**Computing BSc ALL project 1**

**Semester 1**

**Associated modules 120CT & 121COM**

Group Report – Team TWO

**Introduction:**

This report will outline the stages taken to complete the objective of the Activity Led Learning (A.L.L.) project, which was to build a basic game of Nougats and Crosses (or Tic-Tac-Toe) programmed in Python and to be performed on a Raspberri Pi, functioning as the server, to enable multiple users to play against one another from different PC end-systems. As this was a group project, the report will detail from each group member the contributions made, difficulties encountered and the solutions made during the course of this project.

**Project Management:**

To start we made a Product Catalogue, of all tasks that we needed to complete. This included; designing a game board, Ai element, win conditions, menu, player selection, server/ client, etc. This allowed us to have a clear knowledge of what we needed to complete, and could be used as a progress reference.

We used an AGILE project Plan to allocate time to each task, which was subject to change as we discovered what tasks would take a longer or shorter time than expected. We also used the course plan to see what weeks, certain programming techniques were being taught, so that we could use what we had learnt and implement it into the game. The project plan also had the project brief included for easy reference.

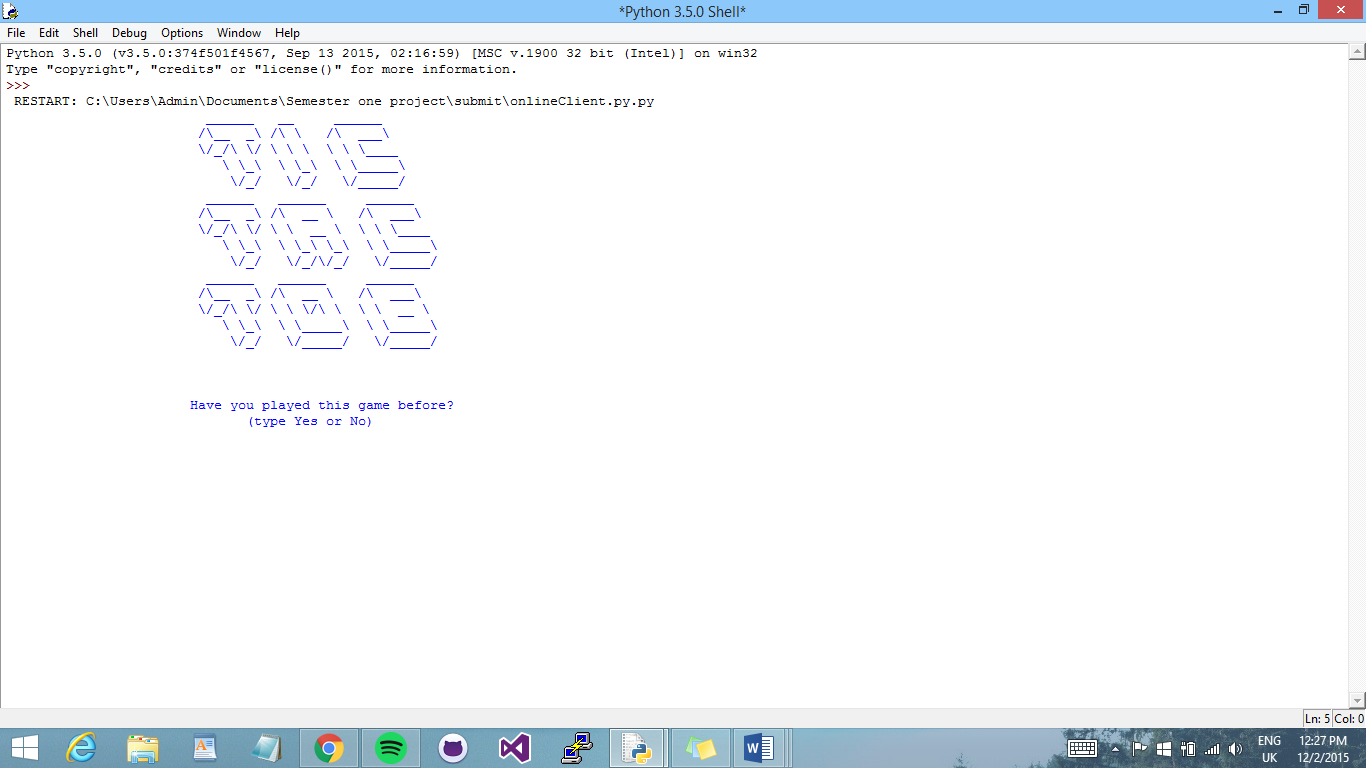
To keep track of tasks and to work on the same files, we used a repository on GitHub. Using this to upload segments of the game, ie; Game Board, win conditions, win results, etc. Which would then be uploaded to the main game file. This was to ensure that each segment would work within the game and avoid errors. It also allowed group members to be able to quickly see what another person has done without having to search through the main game file. We also used a ‘README’ file to use as a checklist for what has and hasn’t been completed, which could be seen within the repository.

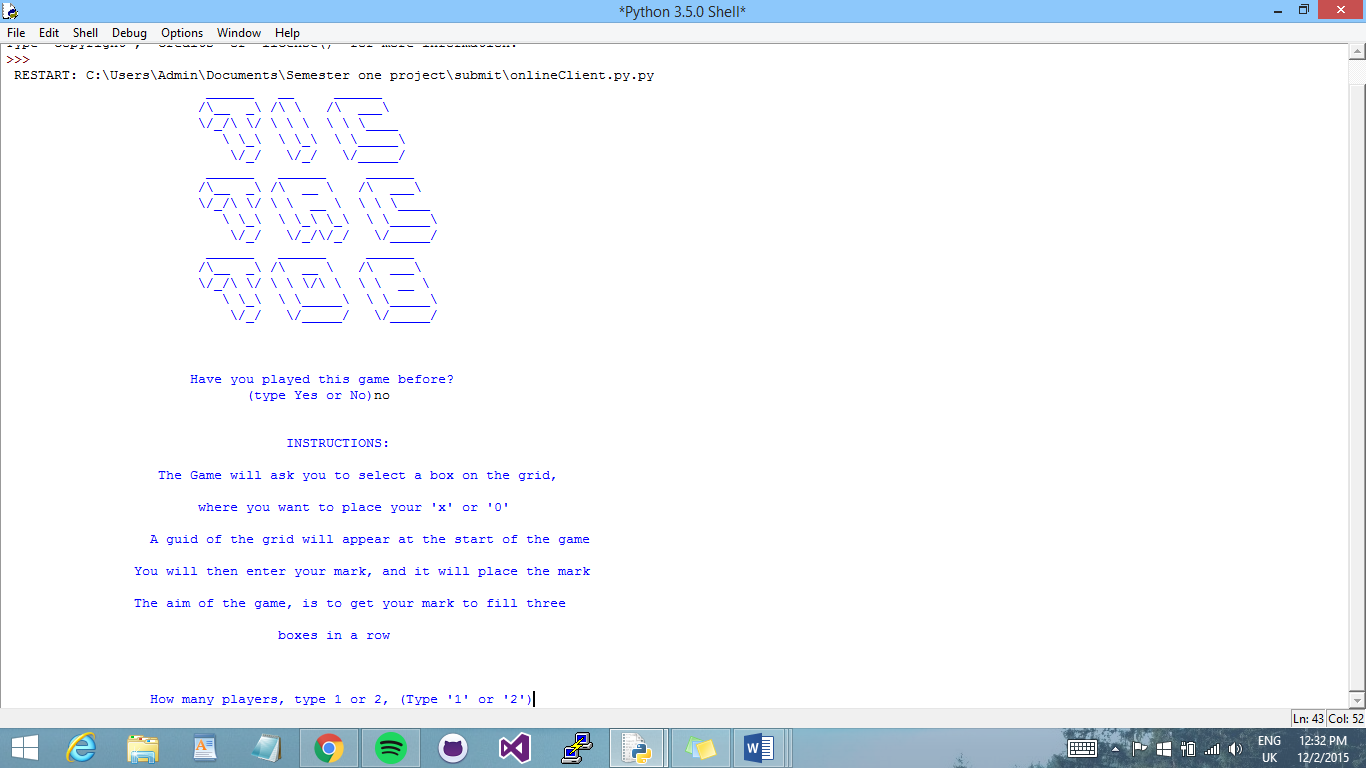
To decide who would work on a specific part of the game, was left open to all group members for them to come forward and state which part that they wished to do. Then they would declare that it was there segment when they uploaded it to GitHub. This allowed everyone to work on the specific part of the game that they felt most comfortable with, but didn’t restrict any person to just doing one certain task.

Despite each group member being able to work on a segment of the game each, often a group member would upload there segment and then another member would add to their code or adapt it slightly. This allowed us to develop segments of the game quickly and to improve each other’s techniques and understanding of the code.

**User Guide:**

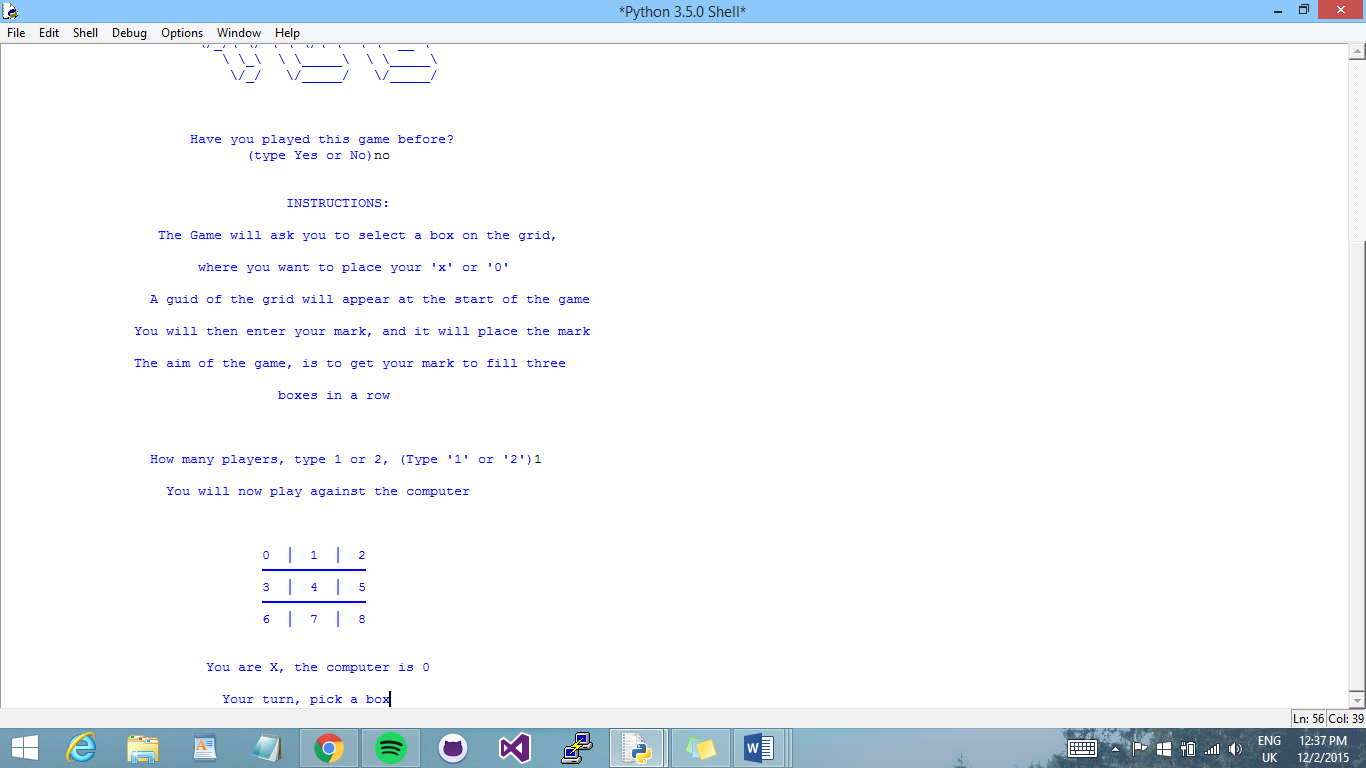
When the game starts, the Title page is show. It will ask if you have played the game before. To reply just type “Yes “ or “No”, on the screen.



