Au Reikura
School of Computing and Mathematical Sciences

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

DEPARTMENT OF
COMPUTER SCIENCE
Te Tari Rorohiko

2025

Contributors

J. Turner

V. Moxham-Bettridge

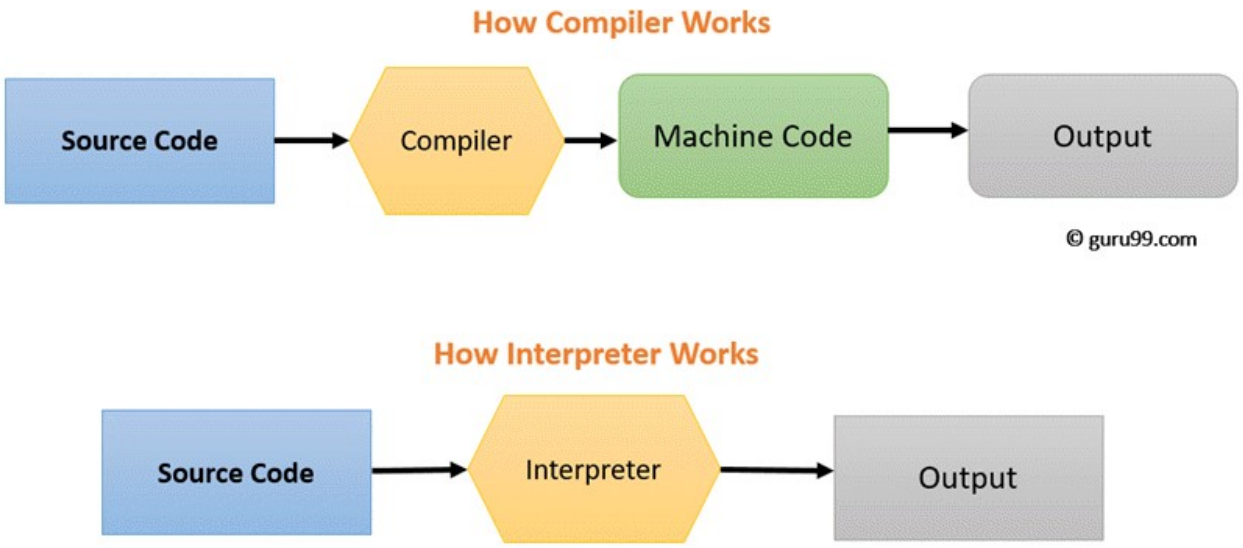
J. Bowen

J. Kasmara

INTRODUCTION TO PYTHON

Today we introduce you to a scripting language very popular in the realms of data science, mathematics and website development, Python. How Python differs from other languages you may have encountered before is that it is a *scripting* language rather than a *programming* language (although this term is often used interchangeably). The difference in terms relates to how they are executed.

Figure 1: A diagram showing the difference between interpretation and compilation1



Interpreted vs Compiled languages

If you're working through this manual, then chances are you have encountered compiled languages before, either from a previous CSNeT or through your own experience. A few examples of these that you are probably familiar with are C# (used for game development with Unity), Java (the language that Minecraft was built with), Go, Rust and so on. The compilation process (see figure 1) starts when the code you write (this is called *source code*) goes through a translator (called the *compiler*) which translates your code to numbers the computer can understand (this is called *machine code,* see figure 2). Once translated, the *machine code* can be run (often by double clicking on a program icon) which results in some *output* being displayed.

[1] Diagram retrieved from https://www.guru99.com/difference-compiler-vs-interpreter.html

Due to this, once a program has been compiled, it can be run as many times as is desired without the need to recompile (unless changes to the source code have been made).

With interpreted languages however, each line of *source code* is read by a translator that translates it into *machine code* that produces some *output* at the time of execution (i.e. after you have clicked on the program's icon). This happens one line at a time which is why interpreted languages are usually slower than their compiled counterparts. Unlike compiled languages, there is no final executable file generated that can be reused, which means the whole translation process takes place every time you run the program. Although Python is typically used in the fields of machine learning and data science, some game studios have used it to build their products, with popular examples being Battlefield 2 and The Sims 4.

