**INTRO**

This part of the document is a documentation of how each line of code of ***Snake and Ladders NEA project*** works written by the *supervisor*.

In this, the various classes, class methods, subroutines and other code will be discussed.

Each line that contains code will be explained.

Its purpose will be outlined and each line will be checked as to what function it serves and where it is used in the ***Snake and Ladders NEA project****.*

Comments are included as lines, but comments will not be explained.

Version 1

This version includes the Player Class as an object and also a few subroutines which allow you to play the game on the default python command line interface (IDLE)

NOTE: This is an Early version of the program and visual features will be added later on in the development.

***Classes:***

* *Player*
* This is currently the only class in the game and is instantiated twice - one for each player
* The constructor takes 3 parameters and assigns them as attributes within the player class
* There is then a moveForward method which allows the player to move forward within the game and takes one parameter which is the number that the player rolled.
* In this there is an IF, ELSE statement which checks if the *currentSpace* attribute added to the *roll* variable is less than or equal to 100
* If it is, increments the *currentSpace* attribute by *roll*
* Does a print statement to let the user know which space they are on now
* Takes an empty input
* Otherwise, does a print statement to let the user know that they cannot move this amount of spaces and which space they are still on. Also takes an empty input.

***Subroutines:***

* *winDisplay*
* This is the first defined procedure (it doesn’t return anything) in the program
* It doesn't take any parameters and is called when the player wins the game
* Prints a statement to notify the user that they have won the game
* Takes an empty input
* Then exits the program
* *displayMenu*
* This is the second defined procedure (it doesn’t return anything) in the program
* It doesn't take any parameters and is called upon starting the program.
* Prints the menu options each on a separate line
* *displayDifficultyMenu*
* This is the third defined procedure (it doesn’t return anything) in the program
* It doesn't take any parameters and is called after the user chooses ‘New game’
* Prints the difficulty options each on a separate line
* *printInstructions*
* This is the fourth defined procedure (it doesn’t return anything) in the program
* It doesn't take any parameters and is called after the user chooses ‘How to play’
* Prints the instructions and then takes an empty input.
* Also calls displayMenu at the end after instructions have been printed.
* *checkChoice*
* This is the fifth defined procedure (it doesn’t return anything) in the program
* It takes one parameter which is the choice that the user entered and is called after ‘displayMenu’
* Initialises a WHILE loop which takes the input of ‘choice’ if it isn’t 1, 2 or 3
* ELSE:
* If the choice is 1 then creates the string “Display difficulty menu”
* If the choice is 2, calls ‘printInstructions’ then stores as empty input as ‘choice’ then calls itself
* Otherwise, exits the program
* *checkDifficultyChoice*
* This is the first defined function (it returns something) in the program
* It takes one parameter which is the choice that the user entered and is called after ‘displayDifficultyMenu’
* Initialises a WHILE loop which takes the input of *choice* if it isn’t 1, 2, 3 or 4
* ELSE:
* If the *choice* is 1, prints “You have selected easy mode!”
* If the *choice* is 2, prints “You have selected normal mode!"
* If the *choice* is 3, prints “You have selected hard mode!"
* Otherwise, prints “You have selected a custom difficulty mode!"
* Returns *choice* to the Main part of the program
* *checkCustomDifficultyChoice*
* This is the second defined function (it returns something) in the program
* It takes four parameters which are the choices for the different special spaces that the user entered and is called after the user chooses ‘Custom’ difficulty
* Initialises a WHILE loop
* Uses exception handling to cast the parameter variables into integers from strings
* If this doesn’t work, takes these inputs again