If you type ‘Yes’ then you will skip the instructions and move on to player selection. If you type ‘No’, the instructions will be shown, to help you play the game. 

After the instructions, you have the player selection options available. You can choose to play one player against the computer, or two player offline or online. If you wish to play two player offline, then the two players must both be on the same computer. However if you choose offline, you can play on two separate computers.

**The Game Board:**



This is the introductory game board. It will be displayed before the start of every game.

It gives reference for players to know which numbers are for which place on the board.

The boxes are in increasing order from right to left. (0 – 9).

During the Game, players will be asked to choose a box to place their ‘X’ or ‘0’, to place your mark just type the number box that you want to place your mark in.

**Playing Against the computer:**

To play against the computer you must select one player. Then the game will start, you the user is player ‘X’ and the computer is player ‘0’. You will go first. As stated before, the reference board is displayed, and the players are declared.

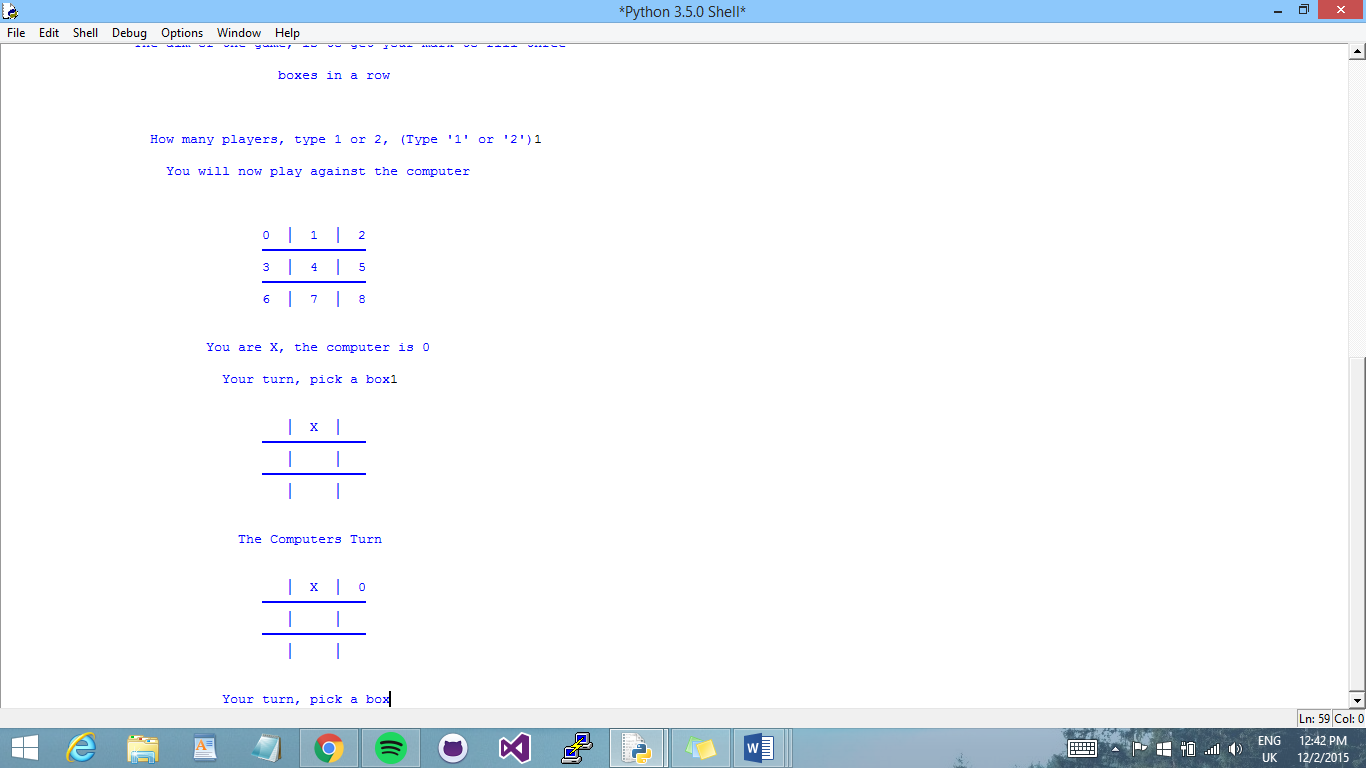
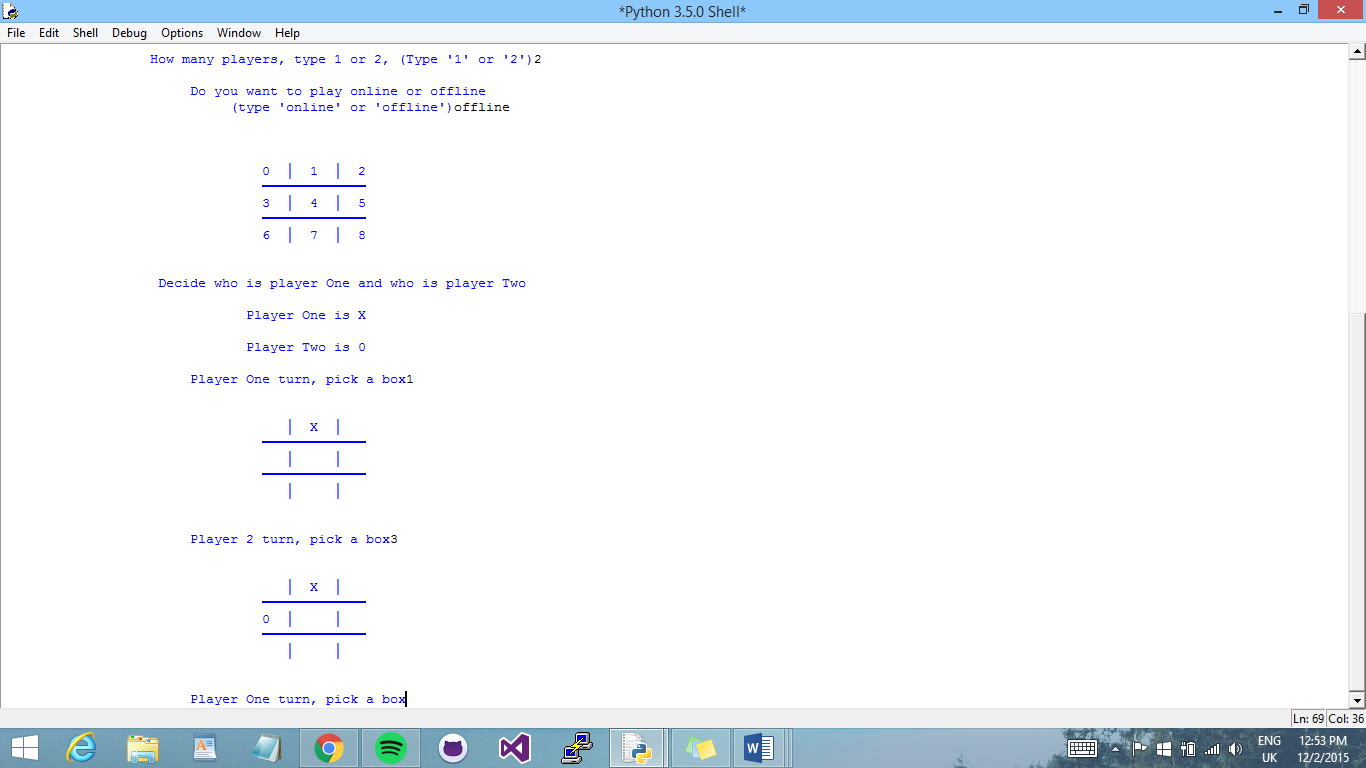
To enter your ‘X’ just type in the number box that you have chosen, and you will see you move displayed on the grid. Player sound will be played. Then it will be the computers turn. There will be a three second delay between the computers turn and yours.

If either the player or the computer, choose a place on the board, that is already taken, it will automatically move on to the next players turn.

**Two Players Offline:**

To play, both players need to be at the same computer, because of this the players can decide between themselves who is player one and who is player two.

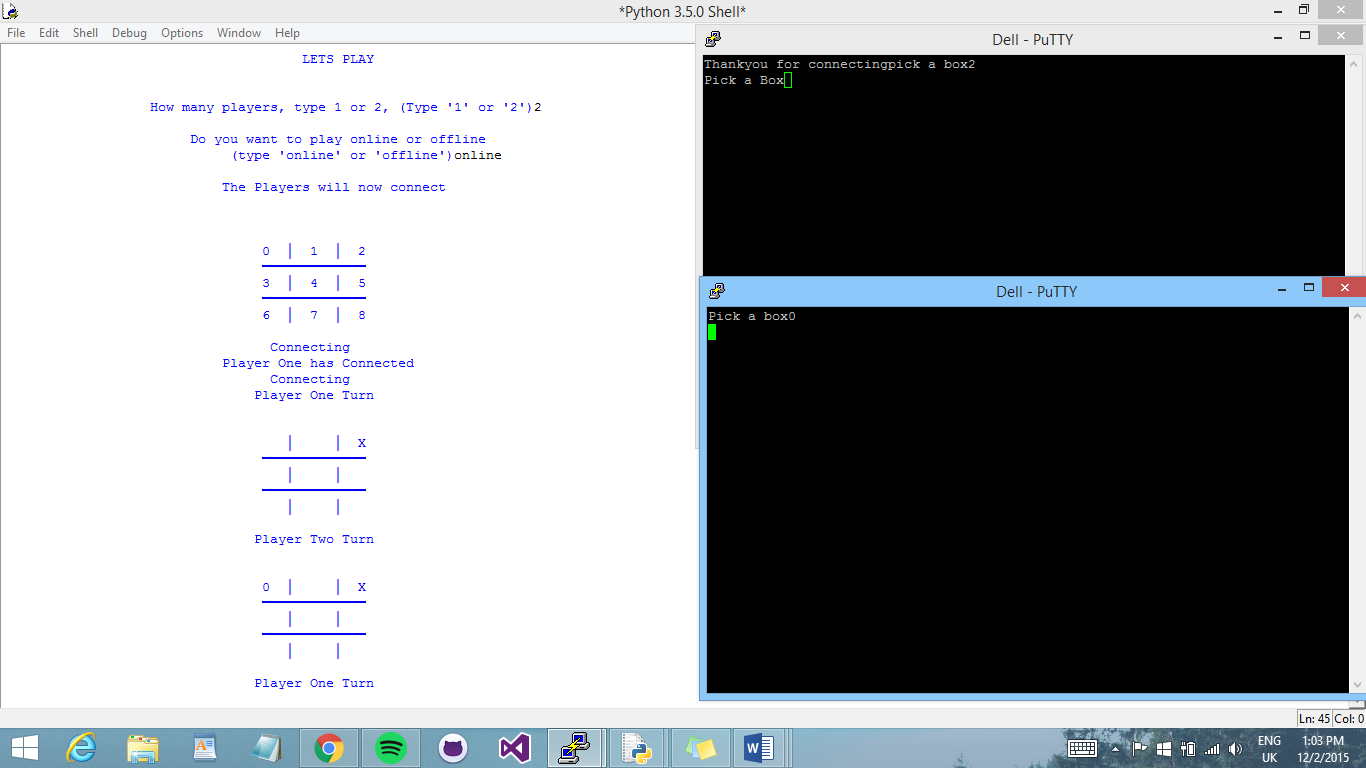
Then the game will start, displaying the reference board. If either player chooses a box that is already taken or unavailable, the next players turn will start.

