Tic Tac Toe Docs

Project documentation with Markdown.

lordksix

Table of contents

1. Tic Tac Toe Docs

This site contains the project documentation for the `Tic Tac Toe` project that is a Tic Tac Toe game wrapped in a package

This is a console application to play Tic Tac Toe with another human or computer, which as a basic AI.

1.1 Table Of Contents

The documentation follows the best practice for project documentation as described by Daniele Procida in the Diátaxis documentation framework and consists of four separate parts:

1. Tutorials
2. How-To Guides
3. Reference
4. Explanation

Quickly find what you're looking for depending on your use case by looking at the different pages.

1.2 Project Overview

1.2.1 Backend package

Package that handles the backend of the game. It is the game engine

Subpackages exported by this subpackage:

- `game` : Handle and render your game with this engine.
- `logic` : Handle model domain and business logic

1.2.2 Frontend package

Package that handles frontend of the game.

Subpackages exported by this subpackage:

- `console` : Handle the frontend console.

1.3 Acknowledgements

I want to thank RealPython for the help, base and inspiration.

2. How-To Guides

This part of the project documentation focuses on a **problem-oriented** approach. You'll tackle common tasks that you might have, with the help of the code provided in this project.

2.1 How is it structured?

Download the code from this GitHub repository and place the `tic-tac-toe-python/` folder in the same directory as your Python script:

```
tic-tac-toe/
│
├── docs/
│   │
│   |── explanation.md
│   |── index.md
│   |── tutorials.md
│   ├── how-to-guides.md
|   └── reference.md
│
└── src/
    ├── frontends/
    │   |
    |   └── console/
    |   │    ├── __init__.py
    |   │    └── __main__.py
    |   │    ├── arg.py
    |   │    └── cli.py
    |   │    ├── players.py
    |   │    └── renderers.py
    |   |
    |   ├── __init__.py
    |   └── play.py
    │
    ├── backend/
    │   │
    │   ├── game/
    │   │   ├── __init__.py
    │   │   ├── engine.py
    │   │   ├── players.py
    │   │   └── renderers.py
    │   │
    │   ├── logic/
    │   │   ├── __init__.py
    │   │   ├── exceptions.py
    │   │   ├── models.py
    │   │   └── validators.py
    │   │
    │   └── __init__.py
    │
    └── pyproject.toml
```

You can run the script `play.py`, to run the most basic and default version of the game:

```
tictactoe -X human -O minimax
```

Where -X mean the player that uses the X mark, options: human, random, minimax Where -O mean the player that uses the O mark, options: human, random, minimax

3. Reference

This part of the project documentation focuses on an **information-oriented** approach. Use it as a reference for the technical implementation of the `Tic Tac Toe` project code.

3.1 Backend subpackages

3.1.1 Game subpackage

Handle and render your game with this engine.

Modules exported by this package:

- `engine` : Provide the class that handles the game.
- `players` : Provide the classes to instantiate players, human or computer.
- `renderers` : Provide classes for visual and state rendering.

This subpackage has the following modules:

1. Engine
2. Players
3. Renderers

3.1.2 Logic subpackage

Handle model domain and business logic

Modules exported by this package:

- `exceptions` : Provide exceptions for that handles the game.
- `minimax` : Provide methods to implement basic AI to computer player
- `validator` : Provide methods to validate game states and grid.
- `models` : Provide classes for domain models.

This subpackage has the following modules:

1. Exceptions
2. Minimax
3. Models
4. Validators

3.2 Frontend

3.2.1 Console subpackage

Package that handles the frontend console.

Modules exported by this package:

- `args` : Provide exceptions for that handles the game.
- `CLI` : Provide methods to implement basic AI to computer player
- `players` : Provide methods to validate game states and grid.
- `renderes` : Provide classes for domain models.

This subpackage has the following modules:

1. Args
2. CLI
3. Players
4. Renderer

4. Explanations

This part of the project documentation focuses on an **understanding-oriented** approach. You'll get a chance to read about the background of the project, as well as reasoning about how it was implemented.

Note: Expand this section by considering the following points:

- Give context and background on your library
- Explain why you created it
- Provide multiple examples and approaches of how to work with it
- Help the reader make connections
- Avoid writing instructions or technical descriptions here

5. Tutorials

This part of the project documentation focuses on a **learning-oriented** approach. You'll learn how to get started with the code in this project.

Note: Expand this section by considering the following points:

- Help newcomers with getting started
- Teach readers about your library by making them write code
- Inspire confidence through examples that work for everyone, repeatably
- Give readers an immediate sense of achievement
- Show concrete examples, no abstractions
- Provide the minimum necessary explanation
- Avoid any distractions

6. Engine module

Provide the class that handles the game.

This module allows the game to run. It is the game engine.

Examples:

```
>>> player1 = RandomComputerPlayer(Mark("X"))
>>> player2 = RandomComputerPlayer(Mark("O"))
>>> TicTacToe(player1, player2, ConsoleRenderer()).play()
```

The module contains the following class: - `TicTacToe`

6.1 `TicTacToe` `dataclass`

A class used to represebt the game engine.

Attributes:

Name	Type	Description
`player1`	`Player`	Player An instance of subclass of the Player class that represents a human or computer.
`player2`	`Player`	Player An instance of subclass of the Player class that represents a human or computer.
`renderer`	`Renderer`	Renderer An instance of subclass of the Renderer class that handles UI rendering.
`error_handler`	`ErrorHandler | None`	ErrorHandler | None = None A placehholder for a callback function that handles InvalidMove exceptions.

Methods:

Name	Description
`play`	Mark = Mark("X")) -> None: Handles the flow of the game. The engine itself
`def get_current_player`	GameState) -> Player: Determines current player base on the current game state

Source code in `src\backend\game\engine.py`

```
@dataclass(frozen=True)
class TicTacToe:
    """A class used to represebt the game engine.

    Attributes:
        player1: Player
            An instance of subclass of the Player class that represents a human or computer.
        player2: Player
            An instance of subclass of the Player class that represents a human or computer.
        renderer: Renderer
            An instance of subclass of the Renderer class that handles UI rendering.
        error_handler: ErrorHandler | None = None
            A placehholder for a callback function that handles InvalidMove exceptions.

    Methods:
        play(self, starting_mark: Mark = Mark("X")) -> None:
            Handles the flow of the game. The engine itself
        def get_current_player(self, game_state: GameState) -> Player:
            Determines current player base on the current game state
    """
    player1: Player
    player2: Player
    renderer: Renderer
    error_handler: ErrorHandler | None = None

    def __post_init__(self):
        """Post instantiation hook that verifies that the player instantiation was corrected
        """
        validate_players(self.player1, self.player2)

    def play(self, starting_mark: Mark = Mark("X")) -> None:
        """Starts and handles the game until the game is over

        Args:
            starting_mark (Mark, optional): Initial Mark. Defaults to Mark("X").
        """
        game_state = GameState(Grid(), starting_mark)
        while True:
            self.renderer.render(game_state)
            if game_state.game_over:
                break
            player = self.get_current_player(game_state)
            try:
                game_state = player.make_move(game_state)
            except InvalidMove as ex:
                if self.error_handler:
                    self.error_handler(ex)

    def get_current_player(self, game_state: GameState) -> Player:
        """Determines current player base on the current game state

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Returns:
            Player: The player that has to play current turn
        """
        if game_state.current_mark is self.player1.mark:
            return self.player1
        return self.player2
```

