Documentation - Royal Game of Ur

Written with reference to the format used in Python's documentation[1]

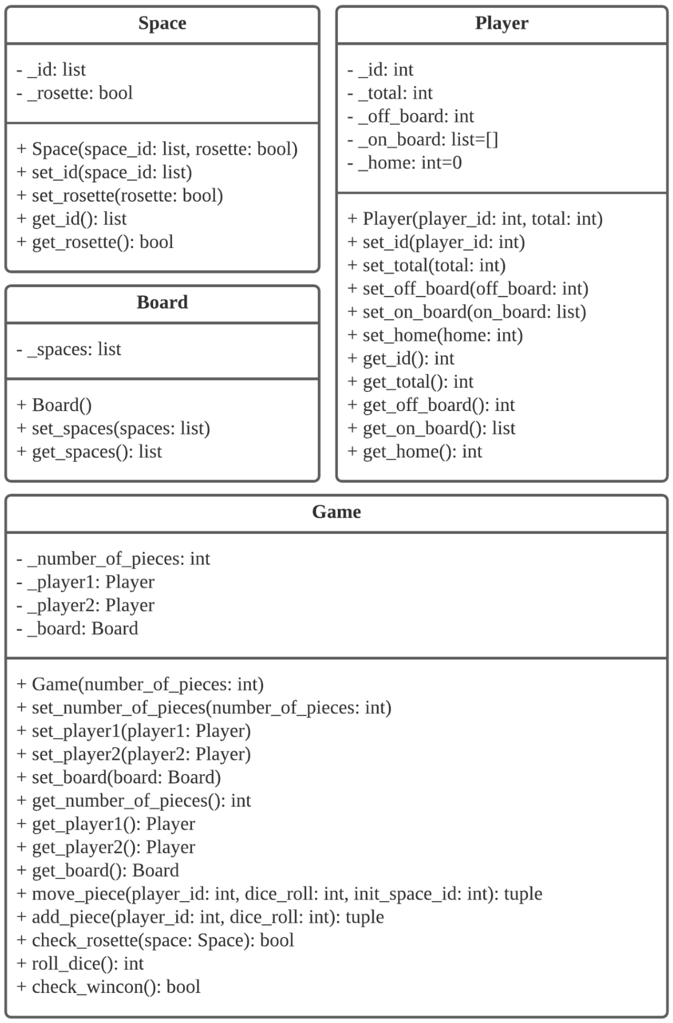
F07 Team 7G Members: Chew Ching Hian 1006083, Caroline Tiu 1006179, Leow Jing Ting 1006392, Cordelia Tan 1006138, Nicholas Ng Zhi Yong 1006021

This game uses the `random` library for the logic aspect of the game, the `socket` library to allow players on different devices to play with each other, and the `tkinter` library for the interface.

royal_classes.py

This file contains classes that are essentially the various components of the game. It is dependent on the `random` library.

Below is the UML class diagram for our classes.



[1] https://docs.python.org/3/library/stdtypes.html

Spaces

Spaces represent specific cells on the board.

class **`Space(`**`space_id, rosette`**`)`**

The first argument of the space constructor is a list, while the second argument is a boolean.

`space_id`

A list containing four items.

- Item 1 represents the identification number of the space with respect to the first player.
- Item 2 represents the number of pieces that the first player has on that space.
- Item 3 represents the identification number of the space with respect to the second player.
- Item 4 represents the number of pieces that the second player has on that space.

`rosette`

A boolean representing whether the space contains a rosette, since a player whose piece lands in a space with a rosette is allowed to have another turn.

`set_id(`*`space_id`***`)`**

This method is a setter method for the user to set the identification of the space without directly accessing the attribute as it is private.

Similar to `space_id` in the constructor, *`space_id`* here is also a list containing four items:

- Item 1 represents the identification number of the space with respect to the first player.
- Item 2 represents the number of pieces that the first player has on that space.
- Item 3 represents the identification number of the space with respect to the second player.
- Item 4 represents the number of pieces that the second player has on that space.

