Multiple Clients Chat Server in Python

Lior Shalom

Full Project Overview

LinkedIn: https://www.linkedin.com/in/lior-shalom-a588b325b/
Email: Liorshalom4@gmail.com

It’s Good to See You!

Welcome! To my **Chat Server** project, a real-time chat application that allows users to connect with others and have fun chatting together.

About the Project

This Python project creates a chat platform for users to chat with each other and have fun. It has two parts: the **Server** and the **Client:**

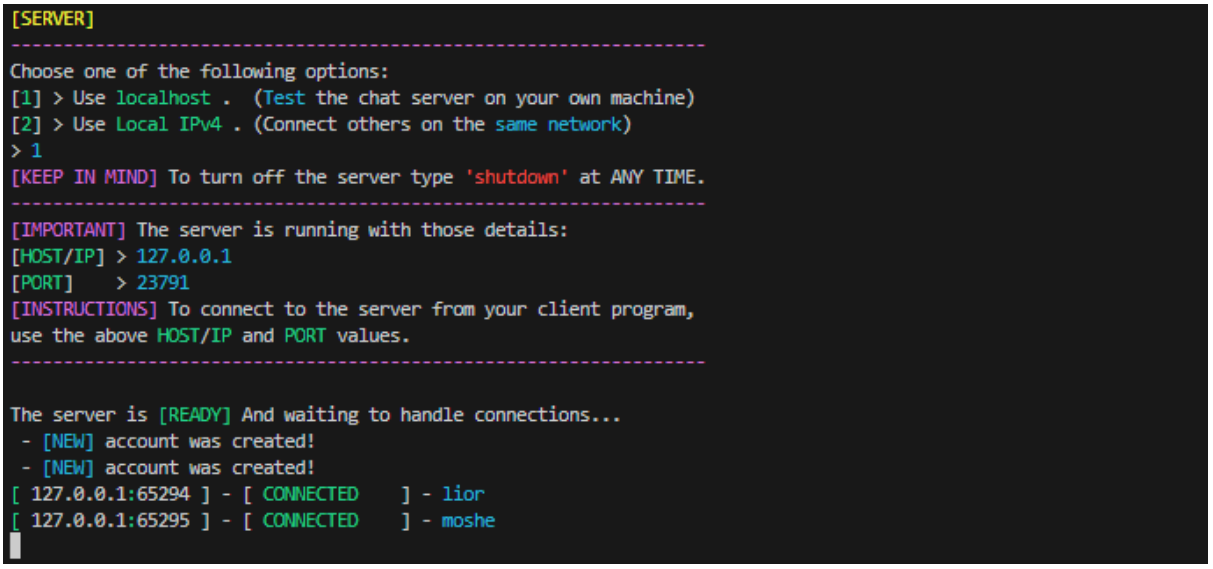
- The **Server:** The ‘heart’ of the chat room, managing all clients information, connections and ensuring smooth communication between users.
- The **Client:** What users use to chat and interact with others. Provides real-time updates on joined and left users and displays the names of each user according to the received messages.

Meet the Server

The **Server** is where all the magic happens! It handles the connections between people and keeps the chat running smoothly.

How it Works:

1. The **Server** automatically finds its **Local IPv4** address, making the setup easy. If a problem occurs, the user can provide the address manually.
2. After configuring the program, the server starts and waits for clients.
3. When a user wants to connect to the server, the user needs to input the server's connection details (host/ip and port) that appear on the server side.
4. New users need to create a new account or to log in if they’ve joined before. The server stores their details for smooth communication.
5. Once connected, you can send messages, and the server will ensure everyone in the chat sees them along with your name.
6. The server is indicated whenever a user joined, left or created an account.
7. The server is able to shut down all programs running on connected devices.