6.1.1 `__post_init__()`

Post instantiation hook that verifies that the player instantiation was corrected

Source code in `src\backend\game\engine.py`

```
def __post_init__(self):
    """Post instantiation hook that verifies that the player instantiation was corrected
    """
    validate_players(self.player1, self.player2)
```

6.1.2 `get_current_player(game_state)`

Determines current player base on the current game state

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Returns:

Name	Type	Description
`Player`	`Player`	The player that has to play current turn

Source code in `src\backend\game\engine.py`

```
def get_current_player(self, game_state: GameState) -> Player:
    """Determines current player base on the current game state

    Args:
        game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
            that can be X, O or spaces) and a starting Mark (default X).

    Returns:
        Player: The player that has to play current turn
    """
    if game_state.current_mark is self.player1.mark:
        return self.player1
    return self.player2
```

6.1.3 `play(starting_mark=Mark('X'))`

Starts and handles the game until the game is over

Parameters:

Name	Type	Description	Default
`starting_mark`	`Mark`	Initial Mark. Defaults to Mark("X").	`Mark('X')`

Source code in `src\backend\game\engine.py`

```
def play(self, starting_mark: Mark = Mark("X")) -> None:
    """Starts and handles the game until the game is over

    Args:
        starting_mark (Mark, optional): Initial Mark. Defaults to Mark("X").
    """
    game_state = GameState(Grid(), starting_mark)
    while True:
        self.renderer.render(game_state)
        if game_state.game_over:
            break
        player = self.get_current_player(game_state)
        try:
            game_state = player.make_move(game_state)
        except InvalidMove as ex:
            if self.error_handler:
                self.error_handler(ex)
```

7. Exceptions module

Provide custom exceptions for the game.

This module allows the raise of custom exceptions.

The module contains the following custom exceptions: - `InvalidGameState` - `InvalidMove` - `UnknownGameScore`

7.1 `InvalidGameState`

Bases: `Exception`

Raised when the game state is invalid.

Source code in `src\backend\logic\exceptions.py`

```
class InvalidGameState(Exception):
    """Raised when the game state is invalid."""
```

7.2 `InvalidMove`

Bases: `Exception`

Raised when the move is invalid.

Source code in `src\backend\logic\exceptions.py`

```
class InvalidMove(Exception):
    """Raised when the move is invalid."""
```

7.3 `UnknownGameScore`

Bases: `Exception`

Raised when the game score is unknown.

Source code in `src\backend\logic\exceptions.py`

```
class UnknownGameScore(Exception):
    """Raised when the game score is unknown."""
```

8. Minimax module

Provide functions with basic AI for computer moves.

This module allows more complexity to the computer moves using different kind of algorithms.

Examples:

```
>>> from tic_tac_toe.logic.minimax import minimax
>>> from tic_tac_toe.logic.models import GameState, Grid, Mark
>>> def preview(cells):
        print(cells[:3], cells[3:6], cells[6:], sep='\n')
>>> game_state = GameState(Grid("XXO O X O"), starting_mark=Mark('X'))
>>> for move in game_state.possible_moves:
        print("Score:", minimax(move, maximizer=Mark('X')))
        preview(move.after_state.grid.cells)
        print('-' * 10)
```

The module contains the following functions: - `find_best_move(game_state: GameState)` - Return the best move available. - `minimax( move: Move, maximizer: Mark, choose_highest_score: bool = False )` - Return 1, 0 or -1 base in the result of the next move.

8.1 `find_best_move(game_state)`

Return the best move available.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X)	*required*

Returns:

Type	Description
`Move | None`	Move | None: Inmutable data Class that is strictly a data transfer object (DTO) whose main

purpose is to carry data. Consists of the mark identifying the player who made a move, a numeric zero-based index in the string of cells, and the two states before and after making a move.

Source code in `src\backend\logic\minimax.py`

```
29  def find_best_move(game_state: GameState) -> Move | None:
30      """Return the best move available.
31
32      Args:
33          game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
34              that can be X, O or spaces) and a starting Mark (default X)
35
36      Returns:
37          Move | None: Inmutable data Class that is strictly a data transfer object (DTO) whose main
38      purpose is to carry data. Consists of the mark identifying the player who made a move, a numeric
39      zero-based index in the string of cells, and the two states before and after making a move.
40      """
41      maximizer: Mark = game_state.current_mark
42      bound_minimax = partial(minimax, maximizer=maximizer)
43      return max(game_state.possible_moves, key=bound_minimax)
```

8.2 `minimax(move, maximizer, choose_highest_score=False)`

Return 1, 0 or -1 base in the result of the next move.

Parameters:

Name	Type	Description	Default
`move`	`Move`	Inmutable data Class that is strictly a data transfer object (DTO) whose main purpose is to carry data. Consists of the mark identifying the player who made a move, a numeric zero-based index in the string of cells, and the two states before and after making a move.	*required*
`maximizer`	`Mark`	Mark for the player	*required*
`choose_highest_score`	`bool`	Defaults to False.	`False`

Returns:

Name	Type	Description
`int`	`int`	returns 1, 0 or -1

Source code in `src\backend\logic\minimax.py`

```
45  def minimax(
46      move: Move, maximizer: Mark, choose_highest_score: bool = False
47  ) -> int:
48      """Return 1, 0 or -1 base in the result of the next move.
49
50      Args:
51          move (Move): Inmutable data Class that is strictly a data transfer object (DTO) whose
52              main purpose is to carry data. Consists of the mark identifying the player who made a
53              move, a numeric zero-based index in the string of cells, and the two states before
54              and after making a move.
55          maximizer (Mark): Mark for the player
56          choose_highest_score (bool, optional): Defaults to False.
57
58      Returns:
59          int: returns 1, 0 or -1
60      """
61      if move.after_state.game_over:
62          return move.after_state.evaluate_score(maximizer)
63      return (max if choose_highest_score else min)(
64          minimax(next_move, maximizer, not choose_highest_score)
65          for next_move in move.after_state.possible_moves
66      )
```

9. Models module

Provide the classes for domain model.

This module allows the creation of intances of Marks, Grids, Move and GameState.

The module contains the following class: - `Mark` - A class that handles user marks. - `Grid` - A inmutable Class that handles the grid information. - `Move` - A inmutable data class that handles move information. - `GameState` - A inmutable data class that handles game state information.

9.1 `GameState` `dataclass`

An inmutable data Class that is strictly a data transfer object (DTO) whose main purpose is to carry data, consisting of the grid of cells and the starting player's mark

Attributes:

Name	Type	Description
`grid`	`Grid`	Grid Represents the grid, 9 elements X, O or space
`starting_mark`	`Mark`	Mark = Mark("X") Represent the starting mark. Default to X

Methods:

Name	Description
`current_mark`	Cached getter of current mark.
`game_not_started`	Cached getter if current state is the initial state.
`game_over`	Cached getter to check if the game is ofver by check if there is a winner or there is a tie.
`tie`	Cached getter to check if there is a tie by checking if there is a winner or grid is empty.
`winner`	Cached getter that check if there is a winner by checking winning patterns.
`possible_moves`	Cached getter of possible moves.
`make_random_move`	Return possible move based on possible moves.
`make_move_to`	int) -> Move: Return the move to make based on index.
`evaluate_score`	Mark) -> int: Returns score based on the result of the move.

