Version 1

a. Features being developed:
   - Basic grid printing
   - Ability to choose a cell and place a cross/circle with coordinates

b. Code:

Main.py

```
from Core.Game import Game # Import game class from 'Core' folder

game = Game() # Create new game instance

while True: # Loop infinitely
    game.run() # Play one game round
```

Core/Game.py

```
from .Matrix import Matrix # Import Matrix class from current folder
import re # Import regular expressions module


class Game: # Define Game class
    def __init__(self): # Define class constructor with empty parameters
        self.matrix = Matrix(3, 3) # Initialise 3x3 member instance of Matrix class
        self.turn_index = 0 # Initialise an integer member instance indicating who's turn it is

    def run(self): # Define run method (one game round)
        self.compose_frame() # Print game state
        self.update_model() # Update game state

    def compose_frame(self): # Define method which prints the game state
        self.matrix.print()

    def update_model(self): # Define update_model method (update game state)
        while True: # Loop while user input is invalid
            match = re.match(r"^\(?(?P<x>[1-3]),? ?(?P<y>[1-3])\)?$", input("Enter coordinate: ")) #
Check whether user input matches conditions

            if match: # Exit loop if input is valid (the match class defines a __bool__ method for
implicit type casting); helps to avoid having to define the variable outside of the loop with a dummy
value (bad code design)
                break

            print("Invalid input. Please try again.") # Indicate to user that their input is invalid

        self.matrix[int(match.group("x")) - 1, int(match.group("y")) - 1] = ["X", "O"][self.turn_index %
2] # Put the player's symbol at the appropriate spot on the grid
        self.turn_index += 1 # Increment the user's turn
```

Core/Matrix.py

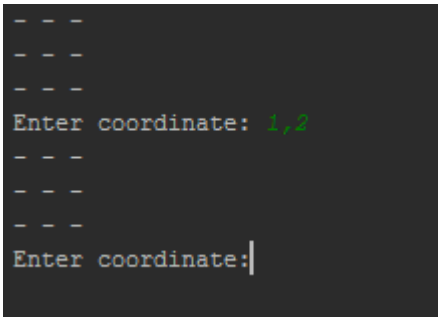
```
class Matrix: # Define Matrix class
    def __init__(self, height, width): # Define constructor accepting a width and height
        self.height = height # Initialise height member
        self.width = width # Initialise width member
        self.matrix = ["-" for _ in range(height * width)] # Initialise empty grid

    def __getitem__(self, item): # Define get indexer accessor to retrieve any symbol on the grid
        return self.matrix[self.map_index(item)]

    def __setitem__(self, key, value): # Define set indexer accessor to set any symbol on the grid
        self.matrix[self.map_index(key)] = value

    def map_index(self, coordinate): # Define method which maps the coordinate to the appropriate 1D
index
        x, y = coordinate
        return x + y * self.width

    def print(self): # Define method which outputs the grid to the screen
        print("\n".join(" ".join(self[column, row] for column in range(self.width)) for row in
range(self.height)))
```