* Returns these to the main part of the program
* *checkGameChoice*
* This is the sixth defined procedure (it doesn’t return anything) in the program
* It takes one parameter and is used to check if the choice given in game is valid
* Initialises a WHILE loop which takes the input of *gameChoice* if it isn’t 1 or 2
* If the WHILE loop isn’t run, then uses an IF statement to check if *gameChoice* is 2
* If the above, IF statement is met, takes an input with a suitable message for exiting the program
* Then, initialises a WHILE loop which takes the input of *answer* if it isn’t ‘Yes’ or ‘No’
* Uses an IF statement to check if answer is ‘Yes’ or ‘yes’
* If the above IF statement is met, exits the program
* If it isn’t met, calls *displayGameMenu* function again and also takes the input of gameChoice and calls *checkGameChoice* again with an argument of *gameChoice*
* *ladderCheck*
* This is the seventh defined procedure (it doesn’t return anything) in the program
* It takes one parameter which is the current turn of the player and is run each turn to check if the player's position is equal to a ladder's position
* Checks if it is **player1’s** turn and if it is runs a linear search matching up to **player1’s** current space
* If the ladder space matches up with **player1’s** current space, then increments **player1’s** current space by 10 and prints a suitable print message using player1’s current space
* If it is **player2’s** turn, runs a linear search matching up to **player2’s** current space
* If the ladder space matches up with **player2’s** current space, then increments **player2’s** current space by 10 and prints a suitable print message using **player2’s** current space
* *ladderSort*
* This is the third defined function (it returns something) in the program
* It takes one parameter which is the ladders array and is used to sort the ladder spaces list into ascending order
* Uses a FOR loop version of bubble sort to sort this array
* Returns the sorted *ladders* array to the main program
* *snakeCheck*
* This is the eighth defined procedure (it doesn’t return anything) in the program
* It takes one parameter which is the current turn of the player and is run each turn to check if the player's position is equal to a snake’s position
* It is the same as the *ladderCheck* in that it does a linear search between the current space of each player and the array
* However, it subtracts 10 from the player’s current position if it matches as it is a snake
* *snakeSort*
* This is the fourth defined function (it returns something) in the program
* It takes one parameter which is the *snakes* array and is used to sort the snake spaces list into ascending order
* Uses a FOR loop version of bubble sort to sort this array
* Returns the sorted *snakes* array to the main program
* *bonusRollSpaceCheck*
* This is the ninth defined procedure (it doesn’t return anything) in the program
* It takes one parameter which is the current turn of the player and is run each turn to check if the player's position is equal to a bonus roll space’s position
* It is the same as the *ladderCheck* in that it does a linear search between the current space of each player and the array
* However, it calls the *bonusRoll* procedure as it is a bonus roll space
* *bonusRoll*
* This is the tenth defined procedure (it doesn’t return anything) in the program
* It takes two parameters which are the roll number and the current turn of the player and is run each turn but only outputs something if the number rolled is equal to 6.
* Initialises a WHILE loop that runs while the roll is equal to 6
* Prints a suitable message for rolling the dice again and then does the dice roll via a random number generation
* Prints a suitable message for the number rolled by the player
* Uses an IF statement to check if *currentTurn* is equal to “Player 1”
* If it is, calls the *moveForward* method of **player1** then calls the *ladderCheck*, *snakeCheck*, *bonusRollSpaceCheck* and *lionCheck* functions
* Uses an IF statement to check if *currentTurn* is equal to “Player 2”
* If it is, calls the *moveForward* method of **player2** then calls the *ladderCheck*, *snakeCheck*, *bonusRollSpaceCheck* and *lionCheck* functions
* *bonusRollSpaceSort*