Source code in `src\backend\logic\models.py`

```python
@dataclass(frozen=True)
class GameState:
    """An inmutable data Class that is strictly a data transfer object (DTO) whose main purpose
    is to carry data, consisting of the grid of cells and the starting player's mark

    Attributes:
        grid: Grid
            Represents the grid, 9 elements X, O or space
        starting_mark: Mark = Mark("X")
            Represent the starting mark. Default to X

    Methods:
        current_mark(self) -> Mark:
            Cached getter of current mark.
        game_not_started(self) -> bool:
            Cached getter if current state is the initial state.
        game_over(self) -> bool:
            Cached getter to check if the game is ofver by check if there is a winner or
            there is a tie.
        tie(self) -> bool:
            Cached getter to check if there is a tie by checking if
            there is a winner or grid is empty.
        winner(self) -> Mark | None:
            Cached getter that check if there is a winner by checking winning patterns.
        possible_moves(self) -> list[Move]:
            Cached getter of possible moves.
        make_random_move(self) -> Move | None:
            Return possible move based on possible moves.
        make_move_to(self, index: int) -> Move:
            Return the move to make based on index.
        evaluate_score(self, mark: Mark) -> int:
            Returns score based on the result of the move.
    """
    grid: Grid
    starting_mark: Mark = Mark("X")

    def __post_init__(self) -> None:
        """Post instantiation hook that verifies that the gamestate is correct
        """
        validate_game_state(self)

    @cached_property
    def current_mark(self) -> Mark:
        """Cached getter of current mark.

        Returns:
            Mark: Mark of current state.
        """
        if self.grid.x_count == self.grid.o_count:
            return self.starting_mark
        return self.starting_mark.other

    @cached_property
    def game_not_started(self) -> bool:
        """Cached getter if current state is the initial state.

        Returns:
            bool: Rather current turn is the first turn or not
        """
        return self.grid.empty_count == 9

    @cached_property
    def game_over(self) -> bool:
        """Cached getter to check if the game is ofver by check if there is a winner or
        there is a tie.

        Returns:
            bool: Rather the game is over or not.
        """
        return self.winner is not None or self.tie

    @cached_property
    def tie(self) -> bool:
        """Cached getter to check if there is a tie by checking if
        there is a winner or grid is empty.

        Returns:
            bool: Rather the game has a winner or grid is empty
        """
        return self.winner is None and self.grid.empty_count == 0

    @cached_property
    def winner(self) -> Mark | None:
        """Cached getter that check if there is a winner by checking winning patterns.

        Returns:
            Mark | None: Could be X, O or None.
        """
        for pattern in WINNING_PATTERNS:
            for mark in Mark:
                if re.match(pattern.replace("?", mark), self.grid.cells):
                    return mark
        return None

    @cached_property
    def winning_cells(self) -> list[int]:
        """Chaced getter with information of position of marks in winning cell

        Returns:
            list[int]: List of positions of marks in winning cell
```

```
        """
        for pattern in WINNING_PATTERNS:
            for mark in Mark:
                if re.match(pattern.replace("?", mark), self.grid.cells):
                    return [
                        match.start()
                        for match in re.finditer(r"\?", pattern)
                    ]
        return []

    @cached_property
    def possible_moves(self) -> list[Move]:
        """Cached getter of possible moves.

        Returns:
            list[Move]: list of possible moves
        """
        moves = []
        if not self.game_over:
            for match in re.finditer(r"\s", self.grid.cells):
                moves.append(self.make_move_to(match.start()))
        return moves

    def make_random_move(self) -> Move | None:
        """Return possible move based on possible moves.

        Returns:
            Move | None: Snapshot of moves.
        """
        try:
            return random.choice(self.possible_moves)
        except IndexError:
            return None

    def make_move_to(self, index: int) -> Move:
        """Return the move to make based on index.

        Args:
            index (int): Position of the move.

        Raises:
            InvalidMove: Exception when a invalid move is selected

        Returns:
            Move: Snapshot of moves
        """
        if self.grid.cells[index] != " ":
            raise InvalidMove("Cell is not empty")
        return Move(
            mark=self.current_mark,
            cell_index=index,
            before_state=self,
            after_state=GameState(
                Grid(
                    self.grid.cells[:index]
                    + self.current_mark
                    + self.grid.cells[index + 1:]
                ),
                self.starting_mark,
            ),
        )

    def evaluate_score(self, mark: Mark) -> int:
        """Returns score based on the result of the move.

        Args:
            mark (Mark): Class that handles user marks.

        Raises:
            UnknownGameScore: Exception when no score can be calculated.

        Returns:
            int: score for the game.
        """
        if self.game_over:
            if self.tie:
                return 0
            if self.winner is mark:
                return 1
            return -1
        raise UnknownGameScore("Game is not over yet")
```

9.1.1 `current_mark: Mark` `cached` `property`

Cached getter of current mark.

Returns:

Name	Type	Description
`Mark`	`Mark`	Mark of current state.

9.1.2 `game_not_started: bool` `cached` `property`

Cached getter if current state is the initial state.

Returns:

Name	Type	Description
`bool`	`bool`	Rather current turn is the first turn or not

9.1.3 `game_over: bool` `cached` `property`

Cached getter to check if the game is ofver by check if there is a winner or there is a tie.

Returns:

Name	Type	Description
`bool`	`bool`	Rather the game is over or not.

9.1.4 `possible_moves: list[Move]` `cached` `property`

Cached getter of possible moves.

Returns:

Type	Description
`list[Move]`	list[Move]: list of possible moves

9.1.5 `tie: bool` `cached` `property`

Cached getter to check if there is a tie by checking if there is a winner or grid is empty.

Returns:

Name	Type	Description
`bool`	`bool`	Rather the game has a winner or grid is empty

9.1.6 `winner: Mark | None` `cached` `property`

Cached getter that check if there is a winner by checking winning patterns.

Returns:

Type	Description
`Mark | None`	Mark | None: Could be X, O or None.

9.1.7 `winning_cells: list[int]` `cached` `property`

Chaced getter with information of position of marks in winning cell

Returns:

Type	Description
`list[int]`	list[int]: List of positions of marks in winning cell

9.1.8 `__post_init__()`

Post instantiation hook that verifies that the gamestate is correct

Source code in `src\backend\logic\models.py`

```
167    def __post_init__(self) -> None:
168        """Post instantiation hook that verifies that the gamestate is correct
169        """
170        validate_game_state(self)
```

9.1.9 `evaluate_score(mark)`

Returns score based on the result of the move.

Parameters:

Name	Type	Description	Default
`mark`	`Mark`	Class that handles user marks.	*required*

Raises:

Type	Description
`UnknownGameScore`	Exception when no score can be calculated.

Returns:

Name	Type	Description
`int`	`int`	score for the game.

Source code in `src\backend\logic\models.py`

```python
def evaluate_score(self, mark: Mark) -> int:
    """Returns score based on the result of the move.

    Args:
        mark (Mark): Class that handles user marks.

    Raises:
        UnknownGameScore: Exception when no score can be calculated.

    Returns:
        int: score for the game.
    """
    if self.game_over:
        if self.tie:
            return 0
        if self.winner is mark:
            return 1
        return -1
    raise UnknownGameScore("Game is not over yet")
```

9.1.10 `make_move_to(index)`

Return the move to make based on index.