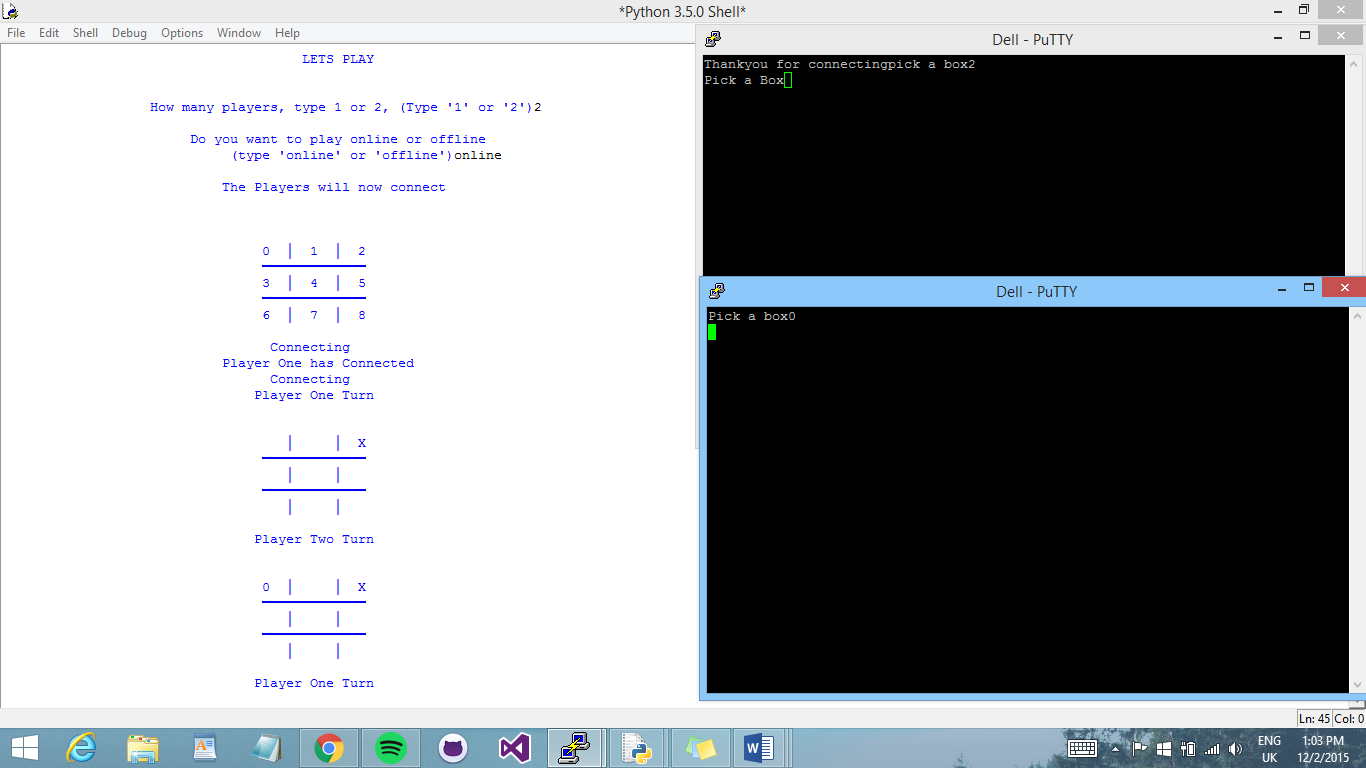


**Two Players Online:**

To play online each player must have access to PuTTy. This can be downloaded online. When the game shows that it is connecting, each player must connect. The first to connect will be player one, and the other player will be player two.

Once both players have connected. Player one will be sent a message to choose a box. Unlike before, when playing online instead of typing input on the game screen, now both players must input there chosen box on the PuTTy window.

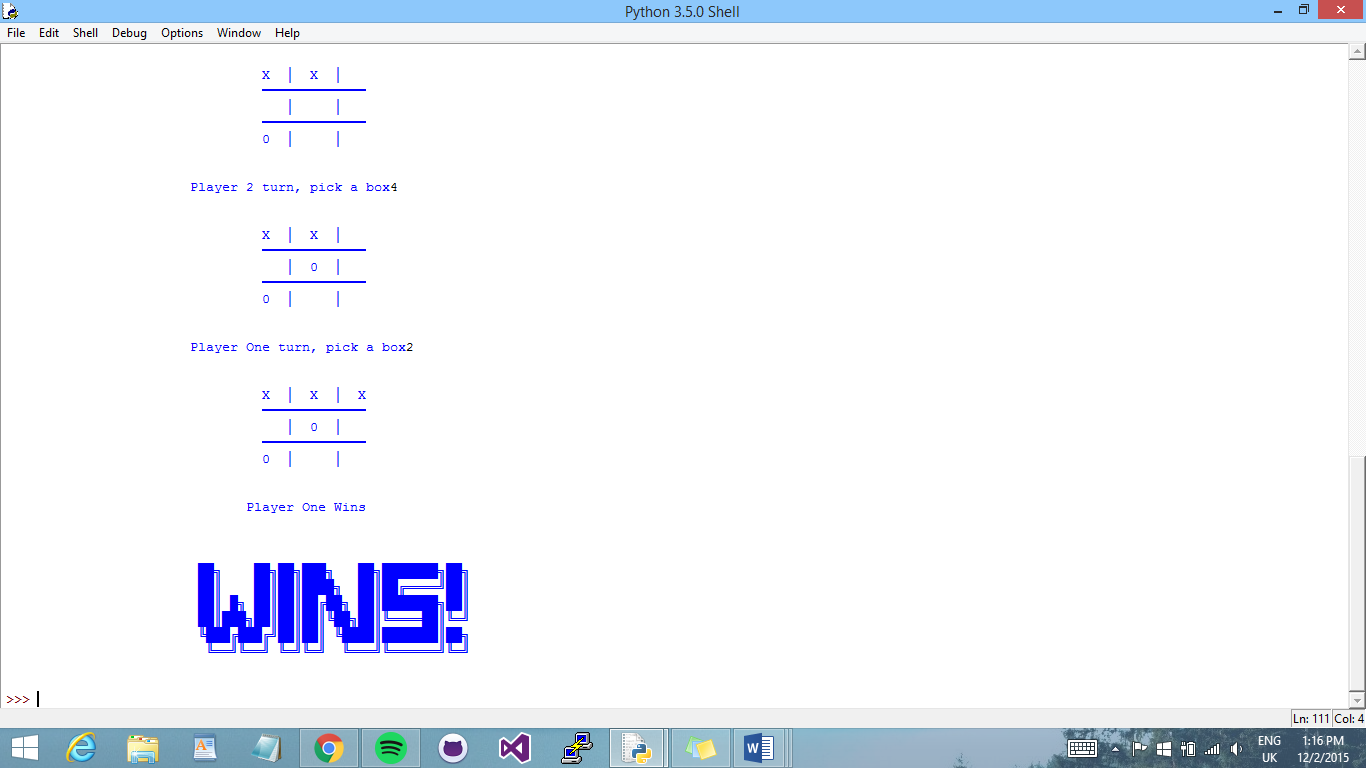




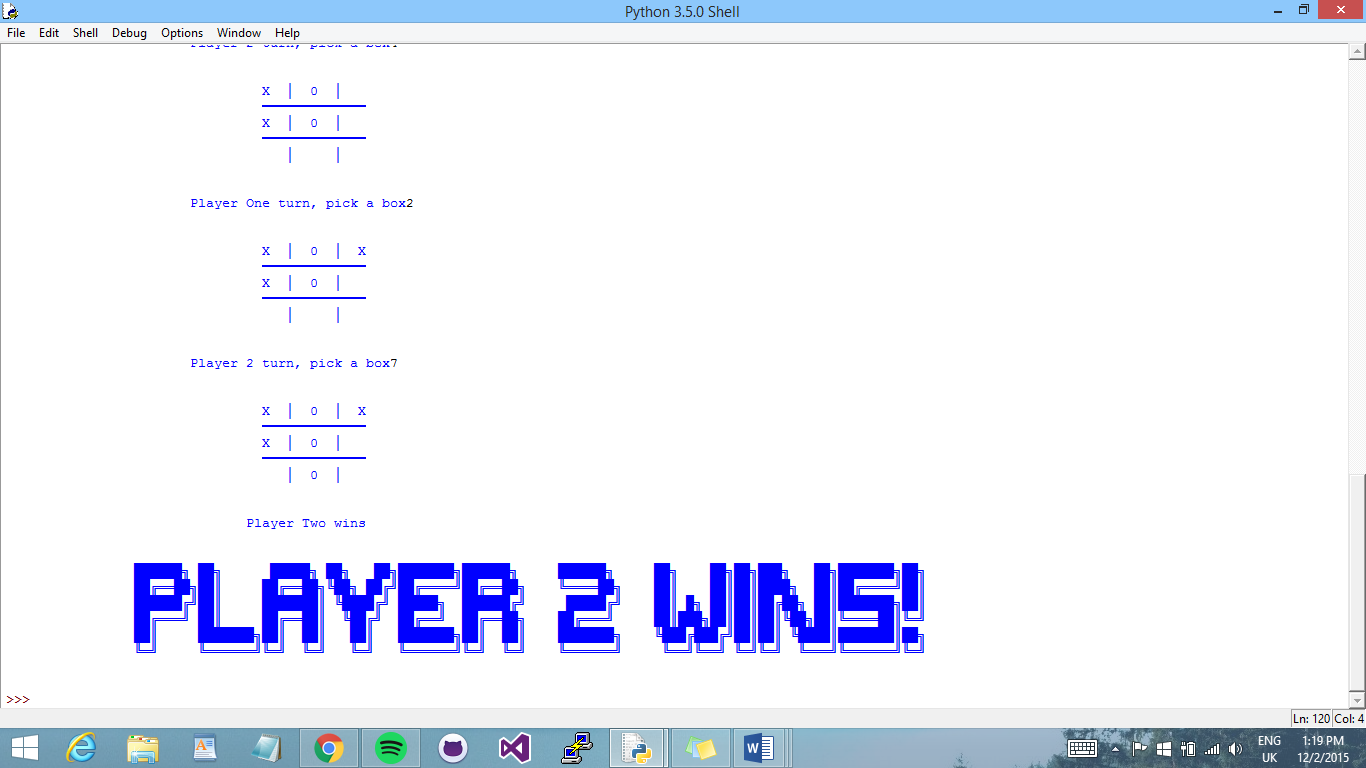
**Winning the game:**

Whether playing one player or two player, the win conditions stay the same. As stated in the instructions for the game, if either player gets there ‘X’ or ‘0’ in three boxes in a row, then they have won the game.

If player one wins then the player one win display will appear:



If player two wins then player two win display will be shown:

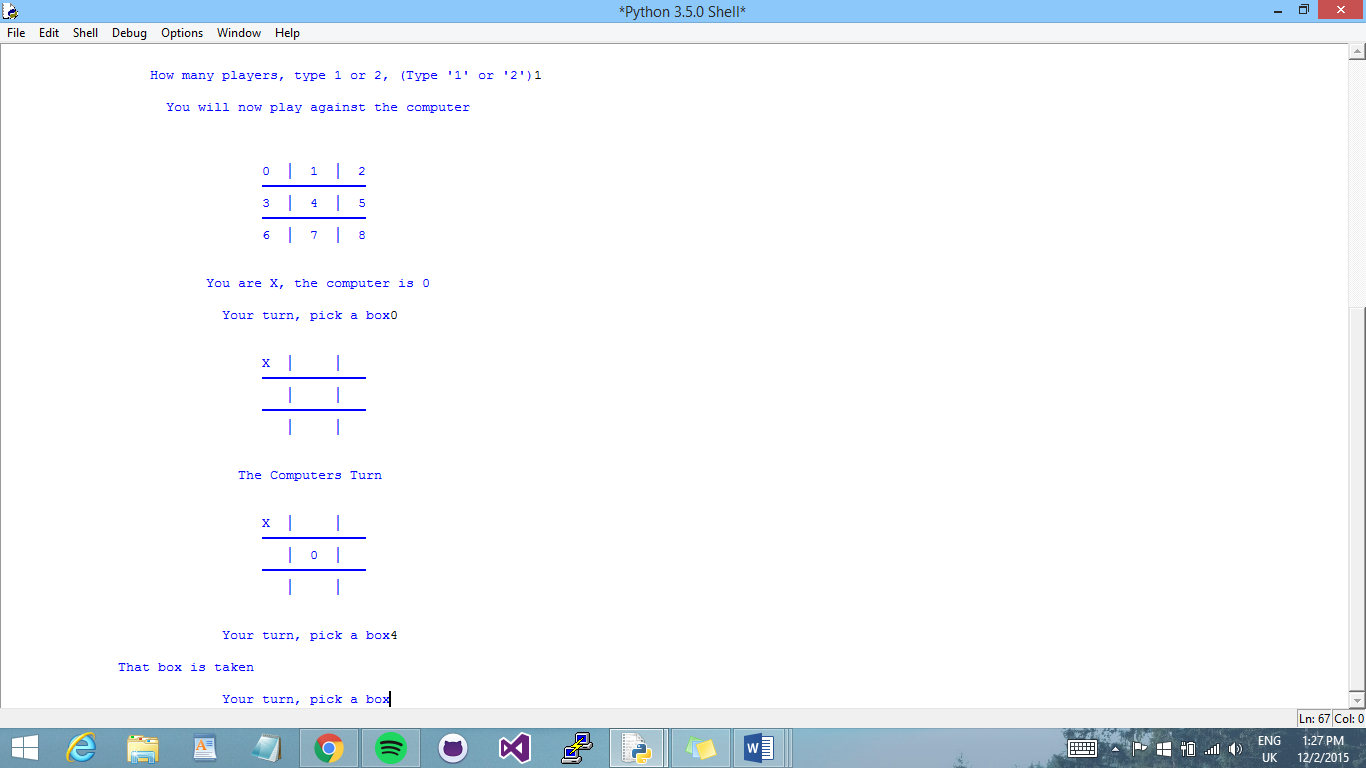


**Testing:**

The following are tests that we ran and what we did to fix them.

