README.md

# Assignment5 - Tic-tac-toe

## BACKGROUND

Tic-tac-toe is a classic, well-known game. Its popularity can partly be attributed to its simplicity. While commercial boards with plastic pieces can be purchased, two players only need a piece of paper and a pen or pencil to play the game. These simple rules, also allow for this game to be easily programmed with a computer. Creating a Tic-tac-toe program that allows two players to play the game is the purpose of this assignment. This assignment will be your first exposure to using dictionaries in your programs. However, you will also have the opportunity to fine-tune your skills applying conditional logic, using loops, handling input/output, and working with variables within a program.

The basics of the game should be familiar. The game starts with a blank board such as the one below:

| | | |
|---|---|---|
| (0,0) | (0,1) | (0,2) |
| (1,0) | (1,1) | (1,2) |
| (2,0) | (2,1) | (2,2) |

The numbers in each cell represent the coordinates (row, column) of each position on the Tic-tac-toe board. For example, the position in the top-left corner represents row 0 and column 0. The position in the middle of the board is at row 1 and column 1. The position in the bottom-right corner represents row 2 and column 2. For this program, these positions are represented in Python by using a tuple. The position on the board at row 1 and column 1 can be represented by the tuple `(1, 1)`. The position on the board at row 0 and column 0 can be represented by the tuple `(0, 0)`. And the position on the board at row 2 and column 2 can be represented by the tuple `(2, 2)`.

When your Tic-tac-toe program is executed, the user will see

```
Let's Play Tic-tac-toe!


     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row:
```

The blank board and prompt for player `X` to choose a row represents round 0 of the game. Each game of Tic-tac-toe is divided into 9 total rounds due to the 9 possible positions where a mark can be set on the board. While 9 rounds are possible in a game, a winning configuration achieved by player `X` or `O` may end the game before all rounds are completed. Player `X` will always go first (in the initial round of the game) in this version of the game. After this player enters a row value, the player will then be prompted to enter a column:

```
Choose your column:
```

When the player enters a valid row (between 0 and 2) and valid column (between 0 and 2) that corresponds to a position on the board that is currently empty, an `X` will be placed in that position on the board. For example, if `X` chooses row 2 and column 1, the following will be displayed on the screen:

```
Let's Play Tic-tac-toe!


     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   |X|

O,
Choose your row:
```

The output of the updated board and the prompt for player `O`'s row choice represents the start of the next round. In this round, player `O` will then be able to choose a row and column.

```
Let's Play Tic-tac-toe!


     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   |X|

O,
Choose your row: 1
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   |O|
    --+-+--
  2   |X|

X,
Choose your row:
```

In the round 1 of the game displayed above, player `O` chose row 1 and column 1 completing the round. The display of the updated board and prompt for player `X`'s choice represents the start of round 2 of the game. The player's alternate taking turns until all positions on the board are filled (9 rounds are completed) or one of the players is able to place three consecutive marks on the board in a horizontal, vertical, or diagonal orientation.

If a player enters a row or column value that is either not on the board (a value other than 0, 1, or 2) or a (row, column) position that is already occupied, the prompt for a row **and** column will be repeated until valid input is provided. For example, from the actions described above, if player `X` attempts to choose the same position as in player `X`'s first turn, the prompt to enter a row will re-appear. In this case, `X` will need to enter a **new** row **and new** column value. These prompts will continue to be displayed until the user provides valid input. An example of the scenario that was just described is illustrated below:

```
Let's Play Tic-tac-toe!


     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   |X|

O,
Choose your row: 1
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   |O|
    --+-+--
  2   |X|

X,
Choose your row: 2
Choose your column: 1

Choose your row:
```

At the start of each round, one player will be allowed to choose a position on the board until one of two situations occurs:

1. The board is occupied such that no additional empty positions are available. For example:

```
     0 1 2
  0  X|O|X
    --+-+--
  1  O|O|X
    --+-+--
  2  X|X|O
```

When this state is reached and neither player has placed the same mark in three consecutive positions on the board, the program will output:

```
It's a draw!
```

2. One of the player's manages to place three marks (`X` or `O`) in three consecutive positions on the board.

The winner will be displayed in this case as either:

```
X wins!
```

when three `X`'s have been placed in three consecutive positions or:

```
O wins!
```

when three `O`'s have been placed in three consecutive positions.