`set_rosette(`*`rosette`***`)`**

This method is a setter method for the user to set the space as a rosette space or a normal space without directly accessing the attribute as it is private.

`rosette` is a boolean.

`get_id()`

This returns a list that represents the identification of the space.

The list returned contains 4 items:

- Item 1 represents the identification number of the space with respect to the first player.
- Item 2 represents the number of pieces that the first player has on that space.
- Item 3 represents the identification number of the space with respect to the second player.
- Item 4 represents the number of pieces that the second player has on that space.

This method is a getter method for the user to access the identification of the space without directly accessing the attribute as it is private.

`get_rosette()`

This returns a boolean that represents whether the space has a rosette.

This method is a getter method for the user to access whether the space has a rosette without directly accessing the attribute as it is private.

Board

Board represents the entire game board. This class is dependent on the `Space` class.

class **`Board()`**

The constructor does not take in any arguments.

This class contains a private attribute that contains a list of 20 `Space` objects, with each objects' first and third item in the `space_id` argument being initialised as the identification number of the space with respect to each player, and the second and fourth item both being initialised as 0, since it represents both players having 0 pieces on every space when the `Board` object is created. The `rosette` argument is set according to the standard layout of the board.

`set_spaces(`*`spaces`*`)`

This method is a setter method for the user to set the list of spaces that the board has without directly accessing the attribute as it is private.

`spaces` is a list containing 20 `Space` objects.

`get_spaces()`

This returns a list containing 20 `Space` objects, each representing a space on the board.

This method is a getter method for the user to access the list of spaces that the board has without directly accessing the attribute as it is private.

Players

This represents the 2 players playing the game.

class **`Player(`**`player_id, total`**`)`**

Both arguments of the constructor are integers.

`player_id`

An integer of value 1 or 2 to represent the identification number of a player. This is because there are only 2 players.

`total`

An integer to represent the total number of pieces that the player has in the game.

The class has other attributes that are private.

The first attribute is an integer representing the number of playable pieces that the player has that are currently off the board. This is initialised as the same value as total since the player starts the game with no pieces on the board.

The second attribute is a list containing integers of the positions of the pieces with respect to the player. The largest possible length of the list during the game should be equivalent to the total number of pieces that the player has. It is initialised as an empty list since the player starts the game with no pieces on the board.

The third attribute is an integer representing the number of pieces that are home, or have successfully completed the entire course on the board. These pieces are no longer playable. It is initialised as 0 since the player starts with no pieces home.

At any point in time in the game, the sum of the number of pieces that the player has that are off the board, on the board, or home should be equivalent to their total number of pieces.

`set_id(`_`player_id`_**`)`**

This method is a setter method for the user to set the identification number of the player without directly accessing the attribute as it is private.

_`player_id`_ is an integer of value 1 or 2.

`set_total(`_`total`_**`)`**

This method is a setter method for the user to set the total pieces that the player has without directly accessing the attribute as it is private.

_`total`_ is a non-negative integer.

`set_off_board(`*`off_board`*`)`

This method is a setter method for the user to set the number of playable pieces that are off the board without directly accessing the attribute as it is private.

`off_board` is a non-negative integer.

`set_on_board(`*`on_board`*`)`

This method is a setter method for the user to set the positions of the player's pieces that are on the board without directly accessing the attribute as it is private.

`on_board` is a list of integers, with each integer representing the position of a piece that the player has on the board.

`set_home(`*`home`*`)`

This method is a setter method for the user to set the number of pieces that are home, which are pieces that have completed an entire course on the board without directly accessing the attribute as it is private.

`home` is a non-negative integer.

`get_id()`

This returns an integer that represents the identification number of the player, which should either be 1 or 2.

This method is a getter method for the user to access the identification number of the player without directly accessing the attribute as it is private.

`get_total()`

This returns an integer that represents the total number of pieces that the player has.

