no player", the "red player"
# and the "yellow player".  Your code should use these constants in
# place of their hard-coded values.  Note that these values are not
# intended to be displayed to a user; they're simply intended as a
# universal way to distinguish which player we're talking about (e.g.,
# which player's turn it is, or which player (if any) has a piece in a
# particular cell on the board).

EMPTY = 0
RED = 1
YELLOW = 2


# These constants specify the minimum and maximum sizes of the board.

MIN_COLUMNS = 4
MAX_COLUMNS = 20

MIN_ROWS = 4
MAX_ROWS = 20


# GameState is a namedtuple that tracks everything important about
# the state of a Connect Four game as it progresses.  It contains
# two fields:
#
#     'board', which is a two-dimensional list of values describing
#     the game board.  Each value represents one cell on the board
#     and is either EMPTY, RED, or YELLOW, depending on whether the
#     cell contains a red piece, a yellow piece, or is empty
#
#     'turn', which specifies which player will make the next move;
#     its value will always be either RED or YELLOW

GameState = namedtuple('GameState', ['board', 'turn'])



# This is the simplest example of how you create new kinds of exceptions
# that are specific to your own program.  We'll see later in the quarter
# in more detail what this notation means; in short, we're introducing
# new types into our program and stating that they have strong similarities
# to the existing type called Exception.


class InvalidMoveError(Exception):
    '''Raised whenever an invalid move is made'''
    pass
```

```
class GameOverError(Exception):
    '''
    Raised whenever an attempt is made to make a move after the game is
    already over
    '''
    pass



# The next several functions are the "public" functions that are provided
# by this module.  You may need to call all of these public functions, but
# will not need (or want) to call any of the others.



def new_game(columns: int, rows: int) -> GameState:
    '''
    Returns a GameState representing a brand new game in which no
    moves have been made yet.
    '''
    _require_valid_column_count(columns)
    _require_valid_row_count(rows)

    return GameState(board = _new_game_board(columns, rows), turn = RED)



def columns(game_state: GameState) -> int:
    '''
    Returns the number of columns on the board represented by the given
    GameSate.
    '''
    return _board_columns(game_state.board)



def rows(game_state: GameState) -> int:
    '''
    Returns the number of rows on the board represented by the given
    GameSate.
    '''
    return _board_rows(game_state.board)



def drop(game_state: GameState, column_number: int) -> GameState:
    '''
    Given a game state and a column number, returns the game state
    that results when the current player (whose turn it is) drops a piece
    into the given column.  If the column number is invalid, a ValueError
    is raised.  If the game is over, a GameOverError is raised.  If a move
    cannot be made in the given column because the column is filled already,
    an InvalidMoveError is raised.
    '''
    _require_valid_column_number(column_number, game_state.board)
    _require_game_not_over(game_state)

    empty_row = _find_bottom_empty_row_in_column(game_state.board, column_number)

    if empty_row == -1:
        raise InvalidMoveError()
```

```
    else:
        new_board = _copy_game_board(game_state.board)
        new_board[column_number][empty_row] = game_state.turn
        new_turn = _opposite_turn(game_state.turn)
        return GameState(board = new_board, turn = new_turn)



def pop(game_state: GameState, column_number: int) -> GameState:
    '''
    Given a game state and a column number, returns the game state that
    results when the current player (whose turn it is) pops a piece from the
    bottom of the given column.  If the column number is invalid, a ValueError
    is raised.  If the game is over, a GameOverError is raised.  If a piece
    cannot be popped from the bottom of the given column because the column
    is empty or because the piece at the bottom of the column belongs to the
    other player, an InvalidMoveError is raised.
    '''
    _require_valid_column_number(column_number, game_state.board)
    _require_game_not_over(game_state)

    if game_state.turn == game_state.board[column_number][rows(game_state) - 1]:
        new_board = _copy_game_board(game_state.board)

        for row in range(rows(game_state) - 1, -1, -1):
            new_board[column_number][row] = new_board[column_number][row - 1]

        new_board[column_number][row] = EMPTY

        new_turn = _opposite_turn(game_state.turn)

        return GameState(board = new_board, turn = new_turn)

    else:
        raise InvalidMoveError()



def winner(game_state: GameState) -> int:
    '''
    Determines the winning player in the given game state, if any.
    If the red player has won, RED is returned; if the yellow player
    has won, YELLOW is returned; if no player has won yet, EMPTY is
    returned.
    '''
    winner = EMPTY

    for col in range(columns(game_state)):
        for row in range(rows(game_state)):
            if _winning_sequence_begins_at(game_state.board, col, row):
                if winner == EMPTY:
                    winner = game_state.board[col][row]
                elif winner != game_state.board[col][row]:
                    # This handles the rare case of popping a piece
                    # causing both players to have four in a row;
                    # in that case, the last player to make a move
                    # is the winner.
                    return _opposite_turn(game_state.turn)

    return winner



# Modules often contain functions, variables, or classes whose names begin
```

```
# with underscores.  This is no accident; in Python, this is the agreed-upon
# standard for specifying functions, variables, or classes that should be
# treated as "private" or "hidden".  The rest of your program should not
# need to call any of these functions directly; they are utility functions
# called by the functions above, so that those functions can be shorter,
# simpler, and more readable.



