

## Solutions 4

Summer 2017

---

### Task 1 - The Game Board

The idea of this task is to aid the grasping of nested lists, being used to represent a 3x3 game board.

```
Python 3.6.0 (default, Dec 24 2016, 18:27:04)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> board = [['O', 'X', None], [None, 'O', 'X'], ['X', None, 'O']]
>>> board[0]
['O', 'X', None]
>>> board[1]
[None, 'O', 'X']
>>> board[2]
['X', None, 'O']
>>> board[1][2]
'X'
>>> |
```

Ln: 13 Col: 4


---

## Task 2 - Tic-Tac-Toe Code

The initial code provided has 8 methods:

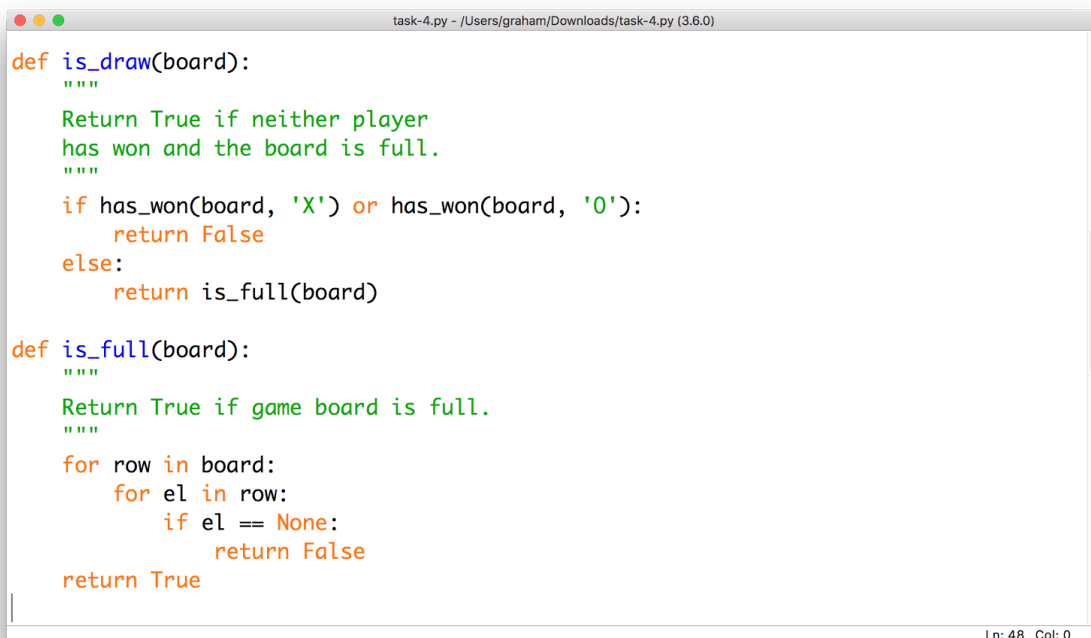
1. *play\_game* is the entry point. It initializes a game board, chooses player X to go first, and then enters the game loop: while the game has not finished, print the board, make one of players move, and then switch players. Once we leave the loop, we print the winner using the *print\_winner* function.
2. *is\_game\_finished* determines if the game is over. A game is over if either player has won, or if there is a draw. Detecting each of these things has been abstracted to individual methods, as we will see.
3. *has\_won* determines if a given player has won the game. The implementation of this method has been left to be completed in the tasks.
4. *is\_draw* determines if the board has been drawn. This happens if neither player has won, and the game board is full. Similar to the *is\_game\_finished* function, each of these checks is delegated to the *has\_won* function and the *is\_full* function.
5. *is\_full* is called from the *is\_draw* function, and will return *True* only when every element of the board is not equal to *None*. The implementation of this function is a task to be completed.
6. *print\_board* does what it says: it prints the board to the screen. The final task is to improve this function so it looks more like a real board.
7. *make\_move* will prompt a move to be entered from the standard input. It takes a board as a formal parameter, and returns a new board after making the move.
8. *print\_winner* prints who won to the screen, using the *has\_won* and *is\_draw* functions to determine the outcome.

So, the *has\_won*, *is\_full* functions have been left to be implemented from scratch, while the rest of the functions have been implemented already. The *print\_board* function can be improved, however, is not necessary to change in order to test the game out.



