## 3-3 Result and Discussion in Task #3:

In this section, we will discuss the implementation of our server and client code. The overall concept of the program is simple: the server generates a random number between 0 and 100, then opens a socket to allow clients to connect and guess the number. The first client to guess the correct number wins the game.

### Server part:



Libraries that we used.



These number will be global (same for all client ).

Player\_list : store players name.

Player\_udp\_addresses : store the address for each client bound with his name.

Winner: flag use for know if there is a winner.

Winner\_name :store the winner name.

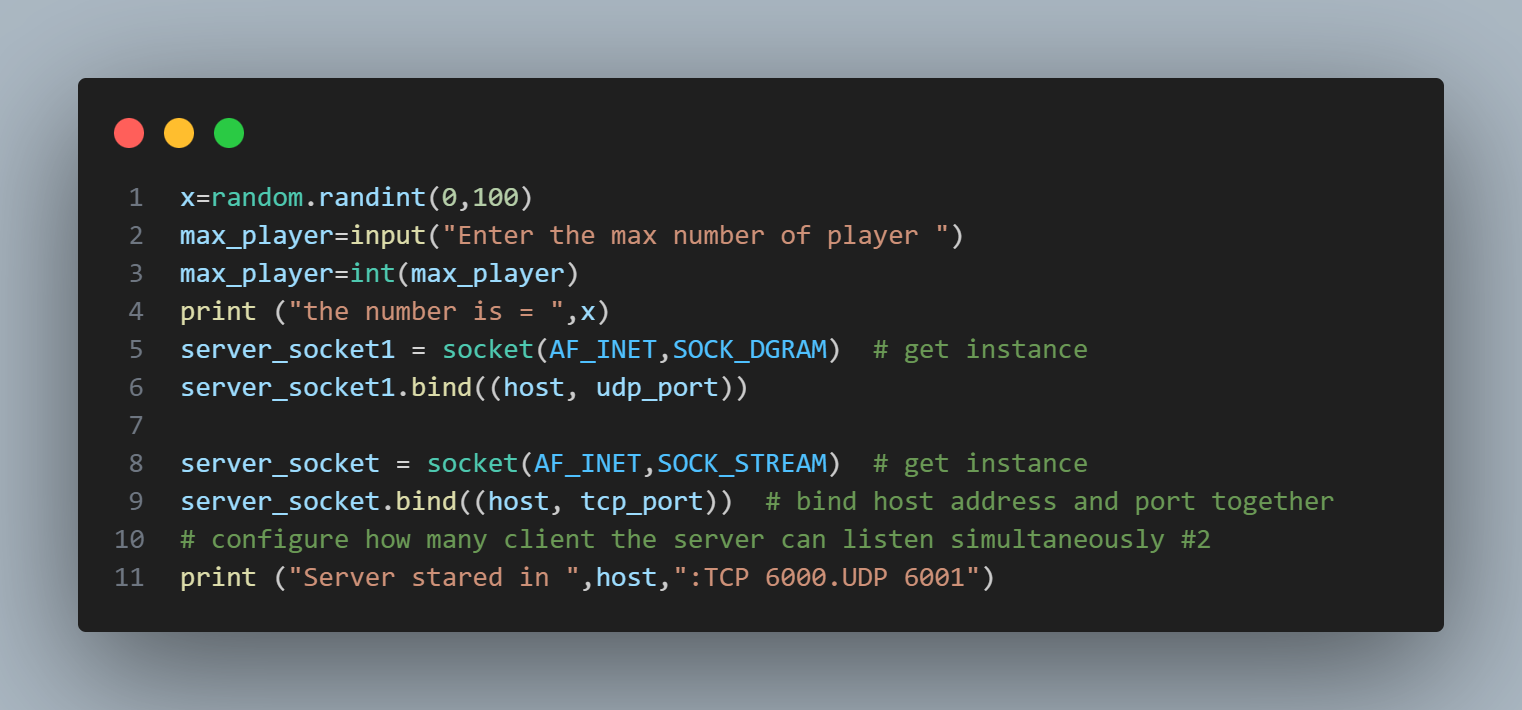
Host: store the IP of the server .

tcp\_port: constant number indicate to the TCP port number.

udp\_port: constant number indicate to UDP port number.

number\_of\_player : indicate the number of client connected with server know.

max player : indicate the maximum of player that will connect with server (game will not start until number of player == max player ).



