Using Python for Research Homework: Week 2

In this homework, we will use the tools we've covered in the past two weeks to create a tic-tac-toe (noughts and crosses) simulator and evaluate basic winning strategies.

Exercise 1

Tic-tac-toe (or noughts and crosses) is a simple strategy game in which two players take turns placing a mark on a 3x3 board, attempting to make a row, column, or diagonal of three with their mark. In this homework, we will use the tools we've covered in the past two weeks to create a tic-tac-toe simulator and evaluate basic winning strategies.

In the following exercises, we will learn to create a tic-tac-toe board, place markers on the board, evaluate if either player has won, and use this to simulate two basic strategies.

Instructions

- For our tic-tac-toe board, we will use a numpy array with dimension 3 by 3.
- Make a function `create_board()` that creates such a board with the value of each cell set to the integer `0`.
- Call `create_board()` and store it.

In [1]:

```python
import numpy as np

def create_board():
    board = np.zeros((3,3), dtype=int)
    return board
```

Exercise 2

Players 1 and 2 will take turns changing values of this array from a 0 to a 1 or 2, indicating the number of the player who places a marker there.

Instructions

- Create a function `place(board, player, position)`, where:
  - `player` is the current player (an integer 1 or 2).
  - `position` is a tuple of length 2 specifying a desired location to place their marker.
  - Your function should only allow the current player to place a marker on the board (change the board position to their number) if that position is empty (zero).
- Use `create_board()` to store a board as `board`, and use `place` to have Player 1 place a marker on location `(0, 0)`.

In [3]:

```python
def place(board, player, position):
    if board[position] == 0:
        board[position] = player
        return board

board = create_board()
place(board, 1, (0, 0))
```

Out[3]:

```
array([[1, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Exercise 3

In this exercise, we will determine which positions are available to either player for placing their marker.

Instructions

- Create a function `possibilities(board)` that returns a list of all positions (tuples) on the board that are not occupied (0). (Hint: `numpy.where` is a handy function that returns a list of indices that meet a condition.)
- `board` is already defined from previous exercises. Call `possibilities(board)` to see what it returns!

In [4]:

```
def possibilities(board):
    not_occupied = np.where(board == 0)
    return list(zip(not_occupied[0], not_occupied[1]))

print(possibilities(board))
```

```
[(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Exercise 4

The next step is for the current player to place a marker among the available positions. In this exercise, we will select an available board position at random and place a marker there.

Instructions

- Write a function `random_place(board, player)` that places a marker for the current player at random among all the available positions (those currently set to 0).
  - Find possible placements with `possibilities(board)`.
  - Select one possible placement at random using `random.choice(selection)`.
- `board` is already defined from previous exercises. Call `random_place(board, player)` to place a random marker for Player 2, and store this as board to update its value.

In [5]:

```
import random
random.seed(1)

def random_place(board, player):
    selections = possibilities(board)
    if len(selections) > 0:
        selection = random.choice(selections)
        place(board, player, selection)
    return board

random_place(board, 2)
```

Out[5]:

```
array([[1, 0, 0],
       [2, 0, 0],
       [0, 0, 0]])
```

Exercise 5

We will now have both players place three markers each.

Instructions

- A new `board` is already given. Call `random_place(board, player)` to place three pieces each on board for players 1 and 2.
- Print board to see your result.

In [6]:

```
random.seed(1)
board = create_board()
for i in range(3):
    for player in [1, 2]:
        random_place(board, player)

#Print board to see your result

print(board)
```

```
[[2 2 1]
 [0 1 0]
 [0 1 2]]
```

Exercise 6

In the next few exercises, we will make functions that check whether either player has won the game.

Instructions

- Make a function `row_win(board, player)` that takes the player (integer) and determines if any row consists of only their marker.
  - Have it return `True` if this condition is met and `False` otherwise.
- `board` is already defined from previous exercises. Call `row_win` to check if Player 1 has a complete row.

In [7]:

```
def row_win(board, player):
    if np.any(np.all(board==player,axis=1)): # this checks if any row contains all
        return True
    else:
        return False

row_win(board, 1)
```

Out[7]:

```
False
```

Exercise 7

In the next few exercises, we will make functions that verify if either player has won the game.

Instructions

- Make a function `col_win(board, player)` that takes the player (integer) and determines if any column consists of only their marker.
  - Have it return `True` if this condition is met and `False` otherwise.
- `board` is already defined from previous exercises. Call `col_win` to check if Player 1 has a complete row.

In [8]:

```
def col_win(board, player):
    if np.any(np.all(board==player,axis=0)): # this checks if any row contains all
        return True
    else:
        return False

col_win(board, 1)
```

Out[8]:

```
False
```

Exercise 8

In the next few exercises, we will make functions that verify if either player has won the game.

Instructions

- Finally, create a function `diag_win(board, player)` that tests if either diagonal of the board consists of only their marker. Have it return `True` if this condition is met, and `False` otherwise.
- `board` has been slightly modified from a previous exercise. Call `diag_win` to check if Player 2 has a complete diagonal.

In [10]:

```
board[1,1] = 2

def diag_win(board, player):
    if np.any(np.diag(board)==player) or np.all(np.diag(np.fliplr(board))==player):
        return True
    else:
        return False

diag_win(board, 2)
```

Out[10]:

```
True
```

Exercise 9

In the next few exercises, we will make functions that check whether either player has won the game.

Instructions

- Create a function `evaluate(board)` that uses `row_win`, `col_win`, and `diag_win` functions for both players. If one of them has won, return that player's number. If the board is full but no one has won, return -1. Otherwise, return 0.

- `board` is already defined from previous exercises. Call evaluate to see if either player has won the game yet.

In [18]:
```
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        # add your code here!
        pass
    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

evaluate(board)
```

Out[18]:
```
0
```

Exercise 10

In this exercise, we will use all the functions we have made to simulate an entire game.

Instructions

- `create_board()`, `random_place(board, player)`, and `evaluate(board)` have been created in previous exercises. Create a function `play_game()` that:
  - Creates a board.
  - Alternates taking turns between two players (beginning with Player 1), placing a marker during each turn.
  - Evaluates the board for a winner after each placement.
  - Continues the game until one player wins (returning 1 or 2 to reflect the winning player), or the game is a draw (returning -1).
- Call play_game 1000 times, and store the results of the game in a list called `results`.

In [21]:
```
random.seed(1)
results = []

def play_game():
    board = create_board()
    for i in range(2):
        random_place(board, i)
    evaluate(board)
    results = results.append(play_game())
    return results

results
```

Out[21]:
```
[]
```

Exercise 11

In the previous exercise, we see that when guessing at random, it's better to go first, as expected. Let's see if Player 1 can improve their strategy.

Instructions

- Create a function `play_strategic_game()`, where Player 1 always starts with the middle square, and otherwise both players place their markers randomly.
- Call `play_strategic_game` 1000 times.

In [ ]:

```
random.seed(1)

# write your code here
```