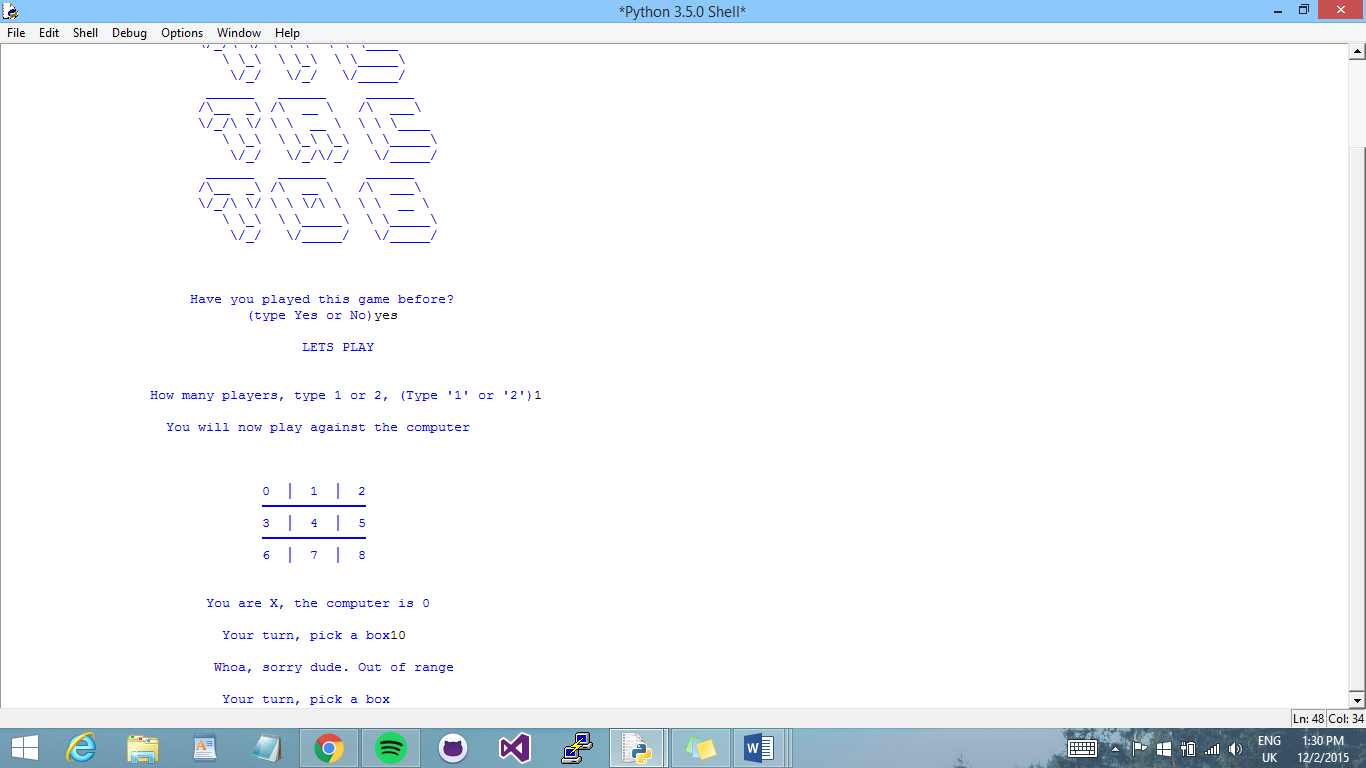
* **Choosing a place on the board that is already taken:**

To ensure that each player cannot put there mark in the same space as another player, we decided to show them the message and then allow them to choose another box on the grid.



* **Choosing a place on the board that is ‘Out of Range’:**

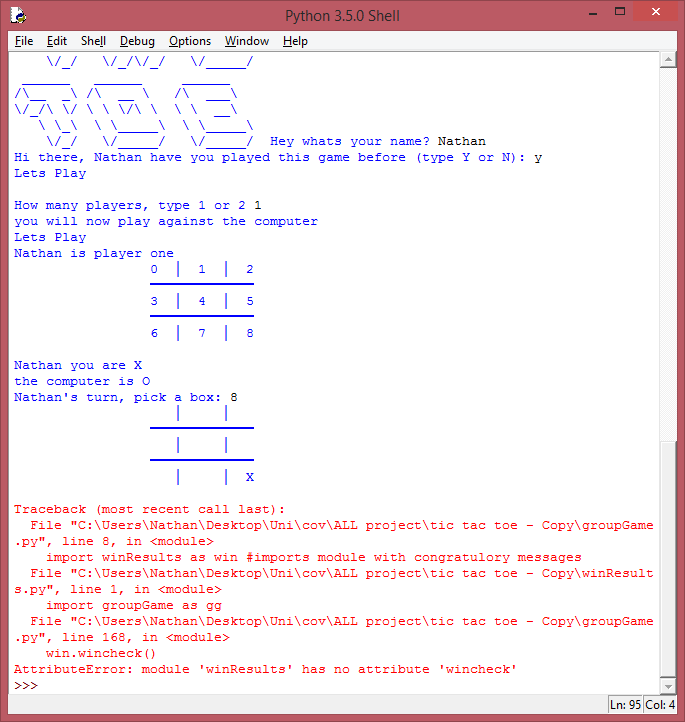
If a player chooses a box that is not on the board, so not in the range of (0-9), it will start the players turn again.



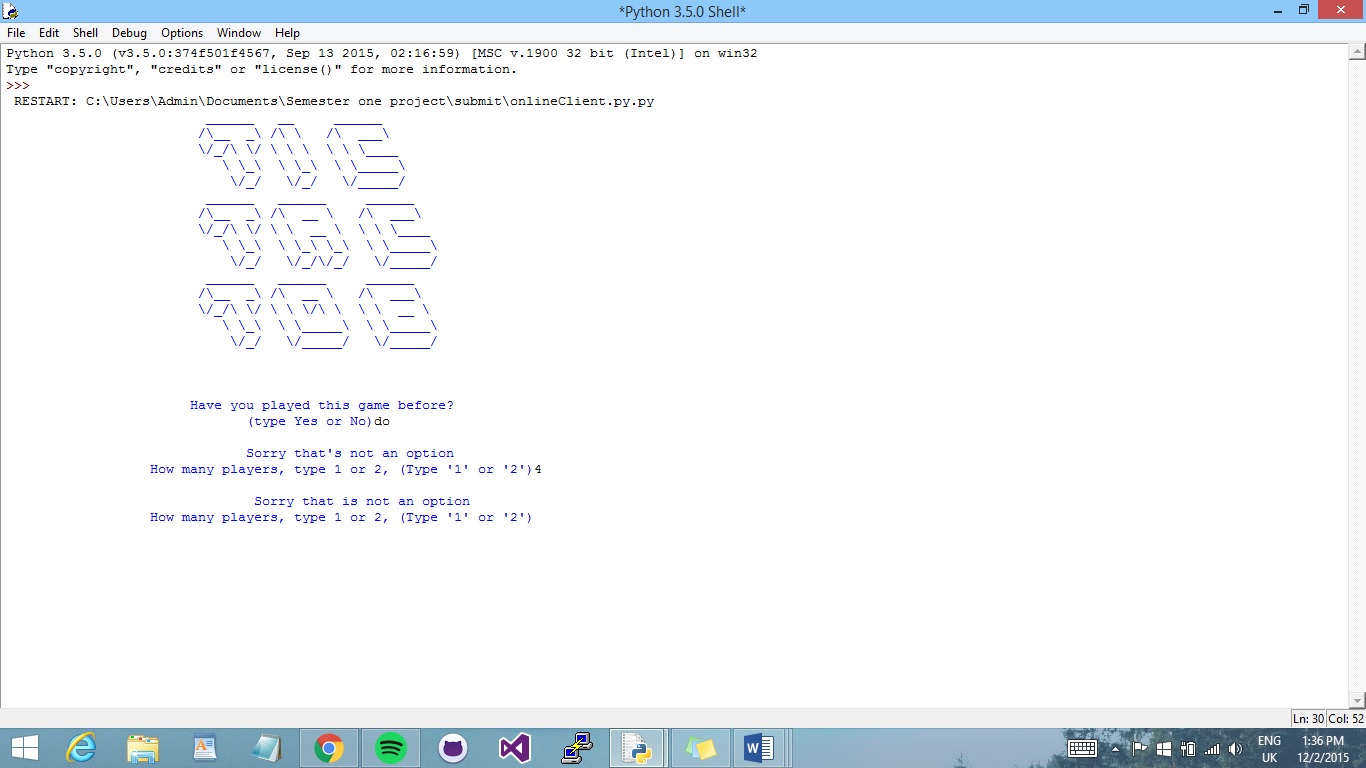
* **Running on different computers:**

When we tried placing the win-condition statements in a separate IDLE module file named *winResults* after encapsulating them inside a function called *winCheck*. We encountered a couple of errors during the course of running the game program. One of which was that the attribute variable ‘name’, assigned within the *winCheck* function, was not found from the separate variable we had made to import into the main game. we approached this error by importing the module file of the main game into the *winResults* module, thus enabling a back-and-forth connection to take place between modules, and the resulting performance of the program found no longer detected this error.





* **Unexpected user input:**



If the user enters unexpected input then, the game will let them know that what they chose is not an option, and continue on instead of producing an error message.

* **Reconnecting players for every turn:**

One of the biggest problems we encountered, was that for online play, the users would have to reconnect to the client every time that it was there turn. To stop this we put the client for each player into two different classes, and defined different methods for each go.

For each players first go, they will both connect but then for each turn after that they do not have to re connect because we used different methods for a first turn and then all other turns.

* **Sending messages and receiving messages:**

Due to the client connections for each player being in different classes. In order to allow the players to send and receive messages in from methods in a different class, we had to define the variable used for the messages to be global, in all classes and methods.

**Work Allocation:**

**Nathan:**

* **Prototype contribution**

The development of the initial, skeletal structure of the game began with the game grid (or board), which was accomplished by defining three lists each operating as the grid rows – thus constructing the grid as a whole when the program is executed (Figure 1). The elements of each list contain string values representing an empty spot on the grid. Following this, the code written for the spot selection (allowing the player to choose where to mark an X or O on the grid) involved two input functions each contained within an int function – to convert the players’ input into an integer from a string (as the inputs are read as strings by default) – and assigned to a variable, indicating a row or column in the grid (Figure 2).

The if-statement following this was written for the program to verify if the chosen spot is within range of the grid (Figure 3). If the condition of this statement results to true, the player will be able to choose which mark to place into the chosen spot; alternatively the program will rerun the spot selection for the player to choose a valid spot in the grid if the condition is otherwise false, indicated by the continue statement. Next, in order to verify if a player has chosen a place in the grid that has already been taken from their opponents’ last turn, three if-statements were included, each evaluating if a certain row was chosen (based on what has been passed on to the prior input functions – Figure 4). The if-statements contain nested if-statements, which is where the checking actually takes place once a certain parent if-statement results to true – in other words, depending on which row has been chosen in the spot selection.