In this implementation, we use Python's random API to generate a random number between 0 and 100. Next, we prompt the user to enter the maximum number of players that will connect to the server. Once the user specifies the number of players, the server opens both a TCP socket and a UDP socket to handle communication. The server then binds both sockets to the host’s IP address, assigning specific port numbers to each. The TCP socket is bound to port 6000, while the UDP socket is bound to port 6001.



In this part of the implementation, we use threading to handle multiple clients concurrently. The number of threads created is based on the max\_players value provided earlier. For each client that connects to the server, a new thread is spawned and assigned to handle that specific client. The thread runs the main game logic by calling the program function, passing the connection socket and the client’s address as arguments.

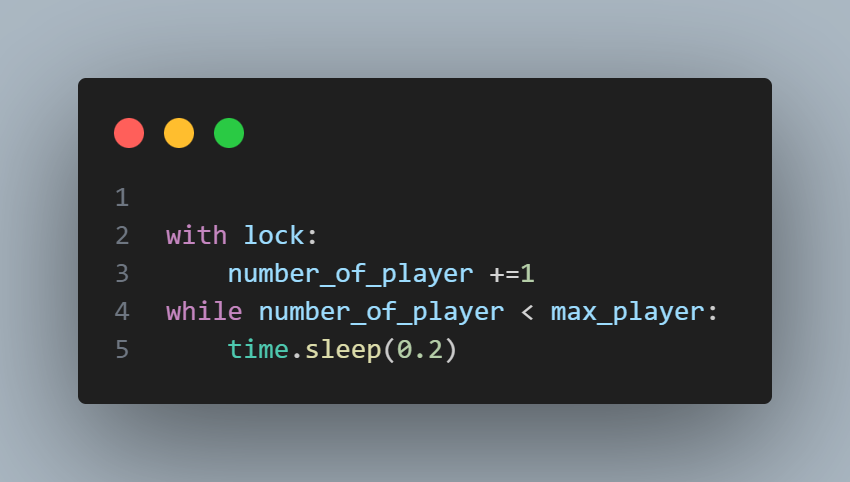


In this part of the code, we handle the process of receiving the player's name. To ensure stability, we use try and except blocks to catch any exceptions that may occur, such as if a client unexpectedly disconnects. This prevents the program from crashing or displaying errors in the terminal.

Additionally, we use a loop to handle cases where a player tries to use a name that has already been taken. If the entered name is already in use, the client is prompted to enter a new one. This logic is also wrapped in a try and except block to gracefully handle any connection issues during the name submission process.



In this part of the program function, we use the UDP socket to receive a message from the client. This message contains the client's name. Using this information, we create a mapping between each client's name and their corresponding address, which is obtained from the UDP socket. This mapping allows the server to identify and communicate with each client based on their name and network address.



This loop ensures that if the number of connected players has not yet reached the maximum allowed, each client's thread will remain in the loop, waiting until the required number of players have joined.



This section represents the main game loop, where each client attempts to guess the number initially generated by the server. At the start of the round, the server begins tracking real-time, setting a 60-second time limit for the game.

For each iteration, the server checks whether the TCP connection with the client is still active if it is, the player continues to participate; otherwise, the player is disconnected. The server then waits for the client to send a guess.

The player's guess is passed to the guess\_random\_number function, which determines whether the guess is correct and returns a corresponding result as a string. This result is then sent back to the client using the UDP socket.

Afterward, the server evaluates several conditions:

* If the **winner flag** is set to True, the server sends the winner's name to all clients via the TCP connection and ends the game.
* If the **time limit of 60 seconds** has been reached, it sends a timeout message to all players and ends the game.
* If the **number of connected players is less than or equal to one**, the server gives the remaining client the option to continue playing. If the client chooses not to, the game ends; otherwise, it continues until there is a winner or the time runs out.
* In other cases, the server gives the player another opportunity to guess after a 10-second wait.