Your program should, ultimately, allow multiple games of Tic-tac-toe to be played. After every game is completed, the user will be prompted for a choice to play again. In the example below, player `X` has won the game and the user is presented with the play again prompt. If the user does not respond with a proper input of `Y` (or `y`) so that a new game is started or `N` (or `n`) so that the program ends, the question will continue to be displayed until valid input is provided:

```
     0 1 2
  0   |O|
    --+-+--
  1   |O|
    --+-+--
  2  X|X|X

X wins!

Play again (Y/N)? G
Play again (Y/N)? 2
Play again (Y/N)? s
Play again (Y/N)? Y
```

Only in the case that the player has entered a valid input (`Y`, in this case) will the board be reset and another game of Tic-tac-toe started:

```

     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row:
```

Notice that the welcome message is not displayed when a new game is started once your program has begun. When the user indicates no desire to play another game, `Goodbye.` is displayed before the program ends.

```
     0 1 2
  0   |O|
    --+-+--
  1   |O|
    --+-+--
  2  X|X|X

  X wins!

  Play again (Y/N)? n

  Goodbye.
```

**Winning Configurations**

There are 8 possible winning configurations:

**0th row win**

```
   0 1 2
0  -|-|-
  --+-+--
1   | |
  --+-+--
2   | |
```

**1st row win**

```
   0 1 2
0   | |
  --+-+--
1  -|-|-
  --+-+--
2   | |
```

**2nd row win**

```
   0 1 2
0   | |
  --+-+--
1   | |
  --+-+--
2  -|-|-
```

**0th col win**

```
   0 1 2
0  -| |
  --+-+--
1  -| |
  --+-+--
2  -| |
```

**1st col win**

```
   0 1 2
0   |-|
  --+-+--
1   |-|
  --+-+--
2   |-|
```

**2nd col win**

```
   0 1 2
0   | |-
  --+-+--
1   | |-
  --+-+--
2   | |-
```

**left-right diagonal win**

```
   0 1 2
0  -| |
  --+-+--
1   |-|
  --+-+--
2   | |-
```

**right-left diagonal win**

```
   0 1 2
0   | |-
  --+-+--
1   |-|
  --+-+--
2  -| |
```

You will need to implement a function that can determine if any of these winning configurations have been achieved by one of the player's in the game.

Detailed instructions for how to implement this program are below.

Let's get started.

## TABLE OF CONTENTS

## STEP 0 - READ THE README

Before writing a single line of Python code for this assignment, you are **strongly** encouraged to read this document in its entirety. The document includes key details about the expected implementation of your program. Diving into the implementation based solely on the description of the program above is likely to result in following a path towards your program's implementation that fails to meet the requirements of this assignment.

If you would like to consume this document in an alternative format, the README has also been converted to a PDF which is available in this repository.

## DECOMPOSITION

In order to make implementation of this program more manageable, the program has been decomposed into a group of functions that solve specific sub-problems for a Tic-tac-toe program. The instructions that follow guide you through the implementation of these functions that will result in a solution that meets the requirements of this assignment.

## INSTRUCTIONS

A single game of Tic-tac-toe is run as a series of rounds. At **most**, a single game will consist of 9 rounds. A constant has been defined in the starter code named `MAX_ROUNDS` that has been assigned this value. This constant will be useful when iterating through each round of the game.

### The Board

While we visualize the Tic-tac-toe board as a 3x3 grid of positions,

| | | |
|---|---|---|
| (0,0) | (0,1) | (0,2) |
| (1,0) | (1,1) | (1,2) |
| (2,0) | (2,1) | (2,2) |

the internal representation of the board is different. The Tic-tac-toe board is represented by a Python dictionary of key-value pairs. Each key for the dictionary is a tuple representing `(row, col)` coordinates where `row` and `col` are integer values between `0` and `2` (inclusive) representing a position on the Tic-tac-toe board. The board is initialized so that each of the 9 values in the dictionary is assigned the space (`' '`) character. When all board positions have been assigned the empty string (`' '`) as a value, the board can be considered empty. The space character is replaced by an `X` or an `O` when the board position is chosen by the respective player (player `X` or player `O`). The key point to understand for this representation of the board is that a position in the (*row*, *column*) coordinate format corresponds directly to a position on the board.