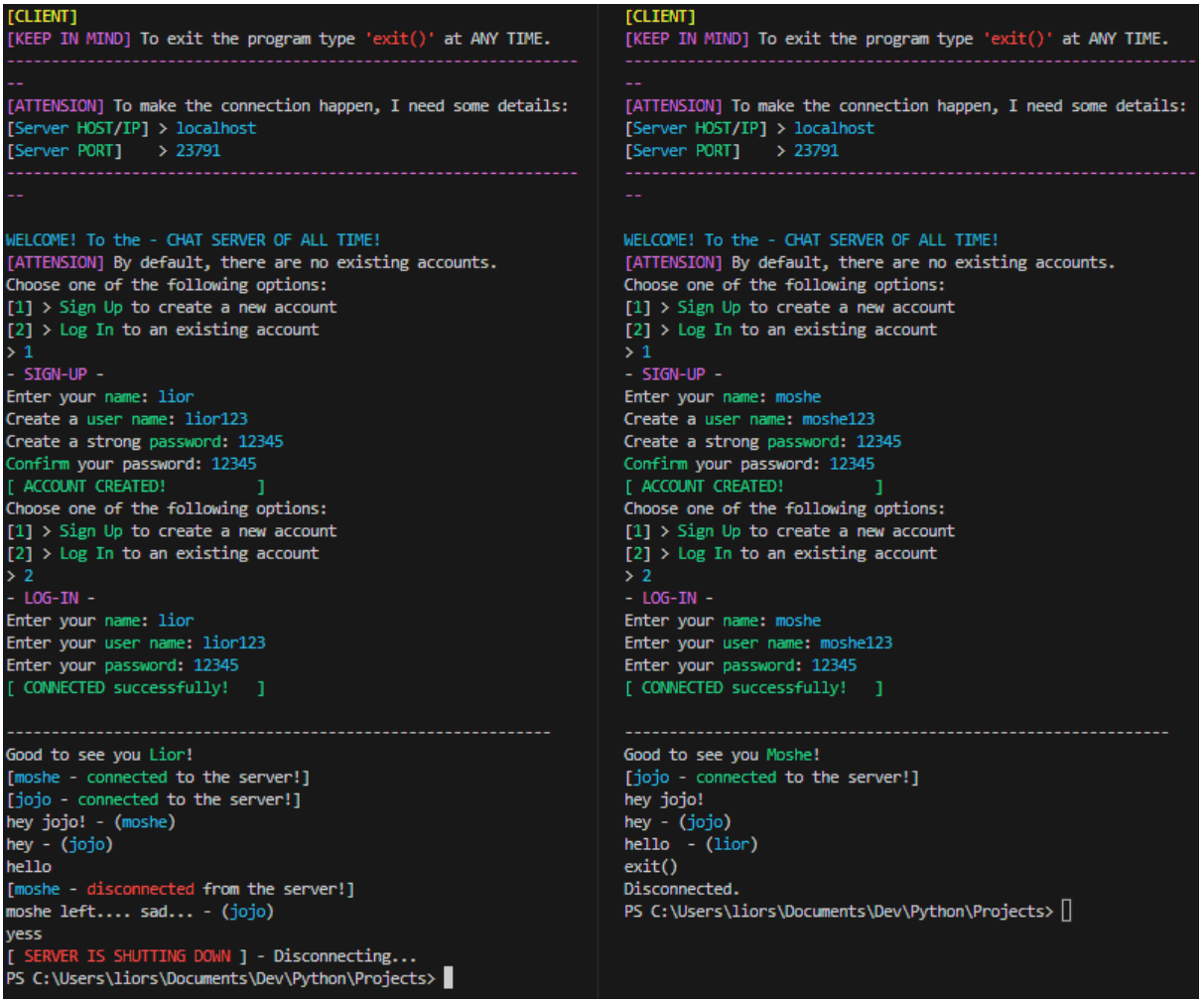


Meet the Client

The **Client** is your side of the chat. It's what you'll use to create a new account, log-in to an existing account, and to send and receive messages to see what others are saying.

How it Works:

1. You open the **Client** on your device.
2. Enter the connection details that appear on the server side.
3. If you're new and you don't have a user yet, you need to sign up to create a new user. If you've been here before, just log in.
4. Once you're in, start typing messages and hit send (enter). All the connected users will see what you write.
5. You'll be noticed in the chat whenever a new user joins or leaves the chat, And you can chat with multiple users (up to 20) at the same time and enjoy fun conversations.



