Remembering Python Fundamentals

Building a console tic tac toe game

Setting up the board and player moves

First, we need a board to play.

Let's assume that empty spots in the board will be 0's, player one will be 1's and player two will be 2's.

Here's a way to display the board in the console:

In [3]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

for row in game:
    print(row)
```

```
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
```

Next up, how will players tell where they'll want to play? Why not by defining coordinates?

So, let's find a way to let them know our board's available coordinates:

In [4]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],]

print("   0  1  2")

for count, row in enumerate(game):
    print(count, row)
```

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

Ok, now players can decide where they want to play by giving us a pair of numbers: one for the column number and another for the row number.

When they provide such numbers, we can use indexes to mark their play in the board. Here's an example:

In [5]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

game[0][1] = 2

print("   0  1  2")

for count, row in enumerate(game):
    print(count, row)
```

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

Applying the DRY principle

Great, so that's how we can actually set a value. Now we're almost ready to start playing on the map, but, every time we do that, we're going to be re-printing out the game after a player plays. That's going to be a lot of repeated code, so how do we eliminate that need? Yep, functions are the answer.

This is our new code:

In [6]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

def game_board():
    print("   0  1  2")
    for count, row in enumerate(game):
        print(count, row)
```

We start the game with:

In [7]:

```
game_board()
```

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

And everytime someone makes a move, we just refresh the board:

In [8]:

```
game[0][1] = 2

game_board()
```

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

Hmm, so now we always have to refresh the board after a move? That's also boring.

What if we add arguments to our function to pass every move? Let's look into it.

In [9]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

def game_board(player, row, column):
    game[row][column] = player
    print("   0  1  2")
    for count, row in enumerate(game):
        print(count, row)

game_board(player=2, row=0, column=1)
```

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

It works! But how do we show an empty board on game start?

We could do this:

In [10]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

game_board(player=0, row=0, column=0)
```

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

But we're smarter than that. Let's set defaults for all arguments, so if we don't specify them it means we want to clean the board.

In [11]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

def game_board(player=0, row=0, column=0):
    game[row][column] = player
    print("   0  1  2")
    for count, row in enumerate(game):
        print(count, row)

# start the game
game_board()
print()
# player makes a move
game_board(player=2, row=0, column=0)
print()
# game restart
game_board()
```

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

   0  1  2
0 [2, 0, 0]
1 [0, 0, 0]
2 [0, 0, 0]

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

Now we have another issue to solve: what if we simply want to show the current game board? Right now there's no way to do it, because we're either marking new moves or cleaning the board.

Easy, let's add a new `just_display` argument, and set it's default as well:

In [12]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

def game_board(player=0, row=0, column=0, just_display=False):
    if not just_display:
        game[row][column] = player
    print("   0  1  2")
    for count, row in enumerate(game):
        print(count, row)

# start the game
game_board()
print()
# player makes a move
game_board(player=2, row=0, column=0)
print()
# just display the current board
game_board(just_display=True)
print()
# game restart
game_board()
```

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

   0  1  2
0 [2, 0, 0]
1 [0, 0, 0]
2 [0, 0, 0]

   0  1  2
0 [2, 0, 0]
1 [0, 0, 0]
2 [0, 0, 0]

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

Dealing with errors

Ok, so what happens if a player tries to make a move outside of our board? Something like:

In [13]:

```
game_board(player=2, row=3, column=0)
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
<ipython-input-13-1f1501995090> in <module>
----> 1 game_board(player=2, row=3, column=0)

<ipython-input-12-69f269df5782> in game_board(player, row, column, just_di
splay)
      5 def game_board(player=0, row=0, column=0, just_display=False):
      6     if not just_display:
----> 7         game[row][column] = player
      8     print("   0  1  2")
      9     for count, row in enumerate(game):

IndexError: list index out of range
```

Oops. Time to do some exception handling:

In [14]:

```
def game_board(player=0, row=0, column=0, just_display=False):
    try:
        if not just_display:
            game[row][column] = player
        print("   0  1  2")
        for count, row in enumerate(game):
            print(count, row)
    except:
        print("A bad thing just happened.")

# let's try again:
game_board(player=2, row=3, column=0)
```

```
A bad thing just happened.
```

Not the greatest of messages, right? Let's elaborate on this:

In [15]:

```
def game_board(player=0, row=0, column=0, just_display=False):
    try:
        if not just_display:
            game[row][column] = player
        print("   0  1  2")
        for count, row in enumerate(game):
            print(count, row)
    except IndexError:
        print("Did you attempt to play a row or column outside the range of 0, 1 or 2?"
)
    except:
        print("Something went terribly wrong.")

# let's try again:
game_board(player=2, row=3, column=0)
```

```
Did you attempt to play a row or column outside the range of 0, 1 or 2?
```

Dealing with cheating

Much better. What if the player tries to make a move on an already chosen place?

