Immutable design in Tic-tac-toe game using python3

Jani Haakana (517534)

BIOI2250 – Introduction to Programming

University of Turku

September 2020

Contents

1 Problem description

In this project I wanted to inspect how immutability can be used in a small game context. Traditionally programmers are encouraged to create objects which interact with other entities or objects throughout the program lifetime. In addition to the programs own state the objects do have their own internal state as well. This creates a situation where the programmer have to handle multiple states independently of each other.

If I would have written this program in traditional way, the Game object would have had methods which mutate the internal state of the game itself. Also, the game drawer might have been a separate object, which would have taken a game object for inspection during the drawing phase.

2 Problem analysis

Usually programmers want to use a programming language which would support or even encourage immutability in the program design. However, in the context of python, the language itself doesn't enforce nor encourage to any specific way of design. And, because of this, the immutable design is heavily dependent on the programmers own discipline, which means it is relatively easy to create a mutation by mistake.

Let's define mutation to the program state first. First of all, if a program is fully immutable, then it wouldn't be a useful program at all, because it couldn't interact with the real world. From this we can define that the program has to have at least some level of mutation in it. Secondly, we can use the categorical theory's view of functions. Function takes an input in one category and can do something to it and return an output which

is in different category. In other words, mutation can happen inside the function, but in a way that given the same arguments the function will output the same result. Function cannot, in any circumstances, interact with anything outside the function context.

3 Solution structure

Already, I have a lot of mutation happening, but I have defined a place for them. In my case I placed the game state in the main game loop, because that is the place where the program interacts with the real world, players in this case. Secondly, the game object will do mutation, but by giving us the same result with every call.

To reach this level of immutability, the game object itself cannot modify its internal state, but the program has to have a way to advance to the next state. Advancing happens by creating a new version of the game object itself. To support this, the game object has two kinds of methods inspecting and "advancing". The former methods will return some view of the current game. For example, the programmer can ask has the current player won the game by calling `.wins()` method or if the board is full with `.full()` method. The latter functions will take the required change in the state and return a new instance of the game object. As an example, the `.move()` function takes a place where the current player wants to input its marker and returns a modified game state or if the user has given an invalid move, then the move function will return the current game object. This is because with the invalid move the state of the game cannot advance to the next state, and therefore it is reasonable to just return the current state and continue the execution like nothing happened.

4 Detailed module listing

Please refer to tic-tac-toe.html file.

5 Usage instructions

Launch the program from the command-line using python interpreter and giving it the file main.py as an argument.

```
python main.py
```

Then players will input their move by giving the game a number between one and nine. The game will then advance the game state and present the new board with the modification to the user. The game will stop with either a winning or in a draw condition.

6 Final words

Having immutable game state, gave me an opportunity to develop things in the `ipython repl` environment. For example: firstly I spun up the ipython repl with command `ipython`, then I started to import all necessary code for my current debugging needs:

```
In [1]: import tic_tac_toe.core as c

In [2]: import tic_tac_toe.draw as d

In [3]: import tic_tac_toe.utils as u
```

In this case I wanted to debug row printing, so I created my own debug version of the draw function, and then I created a state in which I wanted to test it out.

```
In [4]: def draw(grid):
   ...:     bar = ["-" * (3 * 4 -1)]
   ...:     board = u.interleave(bar, [d.show_row(row) for row in grid])
   ...:     return board
   ...:


In [5]: state = \
   ...:   c.Game.new_game() \
   ...:          .move(1) \
   ...:          .move(2) \
   ...:          .move(7) \
   ...:          .move(5) \
   ...:          .move(9)

In [6]: state.draw_board(draw)
Out[6]:
[['   |   |   ', ' X | O |   ', '   |   |   '],
 ['-----------'],
 ['   |   |   ', '   | O |   ', '   |   |   '],
 ['-----------'],
 ['   |   |   ', ' X |   | X ', '   |   |   ']]

In [7]: u.concat(state.draw_board(draw)))
Out[7]:
```

```
['   |   |   ',
 ' X | O |   ',
 '   |   |   ',
 '-----------',
 '   |   |   ',
 '   | O |   ',
 '   |   |   ',
 '-----------',
 '   |   |   ',
 ' X |   | X ',
 '   |   |   ']
```

For future research, I could start to inspect how multiple game states could be beneficial to the game itself. In this context, however, I didn't see any meaningful way to leverage such possibility.

Another research topic could be the main interaction point. In more complex games, the game might have other places where user interactions are required. In such a small example, there weren't any real need to have multiple interaction points and such the main game loop was a perfect place to handle all the required mutation.