In this screenshot example I execute the **'client.py'** program twice on the same computer. You can do the same, to connect to different clients using different computers that are connected to the same lan network.

How to Run the Project

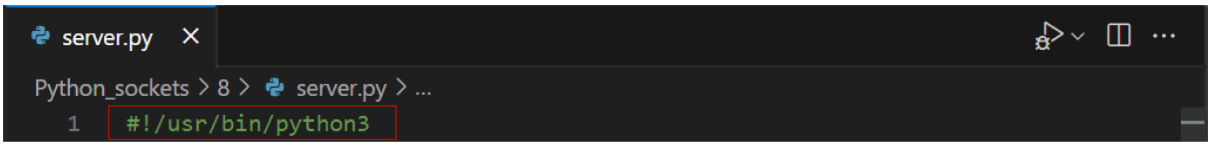
In order to enter the chat room and start chatting with others, follow these simple steps:

1. Make sure you have Python installed on your computer. (At least Python 3)
2. Install the color future by typing **'pip install colorama'** in your CMD.
3. Run the **Server** by opening the **'server.py'** file. That's where you'll see the chat room come to life!
4. To join the chat, open the **'client.py'** file on your device (make sure you're connected to the same network as the server). You can run it multiple times for different users.

I Became the Chat Hero!

In order to make your life easier, I added some configurations to the program. Some of them I mentioned before:

1. The **server** will automatically get the **Local IPv4** address that will allow others in the same network to connect.
2. The program will check and install (if needed) the module **colorama** on both **client** and **server** sides.
3. And the highlight is that I already converted both program codes into executable files (**'.exe'**).
4. For **Unix-based system** users, I added a **shebang** at the beginning of the code (Just copy the code to a text editor and run it as an executable file).



Let the Fun Begin!

That's it! I brought you all the initial information about the project and you're all set to start chatting and having a great time with your friends.
Remember to be friendly and kind to everyone in the chat room. I wish you a happy chatting, and enjoy making new connections.
Later on, I will take you deeper into the guts of the project and reveal to you what is happening behind the scenes.

Let's go Deeper

Now, let's take a closer look at how the code works its magic behind the scenes.

Server Code Overview

The **server.py** code is responsible for handling connections from clients and managing the chat room. It keeps track of the connected clients and their messages, allowing them to chat in real-time.

Dependencies:

First thing first! The server code relies on some Python modules like:

```
2   import socket
3   import select
4   import threading
```

- **'socket'** and **'select'** allow the server to handle multiple connections easily.
- The **'threading'** module provides the server the ability to input **'shutdown'** at any time to close the **server** and to force shutdown for connected clients. Below: **'shutdown_server()'** and his thread (from the main function):

```
148 # SHUTDOWN SERVER SOCKET AND ALL CLIENTS CONNECTED:
149 def shutdown_server(clients_list, server_socket):
150     while True:
151         shut = input(f"{r}")
152         if shut == 'shutdown':
153             # Notify all the clients about the server shutdown:
154             for client in clients_list:
155                 if client != server_socket:
156                     try:
157                         client.send(b'exit_server')
158                     except ConnectionResetError:
159                         pass
160             break
161         else:
162             print(f"{r}!{reset}To turn off the server type {r}'shutdown'{reset}")
163     # Close all client sockets and server socket:
164     for client_socket in clients_list:
165         try:
166             client_socket.close()
167         except OSError:
168             continue
169
170     server_socket.close()
171     exit()
```

```
194     # An option for the user to shutdown the server at any time:
195     threading.Thread(target=shutdown_server, args=(clients_list, server_socket)).start()
```

