Fanorona API for MAIC 2021

Figure: A Fanorona board

1. Fanorona Rules

- The Board
  - The game is played on a 9x5 board where some intersections are connected to each other by diagonals.
  - Each player has 22 pieces.

- The Game
  - First stage: placing the pieces.
    The pieces are placed on all intersections except the central intersection like as shown in the figure below.

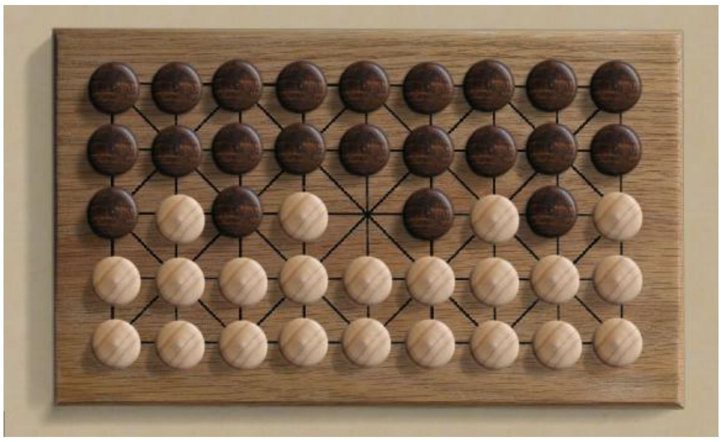


Figure: Initial filling of the board

- Second step: moving and taking pieces.
    - Players decide together the first player.
    - Each piece can move to an adjacent empty intersection along the lines.
    - Capturing moves are obligatory.
    - Pieces are captured by approach or distance.

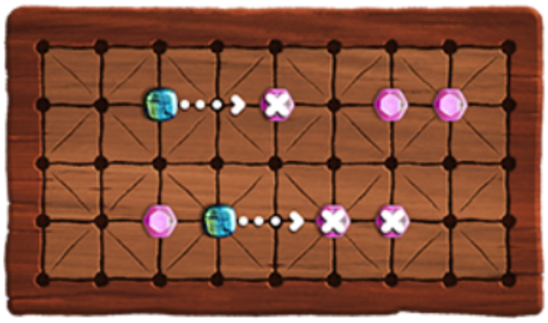


Figure: Approach capture

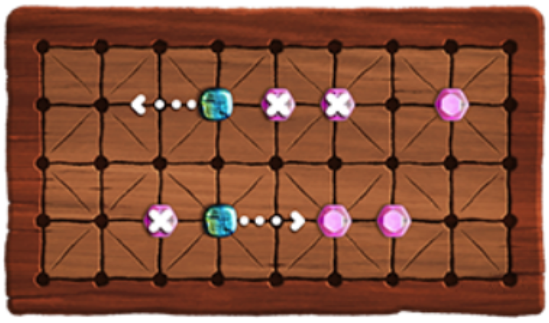
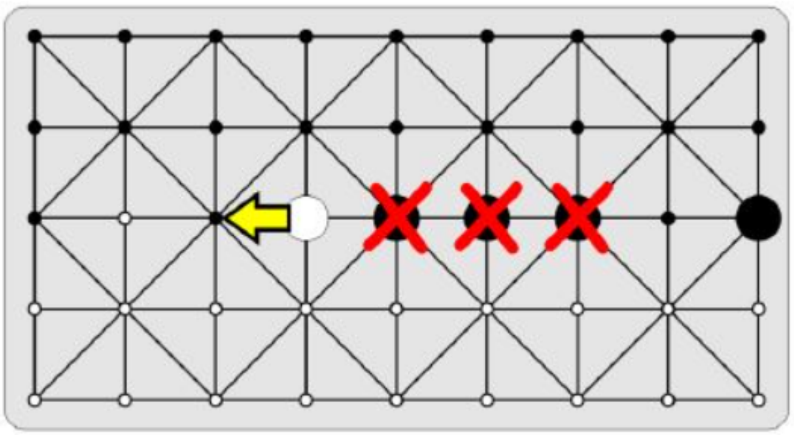


Figure: Distance capture

- When a piece approaches or moves away from an opposing piece, the latter and all the opposing pieces aligned beyond (as long as there is no interruption by an empty point or own piece) are captured.



- When the player can capture both by approach and distance, he must choose one or the other.
- A piece can take several pieces simultaneously provided that the piece does not occupy the same position twice or take the same direction twice consecutively.