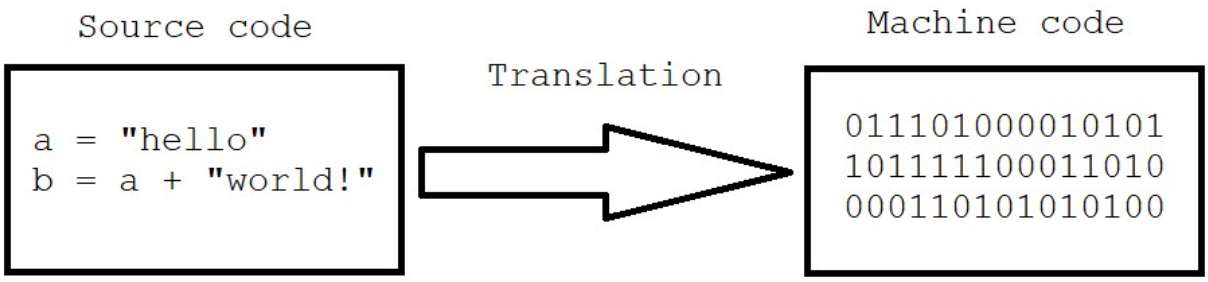


Figure 2: A diagram showing the difference between source code and machine code

The Tic-Tac-Toe game

Your task for this week is to finish a command-line interface (CLI) implementation of the game, Tic Tac Toe[2]. The `tic_tac_toe_game.py` file (the one you will be modifying to do the exercises below) can be found on the Slack channel. Open the file in Visual Studio Code and have a quick look through it to familiarise yourself with the contents before continuing.

How to get started

As this game is CLI based, we first have to open a terminal to run our game in, fortunately Visual Studio Code features various integrated terminals which makes this easy.

[2] Code adapted from https://www.scaler.com/topics/tic-tac-toe-python/

To open a terminal, locate the toolbar at the top of the screen and click on the word 'Terminal'. A dropdown menu will then appear presenting more options; now select 'New Terminal' (see figure 3).

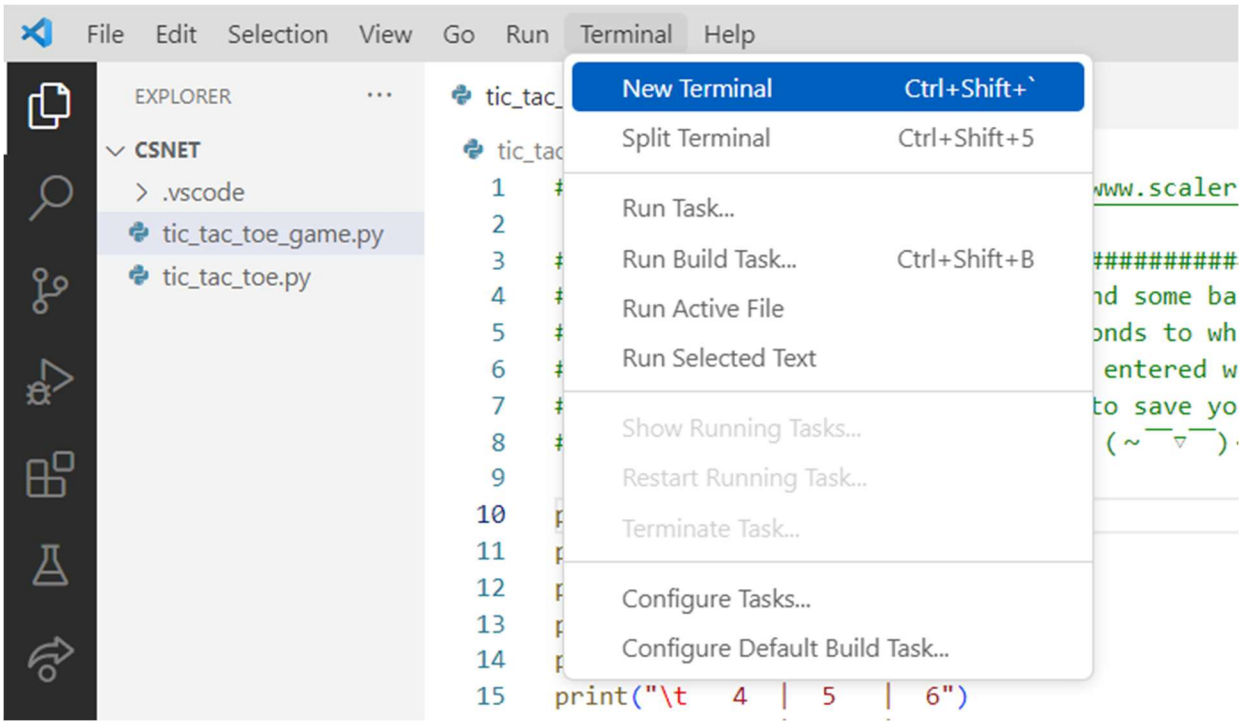


Figure 3: Opening a terminal in Visual Studio Code

A terminal window should now appear at the bottom of the screen (see figure 4).