### Step A - Resetting the board

Function: `reset_board(board)`

```python
def reset_board(board):
    """Resets the board dict to its original state with each
    position being empty (i.e. the (row, column) key has a
    space character (' ') value).

    :param board: a dict of (row, column) tuple keys and string values
    :return: None

    >>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): '
    >>> reset_board(board)
    >>> board
    {(0, 0): ' ', (0, 1): ' ', (0, 2): ' ', (1, 0): ' ', (1, 1): ' ', (1, 2): ' ', (2, 0): ' ', (2, 1): ' ', (2, 2): '
    """
```

While resetting the board is not a task that is required until a game has been completed, it is a good place to start implementing this program as it provides an opportunity to become familiar with the Python `dict` representing the board. The `reset_board()` function has one parameter: `board`. `board` is a dictionary with 9 key-value pairs where the key is a 2-element `tuple` representing a (*row*, *column*) position on the `board`. Each associated value is a string representing the `mark` present at that position on the `board`. The empty string (`' '`) is used when no `mark` has been placed at a position. While there are a few different ways to implement `reset_board()`, be sure to use **one or more loops** in the definition of this function such that the `board` dictionary passed as an argument to this function has a space character (`' '`) as the `value` for each (*row*, *column*) `key`. After a function call to `reset_board()` occurs, no values that are either `X` or `O` will be present in the `board` dictionary.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> reset_board(board)

>>> print(board)
{(0, 0): ' ', (0, 1): ' ', (0, 2): ' ', (1, 0): ' ', (1, 1): ' ', (1, 2): ' ', (2, 0): ' ', (2, 1): ' ', (2, 2): ' '}
```

Once `reset_board()` has been implemented, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step A** tests. Once your program passes the **Step A** tests, proceed to **Step B**.

### Step B - Determine current player for round

Function: `get_current_player(round)`

```python
def get_current_player(round):
    """Returns the mark of the player whose turn it is in the current
    round of the game.

    :param round: integer value representing the round
    :return: 'X' or 'O' depending on round

    >>> round =  3
    >>> get_current_player(round)
    'O'
    >>> round = 8
    >>> get_current_player(round)
    'X'
    """
```

The `get_current_player()` function is used to determine the player whose turn it is in the current round. The `get_current_player()` function has 1 parameter: `round`. `round` is an integer. `get_current_player()` returns `X` when `round` is an even integer

```
>>> round = 8
>>> get_current_player(round)
'X'
```

and `O` when `round` is an odd integer.

```
>>> round = 3
>>> get_current_player(round)
'O'
```

After implementing `get_current_player()`, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step B** tests. Once your program passes the **Step B** tests, proceed to **Step C**.

### Step C - Get player position choice

Function: `get_position_choice(board, player_mark)`

```python
def get_position_choice(board, player_mark):
    """Prompts the user for a valid (row, col) board position. Prompts
    for row and column are repeated until valid position provided. The
    valid (row, col) position chosen is returned.

    :param board: a dict of (row, col) tuple keys and string values
    :param player_mark: 'X' or 'O' depending on round
    :return: (row, col) tuple of integers representing valid position choice
    """
```

With the ability to identify which player is making a position choice in the current round, you can now implement the `get_position_choice()` function. `get_position_choice()` has 2 parameters: `board` and `player_mark`. `board` is the Python `dict` representing the Tic-tac-toe board. `player_mark` is the string `X` or `O` representing the mark for the player making the position choice. The `get_position_choice()` function returns a `(row, col)` (**both integers**) tuple representing a valid position choice made by the player.

A `get_position_choice()` function call outputs the player's mark (followed by a comma, `,`) and a prompt for a row choice:

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): ' ', (2, 1): ' ', (
>>> player_mark = 'X'
>>> get_position_choice(board, player_mark)
X,
Choose your row:
```

The player's mark displayed would change in the case that player `O` is making a position choice.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row:
```

What does not change for each `get_position_choice()` function call is that a comma follows the display of the player's mark and the player is prompted for a row choice as the second line output by a `get_position_choice()` function call. Prompts for both the row and column choice are displayed (along with **a blank line**) before the choice is checked to see if the position choice is valid.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 1
Choose your column: 1

(1, 1)
```

In this case, because the `row` choice (`1`) is in the inclusive range [0, NUM_ROWS - 1] and the `column` choice (`1`) is in the inclusive range [0, NUM_COLS - 1] and position `(1, 1)` is empty on the board, this position choice is valid. As a result, the position choice is returned by the function. Consider the case where the user enters a row value outside of the valid range of row values.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 5
Choose your row:
```

In this situation, the row choice prompt is re-displayed. The row choice prompt will continue to be re-displayed until the player enters a row value that is in the correct range of values.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,

Choose your row: 5
Choose your row: -3
Choose your row:
```

When a valid row value is entered, the column choice prompt is displayed.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 5
Choose your row: -3
Choose your row: 2
Choose your column:
```

Again, the player must enter a column value that is within the range of valid values ([0, NUM_COLS - 1]). If not, the column choice prompt is re-displayed:

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 5
Choose your row: -3
Choose your row: 0
Choose your column: 16
Choose your column:
```