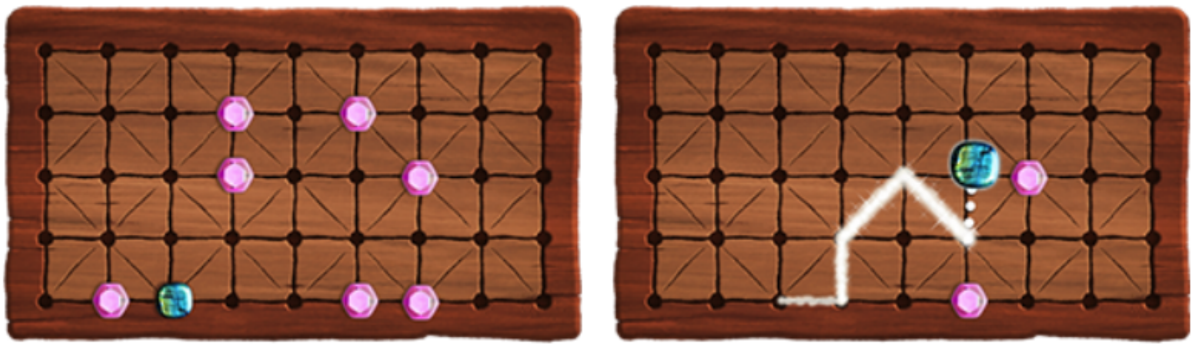
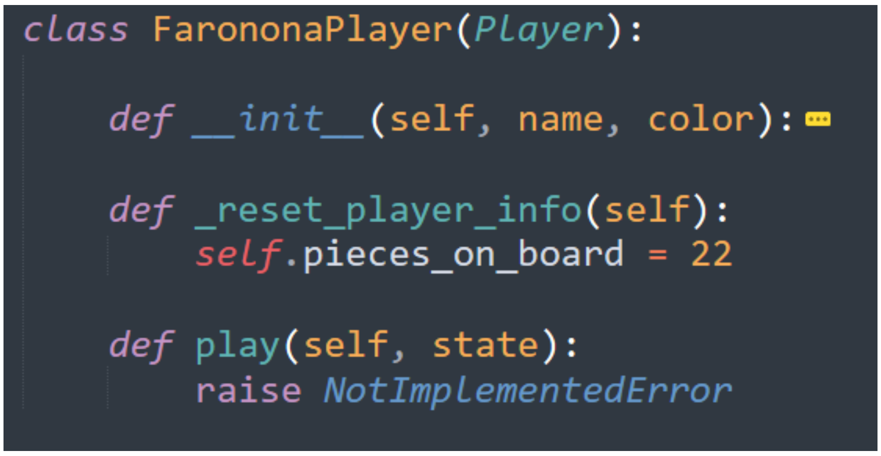


Figure: A multiple capture

2. Your work

- Write a play function for your AI.