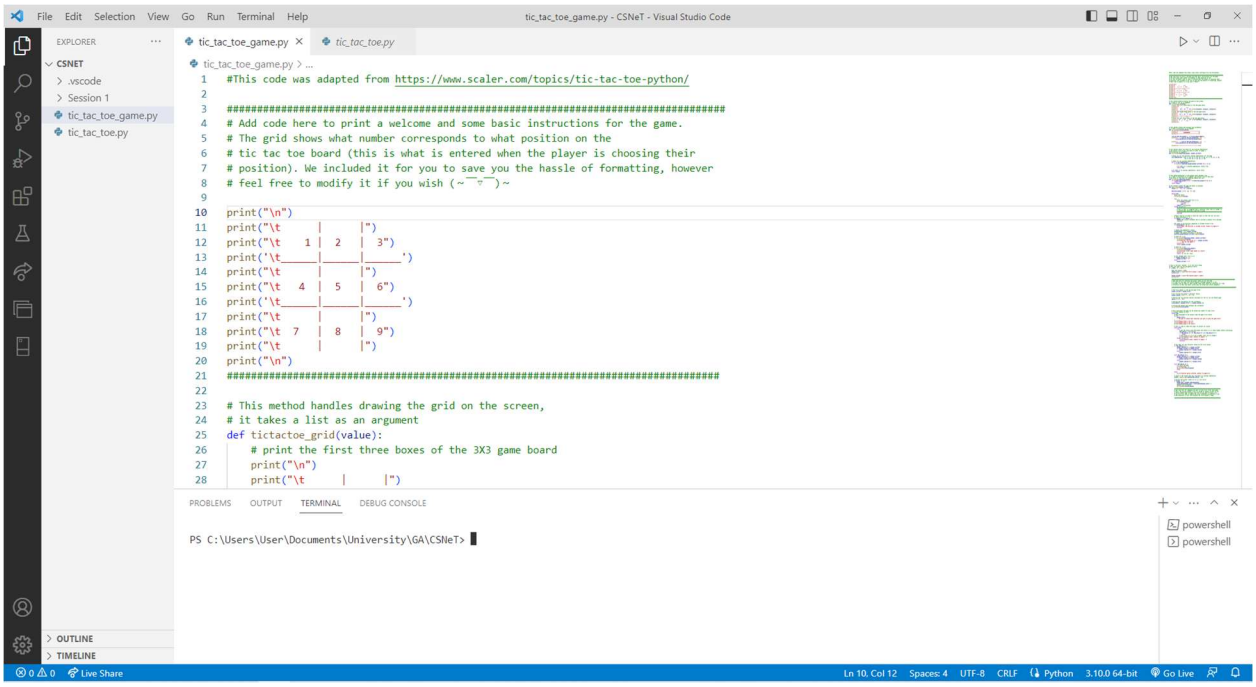


Figure 4: The opened terminal in Visual Studio Code

The terminal opens in a different location to where your file is located, so to make sure we are in the correct place, type '`cd Downloads`' (which means change directory to 'Downloads') and press enter. To then run the game, type '`python tic_tac_toe_game.py`'.

An error message will now appear in the terminal, see figure 5 below.

Figure 5: The error message

The program breaks and does not run, why?

When issues occur unexpectedly, the first place to start is with the error message as it usually provides helpful information about the error (or errors if multiple occur). The message we've received states there is a '`NameError`' error on line 164 of the code due to a variable, '`player_first`', being undefined.

Looking at the code on line 164 (see figure 6 below), we see that the value of `player_first` is to be stored in `player_current`, however `player_first` hasn't been initialised or even created yet hence the '`NameError`' error (i.e. the variable doesn't exist).

```
144  # This is the main 'method', it is the first thing
145  # that runs when tic_tac_toe_game.py starts
146  if __name__ == "__main__":
147
148      #get the players' names
149      print("The First player's name")
150      print("\n")
151
152      print("The Second player's name")
153      print("\n")
154
155      ##############################################################################
156      # Add code here for checking the player names are not the same.
157      # If they are the same, you should display a message saying it is not
158      # allowed, and ask them to input another name (this should be contained in a loop
159      # structure so that this basic action does not break your entire program!)
160
161      ##############################################################################
162
163      # The first player is the one who goes first
164      player_current = player_first
165
```

Figure 6: The section of code the error message was referring to

To fix this, we should store the player's name in a variable called '`player_first`', however, we need to get the name from the player first. To do this, we need to change the `print` statements to `input` statements as that will allow us to display text to the terminal but also receive input from the user.