This method is a getter method for the user to access the total number of pieces that the player has without directly accessing the attribute as it is private.

`get_off_board()`

This returns an integer that represents the number of playable pieces that the player has that are not on the board.

This method is a getter method for the user to access the number of pieces that the player has that are not on the board without directly accessing the attribute as it is private.

`get_on_board()`

This returns a list that represents the various positions of pieces that the player has on the board.

This method is a getter method for the user to access the list of positions of pieces that the player has on the board without directly accessing the attribute as it is private.

`get_home()`

This returns an integer that represents the number of pieces that are home, which means those pieces have completed one course on the board.

This method is a getter method for the user to access the number of pieces that are home without directly accessing the attribute as it is private.

Game

This represents the entire game including the game logic. This class is dependent on the `Player` class and the `Board` class.

class **Game**(`number_of_pieces`)

The argument of the constructor is an integer.

`number_of_pieces`

An integer to represent the total number of pieces that each player has.

This class has two attributes, each containing a `Player` object. Both are initialised using the variable `number_of_pieces`.

There is also an attribute containing a `Board` object.

`set_number_of_pieces(`*`number_of_pieces`***`)`**

This method is a setter method for the user to set the number of pieces that each player has without directly accessing the attribute as it is private.

`number_of_pieces` is an integer.

`set_player1(`*`player1`***`)`**

This method is a setter method for the user to set the first player's `Player` object without directly accessing the attribute as it is private.

`player1` is an instance of the `Player` class, which should have its `player_id` initialised as 1.

`set_player2(`*`player2`*`)`

This method is a setter method for the user to set the second player's `Player` object without directly accessing the attribute as it is private.

`player2` is an instance of the `Player` class, which should have its `player_id` initialised as 2.

`set_board(`*`board`*`)`

This method is a setter method for the user to set the `Board` object without directly accessing the attribute as it is private.

`board` is an instance of the `Board` class.

`get_number_of_pieces()`

This returns an integer representing the total number of pieces that each player has.

This method is a getter method for the user to access the number of pieces without directly accessing the attribute as it is private.

`get_player1()`

This returns a `Player` object, representing the first player.

This method is a getter method for the user to access the first player's `Player` object without directly accessing the attribute as it is private.

`get_player2()`

This returns a `Player` object, representing the second player.

This method is a getter method for the user to access the second player's `Player` object without directly accessing the attribute as it is private.

`get_board()`

This returns a `Board` object, representing the board used for the game.

This method is a getter method for the user to access the `Board` object without directly accessing the attribute as it is private.

`move_piece(``*player_id, dice_roll, init_space_id*`**`)`**

This method returns a tuple with 2 elements in it, with the first item being a boolean and the second item being either a `NoneType` object or a `Space` object. The first item is used to represent whether the function was successful and the second item is the space that a player has moved a piece to, if any.

This method takes in 3 arguments. It is used to move a piece that is currently on the board to another space.

`player_id` is an integer of value 1 or 2, based on the number of players in the game, to represent the identification number of a player. This is because there are only 2 players.

`dice_roll` is an integer of value 0 to 4, based on the possible dice roll values, to represent the number of spaces a player can move during a turn.

`init_space_id` is an integer of value 0 to 13, based on the number of spaces that the player can access, to represent the identification number of the space with respect to each player.

`add_piece(``*player_id, dice_roll*`**`)`**

This method returns a tuple with 2 elements in it, with the first item being a boolean and the second item being either a `NoneType` object or a `Space` object. The first item is used to represent whether the function was successful and the second item is the space that a player has moved a piece to, if any.

This method takes in 2 arguments. It is used to add a piece that is currently off the board onto it.

`player_id` is an integer of value 1 or 2, based on the number of players in the game, to represent the identification number of a player. This is because there are only 2 players.

`dice_roll` is an integer of value 0 to 4, based on the possible dice rolls, to represent the number of spaces a player can move during a turn.

`check_rosette(``*space*`**`)`**

This method returns a boolean that represents whether a space has a rosette or not.