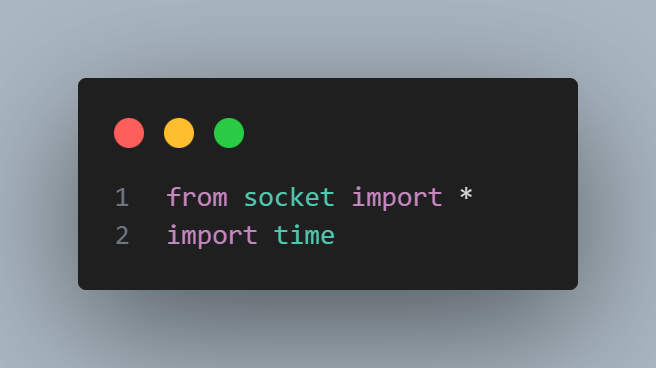
The guess\_random\_number function is responsible for determining whether the player's guess is correct. If the guess matches the randomly generated number, the function sets the winner\_flag to True provided no one has won previously and stores the winner's name in the winner\_name variable. It then returns the string "Correct", which is sent back to the client via the UDP socket as feedback.

If the player's guess is not equal to the target number, the function compares the two values:

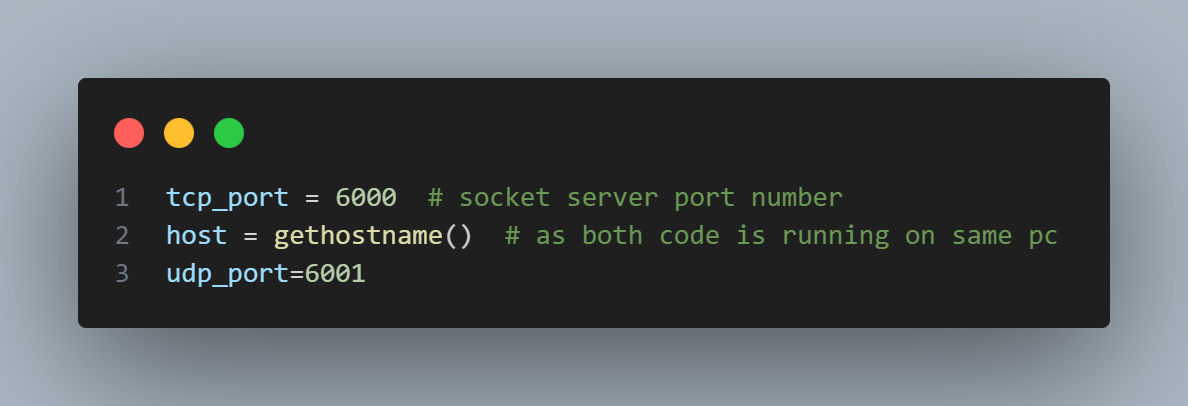
* If the guessed number is **lower** than the correct number, it returns "Higher" to indicate that the player should guess a larger number.
* If the guessed number is **higher** than the correct number, it returns "Lower" to suggest a smaller guess.

Additionally, if the input from the client is invalid such as a non-numeric character the function returns the message "Please send a valid number" to inform the player of the input error.

### Client part :



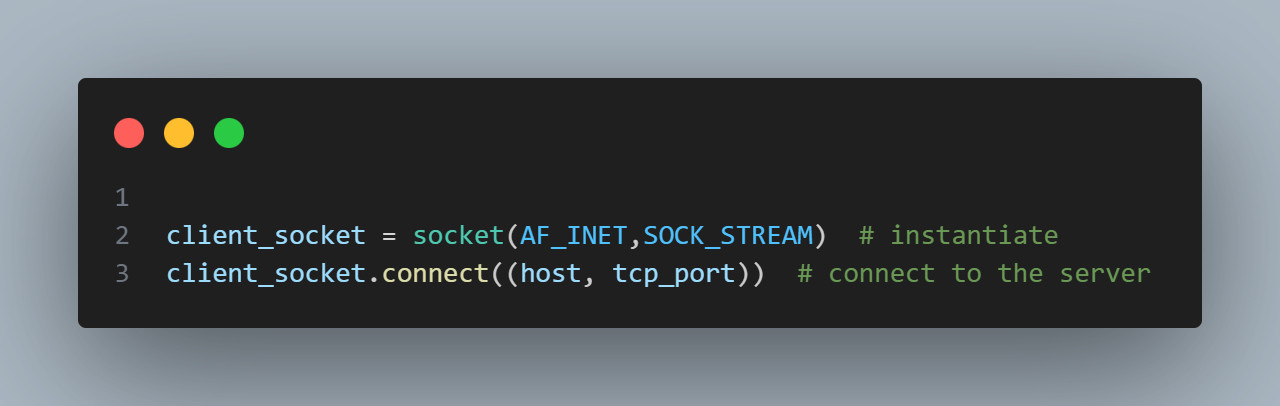
Libraries that we used.