Parameters:

Name	Type	Description	Default
`index`	`int`	Position of the move.	*required*

Raises:

Type	Description
`InvalidMove`	Exception when a invalid move is selected

Returns:

Name	Type	Description
`Move`	`Move`	Snapshot of moves

Source code in `src\backend\logic\models.py`

```python
def make_move_to(self, index: int) -> Move:
    """Return the move to make based on index.

    Args:
        index (int): Position of the move.

    Raises:
        InvalidMove: Exception when a invalid move is selected

    Returns:
        Move: Snapshot of moves
    """
    if self.grid.cells[index] != " ":
        raise InvalidMove("Cell is not empty")
    return Move(
        mark=self.current_mark,
        cell_index=index,
        before_state=self,
        after_state=GameState(
            Grid(
                self.grid.cells[:index]
                + self.current_mark
                + self.grid.cells[index + 1:]
            ),
            self.starting_mark,
        ),
    )
```

9.1.11 `make_random_move()`

Return possible move based on possible moves.

Returns:

Type	Description
`Move | None`	Move | None: Snapshot of moves.

Source code in `src\backend\logic\models.py`

```
254  def make_random_move(self) -> Move | None:
255      """Return possible move based on possible moves.
256
257      Returns:
258          Move | None: Snapshot of moves.
259      """
260      try:
261          return random.choice(self.possible_moves)
262      except IndexError:
263          return None
```

9.2 `Grid` `dataclass`

An inmutable Class that handles the grid. It is instantiate as a empty grid 9 spaces as default. It runs as Post instantiation hook that verifies that grid composition. Allowed cell position: 9 elements (X, O, or space).

Attributes:

Name	Type	Description
`cells`	`str`	str Represents the grid, 9 elements X, O or space.

Methods:

Name	Description
`x_count`	Cached getter of total of X.
`o_count`	Cached getter of total of O
`empty_count`	Cached getter of total of spaces

Raises:

Type	Description
`ValueError`	Raises ValueError if

Source code in `src\backend\logic\models.py`

```python
@dataclass(frozen=True)
class Grid:
    """An inmutable Class that handles the grid. It is instantiate as a empty grid 9 spaces as
    default. It runs as Post instantiation hook that verifies that grid composition. Allowed
    cell position: 9 elements (X, O, or space).

    Attributes:
        cells: str
            Represents the grid, 9 elements X, O or space.

    Methods:
        x_count(self) -> int:
            Cached getter of total of X.
        o_count(self) -> int:
            Cached getter of total of O
        empty_count(self) -> int:
            Cached getter of total of spaces

    Raises:
        ValueError: Raises ValueError if
    """
    cells: str = " " * 9

    def __post_init__(self) -> None:
        """Post instantiation hook that verifies that the grid is compose of 9 elements (X, O, or
        space)"""
        validate_grid(self)

    @cached_property
    def x_count(self) -> int:
        """Cached getter of total of X

        Returns:
            int: Total of X
        """
        return self.cells.count("X")

    @cached_property
    def o_count(self) -> int:
        """Cached getter of total of O

        Returns:
            int: Total of Y
        """
        return self.cells.count("O")

    @cached_property
    def empty_count(self) -> int:
        """Cached getter of total of spaces

        Returns:
            int: Total of spaces
        """
        return self.cells.count(" ")
```

9.2.1 `empty_count: int` `cached` `property`

Cached getter of total of spaces

Returns:

Name	Type	Description
`int`	`int`	Total of spaces

9.2.2 `o_count: int` `cached` `property`

Cached getter of total of O

Returns:

Name	Type	Description
`int`	`int`	Total of Y

9.2.3 `x_count: int` `cached` `property`

Cached getter of total of X

Returns:

Name	Type	Description
`int`	`int`	Total of X

9.2.4 `__post_init__()`

Post instantiation hook that verifies that the grid is compose of 9 elements (X, O, or space)

Source code in `src\backend\logic\models.py`

```
78  def __post_init__(self) -> None:
79      """Post instantiation hook that verifies that the grid is compose of 9 elements (X, O, or
80      space)"""
81      validate_grid(self)
```

9.3 `Mark`

Bases: `StrEnum`

A class that handles user marks. it can be CROSS or X, or NAUGHT or O. Extends enum.StrEnum class. It can be CROSS or X, or NAUGHT or O.

Methods:

Name	Description
`other`	Returns the opposite MARK space).

Returns:

Type	Description
`Mark`	CROSS or X, or NAUGHT or O

Source code in `src\backend\logic\models.py`

```
class Mark(enum.StrEnum):
    """A class that handles user marks. it can be CROSS or X, or NAUGHT or O. Extends enum.StrEnum
        class. It can be CROSS or X, or NAUGHT or O.

    Methods:
        other(self) -> "Mark":
            Returns the opposite MARK
            space).

    Returns:
         (Mark): CROSS or X, or NAUGHT or O
    """
    CROSS = "X"
    NAUGHT = "O"

    @property
    def other(self) -> "Mark":
        """Returns the opposite MARK.

        Returns:
            Mark: can be X or O.
        """
        return Mark.CROSS if self is Mark.NAUGHT else Mark.NAUGHT
```

9.3.1 `other: Mark` `property`

Returns the opposite MARK.

Returns:

Name	Type	Description
`Mark`	`Mark`	can be X or O.

9.4 `Move` `dataclass`

An inmutable data class that is strictly a data transfer object (DTO) whose main purpose is to carry data. Consists of the mark identifying the player who made a move, a numeric zero-based index in the string of cells, and the two states before and after making a move.

Attributes:

Name	Type	Description
`mark`	`Mark`	Mark Represent the mark of the player.
`cell_index`	`int`	int Represent the position to play.
`before_state`	`GameState`	"GameState" Represent the game state before the move.
`after_state`	`GameState`	"GameState" Represent the game state after the move.

Source code in `src\backend\logic\models.py`

```python
@dataclass(frozen=True)
class Move:
    """An inmutable data class that is strictly a data transfer object (DTO) whose main purpose
    is to carry data. Consists of the mark identifying the player who made a move, a numeric
    zero-based index in the string of cells, and the two states before and after making a move.

    Attributes:
        mark: Mark
            Represent the mark of the player.
        cell_index: int
            Represent the position to play.
        before_state: "GameState"
            Represent the game state before the move.
        after_state: "GameState"
            Represent the game state after the move.
    """
    mark: Mark
    cell_index: int
    before_state: "GameState"
    after_state: "GameState"
```

10. Players module

Provide the classes to instantiate players, human or computer.

This module allows the creation of different categories of players.

Examples:

```
>>> player1 = RandomComputerPlayer(Mark("X"))
>>> player2 = MinimaxComputerPlayer(Mark("O"))
```

The module contains the following class: - `Player` - ABC - `ComputerPlayer` - ABC. Extension of class Player. - `RandomComputerPlayer` - Extension of class ComputerPlayer. - `MinimaxComputerPlayer` - ABC. Extension of class ComputerPlayer.

10.1 `ComputerPlayer`

Bases: `Player`

Abstract class for the creation of computer players. Extends as metaclass, abc.ABCMeta. Extends Player abstract class An instance of subclass of the Player class that represents a human or computer.