## Task 3 - Implementing Draws

As we know, there is draw if neither players have won, and the board is full. We have checked if the board is full by using two nested for loops. The outer for loop iterates over the rows of the board, and the inner for loop iterates over the elements of each row. If an element is *None*, then we can return *False* straight away. If we reach the end of the function without returning, then the board must be full, so we return *True*.



```
task-4.py - /Users/graham/Downloads/task-4.py (3.6.0)

def is_draw(board):
    """
    Return True if neither player
    has won and the board is full.
    """
    if has_won(board, 'X') or has_won(board, 'O'):
        return False
    else:
        return is_full(board)

def is_full(board):
    """
    Return True if game board is full.
    """
    for row in board:
        for el in row:
            if el == None:
                return False
    return True

Ln: 48 Col: 0
```

This is not the only possible implementation. More advanced Python users might check membership of *None* in a row using the “in” operator, but should be careful to remember the nested structure of the lists. There are other ways to iterate over the elements of the board too, however, the implementation we’ve given here closest matches what would be expected from pupils, having done this course.

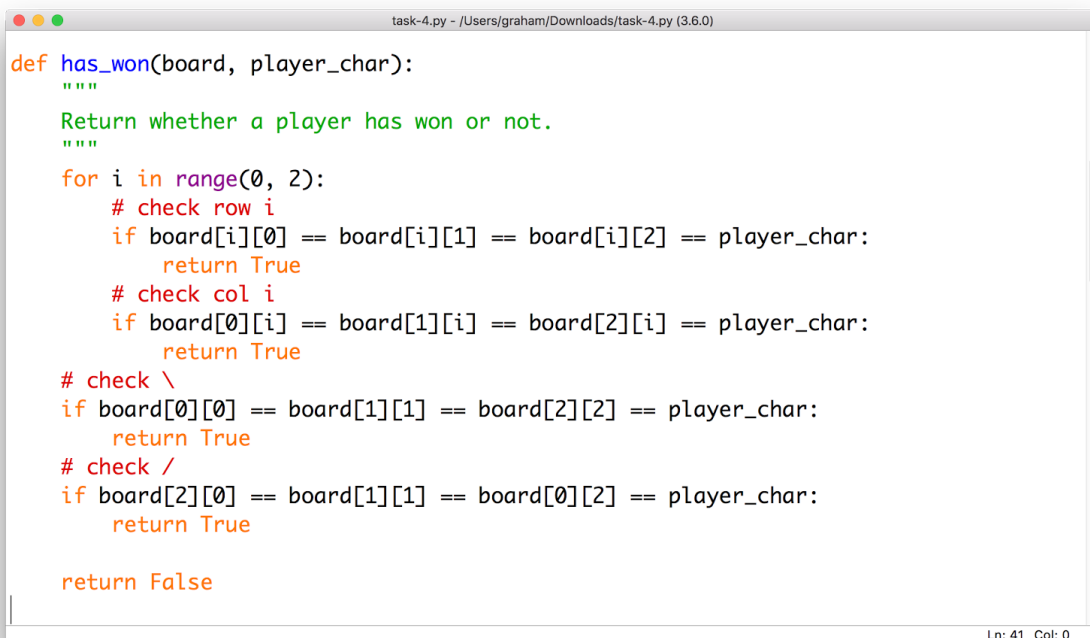
## Task 4 - Implementing Winners

The implementation of this function is quite non-trivial and requires some thought. This task is expected to take up to half an hour by itself, or potentially even longer.

We need to consider the ways in which a user could win:

1. By taking a row
2. By taking a column
3. By taking either diagonal

There are many ways of implementing this, of various sophistications. Some pupils may choose to write code that checks each of the 8 cases separately, however we'd hope that pupils spot the opportunity to use for loops and write code that checks the 3 rows, then the 3 columns, then each diagonal separately. We observe that the board is square, so we only need one for loop for both row and columns.



```
def has_won(board, player_char):  
    """  
    Return whether a player has won or not.  
    """  
    for i in range(0, 2):  
        # check row i  
        if board[i][0] == board[i][1] == board[i][2] == player_char:  
            return True  
        # check col i  
        if board[0][i] == board[1][i] == board[2][i] == player_char:  
            return True  
    # check \  
    if board[0][0] == board[1][1] == board[2][2] == player_char:  
        return True  
    # check /  
    if board[2][0] == board[1][1] == board[0][2] == player_char:  
        return True  
  
    return False
```

Ln: 41 Col: 0

---

## Task 5 - Testing the Code

Pupils check their code, and may have to go back to correct any errors they've made.

## Task 6 - Pretty Printing

This task was intentionally left open ended. Anything that looks more like board than just printing the raw data structure is an acceptable answer.