Tcp\_port : indicate to TCP socket port number.

Udp\_port: indicate to UDP socket port number.

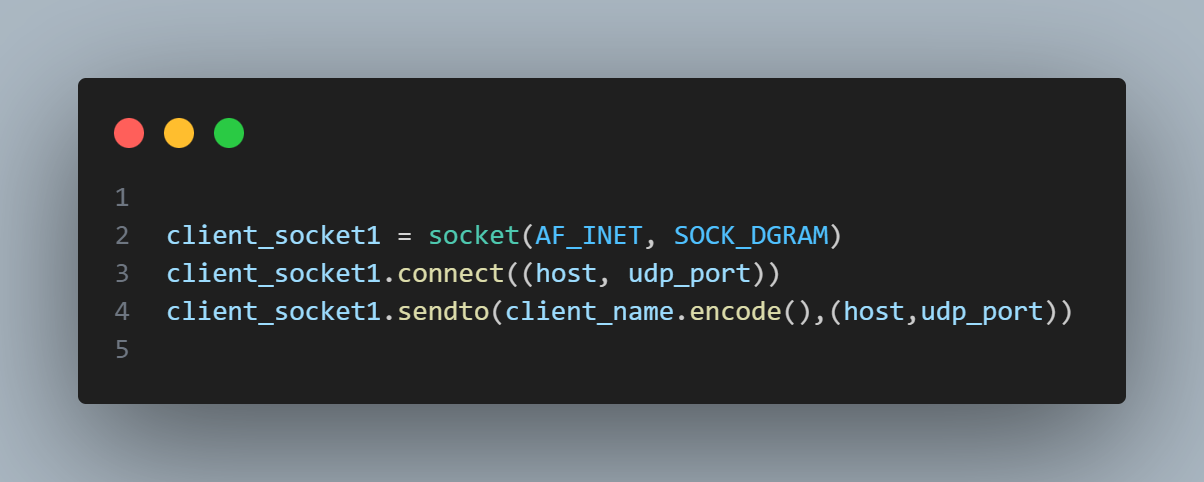
Host : store the IP of the client .



In this part we open the TCP connection between server and client(player).



In this part of the client-side code, the player is prompted to enter a gameplay name. After sending the name to the server via the TCP connection, the client waits for an acknowledgment message. If the server responds with "Enter game", the client saves the entered name in the client\_name variable and proceeds. However, if the acknowledgment is not received, it means that the chosen name is already in use by another player. In that case, the client is asked to enter a different gameplay name.



After that we open a UDP socket using the API shown above also send the client name on it to mapping between client name and his address on server part.



This section handles the main gameplay loop on the client side. At the start of each round, the client sends a message to the server via the TCP connection to indicate that it is still connected. The player then enters a guess, which is sent to the server using the UDP socket. The client then waits for a response from the server, which could be "Higher", "Lower", or "Correct"—indicating how close the guess is to the target number.

This process is repeated every round. In addition to the guess result, the server may also send other important messages:

* **"Timeout"** if the 60-second game period has ended.
* **Winner announcement**, indicating whether this client or another player has won.
* **"You are alone"**, if only one player remains connected, followed by a prompt asking whether they wish to continue or exit.

Finally, a sleep() call is used to enforce a 10-second delay between rounds, maintaining consistent pacing throughout the game.