Attributes:

Name	Type	Description
`mark`		Mark An instance of Mark class that handles user marks

Methods:

Name	Description
`get_move`	GameState) -> Move | None: Return the current computer player's move in the given game state
`get_computer_move`	GameState) -> Move | None: Determines current player base on the current game state. Abstract method

Source code in `src\backend\game\players.py`

```
class ComputerPlayer(Player, metaclass=abc.ABCMeta):
    """Abstract class for the creation of computer players. Extends as metaclass, abc.ABCMeta.
        Extends Player abstract class An instance of subclass of the Player class that represents
        a human or computer.

    Attributes:
        mark: Mark
            An instance of Mark class that handles user marks

    Methods:
        get_move(self, game_state: GameState) -> Move | None:
            Return the current computer player's move in the given game state
        get_computer_move(self, game_state: GameState) -> Move | None:
            Determines current player base on the current game state. Abstract method
    """
    def __init__(self, mark: Mark, delay_seconds: float = 0.25) -> None:
        """
        Args:
            mark (Mark): An instance class that handles user marks
            delay_seconds (float, optional): Represents the delay time for the computer
                to player. Defaults to 0.25.
        """
        super().__init__(mark)
        self.delay_seconds = delay_seconds

    def get_move(self, game_state: GameState) -> Move | None:
        """Return the current computer player's move in the given game state

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Returns:
            Move | None: return a move class or none
        """
        time.sleep(self.delay_seconds)
        return self.get_computer_move(game_state)

    @abc.abstractmethod
    def get_computer_move(self, game_state: GameState) -> Move | None:
        """Return the computer's move in the given game state."""
```

10.1.1 `__init__(mark, delay_seconds=0.25)`

Parameters:

Name	Type	Description	Default
`mark`	`Mark`	An instance class that handles user marks	*required*
`delay_seconds`	`float`	Represents the delay time for the computer to player. Defaults to 0.25.	`0.25`

Source code in `src\backend\game\players.py`

```
def __init__(self, mark: Mark, delay_seconds: float = 0.25) -> None:
    """
    Args:
        mark (Mark): An instance class that handles user marks
        delay_seconds (float, optional): Represents the delay time for the computer
            to player. Defaults to 0.25.
    """
    super().__init__(mark)
    self.delay_seconds = delay_seconds
```

10.1.2 `get_computer_move(game_state)` `abstractmethod`

Return the computer's move in the given game state.

Source code in `src\backend\game\players.py`

```
@abc.abstractmethod
def get_computer_move(self, game_state: GameState) -> Move | None:
    """Return the computer's move in the given game state."""
```

10.1.3 `get_move(game_state)`

Return the current computer player's move in the given game state

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Returns:

Type	Description
`Move | None`	Move | None: return a move class or none

Source code in `src\backend\game\players.py`

```
def get_move(self, game_state: GameState) -> Move | None:
    """Return the current computer player's move in the given game state

    Args:
        game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
            that can be X, O or spaces) and a starting Mark (default X).

    Returns:
        Move | None: return a move class or none
    """
    time.sleep(self.delay_seconds)
    return self.get_computer_move(game_state)
```

10.2 `MinimaxComputerPlayer`

Bases: `ComputerPlayer`

A class for the creation of computer players with move based on minimax algorithm. Extends ComputerPlayer, an abstract class for the creation of computer players.

Methods:

Name	Description
`get_computer_move`	GameState) -> Move | None: Return the current computer player's move in the given game state.

Source code in `src\backend\game\players.py`

```
class MinimaxComputerPlayer(ComputerPlayer):
    """A class for the creation of computer players with move based on minimax algorithm.
    Extends ComputerPlayer, an abstract class for the creation of computer players.

    Methods:
        get_computer_move(self, game_state: GameState) -> Move | None:
            Return the current computer player's move in the given game state.
    """
    def get_computer_move(self, game_state: GameState) -> Move | None:
        """Return the current computer player's move in the given game state using
        minimax algorithm.

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Returns:
            Move | None: return a move class or none.
        """
        if game_state.game_not_started:
            return game_state.make_random_move()
        return find_best_move(game_state)
```

10.2.1 `get_computer_move(game_state)`

Return the current computer player's move in the given game state using minimax algorithm.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Returns:

Type	Description
`Move | None`	Move | None: return a move class or none.

Source code in `src\backend\game\players.py`

```
def get_computer_move(self, game_state: GameState) -> Move | None:
    """Return the current computer player's move in the given game state using
    minimax algorithm.

    Args:
        game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
            that can be X, O or spaces) and a starting Mark (default X).

    Returns:
        Move | None: return a move class or none.
    """
    if game_state.game_not_started:
        return game_state.make_random_move()
    return find_best_move(game_state)
```

10.3 `Player`

Abstract class for the creation of players. Extends as metaclass, abc.ABCMeta.

Attributes:

Name	Type	Description
`mark`		Mark An instance of Mark class that handles user marks.

Methods:

Name	Description
`make_move`	GameState) -> GameState: Handles the current player move.
`get_move`	GameState) -> Move | None: Determines current player base on the current game state. Abstract method.

Source code in `src\backend\game\players.py`

```python
class Player(metaclass=abc.ABCMeta):
    """Abstract class for the creation of players. Extends as metaclass, abc.ABCMeta.

    Attributes:
        mark: Mark
            An instance of Mark class that handles user marks.

    Methods:
        make_move(self, game_state: GameState) -> GameState:
            Handles the current player move.
        get_move(self, game_state: GameState) -> Move | None:
            Determines current player base on the current game state. Abstract method.
    """
    def __init__(self, mark: Mark) -> None:
        """Initializes the instance based on mark provided.
        Args:
            mark (Mark): An instance class that handles user marks.
        """
        self.mark = mark

    def make_move(self, game_state: GameState) -> GameState:
        """Handles the current player move which depends on the get_move method
        implemented in each subclass if it's the given player's turn and whether the move exists.

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Raises:
            InvalidMove: Exception when a invalid move is selected
        Returns:
            GameState: current GameState, consisting of a current Grid (9 elemets that
            that can be X, O or spaces) and a starting Mark (default X).
        """
        if self.mark is game_state.current_mark:
            if move := self.get_move(game_state):
                return move.after_state
            raise InvalidMove("No more possible moves")
        raise InvalidMove("It's the other player's turn")

    @abc.abstractmethod
    def get_move(self, game_state: GameState) -> Move | None:
        """Return the current player's move in the given game state."""
```

10.3.1 `__init__(mark)`

Initializes the instance based on mark provided.

Parameters:

Name	Type	Description	Default
`mark`	`Mark`	An instance class that handles user marks.	*required*

Source code in `src\backend\game\players.py`

```
36  def __init__(self, mark: Mark) -> None:
37      """Initializes the instance based on mark provided.
38      Args:
39          mark (Mark): An instance class that handles user marks.
40      """
41      self.mark = mark
```

10.3.2 `get_move(game_state)` `abstractmethod`

Return the current player's move in the given game state.

Source code in `src\backend\game\players.py`

```
63  @abc.abstractmethod
64  def get_move(self, game_state: GameState) -> Move | None:
65      """Return the current player's move in the given game state."""
```

10.3.3 `make_move(game_state)`

Handles the current player move which depends on the get_move method implemented in each subclass if it's the given player's turn and whether the move exists.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Raises:

Type	Description
`InvalidMove`	Exception when a invalid move is selected

Returns:

Name	Type	Description
`GameState`	`GameState`	current GameState, consisting of a current Grid (9 elemets that
	`GameState`	that can be X, O or spaces) and a starting Mark (default X).