Another set of nested if-statements were coded (Figure 5), each instructing the program to now place the player’s input mark into a spot in the grid once all the prior conditional statements result to desired Boolean outcomes (for the game to continue). For instance, the first statement evaluates if the first row had been chosen, as well as the first, second or third column, then the program will read the if- and else-statement nested inside which will evaluate which mark the player had chosen (from the first written conditional statement – Figure 6) before placing the mark on the grid.

Once the background processes take place, as the program runs, the grid is updated and will be displayed with the player’s mark on the selected spot, which was implemented by coding several print functions each containing the list variables constructing the grid in a format that does not appear as a grid (Figure 7). The elements of the lists are instead displayed as individual string values separated by tabular spacing, using the join function followed by the character code for tabular spacing.

Lastly, the win conditions was first configured by coding several if-statements, each evaluating if a horizontal, vertical or diagonal row of the game grid has been filled with an X or O and indicating which player has won the game, depending on what mark (X or O) the a player was allocated to. If any of the conditions of the if-statements are met, the program should print a congratulatory message for the winner, the game terminates by calling the exit function from the imported sys module. Adding the exit function from this module resolved the previous issue of the program not terminating when a player wins the game; the game would loop instead and run the next player’s turn, seeing as all the coding for the game execution has been nested inside a while statement to enable looping – especially with the use of the continue statement.

* **Modification contribution**

The code for the board was rewritten so that only one list could construct the entire board instead of three to represent each row (Figure 8). Consequently, the spot selection was re-coded to enable players to specify an integer referring to a spot in the grid instead of choosing by column and row (as indicated in the reference (introductory) game grid, shown in Figure 3 in the User Guide section). In addition, the if-statements that run after the spot selection have been completely remodified by including a try-except statement (Figure 9) where, within the try statement code space, an if-statement serves the purpose of checking if a spot in the grid has already been taken from the last player’s turn, which is run only if the player’s input value is an integer referring to an existing grid spot, i.e. if the input integer value refers to an existing element within the list constructing the game grid. If the case is otherwise, then program runs the except statement, as a method of handling the IndexError error type message, and prints a customised message indicating to the player that the spot they have chosen is “out of range” before rerunning the spot selection though the continue statement.

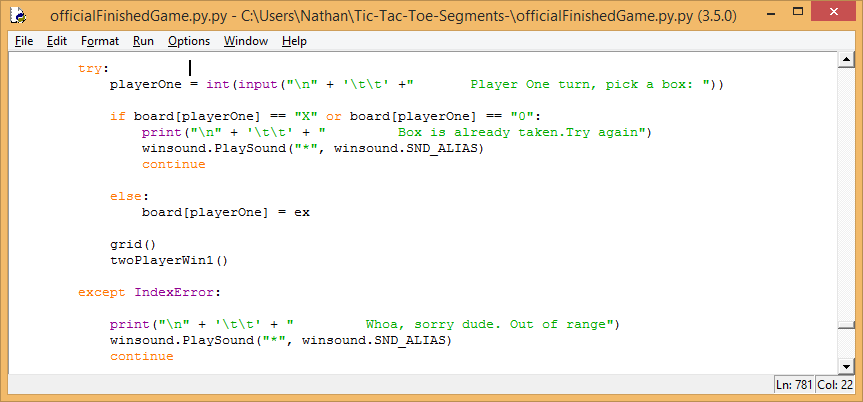


Figure 9

Further modifications involve adjusting the appearance of the game grid by changing the characters representing the grid borders (Figure 10). The hyphens that were previously coded for this purpose have been replaced by the heavy horizontal dash character that could be written as the symbol itself or the ASCII code that represents it and, nevertheless, readable in the Python interpreter (Figure 11). Following this, the character string \t was written in the print function to make tabular spaces before printing the grid, by row, and aligning it at (or roughly close to) the middle of the screen.

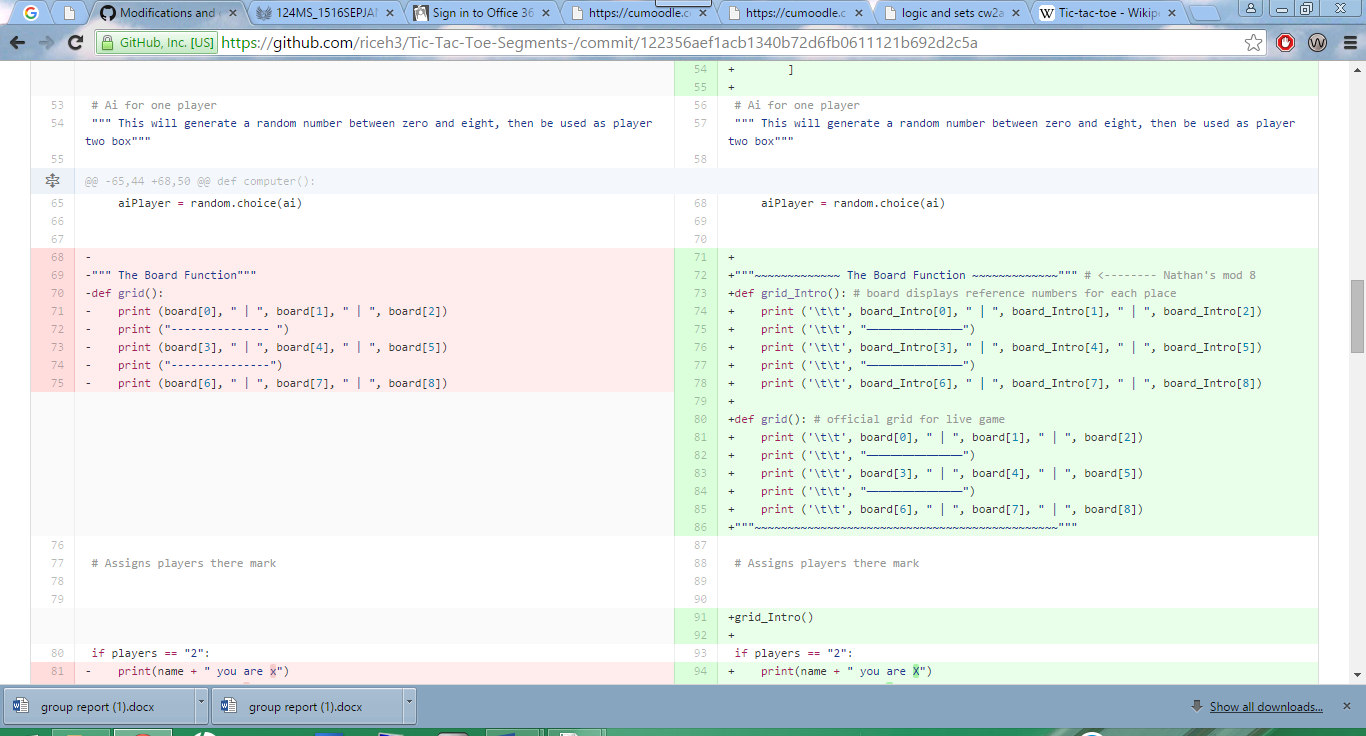
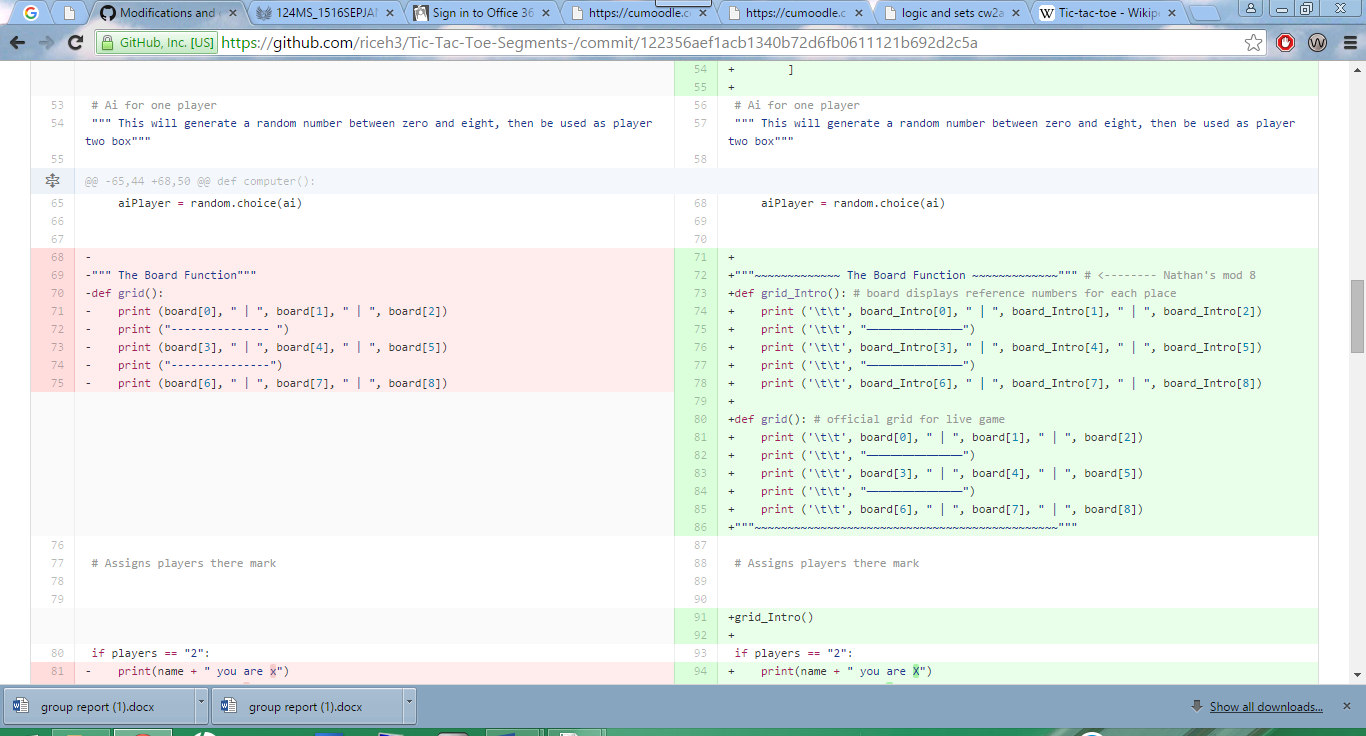


Figure 10

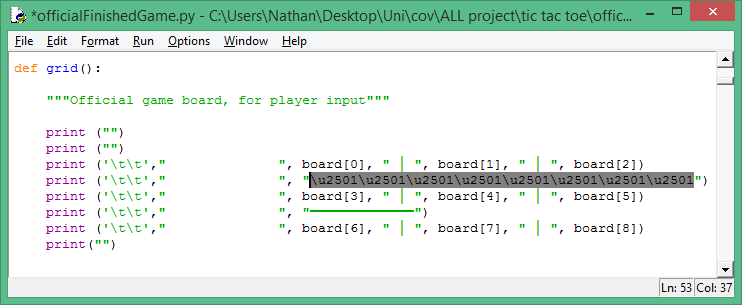


Figure 11

The use of a timer was another effective addition to the game code as this allowed a delay between the player’s turn and CPU’s turn during the game, if the game was set to the one-player mode. This was accomplished by importing the built-in time module (Figure 12) and calling a function from it named sleep which takes an argument as the number of seconds it causes the program to pause before resuming execution. The reason for this addition was to ensure that the player is fully aware that the computer’s turn has taken place. Before this function was added, the computer would execute its turn immediately after the user has placed a mark on the grid, without the player’s realisation of its rapidity.

|  |  |
| --- | --- |
| Imported modules |  |
| Calling of specified function from respective, imported modules |

Figure 12

Another further modifications were made to improve the presentation and readability of content when the game is execution, which includes adding the string character \n in different places in the code (Figure 13) to make multiple new lines between the display of the content – such as showing the instruction texts as a paragraph as well as in a centred alignment using the center function (Figure 14), the gapping between the grid and the next player’s turn, etc.