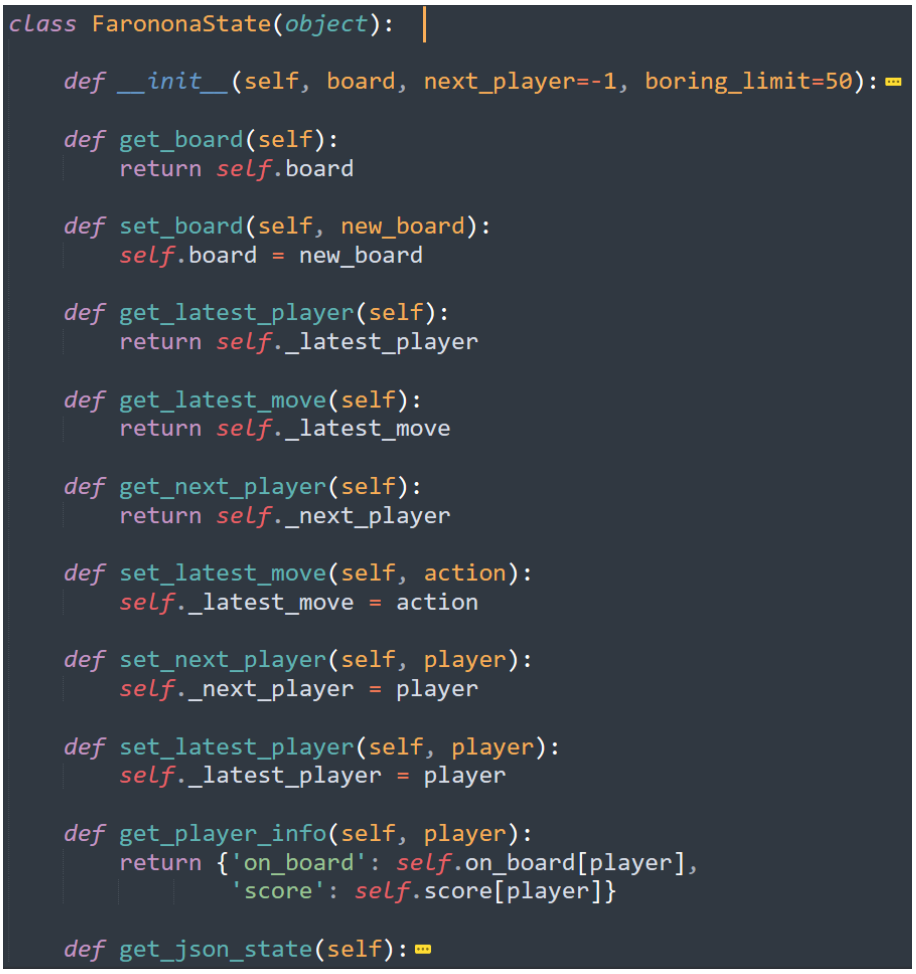


```python
class FarononaPlayer(Player):

    def __init__(self, name, color):

    def _reset_player_info(self):
        self.pieces_on_board = 22

    def play(self, state):
        raise NotImplementedError
```

The AI class will extend the FarononaPlayer class and then you will implement the play function.

3. What will you need?

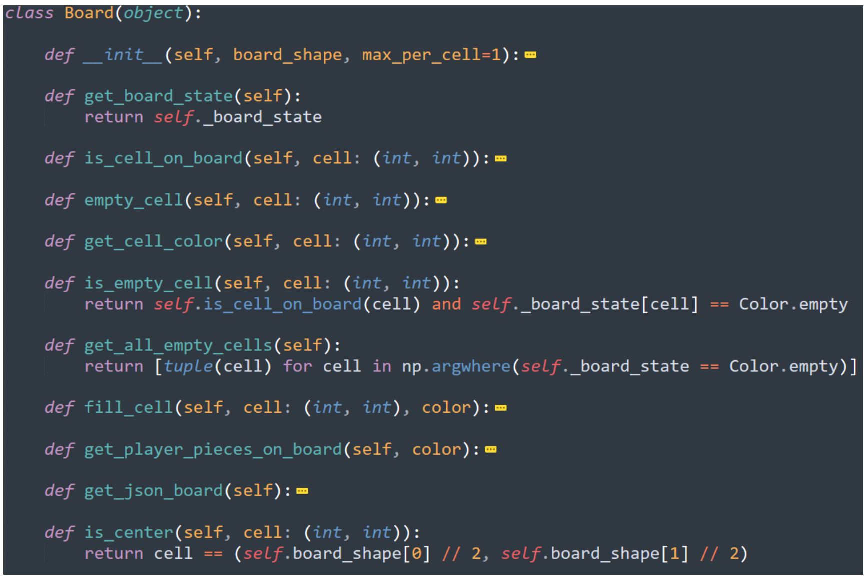
- The state object



```python
class FaronnaState(object):

    def __init__(self, board, next_player=-1, boring_limit=50):

    def get_board(self):
        return self.board

    def set_board(self, new_board):
        self.board = new_board

    def get_latest_player(self):
        return self._latest_player

    def get_latest_move(self):
        return self._latest_move

    def get_next_player(self):
        return self._next_player

    def set_latest_move(self, action):
        self._latest_move = action

    def set_next_player(self, player):
        self._next_player = player

    def set_latest_player(self, player):
        self._latest_player = player

    def get_player_info(self, player):
        return {'on_board': self.on_board[player],
                'score': self.score[player]}

    def get_json_state(self):
```

It keeps the situation of the players and the board at each move.