Only after a valid column value is entered is the `(row, col)` position checked to ensure that the position on the board is empty.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 5
Choose your row: -3
Choose your row: 0
Choose your column: 16
Choose your column: 0

Choose your row:
```

In this case, player `O` has entered a position `(0, 0)` that is not empty as player `X` has already chosen this position in a previous round. Therefore, player `O` will need to enter a new row and column choice.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)
O,
Choose your row: 5
Choose your row: -3
Choose your row: 0
Choose your column: 16
Choose your column: 0

Choose your row: 2
Choose your column: 0

Choose your row:
```

Again, player `O` has chosen a position `(2, 0)` that is not empty, so the row choice prompt is re-displayed. Only after the player has entered a position that includes row and column values that are in the correct range **and** represent an empty position on the board does the `get_position_choice()` function call return that position as a `tuple`.

```
>>> board = {(0, 0): 'X', (0, 1): ' ', (0, 2): ' ', (1, 0): 'O', (1, 1): ' ', (1, 2): 'O', (2, 0): 'X', (2, 1): ' ', (
>>> player_mark = 'O'
>>> get_position_choice(board, player_mark)

O,
Choose your row: 5
Choose your row: -3
Choose your row: 0
Choose your column: 16
Choose your column: 0

Choose your row: 2
Choose your column: 0

Choose your row: 0
Choose your column: 2

(0, 2)
```

Define your `get_position_choice()` function such that this behavior is exhibited when invalid `row` and `column`, and `position` choices are entered.

After implementing `get_position_choice()`, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step C** tests. Once your program passes the **Step C** tests, proceed to **Step D**.

### Step D - Update the board based on player choice

Function: `update_board(board, mark, position)`

```python
def update_board(board, player_mark, position):
    """Updates the value at the key represented by position
    in board dictionary to player_mark.

    :param board: a dict of (row, col) tuple keys and string values
    :param player_mark: 'X' or 'O' depending on round
    :param position: (row, col) tuple representing position
    :return: None
    """
```

Having obtained the player's choice of position, the board can be updated. The `update_board()` function has 3 parameters: `board`, `player_mark`, and `position`. Define `update_board()` such that after a function call to `update_board()`, the value for the key `position` in `board` is the mark for the player in the current round indicated by `player_mark`.

```
>>> board = {(0, 0): ' ', (0, 1): ' ', (0, 2): ' ', (1, 0): ' ', (1, 1): ' ', (1, 2): ' ', (2, 0): ' ', (2, 1): ' ', (
>>> player_mark = 'O'
>>> pos = (1, 2)
>>> board[pos]
' '
>>> update_board(board, player_mark, pos)
>>> board[pos]
'O'
```

After defining `update_board()`, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step D** tests. Once your program passes the **Step D** tests, proceed to **Step E**.

#### Step E - Display the outcome of the game

Function: `display_outcome(round)`

```
def display_outcome(round):
    """Displays an outcome message for a completed Tic-tac-toe game.

    :param round: the final value of the round variable for the game
    :return: None

    >>> round = MAX_ROUNDS
    >>> display_outcome(round)
    It's a draw!
    <BLANKLINE>
    >>> round = 6
    >>> display_outcome(round)
    X wins!
    <BLANKLINE>
    >>> round = 5
    >>> display_outcome(round)
    O wins!
    <BLANKLINE>
    """
```

There are three possible outcomes to any complete Tic-tac-toe game: a win by `X`, a win by `O`, or a draw. The `display_outcome()` function has 1 parameter: `round`. When the value passed as an argument to `round` is equal to `MAX_ROUNDS`, a `display_outcome()` function call will output `It's a draw!`.

```
>>> round = MAX_ROUNDS
>>> display_outcome(round)
It's a draw!
<BLANKLINE>
```

In the case that the value passed to round is less than `MAX_ROUNDS` and even, the `display_outcome()` function call will output `X wins!`.

```
>>> round = 6
>>> display_outcome(round)
X wins!
<BLANKLINE>
```

In the case that the value passed to round is less than `MAX_ROUNDS` and odd, the `display_outcome()` function call will output `O wins!`.

```
>>> round = 5
>>> display_outcome(round)
O wins!
<BLANKLINE>
```

Use a `get_current_player()` function call in this function to output the correct winner when the game does not result in a draw.

After `display_outcome()` has been defined, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step E** tests. Once your program passes the **Step E** tests, proceed to **Step F**.