def _new_game_board(columns: int, rows: int) -> list[list[int]]:
    '''
    Creates a new game board with the specified number of columns.
    Initially, a game board has the size columns x rows and is composed
    only of integers with the value EMPTY.
    '''
    board = []

    for col in range(columns):
        board.append([])
        for row in range(rows):
            board[-1].append(EMPTY)

    return board



def _board_columns(board: list[list[int]]) -> int:
    '''Returns the number of columns on the given game board'''
    return len(board)



def _board_rows(board: list[list[int]]) -> int:
    '''Returns the number of rows on the given game board'''
    return len(board[0])



def _copy_game_board(board: list[list[int]]) -> list[list[int]]:
    '''Copies the given game board'''
    board_copy = []

    for col in range(_board_columns(board)):
        board_copy.append([])
        for row in range(_board_rows(board)):
            board_copy[-1].append(board[col][row])

    return board_copy



def _find_bottom_empty_row_in_column(board: list[list[int]], column_number: int) -> int:
    '''
    Determines the bottommost empty row within a given column, useful
    when dropping a piece; if the entire column in filled with pieces,
    this function returns -1
    '''
    for i in range(_board_rows(board) - 1, -1, -1):
        if board[column_number][i] == EMPTY:
            return i

    return -1
```

```
def _opposite_turn(turn: int) -> int:
    '''Given the player whose turn it is now, returns the opposite player'''
    if turn == RED:
        return YELLOW
    else:
        return RED



def _winning_sequence_begins_at(board: list[list[int]], col: int, row: int) -> bool:
    '''
    Returns True if a winning sequence of pieces appears on the board
    beginning in the given column and row and extending in any of the
    eight possible directions; returns False otherwise
    '''
    return _four_in_a_row(board, col, row, 0, 1) \
           or _four_in_a_row(board, col, row, 1, 1) \
           or _four_in_a_row(board, col, row, 1, 0) \
           or _four_in_a_row(board, col, row, 1, -1) \
           or _four_in_a_row(board, col, row, 0, -1) \
           or _four_in_a_row(board, col, row, -1, -1) \
           or _four_in_a_row(board, col, row, -1, 0) \
           or _four_in_a_row(board, col, row, -1, 1)



def _four_in_a_row(board: list[list[int]], col: int, row: int, coldelta: int, rowdelta: int)
-> bool:
    '''
    Returns True if a winning sequence of pieces appears on the board
    beginning in the given column and row and extending in a direction
    specified by the coldelta and rowdelta
    '''
    start_cell = board[col][row]

    if start_cell == EMPTY:
        return False
    else:
        for i in range(1, 4):
            if not _is_valid_column_number(col + coldelta * i, board) \
                   or not _is_valid_row_number(row + rowdelta * i, board) \
                   or board[col + coldelta *i][row + rowdelta * i] != start_cell:
                return False
        return True



def _require_valid_column_number(column_number: int, board: list[list[int]]) -> None:
    '''Raises a ValueError if its parameter is not a valid column number'''
    if type(column_number) != int or not _is_valid_column_number(column_number, board):
        raise ValueError(f'column_number must be an int between 0 and {_board_columns(board)
- 1}')



def _require_game_not_over(game_state: GameState) -> None:
    '''
    Raises a GameOverError if the given game state represents a situation
    where the game is over (i.e., there is a winning player)
    '''
    if winner(game_state) != EMPTY:
        raise GameOverError()
```

```
def _is_valid_column_number(column_number: int, board: list[list[int]]) -> bool:
    '''Returns True if the given column number is valid; returns False otherwise'''
    return 0 <= column_number < _board_columns(board)



def _is_valid_row_number(row_number: int, board: list[list[int]]) -> bool:
    '''Returns True if the given row number is valid; returns False otherwise'''
    return 0 <= row_number < _board_rows(board)


def _require_valid_column_count(columns: int) -> None:
    '''Raises a ValueError if the given number of columns is invalid.'''
    if columns < MIN_COLUMNS or columns > MAX_COLUMNS:
        raise ValueError(f'columns must be an int between {MIN_COLUMNS} and {MAX_COLUMNS}')


def _require_valid_row_count(rows: int) -> None:
    '''Raises a ValueError if the given number of rows is invalid.'''
    if rows < MIN_ROWS or rows > MAX_ROWS:
        raise ValueError(f'rows must be an int between {MIN_ROWS} and {MAX_ROWS}')
```