- The board object

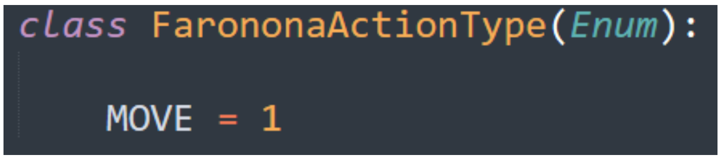


```python
class Board(object):

    def __init__(self, board_shape, max_per_cell=1):

    def get_board_state(self):
        return self._board_state

    def is_cell_on_board(self, cell: (int, int)):

    def empty_cell(self, cell: (int, int)):

    def get_cell_color(self, cell: (int, int)):

    def is_empty_cell(self, cell: (int, int)):
        return self.is_cell_on_board(cell) and self._board_state[cell] == Color.empty

    def get_all_empty_cells(self):
        return [tuple(cell) for cell in np.argwhere(self._board_state == Color.empty)]

    def fill_cell(self, cell: (int, int), color):

    def get_player_pieces_on_board(self, color):

    def get_json_board(self):

    def is_center(self, cell: (int, int)):
        return cell == (self.board_shape[0] // 2, self.board_shape[1] // 2)
```

It keeps all the information related to free intersections, the locations of opposing pawns and own pawns.

4. Actions

- Only one type of action is allowed



```python
class FaronanaActionType(Enum):

    MOVE = 1
```

Figure: Action Type

- Return an action

Figure: example of returned action

- action_type : is always *FaronanaActionType.MOVE*

- at : coordinates of begin move

- to : coordinates of end move

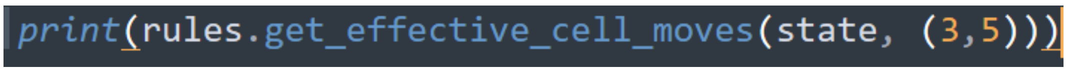
- win_by : parameter to specify when your move can win both by approach (APPROACH) and remote (REMOTE). By default it is APPROACH.

5. Functions

- Methods of FanoronaRules class
  - *get_effective_cell_moves*

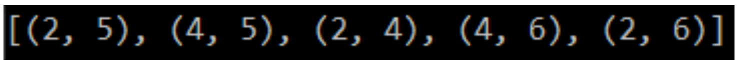
    It takes the state of the game and the coordinates of a piece in parameters to give the only possible movements of that piece on the board. This method returns a list of all the coordinates where the piece can go.

    Example:



```
print(rules.get_effective_cell_moves(state, (3,5)))
```

Result:



```
[(2, 5), (4, 5), (2, 4), (4, 6), (2, 6)]
```

  - *is_legal_move*

    It takes the state, the action and the player and return if it is a legal move.

```
is_legal_move(state, action, player):
```

    return true or false.

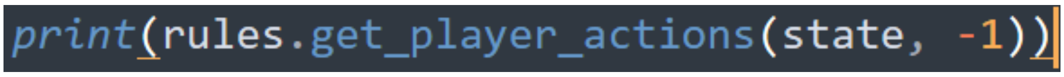
  - *random_play*

    It takes the state, and the player and return a random action.

```
random_play(state, player)
```

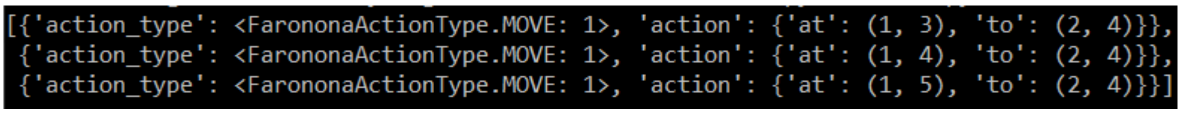
    return true or false

- *get_player_actions*
  It takes the state of the game and the number of the player in parameters to determine for this player, in this state the set of his possible actions. It returns a list of all possible actions (FanoronaAction) at the given state.
  Example:



```
print(rules.get_player_actions(state, -1))
```

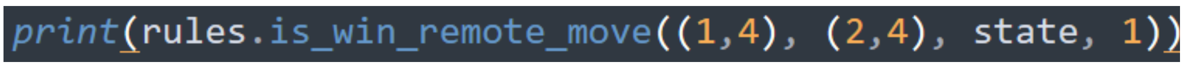
Result:



```
[{'action_type': <FaronaActionType.MOVE: 1>, 'action': {'at': (1, 3), 'to': (2, 4)}},
 {'action_type': <FaronaActionType.MOVE: 1>, 'action': {'at': (1, 4), 'to': (2, 4)}},
 {'action_type': <FaronaActionType.MOVE: 1>, 'action': {'at': (1, 5), 'to': (2, 4)}}]
```