### Step F - Check for a winning configuration

Function: `check_positions(pos1_value, pos2_value, pos3_value)`

```
def check_positions(pos1_value, pos2_value, pos3_value):
    """Returns True when all parameters have a value of 'X' or
    all parameters have a value of 'O'. Returns False for all
    other value combinations.

    :param pos1_value: the first of 3 consecutive board position values
    :param pos2_value: the second of 3 consecutive board position values
    :param pos3_value: the third of 3 consecutive board position values
    :return: True when all 3 values are 'X' or when all 3 values are 'O', False otherwise

    >>> (pos_val1, pos_va2, pos_val3)  = ('X', 'X', 'X')
    >>> check_positions(pos_val1, pos_va2, pos_val3)
    True
    >>> (pos_val1, pos_val2, pos_val3)  = ('O', 'O', 'O')
    >>> check_positions(pos_val1, pos_val2, pos_val3)
    True
    >>> (pos_val1, pos_val2, pos_val3)  = (' ', ' ', ' ')
    >>> check_positions(pos_val1, pos_val2, pos_val3)
    False
    >>> (pos_val1, pos_val2, pos_val3)  = ('O', 'X', 'O')
    >>> check_positions(pos_val1, pos_val2, pos_val3)
    False
    >>> (pos_val1, pos_val2, pos_val3)  = ('X', 'X', ' ')
    >>> check_positions(pos_val1, pos_val2, pos_val3)
    False
    """
```

At this point in the development of your program, it is time to determine if the current game of Tic-tac-toe has a winner. A winning configuration for a game of Tic-tac-toe is achieved when a player's mark has been placed in 3 consecutive positions on the board. The `check_positions()` function is used to determine if 3 consecutive board positions contain the same player mark. The function has 3 parameters: `pos1_value`, `pos2_value`, and `pos3_value`. When the board values passed to these 3 parameters are all `X`, the function returns `True`.

```
>>> (pos_val1, pos_val2, pos_val3)  = ('X', 'X', 'X')
>>> check_positions(pos_val1, pos_val2, pos_val3)
True
```

The function also returns `True` when the board values passed to these 3 parameters are all `O`.

```
>>> (pos_val1, pos_val2, pos_val3)  = ('O', 'O', 'O')
>>> check_positions(pos_val1, pos_val2, pos_val3)
True
```

Any other set of board values that are passed as the values for the 3 parameters for a `check_positions()` function call will return `False`.

```
>>> (pos_val1, pos_val2, pos_val3)  = (' ', ' ', ' ')
>>> check_positions(pos_val1, pos_val2, pos_val3)
False
```

```
>>> (pos_val1, pos_val2, pos_val3)  = ('O', 'X', 'O')
>>> check_positions(pos_val1, pos_val2, pos_val3)
False
```

```
>>> (pos_val1, pos_val2, pos_val3)  = ('X', 'X', ' ')
>>> check_positions(pos_val1, pos_val2, pos_val3)
False
```

When using `check_positions` in your program, it is your responsibility to ensure that the values passed to the function call represent the values at 3 **consecutive** positions on the board.

When `check_positions()` has been implemented, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step F** tests. Once your program passes the **Step F** tests, proceed to **Step G**.

### Step G - Check if the game has ended

Function: `is_game_complete(board)`

```python
def is_game_complete(board):
    """Determines whether or not a winning configuration has been achieved in the game
    represented by the board. Returns True when a winning configuration is detected and
    False when no winning configuration exists on the board.

    :param board: a dict of (row, col) tuple keys and string values
    :return: True when a winning configuration is detected, False otherwise

    >>> board = {(0, 0): ' ', (0, 1): ' ', (0, 2): ' ', (1, 0): ' ', (1, 1): 'X', (1, 2): ' ', (2, 0): ' ', (2, 1): '
    >>> is_game_complete(board)
    False
    >>> board = {(0, 0): 'X', (0, 1): 'X', (0, 2): 'O', (1, 0): 'O', (1, 1): 'O', (1, 2): 'X', (2, 0): 'X', (2, 1): 'X
    >>> is_game_complete(board)
    False
    >>> board = {(0, 0): ' ', (0, 1): 'O', (0, 2): ' ', (1, 0): ' ', (1, 1): 'O', (1, 2): ' ', (2, 0): 'X', (2, 1): 'X
    >>> is_game_complete(board)
    True
    >>> board = {(0, 0): 'O', (0, 1): ' ', (0, 2): 'X', (1, 0): 'X', (1, 1): 'O', (1, 2): 'X', (2, 0): ' ', (2, 1): '
    >>> is_game_complete(board)
    True
    """
```

With `check_positions()` now defined, `is_game_complete()` can be defined for testing whether a winning configuration has occurred in the current game. `is_game_complete()` has 1 parameter: `board`. `is_game_complete()` has the potential to make 8 individual function calls to `check_positions()` (depending on the board configuration) when an `is_game_complete()` function call is made. Define `is_game_complete()` such that detecting any of the winning configurations shown in the *Background* section of this README will cause an `is_game_complete()` function call to return a value of `True`. If after testing for all 8 winning configurations, none are found, the `is_game_complete()` function call will return `False`.