It is used to check whether a player's piece has landed on a space with a rosette. If there is a rosette, the player is allowed to have another turn.

`space` is a `Space` object.

`roll_dice()`

This method returns an integer, to represent the dice roll generated for a player's turn.

In the physical game, there are 4 tetrahedron dice, each with a 50-50 chance of rolling a dotted corner, which represents the value 1, or rolling an empty corner, which represents the value 0.

As such, the minimum value that this function generates is 0, and the maximum value that it generates is 4.

`check_wincon()`

This method returns a boolean which is used to check whether either player has fulfilled the winning condition, meaning that all the player's pieces are home.

menu.py

This file contains the functions to carry out the menu display for our text interface.

`validate_main_opt(`*`opt, moves`***`)`**

This function returns a boolean which is used to check whether a player has fulfilled the conditions for entering an option.

This function takes in 2 arguments. It is used to validate the player's input for the desired action during the turn, that is, moving a piece, adding a piece, passing a turn, or quitting the game.

`opt` is a string which represents the option chosen by the player.

`moves` is an integer which represents the length of the list of options that were given to the player, since the list is dynamic.

`move_input(`*`pieces`***`)`**

This function returns an integer which represents the position of the piece the player wants to move, accessed by indexing the list *`pieces`*.

This function takes in 1 argument. It is used to validate the player's input for the desired piece that is on the board to move during the turn.

`pieces` is a list of integers, with each integer representing the position of a piece that the player has on the board.

`print_menu(`*`off, num`***`)`**

This function returns a list which represents the possible actions a player can select from for their turn.

This function takes in 2 arguments. It is used to display the menu to show the player's options for a turn.

`off` is an integer which represents the number of pieces that are off the board with respect to the player who is currently carrying out their turn.

`num` is an integer which represents the number of pieces that are on the board with respect to the player who is currently carrying out their turn.

menu(*off, pieces, dice_roll***)**

This function returns a string which will be used as the protocol message for the socket later on.

This function takes in 3 arguments. It is used to display the menu and prompt the player for an input until the player is successful in giving a valid input. It is dependent on the above 3 functions, with `print_menu` displaying the options, `validate_main_opt` validating the player's input for their move, and `move_input` validating the player's input for their choice of piece to move.

off is an integer which represents the number of pieces that are off the board with respect to a player.

pieces is a list of integers, with each integer representing the position of a piece that the player has on the board.

dice_roll is an integer of value 0 to 4, based on the possible dice rolls, to represent the number of spaces a player can move during a turn.

socket_func.py

This file is dependent on `menu.py` and contains the necessary functions which will be used for the socket later on.

`create_message(``running, passed, game, message, player_id`**`)`**

This function takes in 5 arguments and returns a tuple with 4 elements in it. The first element of the tuple determines whether the game is still running after the player's input within this function. The second element represents whether the player has chosen to pass their turn. The third element is the protocol messages to be sent between the server and client, with each instruction separated by a semicolon. The fourth element represents the space that the player's piece ends on, if any.

This function is used to create the messages that will be sent between the server and client based on the action that the server or client has chosen to take. Since the player is prompted for their action, this function is dependent on `menu` from `menu.py`.

`running` is a boolean which represents whether the game is still running.

`passed` is a boolean which represents whether the player has chosen to pass a turn.

`game` is a `Game` object from `royal_classes.py`.

`message` is a string which represents the protocol message that will be used for the socket.

`player_id` is an integer of value 1 or 2, based on the number of players in the game, to represent the identification number of a player. This is because there are only 2 players.

`send_message(``running, game, csock, player_id`**`)`**

This function is a function that takes in 4 arguments and returns a boolean that represents whether the game is still running after the player's input.

This function is dependent on `create_message` and it encodes and sends the message created by that function to the server or client. A newline character is appended to the end of the message to signify to the receiver that it can stop listening for more bytes.

`running` is a boolean which represents whether the game is still running.