c. Testing and Debugging

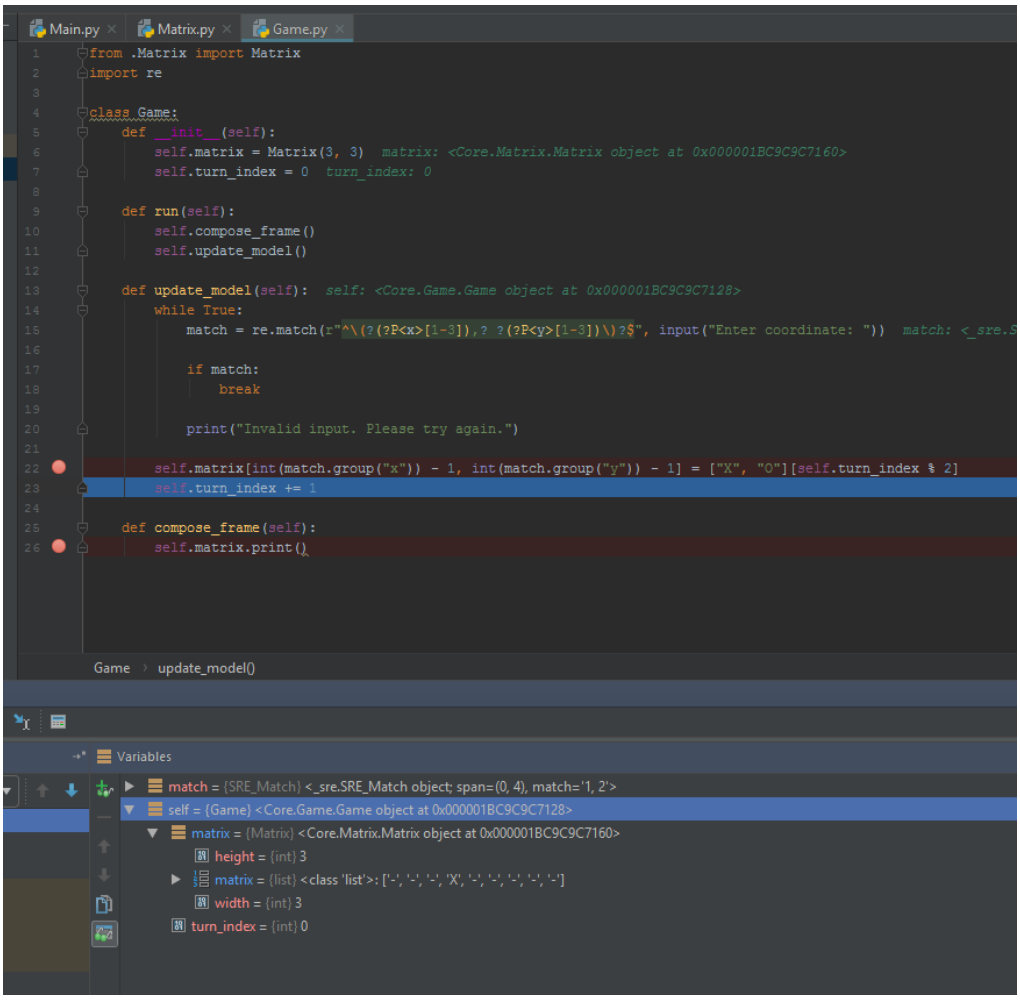
Debugging:

One of the first problems I had with the code was that the game was updating the grid after each game with the user's input, however on the next round, the game grid was empty again. It turns out that I was creating a new instance of the game class on every round, which was subsequently creating a new instance of the matrix, therefore resetting the grid.

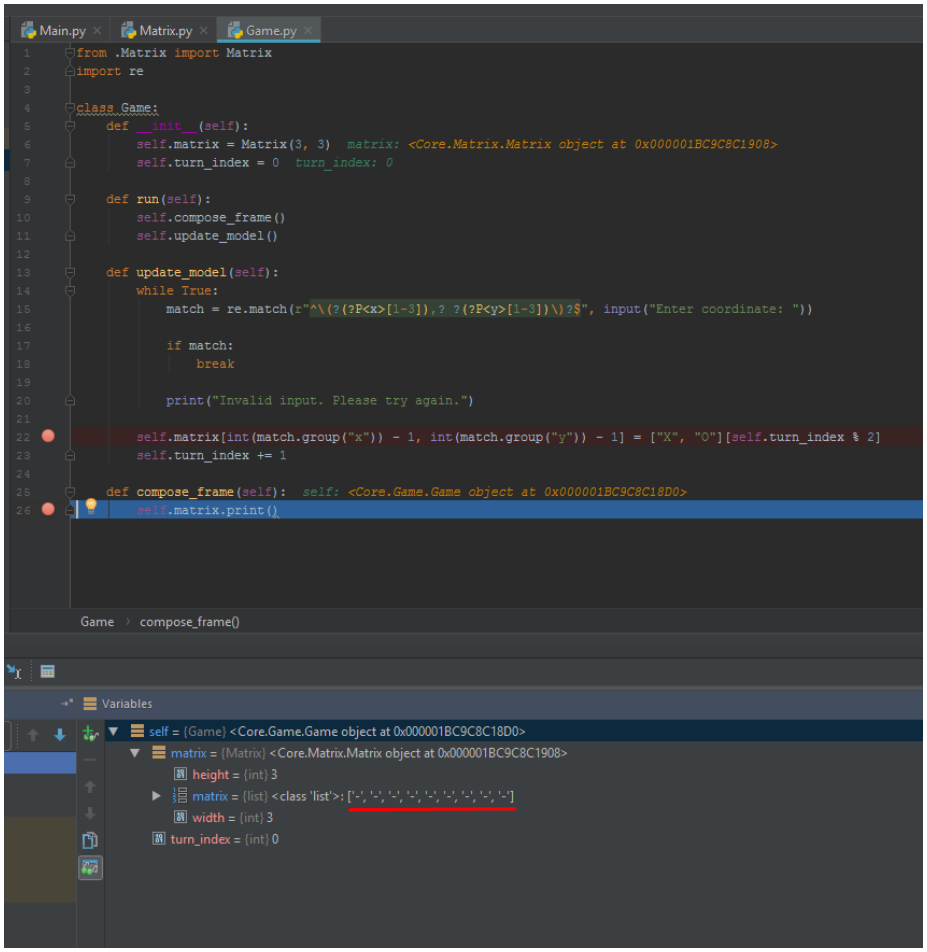
Here is the debugging process:



As you can see, even though a coordinate was being input, the grid was not modified at all.