Change line 149 to '`player_first = input("The first player's name")`' and do the same for the second player on line 152 (i.e. store their name in a variable called '`player_second`').

Now run the program using the command mentioned on the previous page and the game should start successfully. If an error message occurs, it means you have done the previous step wrong

(make sure the variables are named '`player_first`' and '`player_second`' otherwise the game will break).

After entering the players' names, the game should now present a scoreboard (see figure 7), however the message currently displayed is very confusing (i.e. who gets to choose first?).

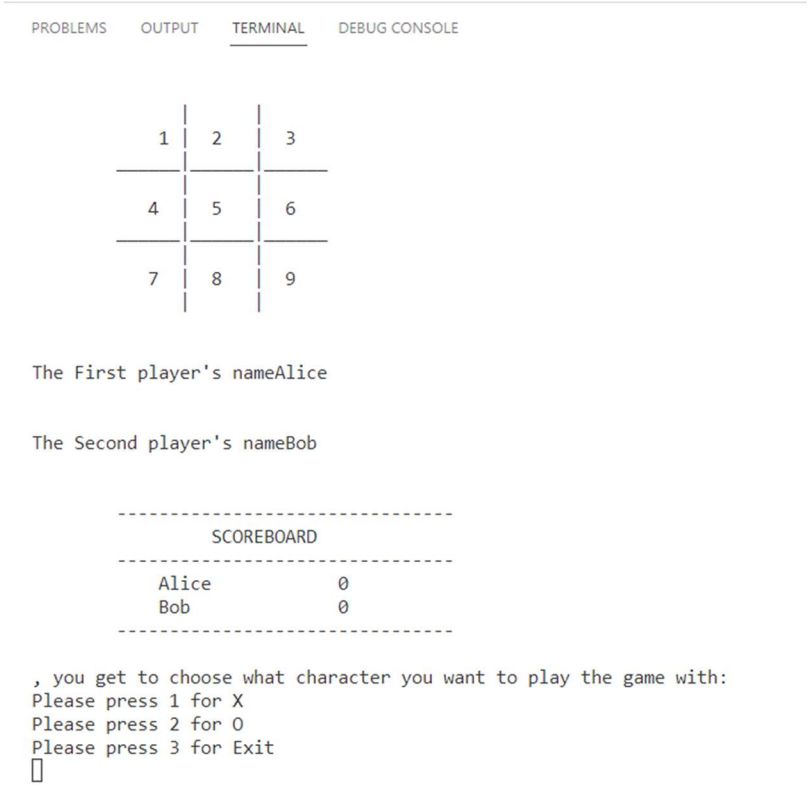


Figure 7: The game's scoreboard

To let the players know who starts, we need to modify the `print` statement on line 183 (see figure 8).

```
179     # This loop means the game can be played any number of times until
180     # a player chooses to exit
181     while True:
182         # Menu displayed to the players when the game first starts
183         print(
184             ", you get to choose what character you want to play the game with:"
185         )
186         print("Please press 1 for X")
187         print("Please press 2 for O")
188         print("Please press 3 for Exit")
189
```

Figure 8: The print statements for the scoreboard

For the sake of simplicity, we'll let the first player start the game. The `print` method in Python makes this easy as all we have to do is pass it the variable that holds the first player's name (i.e. `player_first`); see figure 9.

```
179     # This loop means the game can be played any number of times until
180     # a player chooses to exit
181     while True:
182         # Menu displayed to the players when the game first starts
183         print(
184             player_first,
185             ", you get to choose what character you want to play the game with:"
186         )
187         print("Please press 1 for X")
188         print("Please press 2 for O")
189         print("Please press 3 for Exit")
190
```

Figure 9: Modifying the print statement

Now when it is run, you should see the name that was entered for the first player. Try out the options presented, what happens?