`game` is a `Game` object from `royal_classes.py`.

`sock` is a `socket` object.

`player_id` is an integer of value 1 or 2, based on the number of players in the game, to represent the identification number of a player. This is because there are only 2 players.

read_message(`running, game, message, opp_player_id`**)**

This function takes in 4 arguments and returns a boolean that represents whether the game is still running after receiving the client's or server's message.

This function parses the information that has been sent in the message to carry out the actions for the game. It also prints the statistics of both players after parsing the opponent's moves.

`running` is a boolean which represents whether the game is still running.

`game` is a `Game` object from `royal_classes.py`.

`message` is a string which represents the protocol message that was received from the socket.

`opp_player_id` is an integer of value 1 or 2, based on the number of players in the game, to represent the identification number of a player. This is because there are only 2 players.

server.py

This file uses the `socket` library and is dependent on `royal_classes.py` and `socket_func.py`. It is the server side of the socket.

First, the server binds to the socket based on a tuple containing the Internet protocol (IP) address of the device and the chosen port number. It then listens to the socket to see if there is any client connecting to it. Once it has detected a client, it will accept the client.

At this point, the server creates an instance of the Game object with 7 pieces and rolls the dice to see if the server or client goes first. The client will send over its dice roll and the server will process whether it is less than or greater than its own dice roll. If the dice rolls have the same values, the server will continue asking the client to send its new dice roll until they get different values. If the client's dice roll is larger than the server's, the client will go first and thus the client will have the player identification number of 1 and the server will have the player identification number of 2, and vice versa if the server has a larger dice roll than the client.

Afterwards, the server and the client will communicate with each other such that they will continuously send over the instructions for the game with each instruction separated by a semicolon. The end of the message is signified by the newline character. The receiver of the message will then process the message that was sent and carry out the instructions according to the various protocols. Once the game ends i.e. a player wins or one of the players quits the game, the socket is closed.

The following is a list of all the protocols that are used:

- **ROLL**: The client uses this protocol when it is rolling the dice to determine which player goes first.
- **REROLL**: The server uses this protocol when it requires the client to roll the dice again, which occurs when the server is unable to determine which player goes first.
- **START**: The server uses this protocol to inform the client that the client has the first turn in the game.
- **MOVE**: Both the server and client use this protocol to inform the receiver of the piece that the player is moving as well as how much they are moving the piece by.
- **ADD**: Both the server and client use this protocol to inform the receiver of where the player is adding a new piece to.
- **PASS**: Both the server and client use this protocol to inform the receiver that the player is passing their turn.
- **QUIT**: Both the server and client use this protocol to inform the receiver that the player is quitting the game. The socket will close upon receiving this protocol message.

client.py

This file uses the `socket` library and is dependent on `royal_classes.py` and `socket_func.py`. It is the client side of the socket.

The client connects to the socket based on a tuple containing the IP address of the device and the chosen port number. If the connection is successful, the server creates an instance of the Game object with 7 pieces and rolls the dice to see if the server or client goes first. The client will send over its dice roll and wait for the server to process whether the client or server makes the first move.

The client and server will then communicate with each other with the protocols listed above. This file is similar to `server.py` in the processing of instructions.

gui.py

This file is a static image of the board to help the player visualize the layout of the board and where the position of the game pieces are located.

class **`royal_classes_GUI()`**

The constructor does not take in any arguments.

The class contains 3 private attributes.

The first attribute contains a `Tk` object from the `tkinter` library.

The second attribute contains a `Canvas` object from the `tkinter` library that is 800 by 300 pixels with a black background.

`grid()`

This function displays the board with squares and rosettes at specific locations on the board.

`number_player1()`

This function displays the numbering of spaces for the first player from 0 to 13 on the player's play area. The text is displayed in blue.

`number_player2()`

This function displays the numbering of spaces for the second player from 0 to 13 on the player's play area. The text is displayed in red.

`menu_button()`

This function creates a button at the bottom right of the screen for the user to close the window.