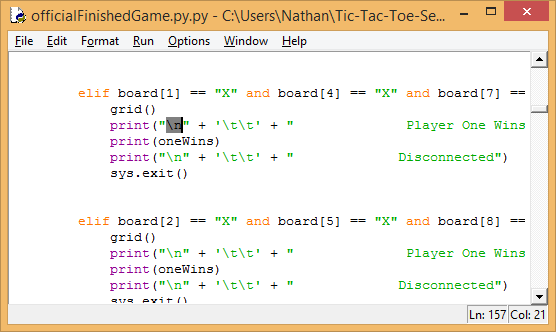


Exhibit A

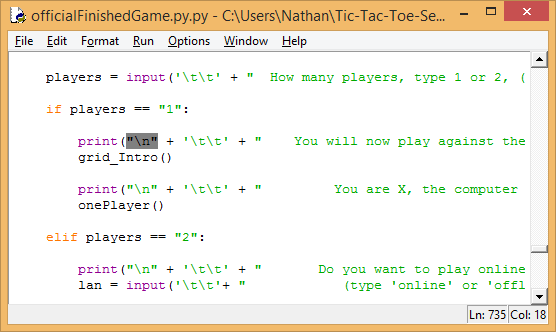


Exhibit B

Figure 13 – example segments where the string character code representation for new-line (‘\n’) is used.

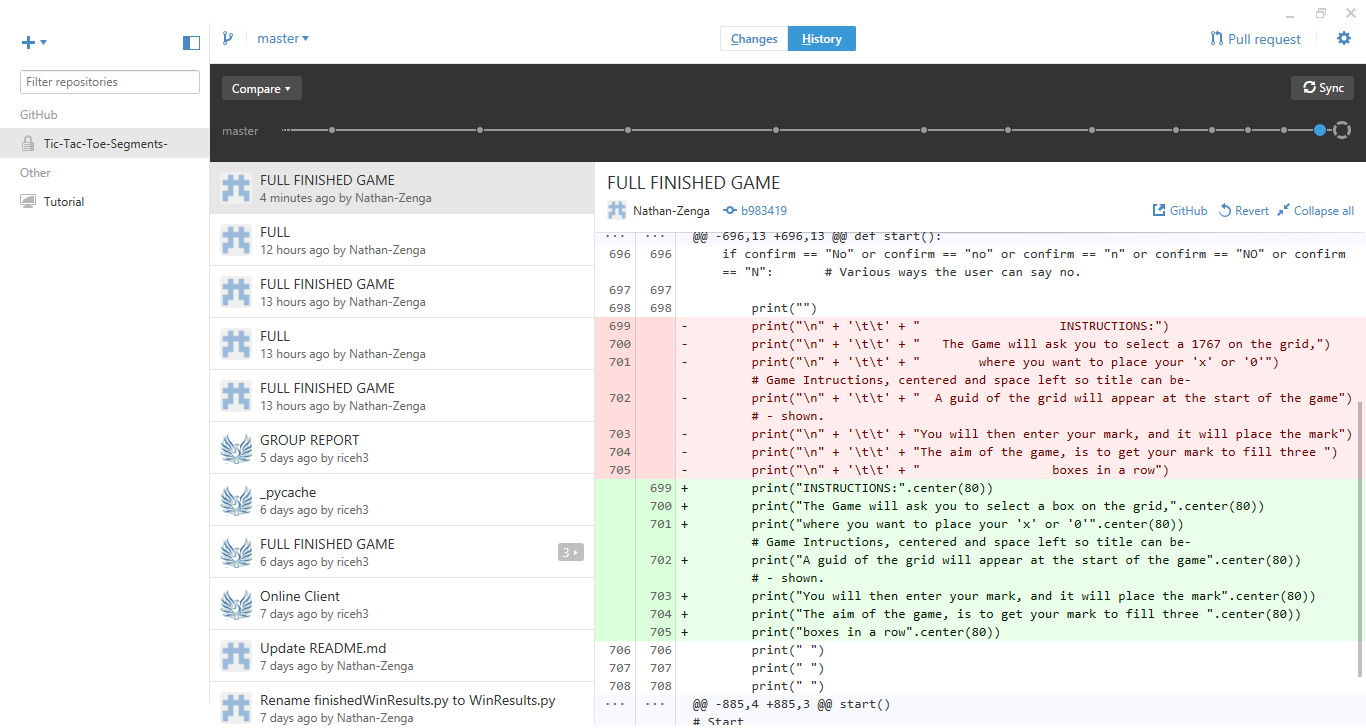


Figure 14 – before and after use of the center function

We have also created a module to import into the main game in order to show a congratulatory message if any of the win conditions result to true. The text design of the congratulatory message was accomplished by using an online application which transforms plain text inputs into artistic typography constructed by ASCII characters (Figure 15, 16).