Two examples of `board`s for which `is_game_complete()` will return `False` are shown below:

```
>>> board = {(0, 0): ' ', (0, 1): ' ', (0, 2): ' ', (1, 0): ' ', (1, 1): 'X', (1, 2): ' ', (2, 0): ' ', (2, 1): ' ', (
>>> is_game_complete(board)
False
>>> board = {(0, 0): 'X', (0, 1): 'X', (0, 2): 'O', (1, 0): 'O', (1, 1): 'O', (1, 2): 'X', (2, 0): 'X', (2, 1): 'X', (
>>> is_game_complete(board)
False
```

Two examples of `board`s for which `is_game_complete()` will return `True` are shown below:

```
>>> board = {(0, 0): ' ', (0, 1): 'O', (0, 2): ' ', (1, 0): ' ', (1, 1): 'O', (1, 2): ' ', (2, 0): 'X', (2, 1): 'X', (
>>> is_game_complete(board)
True
>>> board = {(0, 0): 'O', (0, 1): ' ', (0, 2): 'X', (1, 0): 'X', (1, 1): 'O', (1, 2): 'X', (2, 0): ' ', (2, 1): ' ', (
>>> is_game_complete(board)
True
```

There are many `board` configurations that can result in a game being complete. You should define `is_game_complete()` in a way which detects any winning configuration and returns `True` as soon as as the winning configuration is identified.

When you feel that your definition of `is_game_complete()` is defined properly, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step G** tests. Once your program passes the **Step G** tests, proceed to **Step H**.

#### Step H - Run a game of Tic-tac-toe

Function: `play_tic_tac_toe(board)`

```
def play_tic_tac_toe(board):
    """Controls Tic-tac-toe games. This includes prompting player's for
    position choices, checking for winning game configurations, and outputting
    the outcome of a game.

    :param board: a dict of (row, col) tuple keys and string values
    :return: None
    """
```

At this point in the program's development, you have implemented all of the required functions for executing a single game of Tic-tac-toe. The flow of the program will be controlled by the `play_tic_tac_toe()` function. This function requires one parameter: `board`. The value that you should use for this parameter is available in the program through the `dict` variable `board` which has been defined for you in the `main()` function of the starter code file `tic_tac_toe.py`.

A single game of Tic-tac-toe includes multiple rounds (the maximum number being `MAX_ROUNDS`) that perform the following tasks:

1. Display the current state of the board
2. Determine the current player for the round
3. Get a valid position from the current player
4. Update the board with the player’s mark at the chosen position
5. Check to see if the game is complete

When a game is complete, no additional rounds are to be executed, the final board configuration is displayed, and the outcome of the game is output. Again, the possible outcomes are a win by `X`, a win by `O`, or a draw.

The current state of the board is displayed by calling the function `display_board()`. The `display_board()` function has 1 parameter: `board` which is the Python `dict` representing the Tic-tac-toe board. This function has been implemented for you and only needs to be called from the `play_tic_tac_toe()` function.

Example output from a `play_tic_tac_toe()` function call at this step is shown below.

```
     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 1
Choose your column: 1
```

```
     0 1 2
  0   | |
    --+-+--
  1   |X|
    --+-+--
  2   | |

O,
Choose your row: 0
Choose your column: 0

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   |X|

O,
Choose your row: 0
Choose your column: 1

     0 1 2
  0  O|O|
    --+-+--
  1   |X|
    --+-+--
  2   |X|

X,
Choose your row: 0
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1   |X|
    --+-+--
  2   |X|

O,
Choose your row: 2
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1   |X|
    --+-+--
  2  O|X|

X,
Choose your row: 1
Choose your column: 2

     0 1 2
  0  O|O|X
```

```
    --+-+--
  1   |X|X
    --+-+--
  2  O|X|

O,
Choose your row: 1
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1  O|X|X
    --+-+--
  2  O|X|

O wins!
```

The example demonstrates a game where the player using `O` wins in round 7 (with round 0 being the initial round). Notice that after the position choice that results in three consecutive `O` marks being placed in column 0, the game immediately ends with a display of the final state of the board followed by the outcome message.