Source code in `src\backend\game\players.py`

```
def make_move(self, game_state: GameState) -> GameState:
    """Handles the current player move which depends on the get_move method
    implemented in each subclass if it's the given player's turn and whether the move exists.

    Args:
        game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
            that can be X, O or spaces) and a starting Mark (default X).

    Raises:
        InvalidMove: Exception when a invalid move is selected
    Returns:
        GameState: current GameState, consisting of a current Grid (9 elemets that
        that can be X, O or spaces) and a starting Mark (default X).
    """
    if self.mark is game_state.current_mark:
        if move := self.get_move(game_state):
            return move.after_state
        raise InvalidMove("No more possible moves")
    raise InvalidMove("It's the other player's turn")
```

10.4 `RandomComputerPlayer`

Bases: `ComputerPlayer`

Class for the creation of computer players with random moves. Extends ComputerPlayer abstract class.

Methods:

Name	Description
`get_computer_move`	GameState) -> Move | None: Return the current computer player's random move in the given game state.

Source code in `src\backend\game\players.py`

```
class RandomComputerPlayer(ComputerPlayer):
    """Class for the creation of computer players with random moves. Extends ComputerPlayer
    abstract class.

    Methods:
        get_computer_move(self, game_state: GameState) -> Move | None:
            Return the current computer player's random move in the given game state.
    """
    def get_computer_move(self, game_state: GameState) -> Move | None:
        """Return the current computer player's random move in the given game state.

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Returns:
            Move | None: return a move class or none
        """
        return game_state.make_random_move()
```

10.4.1 `get_computer_move(game_state)`

Return the current computer player's random move in the given game state.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Returns:

Type	Description
`Move | None`	Move | None: return a move class or none

Source code in `src\backend\game\players.py`

```
117  def get_computer_move(self, game_state: GameState) -> Move | None:
118      """Return the current computer player's random move in the given game state.
119
120      Args:
121          game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
122              that can be X, O or spaces) and a starting Mark (default X).
123
124      Returns:
125          Move | None: return a move class or none
126      """
127      return game_state.make_random_move()
```

11. Renderers module

Provide classes for visual and state rendering.

This module allows the creation of UI rendering.

The module contains the following class: - `Renderer`

11.1 `Renderer`

Abstract class for the creation of visual and state rendering. Extends as metaclass, abc.ABCMeta.

Methods:

Name	Description
`def render`	GameState) -> None: Render the current game state.

Source code in `src\backend\game\renderers.py`

```python
class Renderer(metaclass=abc.ABCMeta):
    """Abstract class for the creation of visual and state rendering. Extends as
        metaclass, abc.ABCMeta.

    Methods:
        def render(self, game_state: GameState) -> None:
            Render the current game state.
    """
    @abc.abstractmethod
    def render(self, game_state: GameState) -> None:
        """Render the current game state."""

    def placerholder(self) -> None:
        """Render the current game state."""
```

11.1.1 `placerholder()`

Render the current game state.

Source code in `src\backend\game\renderers.py`

```python
    def placerholder(self) -> None:
        """Render the current game state."""
```

11.1.2 `render(game_state)` `abstractmethod`

Render the current game state.

Source code in `src\backend\game\renderers.py`

```python
    @abc.abstractmethod
    def render(self, game_state: GameState) -> None:
        """Render the current game state."""
```

12. Validators module

Provide functions to validate grid and gamestates

This module allows more validate different game states and the UI grid.

Examples:

```
>>> from tic_tac_toe.logic.models import Grid
>>> # Create an empty grid
>>> Grid()
Grid(cells='         ')
```

```
>>> # Create a grid of a particular cell combination
>>> Grid("XXOXO O  ")
Grid(cells='XXOXO O  ')
```

```
>>> # Don't create a grid with too few cells
>>> Grid("XO")
Traceback (most recent call last):
...
ValueError: Must contain 9 cells of: X, O, or space
```

The module contains the following functions: - `validate_grid(grid: Grid)` - Verify that the grid is compose of 9 elements (X, O, or space). - `validate_game_state(game_state: GameState)` - Verify a correct gamestate, raises exceptions if is not. - `validate_number_of_marks(grid: Grid)` - Verify the correct quantity of X and O. Differente between Xs and Os must not exceed 1. - `validate_starting_mark(grid: Grid, starting_mark: Mark)` - Verify that the correct starting mark is selected. - `validate_winner(`
`    grid: Grid, starting_mark: Mark, winner: Mark | None`
`)` - Validate winner by verifying total of Xs and Os. - `validate_players(player1: Player, player2: Player)` - Validate the correct instantiation of Players.

12.1 `validate_game_state(game_state)`

Verify a correct gamestate, raises exceptions if is not.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X)	*required*

Source code in `src\backend\logic\validators.py`

```
60  def validate_game_state(game_state: GameState) -> None:
61      """Verify a correct gamestate, raises exceptions if is not.
62
63      Args:
64          game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
65              that can be X, O or spaces) and a starting Mark (default X)
66      """
67      validate_number_of_marks(game_state.grid)
68      validate_starting_mark(game_state.grid, game_state.starting_mark)
69      validate_winner(
70          game_state.grid, game_state.starting_mark, game_state.winner
71      )
```

12.2 `validate_grid(grid)`

Verify that the grid is compose of 9 elements (X, O, or space). Raises ValueError it the composition is incorrect

Parameters:

Name	Type	Description	Default
`grid`	`Grid`	Grid with 9 elements(X, O or space)	*required*

Raises:

Type	Description
`ValueError`	"Must contain 9 cells of: X, O, or space"

Source code in `src\backend\logic\validators.py`

```
47  def validate_grid(grid: Grid) -> None:
48      """Verify that the grid is compose of 9 elements (X, O, or
49      space). Raises ValueError it the composition is incorrect
50
51      Args:
52          grid (Grid): Grid with 9 elements(X, O or space)
53
54      Raises:
55          ValueError: "Must contain 9 cells of: X, O, or space"
56      """
57      if not re.match(r"^[\sXO]{9}$", grid.cells):
58          raise ValueError("Must contain 9 cells of: X, O, or space")
```

12.3 `validate_number_of_marks(grid)`

Verify the correct quantity of X and O. Differente between Xs and Os must not exceed 1.

Parameters:

Name	Type	Description	Default
`grid`	`Grid`	Grid with 9 elements(X, O or space).	*required*

Raises:

Type	Description
`InvalidGameState`	Exception that represent a invalidad game state.

Source code in `src\backend\logic\validators.py`

```
73  def validate_number_of_marks(grid: Grid) -> None:
74      """Verify the correct quantity of X and O. Differente between Xs and Os
75      must not exceed 1.
76
77      Args:
78          grid (Grid): Grid with 9 elements(X, O or space).
79
80      Raises:
81          InvalidGameState: Exception that represent a invalidad game state.
82      """
83      if abs(grid.x_count - grid.o_count) > 1:
84          raise InvalidGameState("Wrong number of Xs and Os")
```

12.4 `validate_players(player1, player2)`

Validates the correct instantiation of Players. Players must use different marks

Parameters:

Name	Type	Description	Default
`player1`	`Player`	An instance of subclass of the Player class that represents a human or computer	*required*
`player2`	`Player`	An instance of subclass of the Player class that represents a human or computer	*required*

Raises:

Type	Description
`ValueError`	Exception that represent a invalidad game state.