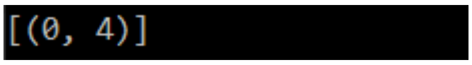
- *is_win_approach_move / is_win_remote_move*
  They take the coordinates of a piece, the coordinates of the intersection the piece is supposed to go, the state of the game and the number of the player. They return a list of the coordinates of the opposing pieces that can be captured, if any.
  ** is_win_approach_move : returned captured moves by approach.*
  ** is_win_remote_move : returned captured moves by remote or distance.*

  Example:



```
print(rules.is_win_remote_move((1,4), (2,4), state, 1))
```

Result:

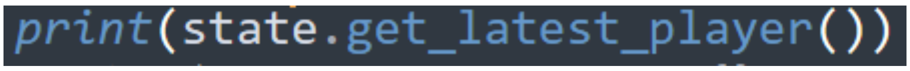


```
[(0, 4)]
```

- Methods of FanoronaState class

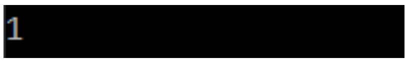
- *get_latest_player / get_next_player*
  Returns respectively the latest player and the next player number.

  Example:



```
print(state.get_latest_player())
```

  Result:

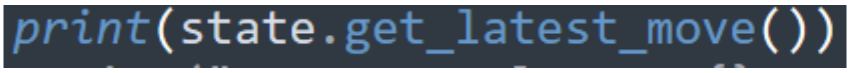


```
1
```

- *get_latest_move*

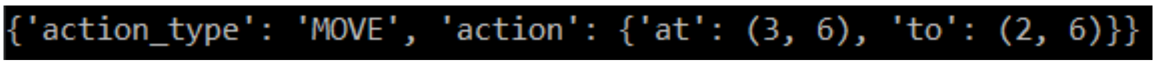
Returns the last move(an object of FanoronaAction) in the game.

  Example:



```
print(state.get_latest_move())
```

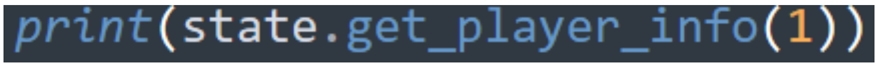
  Result:



```
{'action_type': 'MOVE', 'action': {'at': (3, 6), 'to': (2, 6)}}
```

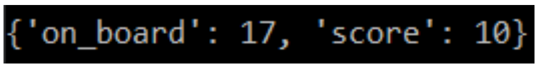
- *get_player_info*
  Takes the number of the player and returns a dictionary contains player's score and his pieces on the board.

  Example:



```
print(state.get_player_info(1))
```

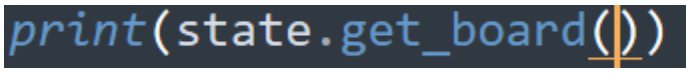
  Result:



```
{'on_board': 17, 'score': 10}
```

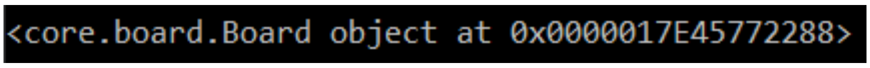
- *get_board*
  Returns the board object.

  Example:



```
print(state.get_board())
```

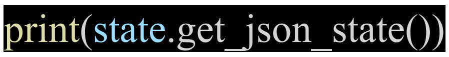
  Result:



```
<core.board.Board object at 0x0000017E45772288>
```

- *get_json_state()*
  Returns the state in json format.

  Example:



```
print(state.get_json_state())
```

  Result:

```
{
    "latest_player": -1,
    "latest_move": {"action_type": "MOVE", "action": {"at": ["1", "4"], "to":
    ["2", "4"]}},
    "next_player": 1,
    "score": {"-1": 2, "1": 0},
    "on_board": {"-1": 22, "1": 20},
    "boring_moves": 1,
    "just_stop": 50,
    "board": [["white", "white", "white", "white", "white", "white", "white",
    "white",     "white"], ["white", "white", "white", "white", "empty",
    "white", "white", "white", "white"], ["white", "green", "white", "green",
    "white", "green", "white", "green", "white"], ["green", "green", "green",
    "green", "empty", "green", "green", "green", "green"], ["green", "green",
    "green", "green", "empty", "green", "green", "green", "green"]]
}
```