If you had tried them all, you should have noticed that the first and second options work but the last option caused an error. Ignoring that for now, have a look at the series of `if` statements on lines 206 - 225 (see figure 10 on the next page). Although the third option is meant to quit the game, does it do so?

```
205         # the logic for the character chosen by the first player
206         if the_choice == 1:
207             player_choice['X'] = player_current
208             if player_current == player_first:
209                 player_choice['O'] = player_second
210             else:
211                 player_choice['O'] = player_first
212
213         elif the_choice == 2:
214             player_choice['O'] = player_current
215             if player_current == player_first:
216                 player_choice['X'] = player_second
217             else:
218                 player_choice['X'] = player_first
219
220         elif the_choice == 3:
221             # to quit the game
222             print("Exiting...")
223
224         else:
225             print("Invalid option entered, please try again\n")
226
```

Figure 10: The logic for the menu of options presented to the player

The answer is no, the game doesn't quit. It prints '`Exiting...`' but doesn't actually exit due to no code instructing it to do so. To fix this, we need to add a `break` statement which means we exit the block of `if` statements and continue on line 228. As this menu appears before each game starts, we should also display the scoreboard to the terminal so the players can see their scores (this is very handy if they have played multiple games already), as shown on line 223 (see figure 11).

```
220         elif the_choice == 3:
221             # to quit the game
222             print("Exiting...")
223             my_scoreboard(score_board)
224             break
225
```

Figure 11: Adding the required functionality for the 'Exit' option

If you run the game and select the third option now, it should successfully quit (see figure 12). When the user's current path shows in the terminal followed by a square cursor (the line highlighted), it indicates the previous task has finished (the game, in our case).

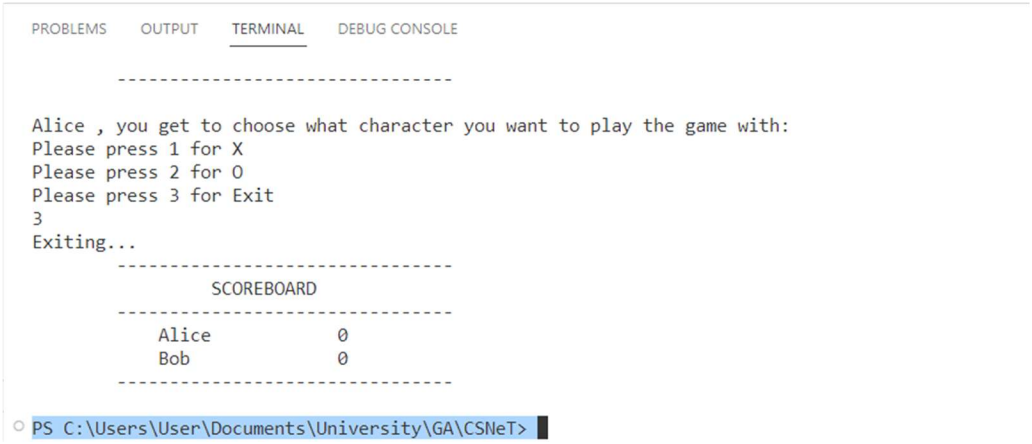


Figure 12: The terminal once the 3rd option has been selected

Take a moment to play the game and try to identify any flaws or 'buggy' actions before continuing.

The first major flaw of the game occurs when a player is selecting a position to place their character. If you enter any of the expected positions (i.e. 1 - 9), it works fine, however, when you enter any other number, the game crashes (see figure 13).