* This is the fifth defined function (it returns something) in the program
* It takes one parameter which is the *bonusRollSpaces* array and is used to sort the bonus roll spaces list into ascending order
* Uses a FOR loop version of bubble sort to sort this array
* Returns the sorted *bonusRollSpaces* array to the main program
* *lionSort*
* This is the sixth defined function (it returns something) in the program
* It takes one parameter which is the *lions* array and is used to sort the lion spaces list into ascending order
* Uses a FOR loop version of bubble sort to sort this array
* Returns the sorted *lions* array to the main program
* *lionCheck*
* This is the tenth defined procedure (it doesn’t return anything) in the program
* It takes one parameter which is the current turn of the player and is run each turn to check if the player's position is equal to a bonus roll space’s position
* It is the same as the *ladderCheck* in that it does a linear search between the current space of each player and the array
* However, it stuns the player (they miss a turn) as it is a lion space
* *probe*
* This is the seventh defined function (it returns something) in the program
* It takes 5 parameters which are the space in the array, the available spaces array, the difference between the space and the previous space, the type of space e.g. “ladder”, “snake” and also the count which determines if its nth position in the array is even or odd. It is used to try to space out the spaces between the ladder and snake, lion and bonus roll spaces.
* Firstly *count* is incremented by 1 and a variable is defined that is *count* MOD 2
* This new variable called *thisCount* determines if the space’s position in the array is even or odd
* A WHILE loop is initialised that runs while *difference* (parameter variable) is less than 5
* Within this WHILE loop, there is an IF, ELIF statement
* This checks if *thisCount* is an even number or an odd number
* If it is even, then it increments *space* by 1 and if it is odd, decrements *space* by 1
* The next few lines use a WHILE loop to keep regenerating spaces if they are not included in the *availableSpaces* array
* Returns the *space*, *count* and *availableSpaces* variables to *checkArrays*
* *checkArrays*
* This is the eighth defined function (it returns something) in the program
* It takes 3 parameters which are the array being checked, the *availableSpaces* array and also what type of spaces the array being checked contains e.g. ladders, snakes, lions. It is used to check to see if the ladder spaces, snake spaces, bonus roll spaces and lion spaces lists need probing and sees whether or not they can be probed.
* Sets count to 0 to begin with
* Then uses a FOR loop to iterate through the *array* (parameter variable)
* Uses TRY and EXCEPT exception handling to catch any errors when probing the *array*
* In the try statement, works out the difference, gets the space, calls the probe method and then sets the new *space* to the same position in the *array*
* Returns the amended *array* and *availableSpaces* array to the main part of the program
* *startGame*
* This is the eleventh and final defined procedure (it doesn’t return anything) in the program
* It doesn’t take any parameters and is initially used to decide the player order and is then used to display the player details after every round
* Firstly, the *turnDecision* was stored as a random integer between 1 and 2
* Then an IF, ELSE statement was used to decide the player order depending on *turnDecision*
* Initialises a WHILE loop that runs infinitely
* Sets *currentTurn* to be the first item in the array *turnOrder*
* Uses an IF, ELIF statement to check if either player is stunned
* If they are stunned, prints a suitable print statement, takes an empty input and progresses to the next player’s turn
* Prints the details of the round including each player’s current position on the board
* Calls the functions for the menu and the roll of the player
* Uses an IF statement to check if it is **player1’s** turn and if it is moves them forward before calling the space checks to check if they landed on a special square and move them accordingly
* Does the same for player 2 and finally switches the player turn