- Additionally, it uses the **'colorama'** module for colorful text output:

```
21  # COLORS:
22  from colorama import Fore, Style
23
24  r = Fore.LIGHTRED_EX        # RED
25  g = Fore.LIGHTGREEN_EX      # GREEN
26  b = Fore.LIGHTCYAN_EX       # BLUE
27  p = Fore.LIGHTMAGENTA_EX    # PURPLE
28  y = Fore.LIGHTYELLOW_EX     # YELLOW
29
30  reset = Style.RESET_ALL     # RESET
```

The Main Function - 'main()'

The **'main()'** function acts as the heart and soul of the server:

```
174  def main():
175      # An instance of the Client_info class to manage the clients information.
176      client_info = Client_info()
177
178      # Define TCP server socket:
179      server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
180
181      # Bind server to address:
182      server_socket.bind(SERVER_ADDRESS)
183
184      # Set the server to Non-Blocking:
185      server_socket.setblocking(False)
186
187      # Listen for connections:
188      server_socket.listen(20)
189      print(f"\n{reset}The server is {g}[READY]{reset} And waiting to handle connections...")
190
191      # List connected sockets:
192      clients_list = [server_socket]
193
194      # An option for the user to shutdown the server at any time:
195      threading.Thread(target=shutdown_server, args=(clients_list, server_socket)).start()
196
```

As I mentioned in the code:

1. **Client_info():** The **'main()'** function sets up the **'Client_info'** class, a lovely helper that manages and secures all the information of the clients.
2. **socket.socket():** Create a TCP socket for the server.
3. **.bind():** Bind the **server_socket** to the specified address.
4. **.setblocking():** Switches the **server_socket** to non-blocking mode, to ensure he will have the ability to manage multiple connections smoothly. By default, when creating a socket, the value in the **'.setblocking()'** function will be **'True'**.
5. **.listen():** Listening for incoming client connections (up to 20 at a time).
6. **clients_list:** Maintains a list of clients, including the server socket.
7. **threading.Thread():** A separate thread is opened to operate the shutdown function concurrently (to enable the server to shutdown at any time).

'main()' - select module

The use of the **select** module is very important to our server chat. It gives the server the ability to switch between actions if needed in real-time thanks to the **'.setblocking()'** function that is set to **'False'**.
It relies during activity on this list that specifies the readable connected sockets:

```
184      # List connected sockets:
185      clients_list = [server_socket]
```

NOTE:
This part of the program is very big, so I couldn't provide a full view of it. To view this part, you'll need to open it from the attached code files.

'main()' - select module - [Continue]

The **'select.select()'** is being indicated about **readable** sockets that are listed in the **clients_list** list.

```
190     # The 'select' module gives the server the ability to handle multiple connections in a single thread.
191     while clients_list:
192         try:
193             readable, _, _ = select.select(clients_list, [], [], 0.2)
194 
195             for sock in readable:
196                 # Accept client connections:   [SERVER]
197                 if sock == server_socket:
198                     client_socket, client_address = server_socket.accept()
199 
200                     # Add client to list:
201                     clients_list.append(client_socket)
202 
203                 # Handle client communication: [CLIENTS]
204                 else:
205                     try:
206                         # RECEIEVE:
207                         data = sock.recv(1024)
208                         # Encode data into text:
209                         text = data.decode('utf-8')
210 
211                         if text.startswith('[NEW]'):
```

Let's clear it up for you:

1. The **select.select()** waits for the **server_socket** to become readable or for a timeout (0.2 seconds).
2. If the **server** socket becomes **readable**, it means there is a new connection request. The server accepts the **client_socket** connection, adds the new **client_socket** to the **clients_list**, and continues waiting for connections.
3. If one of the **client_socket** that are listed in the **clients_list** list becomes readable, it means a client has sent data to the **server_socket**. The **server_socket** receives the data, decodes, and processes it accordingly.

Message Types

There is a different purpose for each message format the server receives:

[NEW] **(New users)** sign-up or log-in.
[ACTIVE] **(Active users)** Regular chat-room messages.
:REMOVE: **(Remove)** Remove the socket from the **clients_list**.