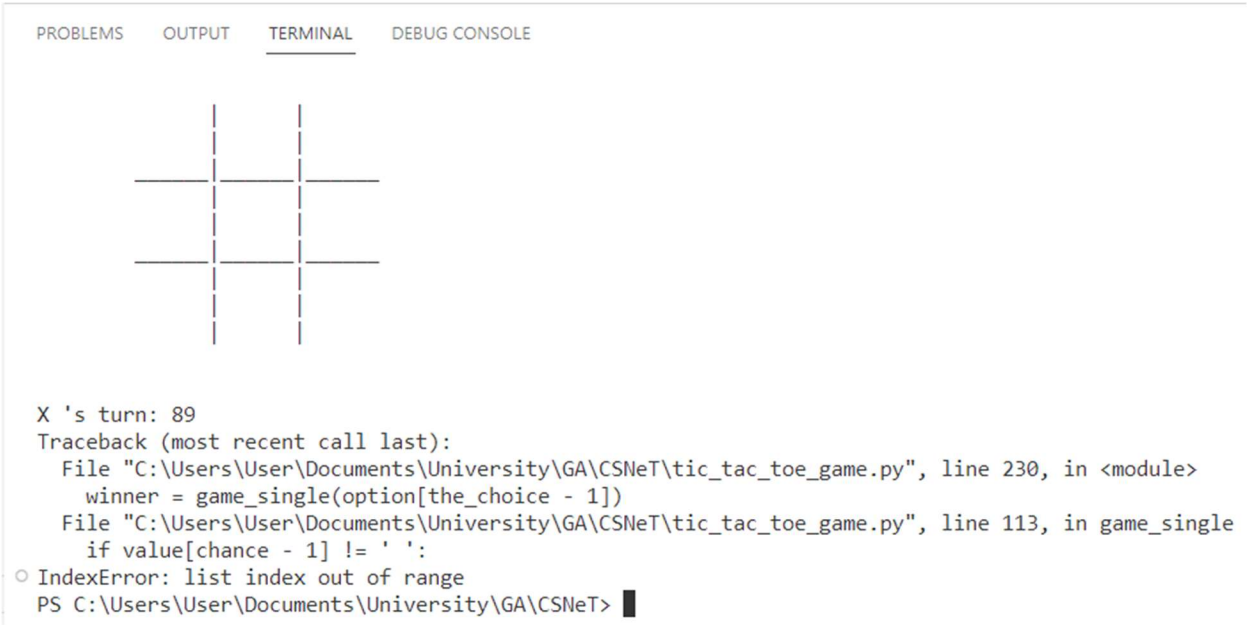


Figure 13: The error encountered when selecting an invalid position

To prevent this from happening, we need to check whether the position is valid before attempting to place a token there. This is done by adding an if statement which checks if the input is invalid (i.e. outside the bounds of 1 - 9), see figure 14.

```
111         # We need to check the input so that the user can only
112         # select positions 1 - 9
113         if chance < 1 or chance > 9:
114             ###### Add a print statement here to provide a helpful error message
115             continue
116
```

Figure 14: Adding the check for the position entered

How the `if` statement works is by performing up to three separate `boolean` evaluations (a `boolean` is a data type that only has 2 values, either `true` or `false`). Firstly, as statements are read from left to right, we confirm if `chance` (which holds the position selected) is *less than 1*. If it is less than 1, it evaluates to `true` (else it's `false` if not). Next, the logical operation is identified, and in this statement it is an `or`. This means for the `if` statement to execute (i.e. reach line 115), *1 or more* of the conditions need to evaluate to `true`. Therefore, if the first condition (`chance < 1`) is `true`, then it has satisfied the logical `or` operation so the `if` statement

executes and the second condition (`chance > 9`) is skipped. However, if the first condition was `false`, the second condition would need to be `true` for the `if` statement to execute. This means if both conditions were `true`, only the first one would need to be evaluated (and the second one skipped) for the `if` statement to execute. Hopefully it is obvious that if both conditions were `false`, all evaluations would return `false` (resulting in no execution of the `if` statement).

To inform the user of their invalid input, you should include a `print` statement with a helpful message on line 114 (do this before continuing).

You may have wondered what `continue` is for on line 115, this is used to change the flow of execution. Similar to that of a `break` statement (which breaks out of the current scope), the `continue` statement skips the remaining code in a loop's current iteration. Using this code as an example, if we're on line 115, after the `continue` executes we'll be on line 94 about to start the '`while True`' loop again.

Test the position selection, does it break when you enter '101'? (Note: It shouldn't, if it does then you have done something wrong). Does your message inform the user of the invalid input?

Now you have completed the introduction and have made some basic modifications, have a go at the following exercises to further develop the Tic Tac Toe game.

Exercises:

1. Add a welcome message when the game loads (see comments on line 3)
2. Add an informative message when the user enters a non-integer value for position selection (see comments on line 105), test to ensure it works.
3. While playing the game, try to type the same name in for player one and player two, what happens? How do you fix it? (see comments on line 160).
4. Currently only the first player can start the game, how could you change this so that each time a game is played it alternates between players? (see comments on line 244).

Summary

Today's session introduced you to Python programming through a simple CLI implementation of the game Tic Tac Toe. We explored the differences in language execution (interpretation vs compilation) and utilised various Python statements to get the game working. We touched on concepts such as the boolean data type, logical operations, controlling execution flow and input checking. Next week we will be using Python in the context of programming Micro:bits.