An example of a game resulting in a draw is displayed below.

```
     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 1
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   |X|
    --+-+--
  2   | |

O,
Choose your row: 0
Choose your column: 0

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   |X|
O,
Choose your row: 0
Choose your column: 1

     0 1 2
  0  O|O| 
    --+-+--
  1   |X| 
    --+-+--
  2   |X| 

X,
Choose your row: 0
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1   |X| 
    --+-+--
  2   |X| 

O,
Choose your row: 2
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1   |X| 
    --+-+--
  2  O|X| 

X,
Choose your row: 1
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1  X|X| 
    --+-+--
  2  O|X| 

O,
Choose your row: 1
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1  X|X|O
    --+-+--
  2  O|X| 

X,
Choose your row: 2
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1  X|X|O
    --+-+--
  2  O|X|X

It's a draw!
```

After this initial implementation of `play_tic_tac_toe()`, try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step H** tests. Once your program passes the **Step H** tests, proceed to **Step I**.

#### Step I - Allow user to end program

Function: `is_program_finished()`

```
def is_program_finished():
    """Prompts the user with the message "Play again (Y/N)?". The question is repeated
    until the user enters a valid response (one of Y/y/N/n). The function
    returns False if the user enters 'Y' or 'y' and returns True if the user
    enters 'N' or 'n'.

    :return response: boolean representing program completion status
    """
```

The user is given the chance to play multiple games of Tic-tac-toe during one program run. Therefore, at the completion of a game, the program needs to determine whether to start a new game or end the program. The `is_program_finished()` function has no parameters and returns `True` when the user indicates the desire to stop playing. `is_program_finished()` returns `False` when the user indicates the desire to play again. This function uses the prompt `Play again (Y/N)?` to determine if the user wants to continue playing or end the program. The user must enter `Y` or `y` to play again. The user enters `N` or `n` to end the program. Any other input from the user causes the prompt `Play again (Y/N)?` to be displayed again until a valid input value is provided by the user.

```
>>> is_program_finished()
Play again (Y/N)? w
Play again (Y/N)? p
Play again (Y/N)? y

False
```

Take note that a blank line is included before the function returns a value. The example above demonstrates that while the user is prompted to enter `Y` to play again, `y` is also accepted. When the user enters `n` or `N` the function will return `True`.

```
>>> is_program_finished()
Play again (Y/N)? n

True
```

After implementing the `is_program_finished()` function, all **Step I** tests should pass. Try submitting your `tic_tac_toe.py` file to Gradescope to see if your implementation passes the **Step I** tests. Once your program passes the **Step I** tests, proceed to **Step J**.

#### Step J - Allow multiple games of Tic-tac-toe

Function to update: `play_tic_tac_toe(board)`

At this point, your program can execute one complete game of Tic-tac-toe before the program ends. However, the final program requires that multiple games can be played. After completing **Step J**, your program will have this functionality. Specifically, your program will pass the tests for this step when the following occurs:

At the start of the program, `Let's Play Tic-tac-toe!` is output once by a `play_tic_tac_toe()` function call. The welcome message is **always** followed by a blank line. The welcome message is displayed only once during each execution of the program -- prior to the start of the initial game.

When the program ends, `Goodbye.` is the final message output by a `play_tic_tac_toe()` function call.

Update your definition of `play_tic_tac_toe()`, such that the following requirements are met:

1. At least one game of Tic-tac-toe must be run each time your program is executed.
2. After completing a game, the program asks the user whether or not to play again using the prompt `Play again (Y/N)?`. Make a function call to `is_program_finished()` for displaying this prompt.
3. When answering this question affirmatively (by entering `Y` or `y`), the board is reset and a new Tic-tac-toe game starts.
4. When the user indicates a desire to end the game (by entering `N` or `n`), the program displays `Goodbye.` before ending.

```
Let's Play Tic-tac-toe!

     0 1 2
  0   | |
    --+-+--
  1   | |
    --+-+--
  2   | |

X,
Choose your row: 1
Choose your column: 1

     0 1 2
  0   | |
    --+-+--
  1   |X|
    --+-+--
  2   | |

O,
Choose your row: 0
Choose your column: 0

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   |X|

O,
Choose your row: 0
Choose your column: 1

     0 1 2
  0  O|O|
    --+-+--
  1   |X|
    --+-+--
  2   |X|

X,
Choose your row: 0
Choose your column: 2

     0 1 2
  0  O|O|X
```

```
     --+-+--
  1    |X|
     --+-+--
  2    |X|

O,
Choose your row: 2
Choose your column: 0

      0 1 2
  0   O|O|X
     --+-+--
  1    |X|
     --+-+--
  2   O|X|

X,
Choose your row: 1
Choose your column: 0

      0 1 2
  0   O|O|X
     --+-+--
  1   X|X|
     --+-+--
  2   O|X|

O,
Choose your row: 1
Choose your column: 2

      0 1 2
  0   O|O|X
     --+-+--
  1   X|X|O
     --+-+--
  2   O|X|

X,
Choose your row: 2
Choose your column: 2

      0 1 2
  0   O|O|X
     --+-+--
  1   X|X|O
     --+-+--
  2   O|X|X

It's a draw!

Play again (Y/N)? Y

      0 1 2
  0    | |
     --+-+--
  1    | |
     --+-+--
  2    | |

X,
Choose your row: 1
Choose your column: 1

      0 1 2
  0    | |
     --+-+--
  1    |X|
    --+-+--
  2   | |

O,
Choose your row: 0
Choose your column: 0

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   | |

X,
Choose your row: 2
Choose your column: 1

     0 1 2
  0  O| |
    --+-+--
  1   |X|
    --+-+--
  2   |X|

O,
Choose your row: 0
Choose your column: 1

     0 1 2
  0  O|O|
    --+-+--
  1   |X|
    --+-+--
  2   |X|

X,
Choose your row: 0
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1   |X|
    --+-+--
  2   |X|

O,
Choose your row: 2
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1   |X|
    --+-+--
  2  O|X|

X,
Choose your row: 1
Choose your column: 2

     0 1 2
  0  O|O|X
    --+-+--
  1   |X|X
    --+-+--
  2  O|X|
O,
Choose your row: 1
Choose your column: 0

     0 1 2
  0  O|O|X
    --+-+--
  1  O|X|X
    --+-+--
  2  O|X|

O wins!

Play again (Y/N)? n

Goodbye.
```

Make sure to call `play_tic_tac_toe()` (with the provided `board` dictionary) from within the program's `main()` function.

When `play_tic_tac_toe()` has been updated to enable multiple rounds of Tic-tac-toe as demonstrated in this step, all **Step J** tests should pass. Try submitting your `tic_tac_toe.py` file to Gradescope to see if all **Step J** tests pass. This indicates that you program is functioning as expected. Congratulations!

## EVALUATION

This assignment will use Gradescope's Autograder tool. Therefore, you will receive immediate feedback when submitting this assignment. You should strive to have your submitted code pass all of the tests run by the Gradescope Autograder by the time you have completed **Step J**. When you have a program implementation that passes all of the Autograder tests, ensure that your program can be executed on its own (without producing any errors) and **behaves in the way described in this document**. This is a requirement for getting full credit on the assignment.

It is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment will be based on how well you follow certain coding conventions. Make sure to follow the standards discussed in class, and before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned below.

#### Comments

To make your program easier to read, you should add comments before (docstrings) and inside your functions to make your intention clearer. Good comments give the reader a clue about what a function does and, in some cases, how it works.

Be sure to include header comments (including your name) for the submitted file.

#### Function design

Function names should include verbs indicating what is accomplished by the function. Most of the functions in this assignment have been named for you. However, if you add any additional functions to further decompose your solution, be sure to name these functions appropriately:

- function names should include verbs
- all words in a function name should be written in lower-case letters
- multiple words in a function name should be separated by underscores

Functions should be decomposed such that the length of each is small. This is an important part of proper decomposition. There is a bit of an art to this but you should ensure that your functions solve well-defined sub-problems. The function definitions should not, in general, have 10s of lines of codes. This is especially true for the programs written in this class.

#### Function use

- Functions that are defined for the program should be used. If program behavior captured by a function is duplicated by code in another part of the program, your submission will be penalized.
- Functions should not include function calls to themselves (recursive definitions). This likely means that you need to use a loop in your program and are not doing so. Such recursive function definitions will be penalized.

#### Variable names

Variable naming conventions are as follows:

- names should be descriptive
- all characters in the name should be written in lowercase (with the exception of constant variables)
- multiple words should be separated by underscores ('_').

#### Spacing

Indentation should be consistent in the files that are submitted. The convention is that each block of indented code is indented using four spaces. Your program should follow this convention.

Spaces should be used between operators and operands in the expressions written in your code.

#### Magic Numbers

Your program should not contain any constant numeric values. Constant variables are to be defined and used in place of hard-coding numeric constants into the program.

## ASSIGNMENT SUBMISSION

You should submit the following file on Gradescope (do not include any files not included in the list below):

- **tic_tac_toe.py**

The Autograder will assign points to passed tests. These results should be used as a guide for keeping you on track towards an implementation of this assignment that meets all of the requirements laid out above.

Good luck!