I ran the debugger, and noticed that the matrix was in fact being filled correctly.



However, as soon as the next round occurs (composing the game frame), the matrix is reset to the defaults.

I looked in *Main.py* to check what the problem was, and realised that a new instance of Game was being created on every iteration.

```
while True:
    Game.Game().run()
```

After fixing the problem, the game printed correctly.

```
Enter coordinate: 21
X O X
- O -
- - -
Enter coordinate: 32
X O X
- O X
- - -
Enter coordinate: 97
Invalid input. Please try again.
Enter coordinate: 23
X O X
- O X
- O -
Enter coordinate:
```

In addition, in order to test and debug the regular expression which I used to validate user input, I used regex101.com and provided a variety of valid and invalid entries, to see which ones should fit and which ones should not, and built the regular expression around this.

Version 2:

a. Features being developed:
   - The ability to enter an index instead of a 2D coordinate

b. Code:

   Main.py is practically set in stone, so there have been no changes to that code file. Only changes since version 1 will be displayed.

Changes in *Game.py*:

```
def update_model(self): # Define update_model method (update game state)
    def set_symbol(coordinate): # Define local function which places the player's symbol at the
appropriate spot on the grid
        self.matrix[coordinate] = ["X", "O"][self.turn_index % 2]

    while True: # Loop while user input is invalid
        input_string = input("Enter coordinate: ") # Store input for use in two places

        match = re.match(r"^\(?(?P<x>[1-3]),? ?(?P<y>[1-3])\)?$", input_string) # Check whether user
input matches 2D coordinate

        if match:
            set_symbol((int(match.group("x")) - 1, int(match.group("y")) - 1)) # Get 2D index and place
user symbol in matrix
            break

        match = re.match("^(?P<index>[1-9])$", input_string) # Check whether user input matches 1D
coordinate

        if match:
            set_symbol(int(match.group("index")) - 1) # Get index and place user symbol in matrix
            break

        print("Invalid input. Please try again.") # Indicate to user that their input is invalid

    self.turn_index += 1 # Increment the user's turn
```

Changes in *Matrix.py*:

```
def map_index(self, coordinate): # Define method which maps the coordinate to the appropriate 1D index
    if type(coordinate) is tuple: # If a 2D coordinate is given, map this coordinate
        x, y = coordinate
        return x + y * self.width
    elif type(coordinate) is int: # If a 1D coordinate is given, it is already mapped, so just return it
        return coordinate

    # Prevent logic errors by asserting the type of the input variable
    assert type(coordinate) is tuple or type(coordinate) is int # This line can only execute if a logic
error has been encountered
```

c. Testing and Debugging

Initially I thought of using the same regex to match both types of input. This isn't a difficult task, as the or operator ('|') can be used for conditional operation. However, this would be problematic to my program, as a method needs to be identified whereby the input choice can be gathered.

To solve this, I decided to use two regexes, and branch the code with appropriate flow control to provide different logic for each case.

First, I defined a regular expression to match the input:

REGULAR EXPRESSION

```
/ ^(?P<index>[1-9])$
```

TEST STRING

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

After implementing the initial program, I noticed a small error: the user-entered coordinates were being zero-indexed. In other words, entering coordinate '5' would fill cell #6, which has an index of 5. This is also problematic for the regular expression, as users can't physically enter a coordinate which fills the first cell, and the regular expression would validate coordinates which are out of bounds.

```
- - -
- - -
- - -
Enter coordinate: 4
- - -
- X -
- - -
Enter coordinate: 5
- - -
- X O
- - -
Enter coordinate: 6
- - -
- X O
X - -
```

This was a fairly simple fix, by taking away 1 from the user input.

```
set_symbol(int(match.group("index")) - 1)
```

```
- - -
- - -
- - -
Enter coordinate: 0
Invalid input. Please try again.
Enter coordinate: 1
X - -
- - -
- - -
Enter coordinate: 9
X - -
- - -
- - O
Enter coordinate:
```

Once this was complete, this version was also complete.