Source code in `src\backend\logic\validators.py`

```
129  def validate_players(player1: Player, player2: Player) -> None:
130      """Validates the correct instantiation of Players. Players must use different marks
131
132      Args:
133          player1 (Player): An instance of subclass of the Player class that
134              represents a human or computer
135          player2 (Player): An instance of subclass of the Player class that
136              represents a human or computer
137
138      Raises:
139          ValueError: Exception that represent a invalidad game state.
140      """
141      if player1.mark is player2.mark:
142          raise ValueError("Players must use different marks")
```

12.5 `validate_starting_mark(grid, starting_mark)`

Verifies that the correct starting mark is selected,

Parameters:

Name	Type	Description	Default
`grid`	`Grid`	Grid with 9 elements(X, O or space)	*required*
`starting_mark`	`Mark`	Mark X	*required*

Raises:

Type	Description
`InvalidGameState`	Exception that represent a invalidad game state.

Source code in `src\backend\logic\validators.py`

```python
def validate_starting_mark(grid: Grid, starting_mark: Mark) -> None:
    """Verifies that the correct starting mark is selected,

    Args:
        grid (Grid): Grid with 9 elements(X, O or space)
        starting_mark (Mark): Mark X

    Raises:
        InvalidGameState: Exception that represent a invalidad game state.
    """
    if grid.x_count > grid.o_count:
        if starting_mark != "X":
            raise InvalidGameState("Wrong starting mark")
    elif grid.o_count > grid.x_count and starting_mark != "O":
        raise InvalidGameState("Wrong starting mark")
```

12.6 `validate_winner(grid, starting_mark, winner)`

Validate winner by verifying total of Xs and Os.

Parameters:

Name	Type	Description	Default
`grid`	`Grid`	Grid with 9 elements(X, O or space)	*required*
`starting_mark`	`Mark`	Mark that represents the winner	*required*

Raises:

Type	Description
`InvalidGameState`	Exception that represent a invalidad game state.

Source code in `src\backend\logic\validators.py`

```python
def validate_winner(
    grid: Grid, starting_mark: Mark, winner: Mark | None
) -> None:
    """Validate winner by verifying total of Xs and Os.

    Args:
        grid (Grid): Grid with 9 elements(X, O or space)
        starting_mark (Mark): Mark that represents the winner

    Raises:
        InvalidGameState: Exception that represent a invalidad game state.
    """
    if winner == "X":
        if starting_mark == "X":
            if grid.x_count <= grid.o_count:
                raise InvalidGameState("Wrong number of Xs")
        else:
            if grid.x_count != grid.o_count:
                raise InvalidGameState("Wrong number of Xs")
    elif winner == "O":
        if starting_mark == "O":
            if grid.o_count <= grid.x_count:
                raise InvalidGameState("Wrong number of Os")
        else:
            if grid.o_count != grid.x_count:
                raise InvalidGameState("Wrong number of Os")
```

13. Args module

Provide the classes and functions to handle CLI arguments and options.

This module allows the handle CLI arguments and options.

The module contains the following classes and functions: - `Args(NamedTuple)` - A class to create a namedtuple to handle arguments for CLI - `parse_args` - Returns type handled tuple with information about the players and initial Mark.

13.1 `Args`

Bases: `NamedTuple`

A class that handle arguments for CLI. Extends NamedTuple

Attributes:

Name	Type	Description
`player1`	`Player`	Player An instance of subclass of the Player class that represents a human or computer.
`player2`	`Player`	Player An instance of subclass of the Player class that represents a human or computer.
`renderer`	`Player`	Renderer An instance of subclass of the Renderer class that handles UI rendering.

Source code in `src\frontend\console\args.py`

```
28  class Args(NamedTuple):
29      """A class that handle arguments for CLI. Extends NamedTuple
30
31      Attributes:
32          player1: Player
33              An instance of subclass of the Player class that represents a human or computer.
34          player2: Player
35              An instance of subclass of the Player class that represents a human or computer.
36          renderer: Renderer
37              An instance of subclass of the Renderer class that handles UI rendering.
38      """
39      player1: Player
40      player2: Player
41      starting_mark: Mark
```

13.2 `parse_args()`

Returns type handled tuple with information about the players and initial Mark.

Returns:

Name	Type	Description
`Args`	`Args`	tuple[Player, Player, Mark] tuple with players and Mark

Source code in `src\frontend\console\args.py`

```
43  def parse_args() -> Args:
44      """Returns type handled tuple with information about the players and initial Mark.
45
46      Returns:
47          Args: tuple[Player, Player, Mark] tuple with players and Mark
48      """
49
50      parser = argparse.ArgumentParser()
51      parser.add_argument(
52          "-X",
53          dest="player_x",
54          choices=PLAYER_CLASSES.keys(),
55          default="human",
56      )
57      parser.add_argument(
58          "-O",
59          dest="player_o",
60          choices=PLAYER_CLASSES.keys(),
61          default="minimax",
62      )
63      parser.add_argument(
64          "-s",
65          "--starting",
66          dest="starting_mark",
67          choices=Mark,
68          type=Mark,
69          default="X",
70      )
71      args = parser.parse_args()
72
73      player1 = PLAYER_CLASSES[args.player_x](Mark("X"))
74      player2 = PLAYER_CLASSES[args.player_o](Mark("O"))
75
76      if args.starting_mark == "O":
77          player1, player2 = player2, player1
78
79      return Args(player1, player2, args.starting_mark)
```

14. CLI module

Provide functions to handle CLI start game.

This module allows the handle CLI start game

Examples:

```
>>> python -m console -X human -O human
>>> tictactoe -X human -O human
```

The module contains the following classes and functions: - `main` - Handle start game from CLI

14.1 `main()`

Handle start game from CLI

Source code in `src\frontend\console\cli.py`

```
18  def main() -> None:
19      """Handle start game from CLI
20      """
21      player1, player2, starting_mark = parse_args()
22      TicTacToe(player1, player2, ConsoleRenderer()).play(starting_mark)
```

15. Players module

Provide the classes handle human players.

This module allows the handle CLI arguments and options.

The module contains the following classes: - `ConsolePlayer(Player)` - A class that represents human players.

The module contains the following functions: - `grid_to_index(grid: str) -> int:` - Return infex of the next move.

15.1 `ConsolePlayer`

Bases: `Player`

A class that represents human players. Extend abstract class for the creation of players.

Methods:

Name	Description
`get_move`	GameState) -> Move | None: Return the current player's move based on the human player choice.

Source code in `src\frontend\console\players.py`

```python
class ConsolePlayer(Player):
    """A class that represents human players. Extend abstract class for the creation of players.

    Methods:
        get_move(self, game_state: GameState) -> Move | None:
            Return the current player's move based on the human player choice.
    """
    def get_move(self, game_state: GameState) -> Move | None:
        """Return the current player's move based on the human player choice.

        Args:
            game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
                that can be X, O or spaces) and a starting Mark (default X).

        Returns:
            Move | None: return a move class or none.
        """
        while not game_state.game_over:
            try:
                index = grid_to_index(input(f"{self.mark}'s move: ").strip())
            except ValueError:
                print("Please provide coordinates in the form of A1 or 1A")
            else:
                try:
                    return game_state.make_move_to(index)
                except InvalidMove:
                    print("That cell is already occupied.")
        return None
```

15.1.1 `get_move(game_state)`

Return the current player's move based on the human player choice.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Returns:

Type	Description
`Move | None`	Move | None: return a move class or none.