***Other code:***

This section includes code that is outside a subroutine or a class. This is labelled as the ‘Main’ section and includes function calls as well as class instantiation, selection, iteration and sequence.

* Firstly, the *Player* class is instantiated twice as **player1** and **player2**
* A few arrays are set up for the different spaces on the board
* Then *displayMenu*, *checkChoice* and *displayDifficultyMenu* procedures are called
* Next, the *checkDifficultyChoice* function is called as stored as *difficultyChoice*
* Then, a series of IF statements are carried out using the *difficultyChoice* variable
* These set up the number of special spaces on the board
* Then an ELSE statement is carried out following this which allows the user to input their own values for the number of special spaces on the board given that they choose “Custom” as their difficulty choice
* Soon after, a number of count controlled iteration loops (FOR loops) are used to generate the special spaces using the numbers from before
* Next, the *checkArrays* function is called for each type of special space and the output stored
* Then the first item of the output arrays for the special spaces are stored in their respective variables
* Next, the sort functions for each of these arrays are run and stored as their respective variables
* Finally, the *startGame* procedure is run which continues running the game until a player wins

Version 2

This version includes the multiple object including objects defined within objects without constructor methods and also a few class methods which allow you to play the game on the default python command line interface (IDLE)

NOTE: This is an Early version of the program and visual features may or may not be added later on in the development because of time constraints

***Player class***

* This is mostly unchanged since version 1 but I will still detail it
* The constructor now takes 5 parameters and assigns them as attributes within the player class
* New attributes include *movesMade* and *inventory* which are used in conjunction with the new special dice feature
* *moveForward method*
  + There is then a moveForward method which allows the player to move forward within the game and takes one parameter which is the number that the player rolled.
  + In this there is an IF, ELSE statement which checks if the *currentSpace* attribute added to the *roll* variable is less than or equal to *boardSize* (global variable)
  + If it is, increments the *currentSpace* attribute by *roll*
  + Does a print statement to let the user know which space they are on now
  + Also increments the *movesMade* attribute by 1
  + Takes an empty input
  + Also uses a nested IF statement to check if the *currentSpace* attribute is equal to *boardSize*
  + If it is, stores a new variable called *winner* as the *name* attribute
  + Then passes this attribute to the *winDisplay* method of the **Game** class
  + Otherwise, does a print statement to let the user know that they cannot move this amount of spaces and which space they are still on. Also takes an empty input.

***Menu class***

* This is the object for the **Menu** class and includes the **GameSetup** class within it
* It doesn’t have a constructor so therefore takes no parameters
* *displayMenu method*
  + This is the first PUBLIC method of the **Menu** class and takes no parameters
  + Prints the menu options each on a separate line
* *displaySizeMenu method*
  + This is the second PUBLIC method of the **Menu** class and takes no parameters
  + Prints a print statement suitable for choosing the size of the board
* *displayDifficultyMenu method*
  + This is the third PUBLIC method of the **Menu** class and takes no parameters
  + Prints the difficulty options each on a separate line
* *printInstructions method*
  + This is the fourth PUBLIC method of the **Menu** class and takes no parameters
  + Prints the instructions and then takes an empty input.
  + Also calls the *displayMenu* method of **Menu** at the end after instructions have been printed.
* *checkChoice method*
  + This is the fifth PUBLIC method of the **Menu** class and takes 1 parameter
  + Initialises a WHILE loop which takes the input of *choice* if it isn’t 1, 2 or 3
  + ELSE:
  + If the *choice* is 1 then prints “You have selected to start a new game!”
  + If the *choice* is 2, calls the *printInstructions* method of **Menu** then stores an empty input as *choice* then calls itself
  + Otherwise, exits the program
* *checkBoardSize method*
  + This is the sixth PUBLIC method of the **Menu** class and takes 1 parameter
  + Initialises a WHILE loop
  + Uses exception handling to cast the parameter variable into an integer from a string
  + Also uses an IF statement to check if each value isn’t between 100 and 200
  + If this is met, prints a suitable statement then takes this input again
  + If it isn’t met, calls the *roundBoardSize* method of **GameSetup**, prints a suitable statement then RETURNS the board size value and breaks the validation WHILE loop
  + If there is an exception, prints a suitable statement and takes this input again
* *checkDifficultyChoice method*
  + This is the seventh PUBLIC method of the **Menu** class and takes 1 parameter
  + Initialises a WHILE loop which takes the input of *choice* if it isn’t 1, 2, 3 or 4
  + ELSE:
  + If the *choice* is 1, prints “You have selected easy mode!”
  + If the *choice* is 2, prints “You have selected normal mode!"
  + If the *choice* is 3, prints “You have selected hard mode!"
  + Otherwise, prints “You have selected a custom difficulty mode!"
  + Returns *choice* to the Main part of the program
* *checkCustomDifficultyChoice method*
  + This is the eighth PUBLIC method of the **Menu** class and takes 4 parameters
  + Initialises a WHILE loop
  + Uses exception handling to cast the parameter variables into integers from strings
  + Also uses an IF statement to check if each value isn’t between 0 and 10
  + If this is met, prints a suitable statement then takes these inputs again
  + If it isn’t met, breaks the validation WHILE loop
  + If there is an exception, takes these inputs again
  + Returns these to the main part of the program