Version 3:

a. Features being developed:
   - Inability to place symbols at a filled coordinate
   - Ability to win or draw the game

b. Code:

Main.py:

```
from Core.Game import Game # Import game class from 'Core' folder

game = Game() # Create new instance

while game.run(): # Loop until game has finished
    pass
```

Changes in *Game.py*:

```
def run(self):  # Define run method (one game round)
    self.compose_frame()  # Print game state

    if self.process_game_state(): # Check whether game has finished
        return True

    self.update_model()  # Update game state

    return False # Return that the game is still ongoing

def process_game_state(self):
    def process_succession(iterable, generator): # Local function which uses meta-programming principles
to perform row, column, and diagonal checks
        for item in iterable: # For each diagonal, row, or column
            # Applies generator function to each item, and converts the collection into a set to check
whether all
            # symbols are the same
            symbols = "".join(set(generator(index, item) for index in range(3)))

            if len(symbols) == 1 and symbols != "-": # If all of the symbols are the same (but not '-')
                print(f"{symbols} wins!") # The only symbol in the set wins
                return True # Indicate win

    if any(
        [
            process_succession(range(3), lambda column, _: self.matrix[column, 0]), # Check columns for
win
            process_succession(range(3), lambda row, _: self.matrix[0, row]), # Check rows for win
            process_succession([0, 2], lambda index, diagonal: self.matrix[index + [diagonal, 0, -
diagonal][index], index]) # Check diagonals for win
        ]
    ): # Check whether any row, column, or diagonal has a win state
        return True # Return game end state
```

```
    if all(map(lambda symbol: symbol != "-", self.matrix)): # Check if no empty cells are left
        print("Draw!")
        return True # Return game end state

    return False # Return game continuing state

# compose_frame omitted

def update_model(self):  # Define update_model method (update game state)
    def set_symbol(coordinate):  # Define local function which places the player's symbol at the
appropriate spot on the grid
        self.matrix[coordinate] = ["X", "O"][self.turn_index % 2]

    def is_bad_coordinate(coordinate):
        return self.matrix[coordinate] != "-"

    while True:  # Loop while user input is invalid
        input_string = input("Enter coordinate: ")  # Store input for use in two places

        match = re.match(r"^\(?(?P<x>[1-3]),? ?(?P<y>[1-3])\)?$",
                         input_string)  # Check whether user input matches 2D coordinate

        if match:
            coordinate = (int(match.group("x")) - 1, int(match.group("y")) - 1)

            if is_bad_coordinate(coordinate):
                print("Bad coordinate. Please try again.")
                continue

            set_symbol(coordinate)  # Get 2D index and place user symbol in matrix
            break

        match = re.match("^(?P<index>[1-9])$", input_string)  # Check whether user input matches 1D
coordinate

        if match:
            coordinate = int(match.group("index")) - 1

            if is_bad_coordinate(coordinate):
                print("Bad coordinate. Please try again.")
                continue

            set_symbol(coordinate)  # Get index and place user symbol in matrix
            break

        print("Invalid input. Please try again.")  # Indicate to user that their input is invalid

    self.turn_index += 1  # Increment the user's turn
```

c. Testing and Debugging

The first feature I started developing was the inability to place symbols at a filled coordinate. Due to the simplicity of the code, this was an immediate success:

```
- - -
- - -
- - -
Enter coordinate: 1
X - -
- - -
- - -
Enter coordinate: 1
Bad coordinate. Please try again.
Enter coordinate: 2
X O -
- - -
- - -
Enter coordinate:
```

However, for the game winning feature, the story was different. On the first test, this was the result:

```
- - -
- - -
- - -
Enter coordinate: 1
{'-'} wins!
X - -
- - -
- - -
Enter coordinate:
```

My program was detecting that the '-' player was winning, as it had columns, rows, or diagonals which were the same. Again, this was a simple fix. I added a secondary condition (using the 'and' operator) which makes sure that all symbols are not dashes.

Following this, the game was perfectly playable:

```
Enter coordinate: 2
X X -
O - -
- - -
Enter coordinate: 5
X X -
O O -
- - -
Enter coordinate: 3
X wins!
X X X
O O -
- - -
```

The only problem was that the game would not end when the player would win. To fix this, main just had to loop until game.run() returns false.

I also made sure that the game board was printed first, before displaying the win state, to allow the user to see the boards state at the end of the game, instead of exiting immediately.

Following this, the game was complete:

```
Enter coordinate: 2
X X -
O - -
- - -
Enter coordinate: 5
X X -
O O -
- - -
Enter coordinate: 3
X X X
O O -
- - -
X wins!

Process finished with exit code 0
```

```
Enter coordinate: 5
X - -
O X -
- - -
Enter coordinate: 6
X - -
O X O
- - -
Enter coordinate: 9
X - -
O X O
- - X
X wins!

Process finished with exit code 0
```

```
Enter coordinate: 7
X O X
- O -
X X O
Enter coordinate: 4
X O X
O O -
X X O
Enter coordinate: 6
X O X
O O X
X X O
Draw!

Process finished with exit code 0
```