Source code in `src\frontend\console\players.py`

```
24  def get_move(self, game_state: GameState) -> Move | None:
25      """Return the current player's move based on the human player choice.
26
27      Args:
28          game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
29              that can be X, O or spaces) and a starting Mark (default X).
30
31      Returns:
32          Move | None: return a move class or none.
33      """
34      while not game_state.game_over:
35          try:
36              index = grid_to_index(input(f"{self.mark}'s move: ").strip())
37          except ValueError:
38              print("Please provide coordinates in the form of A1 or 1A")
39          else:
40              try:
41                  return game_state.make_move_to(index)
42              except InvalidMove:
43                  print("That cell is already occupied.")
44      return None
```

15.2 `grid_to_index(grid)`

Return infex of the next move. Input must be in format A1 or 1A. Letters can be A, B or C, and number 1, 2, or 3.

Parameters:

Name	Type	Description	Default
`grid`	`str`	String with the position option from human input	*required*

Raises:

Type	Description
`ValueError`	Exception when a value of the index is outside bounds.

Returns:

Name	Type	Description
`int`	`int`	index of move

Source code in `src\frontend\console\players.py`

```
46  def grid_to_index(grid: str) -> int:
47      """Return infex of the next move. Input must be in format A1 or 1A.
48      Letters can be A, B or C, and number 1, 2, or 3.
49
50      Args:
51          grid (str): String with the position option from human input
52
53      Raises:
54          ValueError: Exception when a value of the index is outside bounds.
55
56      Returns:
57          int: index of move
58      """
59      if re.match(r"[abcABC][123]", grid):
60          col, row = grid
61      elif re.match(r"[123][abcABC]", grid):
62          row, col = grid
63      else:
64          raise ValueError("Invalid grid coordinates")
65      return 3 * (int(row) - 1) + (ord(col.upper()) - ord("A"))
```

16. Renderers module

Module with classes and methods to handle UI Provide the classes handle human players.

This module allows the handle CLI arguments and options.

The module contains the following classes: - `ConsoleRenderer(Renderer)` - A class to handler render UI in the console.

The module contains the following functions: - `clear_screen() -> None:` - Clear console, like command reset on modern Linux systems. - `blink(text: str) -> str:` - Modify information to be print to console and add slow blinking - `print_blinking(cells: Iterable[str], positions: Iterable[int]) -> None:` - Add blinking ANSI code to positions in cells. - `print_solid(cells: Iterable[str]) -> None:` - Render game UI. It uses textwrap and format to replace on the correct position..

16.1 `ConsoleRenderer`

Bases: `Renderer`

A class to handler render UI in the console. Extend abstract class Renderar for the creation of visual and state rendering

Methods:

Name	Description
`render`	GameState) -> None: Renders a new UI depending on game state.

Source code in `src\frontend\console\renderers.py`

```
25  class ConsoleRenderer(Renderer):
26      """A class to handler render UI in the console. Extend abstract class Renderar
27          for the creation of visual and state rendering
28
29      Methods:
30          render(self, game_state: GameState) -> None:
31              Renders a new UI depending on game state.
32      """
33      def render(self, game_state: GameState) -> None:
34          """Renders a new UI depending on game state.
35
36          Args:
37              game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
38                  that can be X, O or spaces) and a starting Mark (default X).
39          """
40          clear_screen()
41          if game_state.winner:
42              print_blinking(game_state.grid.cells, game_state.winning_cells)
43              print(f"{game_state.winner} wins \N{party popper}")
44          else:
45              print_solid(game_state.grid.cells)
46              if game_state.tie:
47                  print("No one wins this time \N{neutral face}")
48
49      def placeholder2(self) -> None:
50          """This is a placeholder
51          """
```

16.1.1 `placeholder2()`

This is a placeholder

Source code in `src\frontend\console\renderers.py`

```
49  def placeholder2(self) -> None:
50      """This is a placeholder
51      """
```

16.1.2 `render(game_state)`

Renders a new UI depending on game state.

Parameters:

Name	Type	Description	Default
`game_state`	`GameState`	current GameState, consisting of a current Grid (9 elemets that that can be X, O or spaces) and a starting Mark (default X).	*required*

Source code in `src\frontend\console\renderers.py`

```
33  def render(self, game_state: GameState) -> None:
34      """Renders a new UI depending on game state.
35
36      Args:
37          game_state (GameState): current GameState, consisting of a current Grid (9 elemets that
38              that can be X, O or spaces) and a starting Mark (default X).
39      """
40      clear_screen()
41      if game_state.winner:
42          print_blinking(game_state.grid.cells, game_state.winning_cells)
43          print(f"{game_state.winner} wins \N{party popper}")
44      else:
45          print_solid(game_state.grid.cells)
46          if game_state.tie:
47              print("No one wins this time \N{neutral face}")
```

16.2 `blink(text)`

Modify information to be print to console and add slow blinking [5m slow blink and [0m reset.

Source code in `src\frontend\console\renderers.py`

```
57  def blink(text: str) -> str:
58      """Modify information to be print to console and add slow blinking [5m slow blink
59      and [0m reset.
60      """
61      return f"\033[5m{text}\033[0m"
```

16.3 `clear_screen()`

Clear console, like command reset on modern Linux systems.

Source code in `src\frontend\console\renderers.py`

```
52  def clear_screen() -> None:
53      """Clear console, like command reset on modern Linux systems.
54      """
55      print("\033c", end="")
```

16.4 `print_blinking(cells, positions)`

Add blinking ANSI code to positions in cells.

Parameters:

Name	Type	Description	Default
`cells`	`Iterable[str]`	Lits of all cells	*required*
`positions`	`Iterable[int]`	List of positions	*required*

Source code in `src\frontend\console\renderers.py`

```
63  def print_blinking(cells: Iterable[str], positions: Iterable[int]) -> None:
64      """Add blinking ANSI code to positions in cells.
65
66      Args:
67          cells (Iterable[str]): Lits of all cells
68          positions (Iterable[int]): List of positions
69      """
70      mutable_cells = list(cells)
71      for position in positions:
72          mutable_cells[position] = blink(mutable_cells[position])
73      print_solid(mutable_cells)
```

16.5 `print_solid(cells)`

Render game UI. It uses textwrap and format to replace on the correct position.

Parameters:

Name	Type	Description	Default
`cells`	`Iterable[str]`	*description*	*required*

Source code in `src\frontend\console\renderers.py`

```
75  def print_solid(cells: Iterable[str]) -> None:
76      """Render game UI. It uses textwrap and format to replace on the correct position.
77
78      Args:
79          cells (Iterable[str]): _description_
80      """
81
82      print(
83          textwrap.dedent(
84              """\
85                A   B   C
86               -----------
87          1 ¦  {0} | {1} | {2}
88            ¦ ---+---+---
89          2 ¦  {3} | {4} | {5}
90            ¦ ---+---+---
91          3 ¦  {6} | {7} | {8}
92      """
93          ).format(*cells)
94      )
```