***GameSetup class***

* This is the object for the **GameSetup** class within the **Menu** class
* It doesn’t have a constructor so therefore takes no parameters
* *roundBoardSize method*
  + This is the first PUBLIC method of the **GameSetup** class and takes 1 parameter - the board size that the player chose after choosing “Custom” difficulty
  + First, it divides the *boardSize* by 10, then rounds it, then multiplies it by 10
  + It returns *boardSize* to the method that called it
* *ladderSort method*
  + This is the second PUBLIC method of the **GameSetup** class and takes 1 parameter - the ladders array
  + Uses a FOR loop version of bubble sort to sort this array
  + Returns the sorted *ladders* array to the method that called it
* *snakeSort method*
  + This is the third PUBLIC method of the **GameSetup** class and takes 1 parameter - the snakes array
  + Uses a FOR loop version of bubble sort to sort this array
  + Returns the sorted *snakes* array to the main program
* *bonusRollSpaceSort method*
  + This is the fourth PUBLIC method of the **GameSetup** class and takes 1 parameter - the bonusRollSpaces array
  + Uses a FOR loop version of bubble sort to sort this array
  + Returns the sorted *bonusRollSpaces* array to the main program
* *lionSort method*
  + This is the fifth PUBLIC method of the **GameSetup** class and takes 1 parameter - the lions array
  + Uses a FOR loop version of bubble sort to sort this array
  + Returns the sorted *lions* array to the main program
* *probe method*
  + This is the sixth PUBLIC method of the **GameSetup** class and takes 5 parameters which are the space in the array, the available spaces array, the difference between the space and the previous space, the type of space e.g. “ladder”, “snake” and also the count which determines if its nth position in the array is even or odd
  + Firstly *count* is incremented by 1 and a variable is defined that is *count* MOD 2
  + This new variable called *thisCount* determines if the space’s position in the array is even or odd
  + A WHILE loop is initialised that runs while *difference* (parameter variable) is less than 5
  + Within this WHILE loop, there is an IF, ELIF statement
  + This checks if *thisCount* is an even number or an odd number
  + If it is even, then it increments space by 1 and if it is odd, decrements space by 1
  + The next few lines use a WHILE loop to keep regenerating spaces if they are not included in the *availableSpaces* array
  + Returns the *space*, *count* and *availableSpaces* variables to *checkArrays*
* *checkArrays method*
  + This is the seventh PUBLIC method of the **GameSetup** class and takes 3 parameters which are the array being checked, the *availableSpaces* array and also what type of spaces the array being checked contains e.g. ladders, snakes, lions
  + Sets count to 0 to begin with
  + Then uses a FOR loop to iterate through the *array* (parameter variable)
  + Uses TRY and EXCEPT exception handling to catch any errors when probing the *array*
  + In the try statement, works out the difference, gets the space, calls the probe method and then sets the new *space* to the same position in the *array*
  + Returns the amended *array* and *availableSpaces* array to the main part of the program
* *generateBoard method*
  + This is the eighth PUBLIC method of the **GameSetup** class and takes no parameters
  + Uses a FOR loop between 1 and *boardSize* + 1 to generate the board
  + Appends each space to the *availableSpaces* array
* *generateSpecialSpaces method*
  + This is the ninth PUBLIC method of the **GameSetup** class and takes no parameters
  + Uses 4 FOR loops to generate a random number for each special spaces and then append it to its specific array - it also removes it from the *availableSpaces* array
  + Inside these 4 FOR loops it uses exception handling in case a syntax error occurs when generating a random number for these special spaces
  + For the snakes array, it generates between 1 & boardSize - 11, for snakes between 11 & boardSize -1 and for bonusRollSpaces and lions between 1 and boardSize - 1