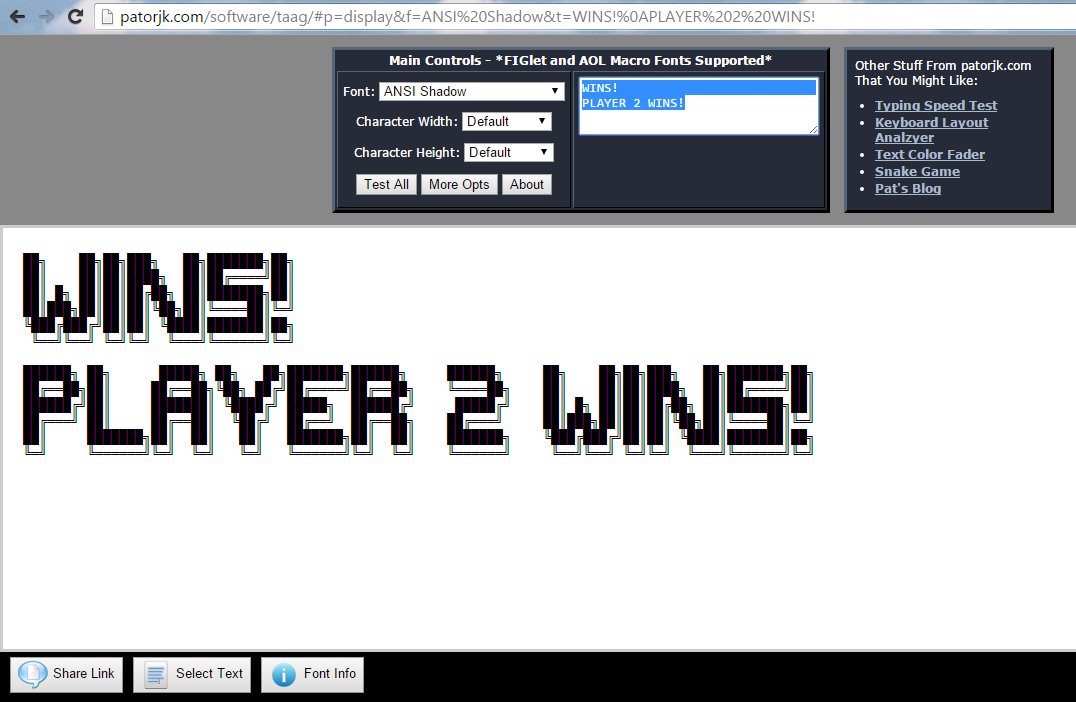


Figure 15

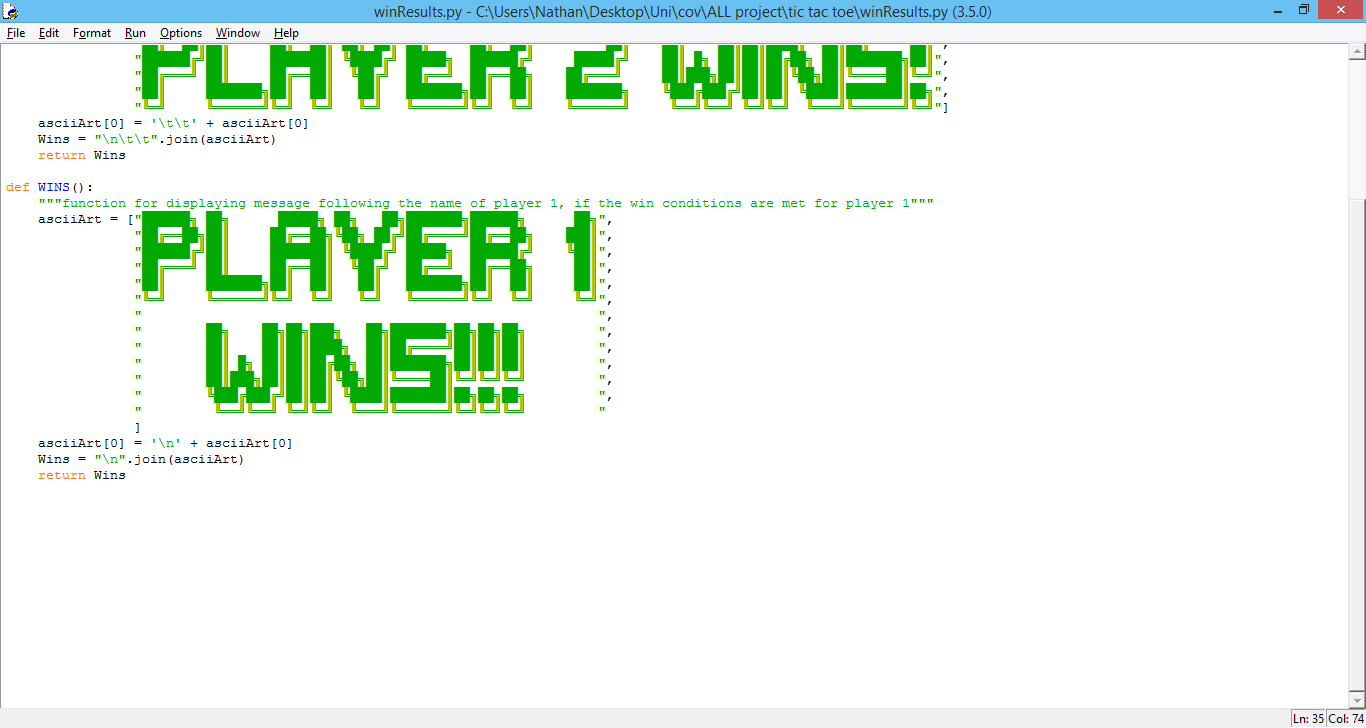
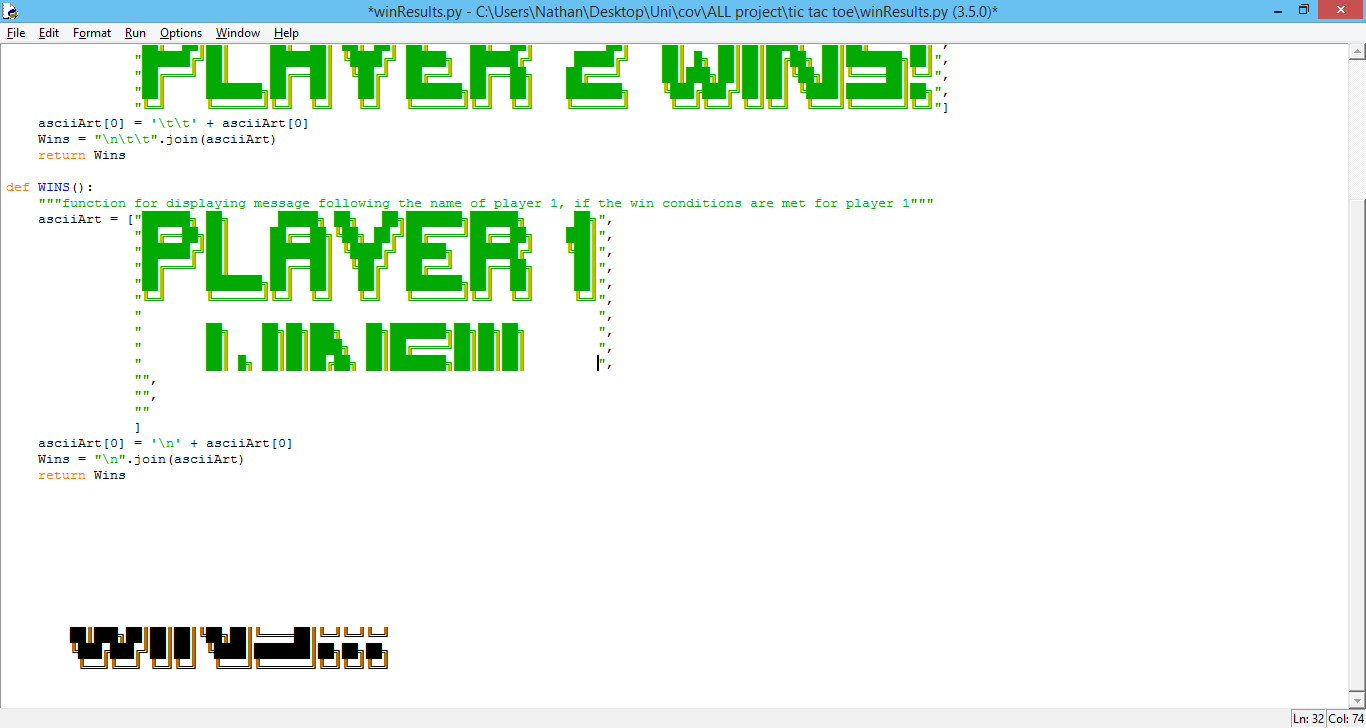
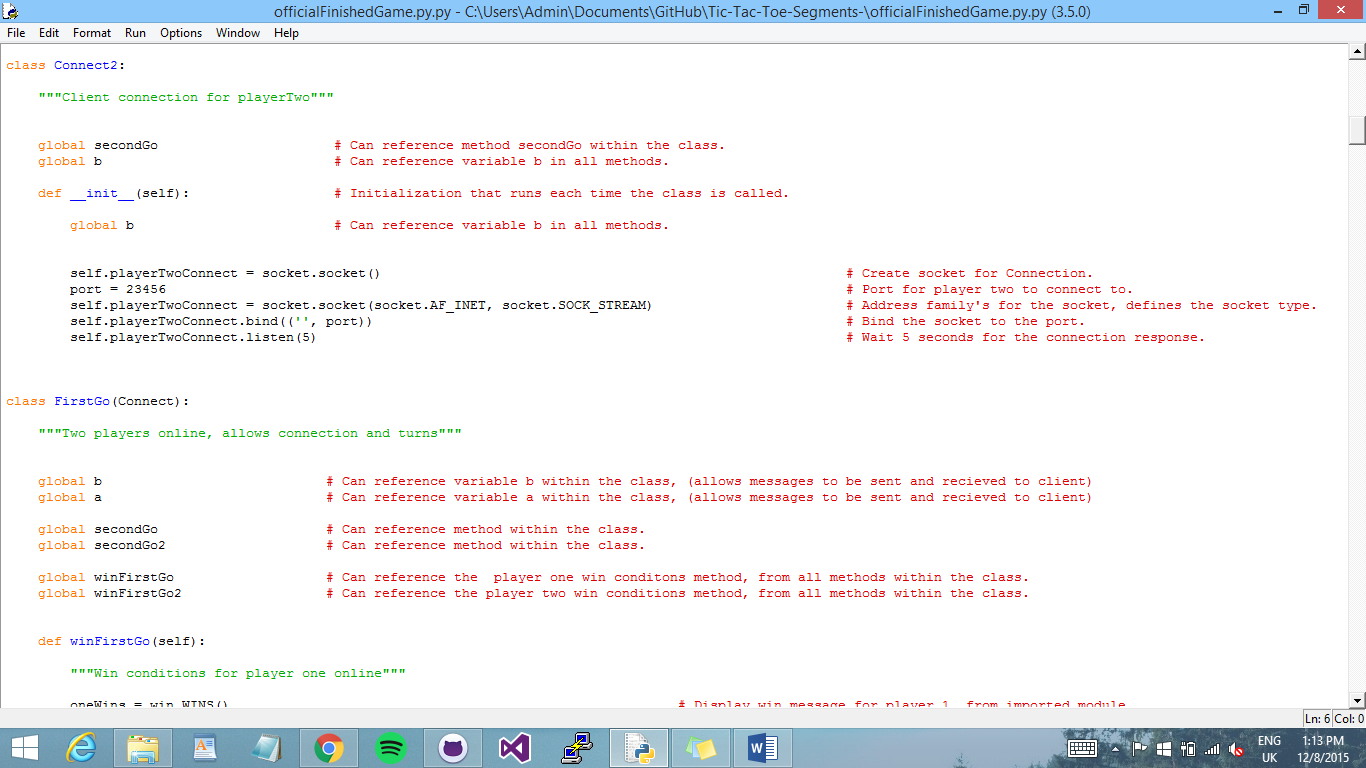
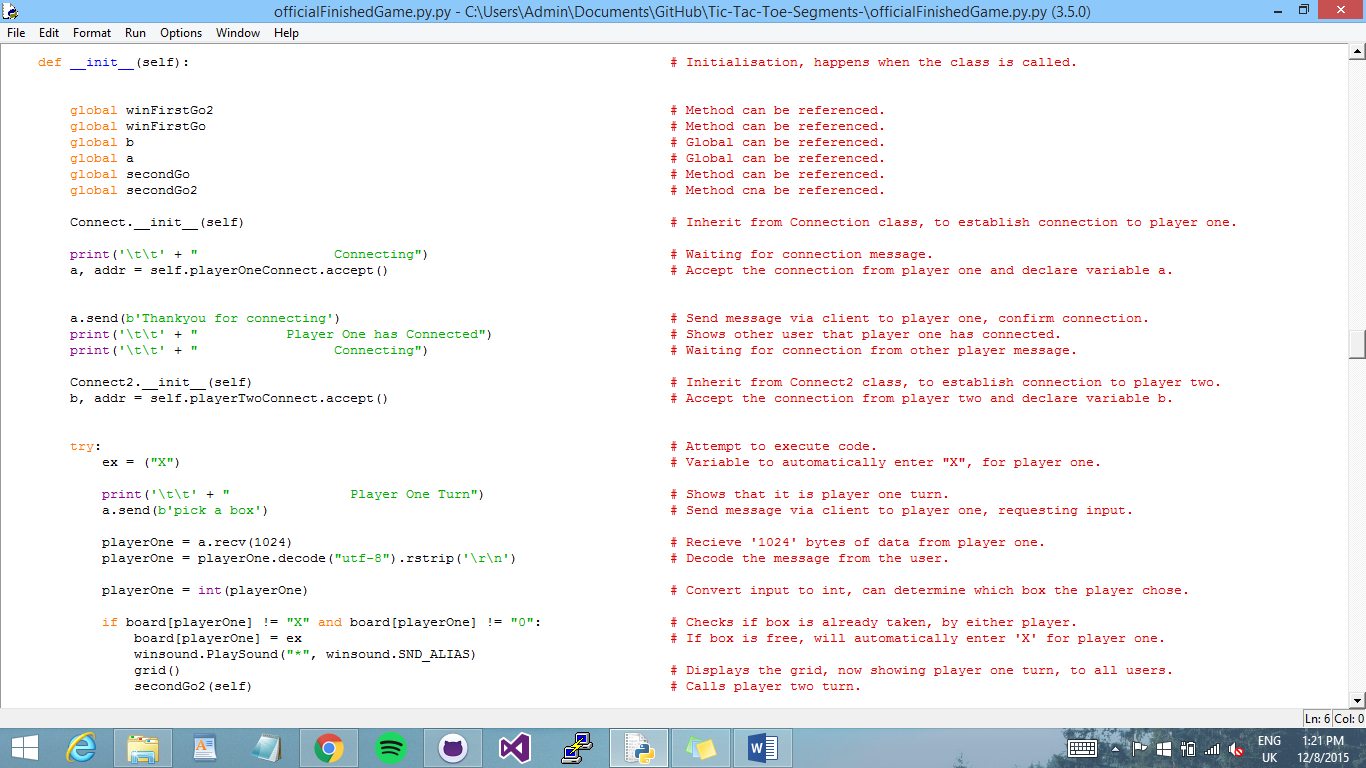


Figure 16

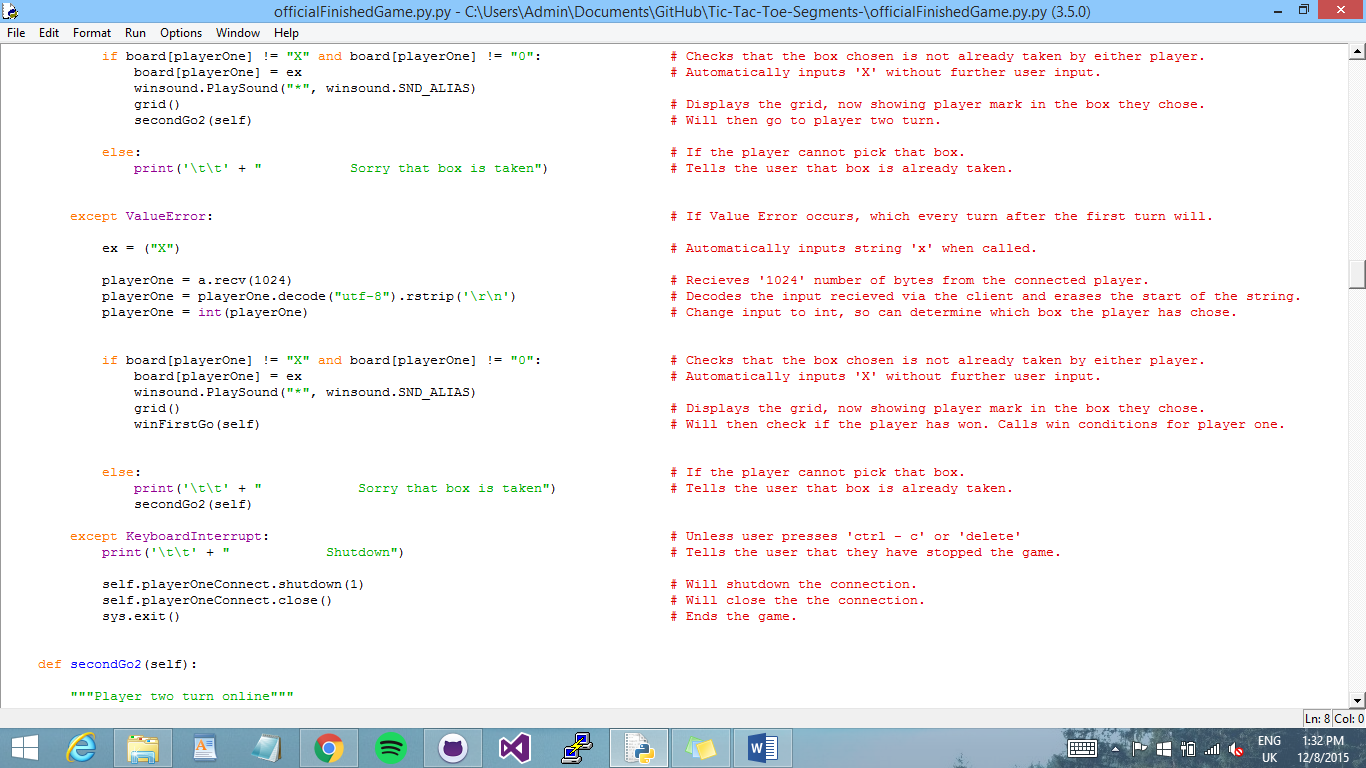
**Hannah:**

Client connection:

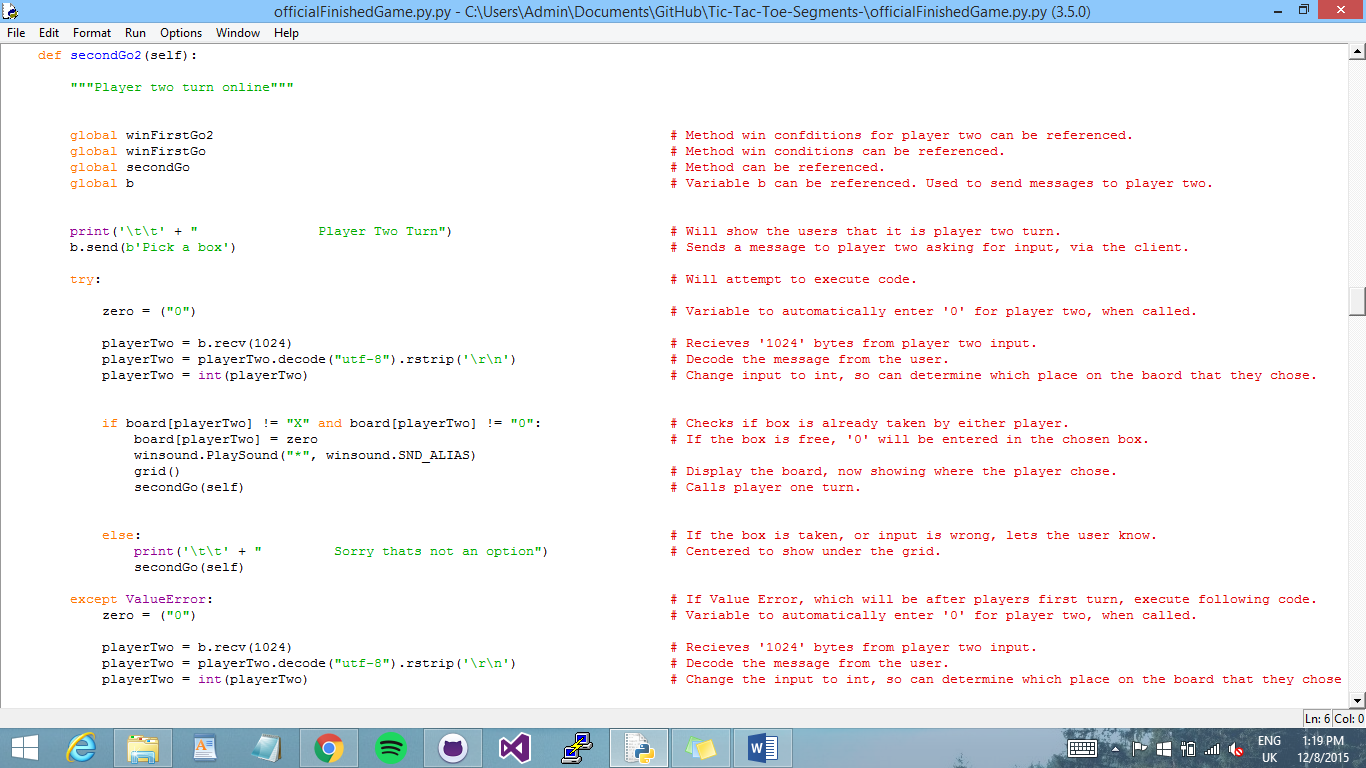
Each player connects through a class. When the class is called it will automatically allow a connection to the player through PuTTy. Each player has a different port to connect to.



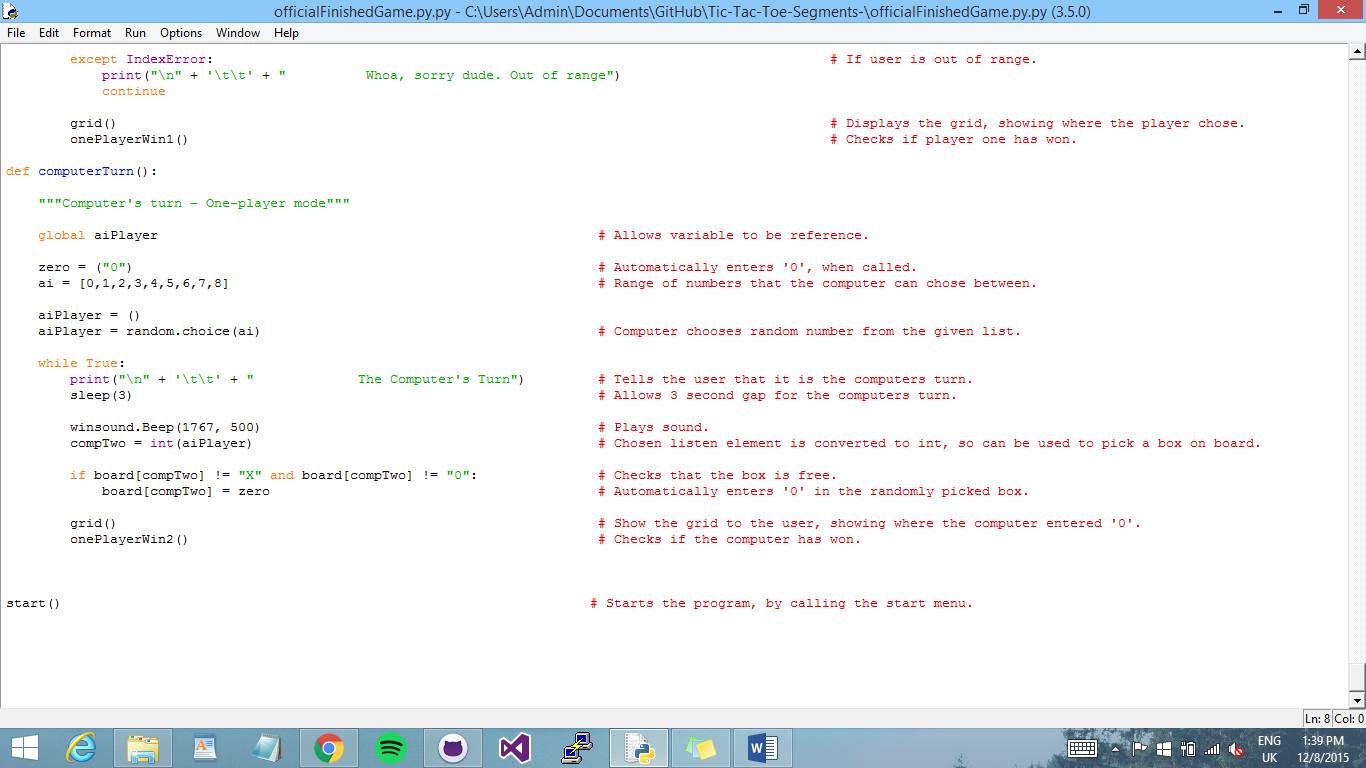
When online play is selected the initialization of this class is called. It inherits from the previously shown class to allow the connection. Then the message has to be sent and received to and from the player. The messages received from the user must always be decoded. Once they are decoded the code is very similar to how the input is accepted for the other play options. However after each players first go there will be a Value Error, so a different method from the class will be called instead.



Due to the Value Error we do not want to call the same method for a players first go, so for every turn after the players first turn there is a different method to be called instead.

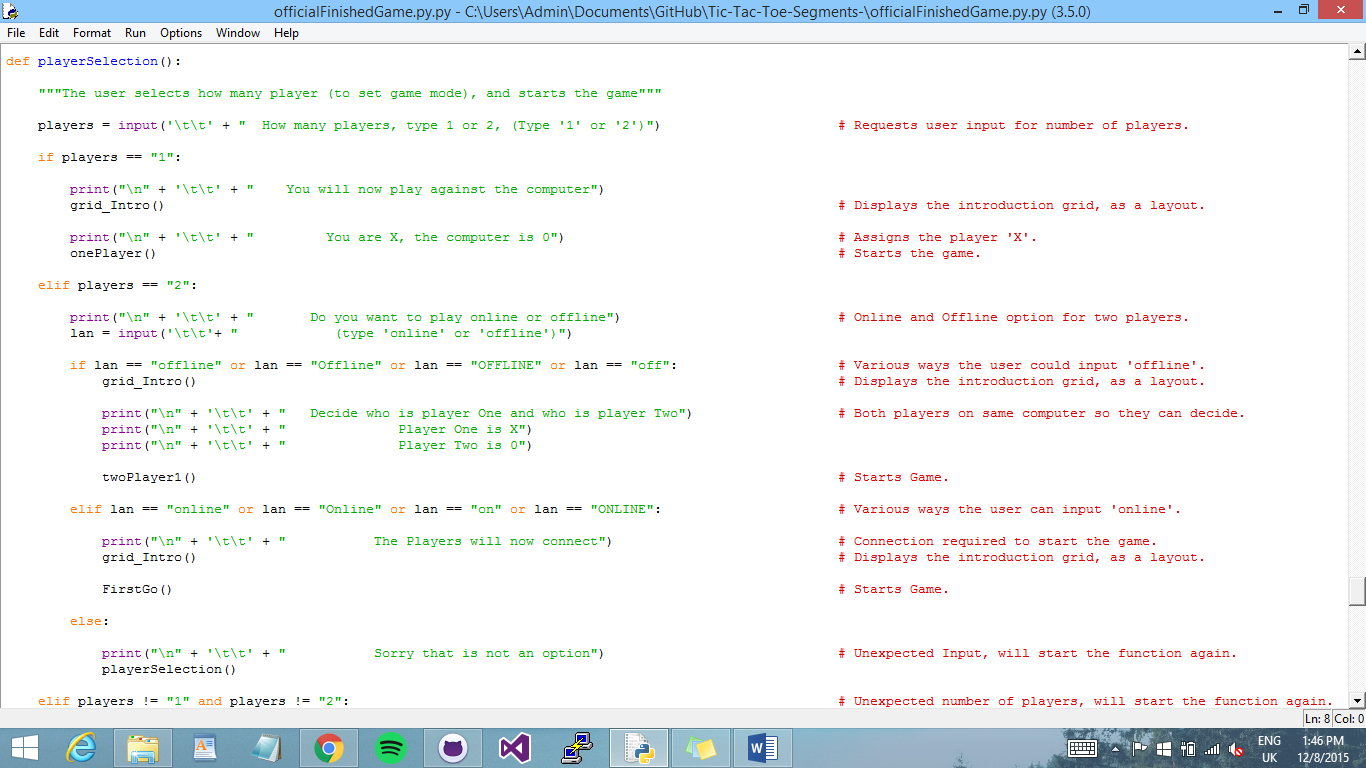


For online play each player also has their own win conditions due to at the end of each win condition the method called has to be different than the one called for the other type of play.

Ai Play:

For the computer to choose input we imported random and used choice to select a number out of the given range.

There is also a three second gap between the turn. So the player can see what has happened.

**Player Selection:**

We allowed multiple ways for the user to input their selection, this was to avoid an error as much as possible. If an error did occur then the player selection would start again, so the game could continue.

**Yavor:**

* Client Connection
  + - Allow players to play online. Worked with Hannah on it .Tested multiple ways to do it and after some time we decided to collaborate on it. And we eventually made it work properly.
* Team work spirit
* Gave many ideas through the making of the game. Tried to keep the team together and eventually three of the people worked together. Did his best on this project and eventually understood it.

**Joe:**

* Original game
  + - Made his own single player game, this being the basis of some of the code of the finished game

Originally each member of the group created their own individual game then we came together and combined our ideas. Joe's original game test worked as a single player Tic Tac Toe game, some of the code of this game formed part of the finished game.

* Sound effects
  + - Sounds added to the finished game

Also added sounds to the finished game this was done by looking through the online documentation and finding a module called winsound. To learn more about this module the help guide in the python interpreter was used. From this we were able to learn about the modules functions that could be used. We imported the function module into the game and was able to create a beep sound for the game and we also allowed the windows sound to be played when an error occurred in the game.

* Testing
  + - Testing carried out to see on the finished game to see if any unexpected bugs

Carried out his own testing on the finished product by making a test plan with the key functions of the game and their expected outcomes (black box testing). After the test plan was complete he went through the game carrying out each test and noting the actual outcome.

**Abdul:**

* Sound effects

Sound effect of beep has been added and tested to the game.

Abdul and Joe mainly focused on the sound effects of the game, Abdul was able to help Joe to import the code for a beeping sound effect in to the game. The sound effect comes into play whenever the player makes their move, not only that different sound effects has been used for each player. This makes it easier for the player to know when their opponent has made their move.

As a team, Nathan, Yavor and Hannah worked well collaboratively, using GitHub and communicating. We knew what parts of the game each other were working on. Once one of us added a segment to the game, the other would always look at the addition and discuss how it was added to the game.

Joe and Abdul work together to bring sound effects to the game communicating well with each other and sharing what they did with the rest of the group via GitHub.

**Drawbacks**

We attempted to use the TkInter module to improve the interface, yet complications arose due to the formatting of the buttons whilst trying to adjust the padding functions from the module.