Time to set up some validations and raise warnings:

In [16]:

```
game = [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]

# Our custom exception can be as simple as:
class CheatingError(Exception):
    pass

def game_board(player=0, row=0, column=0, just_display=False):
    try:
        if not just_display:
            if game[row][column] == 0:
                game[row][column] = player
            else:
                # calling our custom exception
                raise CheatingError
        print("   0  1  2")
        for count, row in enumerate(game):
            print(count, row)
    except IndexError:
        print("Did you attempt to play a row or column outside the range of 0, 1 or 2?"
)
    # handling our custom exception
    except CheatingError:
        print("That spot is already taken. Focus!")
    except:
        print("Something went terribly wrong.")

# player 2 makes a move
game_board(player=2, row=1, column=1)
print()
# player 1 tries to make the same move
game_board(player=1, row=1, column=1)
print()
```

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

That spot is already taken. Focus!
```

Calculating horizontal winner

Imagine the board is like this:

In [17]:

```
game = [[1, 1, 1],
        [2, 2, 0],
        [1, 2, 0]]
```

Clearly player 1 has won with the top row. We could determine that with the following function:

In [ ]:

```
def win(current_game):
    for row in current_game:
        print(row)
        # how might we check all items in this row? We could do something like:
        column_1 = row[0]
        column_2 = row[1]
        column_3 = row[2]
        if column_1 == column_2 == column_3:
            print("WINNER!")
win(game)
```

It works. But do you see the problem in that function? When the board is empty (all zeros) there's a winner too!

We could just make a simple change and make it work: `if column_1 == column_2 == column_3 and column_1 != 0:`

Still, the code feels very static, there's no room for dynamics here, like for example what if we wanted to have a dynamic board? In some games it would be 3x3 and in others 5x5?

In [18]:

```
game = [[1, 1, 1],
        [2, 2, 0],
        [1, 2, 0]]

def win(current_game):
    for row in game:
        print(row)
        # this is where we introduce the dynamics
        if row.count(row[0]) == len(row) and row[0] != 0:
            # let's print a nicer message
            print(f"Player {row[0]} is the winner horizontally!")

win(game)
```

```
[1, 1, 1]
Player 1 is the winner horizontally!
[2, 2, 0]
[1, 2, 0]
```

So much better, don't you agree?

Calculating vertical winner

Imagine the board is like this:

In [21]:

```
game = [[1, 0, 1],
        [1, 2, 0],
        [1, 2, 0]]
```

Try to understand what's going on here:

In [23]:

```
def win(current_game):
    for col in range(len(current_game[0])):
        check = []
        for row in current_game:
            check.append(row[col])
        if check.count(check[0]) == len(check) and check[0] != 0:
            print(f"Player {check[0]} is the winner vertically!")

win(game)
```

```
Player 1 is the winner vertically!
```

Calculating diagonal winner

Imagine the board is like this:

In [24]:

```
game = [[1, 0, 1],
        [1, 1, 2],
        [2, 2, 1]]
```

Let's start with this code:

In [25]:

```
diags = []
for ix in range(len(game)):
    diags.append(game[ix][ix])

print(diags)
```

```
[1, 1, 1]
```

We got the result we wanted. So if we add the same check as we did before:

In [26]:

```
if diags.count(diags[0]) == len(diags) and diags[0] != 0:
    print("Winner!")
```

```
Winner!
```

It works!

So the final code will be:

In [27]:

```
game = [[1, 0, 1],
        [1, 1, 2],
        [2, 2, 1]]

diags = []
for ix in range(len(game)):
    diags.append(game[ix][ix])

if diags.count(diags[0]) == len(diags) and diags[0] != 0:
    print(f"Player {diags[0]} is the winner diagonally!")
```

```
Player 1 is the winner diagonally!
```

Calculating diagonal winner (right to left)

Ok, we got the left to right diagonal done. What about the right to left diagonal?

We need some way to iterate backwards over the range of the len. I wonder if there's a builtin function for this...of course there is, it's called `reversed` !

Let's check if it works:

In [28]:

```
game = [[1, 0, 2],
        [1, 2, 2],
        [2, 2, 1]]

for i in reversed(range(len(game))):
    print(i)
```

```
2
1
0
```

Ok, so our columns will be in reverse, but our rows must continue to be in the original order:

In [29]:

```
cols = reversed(range(len(game)))
rows = range(len(game))
```

And then we would iterate them like this, to do the same as before:

In [30]:

```
for idx in rows:
    print(idx, cols[idx])
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-30-009235d05041> in <module>
      1 for idx in rows:
----> 2     print(idx, cols[idx])

TypeError: 'range_iterator' object is not subscriptable
```

What does this mean? That the `reversed` function returns an iterator, not a sequence. So you can't use `cols` immediatly, you'd have to iterate over them first.

Unless...you cast/convert the result of `reversed` to a `list` !

In [31]:

```
game = [[1, 0, 2],
        [1, 2, 2],
        [2, 2, 1]]

cols = list(reversed(range(len(game))))
rows = range(len(game))

for idx in rows:
    print(idx, cols[idx])
```

```
0 2
1 1
2 0
```

It worked! Each pair of numbers above is indeed the index we need to evaluate the spots in the board in a right to left diagonal.

Now we're ready to check if there's a winner:

In [32]:

```
game = [[1, 0, 2],
        [1, 2, 2],
        [2, 2, 1]]

cols = list(reversed(range(len(game))))
rows = range(len(game))
diags = []

for idx in rows:
    diags.append(game[idx][cols[idx]])

if diags.count(diags[0]) == len(diags) and diags[0] != 0:
    print(f"Player {diags[0]} is the winner diagonally!")
```

```
Player 2 is the winner diagonally!
```

Combining all the wins together

Ok, now we're ready to create a function that will contain all the possible wins:

In [ ]:

```
game = [[2, 2, 1],
        [0, 1, 2],
        [1, 2, 1]]


def win():
    outcome = ''
    # horizontal
    for row in game:
        print(row)
        if row.count(row[0]) == len(row) and row[0] != 0:
            outcome = f"Player {row[0]} is the winner horizontally!"
    # vertical
    for col in range(len(game[0])):
        check = []
        for row in game:
            check.append(row[col])
        if check.count(check[0]) == len(check) and check[0] != 0:
            outcome = f"Player {check[0]} is the winner vertically!"

    diags = []
    # / diagonal
    cols = list(reversed(range(len(game))))
    rows = range(len(game))

    for idx in rows:
        diags.append(game[idx][cols[idx]])

    if diags.count(diags[0]) == len(diags) and diags[0] != 0:
        outcome = f"Player {diags[0]} has won Diagonally (/)"

    # \ diagonal
    for ix in range(len(game)):
        diags.append(game[ix][ix])

    if diags.count(diags[0]) == len(diags) and diags[0] != 0:
        outcome = f"Player {diags[0]} has won Diagonally (\\)"

    return outcome

win()
```

Putting it all together

We have to implement some things to complete the game:

- which player will randomly start the game
- the automatic switch of players between moves
- give the option to retry after a failed move
- determining if anyone wins after a move

Also let's start gluing the code:

In [ ]:

```
# we'll use this builtin package to easily cycle between players and switch them
from itertools import cycle

# the win function
def win(current_game):
    outcome = ''
    # horizontal
    for row in game:
        if row.count(row[0]) == len(row) and row[0] != 0:
            outcome = f"Player {row[0]} is the winner horizontally!"
    # vertical
    for col in range(len(game[0])):
        check = []
        for row in game:
            check.append(row[col])
        if check.count(check[0]) == len(check) and check[0] != 0:
            outcome = f"Player {check[0]} is the winner vertically!"

    # / diagonal
    diags = []
    cols = list(reversed(range(len(game))))
    rows = range(len(game))

    for idx in rows:
        diags.append(game[idx][cols[idx]])

    if diags.count(diags[0]) == len(diags) and diags[0] != 0:
        outcome = f"Player {diags[0]} has won Diagonally (/)"

    # \ diagonal
    diags = []
    for ix in range(len(game)):
        diags.append(game[ix][ix])

    if diags.count(diags[0]) == len(diags) and diags[0] != 0:
        outcome = f"Player {diags[0]} has won Diagonally (\\)"

    return outcome


# the custom exception for the cheating error
class CheatingError(Exception):
    pass

# our board and game manager
def game_board(game_map, player=0, row=0, column=0, just_display=False):
    try:
        if not just_display:
            if game_map[row][column] == 0:
                game_map[row][column] = player
            else:
                raise CheatingError
        print("   0  1  2")
        for count, row in enumerate(game_map):
            print(count, row)
    except IndexError:
        print("Did you attempt to play a row or column outside the range of 0, 1 or 2?"
)
        return False
```

```
    except CheatingError:
        print("That spot is already taken. Focus!")
        return False
    except:
        print("Something went terribly wrong.")
        return False
    return True

# the main loop of the game
play = True
players = [1, 2]
while play:
    game = [[0, 0, 0],
            [0, 0, 0],
            [0, 0, 0]]

    game_won = False
    player_cycle = cycle(players)
    game_board(game, just_display=True)
    while not game_won:
        current_player = next(player_cycle)
        played = False
        while not played:
            print(f"Player: {current_player}")
            column_choice = int(input("Which column? "))
            row_choice = int(input("Which row? "))
            played = game_board(game, player=current_player, row=row_choice, column=col
umn_choice)
            has_won = win(game)
        if has_won != '':
            game_won = True
            print(has_won)
            again = input("The game is over, would you like to play again? (y/n) ")
            if again.lower() == "y":
                print("Restarting!")
            elif again.lower() == "n":
                print("Byeeeee!!!")
                play = False
            else:
                print("Not a valid answer, but... c ya!")
                play = False
```

There's still room for improvement, but for now it's ready to experiment.

Save this code as a *.py file and run in on a console.

Final words

Building this game allowed us to touch several features of the python language, such as:

- playing with conditionals
- iterating variables
- building functions that apply some of the DRY principles (like the use of default argument values that allow the functions to work under several scenarios)
- using builtin functions (`reversed` , `input` , etc) and packages (`itertools`) from the standard library
- handling exceptions and creating custom ones