***Game class***

* This is the object for the **Game** class and includes the **GameUpdate** class within it
* It doesn’t have a constructor so therefore takes no parameters
* *displayGameMenu method*
  + This is the first PUBLIC method of the **Game** class and takes no parameters
  + It is run every turn and prints out the 3 options while in a game
* *winDisplay method*
  + This is the second PUBLIC method of the **Game** class and takes 1 parameter - the player who has the wins the game
  + It is run if a player has won the game and prints out the winner of the game, the amount of moves each player has made and then takes an empty input
  + After this it exits the game
* *startGame method*
  + This is the third PUBLIC method of the **Game** class and takes no parameters
  + This is the same as the *startGame* procedure in version 1 except now it accounts for the additional feature of a special dice and so after every turn the method of the **GameUpdate** class *specialDiceGivingCheck* is called with the parameter of *turnCount*
  + Other than this there are no considerable differences

***GameUpdate class***

* This is the object for the **GameUpdate** class within the **Game** class
* It doesn’t have a constructor so therefore takes no parameters
* *roll method*
  + This is the first PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *currentTurn* of the player
  + It generates a random number for the roll and prints a suitable statement for the roll
  + Next, it calls the *spaceMoveAndCheck* method of the **GameUpdate** class
* *specialDiceRoll method*
  + This is the second PUBLIC method of the **GameUpdate** class and takes 2 parameters - the *currentTurn* of the player and the choice that the player chose in the game
  + It is run if a player wants to use a special dice and uses a series of IF statements to determine which dice to roll
  + Firstly, it checks the player turn using an IF, ELSE statement
  + Then checks which inventory dice (max size of inventory = 2) is chosen
  + For each dice (each IF statement) carries out the same lines of code as the *roll*  method except modified fo the chosen special dice
  + The ELSE statement does the same thing for Player 2
* *checkGameChoice method*
  + This is the third PUBLIC method of the **GameUpdate** class and takes 2 parameters - the *currentTurn* of the player and the choice that the player chose in the game
  + This is the same as the *checkGameChoice* procedure in version 1 except now it accounts for the additional feature of a special dice
* *ladderCheck method*
  + This is the fourth PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *currentTurn* of the player
  + This is exactly the same as in version 1 so will not be detailed further
* *snakeCheck method*
  + This is the fifth PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *currentTurn* of the player
  + This is exactly the same as in version 1 so will not be detailed further
* *bonusRollSpaceCheck method*
  + This is the sixth PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *currentTurn* of the player
  + This is exactly the same as in version 1 so will not be detailed further
* *bonusRoll method*
  + This is the seventh PUBLIC method of the **GameUpdate** class and takes 2 parameters - the *currentTurn* of the player and the roll that the player got in the game
  + This is the same as the *bonusRoll* procedure in version 1 except now it accounts for the additional feature of a special dice
* *lionCheck method*
  + This is the eighth PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *currentTurn* of the player
  + This is exactly the same as in version 1 so will not be detailed further
* *spaceMoveAndCheck method*
  + This is the ninth PUBLIC method of the **GameUpdate** class and takes 2 parameters - the *currentTurn* of the player and the roll that the player got in the game
  + This uses an IF, ELIF statement to check the *currentTurn* of the player.
  + In the **player1** IF statement, keeps track of the player’s original position, calls the *moveForward* method of the **player** class, calls the necessary methods to check if a player has landed on any special spaces etc
  + In the **player2** ELIF statement, does the same for **player2**
* *specialDiceGivingCheck method*
  + This is the tenth PUBLIC method of the **GameUpdate** class and takes 1 parameter - the *turnCount* of the game
  + This checks if *turnCount* is a multiple of 5 and if it is, gives a randomly generated special dice to one of the players or both depending on their positions on the board
  + Also calls the *inventoryCheck* method
* *inventoryCheck method*
  + This is the eleventh PUBLIC method of the **GameUpdate** class and takes no parameters
  + This checks if the player inventory is full and if it is prompts them to replace one of their special dice to make room fo the new one

***Other code:***

This section includes code that is outside a subroutine or a class. This is labelled as the ‘Main’ section and includes function calls as well as class instantiation, selection, iteration and sequence.

This is mostly the same as in version 1 so will not be detailed further