1. **'[NEW]:signup:name:user_name:password:client_socket':** Sent by a client when signing up for the first time. The **server** extracts the client's name, username, password, and socket to store their information for future use.
2. **'[NEW]:login:name:user_name:password:client_socket':** Sent by a client that wants to log in. The **server_socket** checks if the provided credentials match any stored client information (using the **Client_info**) and responds accordingly.
3. **'[ACTIVE]:name:user_name:password:private_sock:message':** Can be a regular message in the chat-room or an **'exit()'** request if a client wants to exit the chat. The **server** broadcasts the regular message to all connected clients except the sender or respectively will remove the client information from the **clinet_info_dict** and notifies all other clients about the disconnection.

'Client_info' Class

The **'Client_info'** class is **one of the most important factors** in the code responsible for storing and managing information about connected clients.

```python
# CLASS THAT HANDLES CLIENTS INFORMATION:
class Client_info:
    def __init__(self):
        # Dictionary to store client information:
        self.clients_info_dict = {}

    # This method is used to add client information to the 'clients_info_dict' dictionary:
    def sign_up_operation(self, name, user_name, password, client_socket):
        # Add client information to the dictionary with the 'client_socket' as a key:
        self.clients_info_dict[(user_name, password)] = (name, client_socket)
        print(f'{reset} - {b}[NEW]{reset} account was created!')

    # This method is used to check if the provided credentials match any of the stored client information:
    def log_in_operation(self, name, user_name, password, client_socket):
        if (user_name, password) in self.clients_info_dict and self.clients_info_dict[(user_name, password)] == (name, ''):
            self.clients_info_dict[(user_name, password)] = (name, client_socket)
            return True

        # If no match is found, it returns False:
        return False

    def client_exit(self, name, user_name, password, client_socket):
        # print(f"{r}exiting{reset}")
        if (user_name, password) in self.clients_info_dict and self.clients_info_dict[(user_name, password)] == (name, client_socket):
            self.clients_info_dict[(user_name, password)] = (name, '')
```

It has the following methods:

1. **sign_up_operation:** Is used to add client information to the **'clients_info_dict'** dictionary when a client signs up.
2. **log_in_operation:** This method is used to check if the provided login credentials match any of the stored client information when a client logs in.
3. **'client_exit':** When a user disconnects, this method is called in order to remove their information from the **'clients_info_dict'** (it will remove only the socket details in order to indicate that the account is in use or not).

How it Works - (Client_info)

When a new user creates a new account, four operations are performed on the client side: connect, send, receive-confirmation, and disconnect.
Accordingly, the server will: accept-client, recv-data, store-data, send-confirm.
The **server_socket** will send the details to the **Client_info** class:

Before: (the dictionary is empty)

```
> clients_info_dict: {}
```

After creating account:

```
> clients_info_dict: {('lior123', '12345'): ('lior', <socket.socket fd=91...', 53594)>)}
```

Then, the client disconnect and the method **client_exit** of the class is called to remove the **client_socket** details (replace with empty string):

After user disconnected:

```
> clients_info_dict: {('lior123', '12345'): ('lior', '')}
```

When someone wants to log in to an existing account, after the class checks if exists, the class will replace the empty string with the **client_socket** details:

After user log in:

```
> clients_info_dict: {('lior123', '12345'): ('lior', <socket.socket fd=91...', 53594)>)}
```

In this way, the server is able to know if a user is logged in to an account or not.

‘choose_address()’ Function

Choosing the server address is a very important step, and I made it so easy for you that you don't have to do anything. This function allows the user to choose the address to run the server on: the **‘localhost’ (127.0.0.1)** or the **local IPv4** address. The user input his choice and the program returns the address.

```
32  # Use 'localhost' address or Local IPv4 address:
33  def choose_address():
34      while True:
35          try:
36              print('Choose one of the following options:')
37              print(f'[{g}1{reset}] > Use {g}localhost {reset}.  ({b}Test{reset} the chat server on your own machine)')
38              print(f'[{g}2{reset}] > Use {g}Local IPv4 {reset}. (Connect others on the {b}same network{reset})')
39              choice = input(f"> {b}")
40              print(f'{reset}', end='\r')
41              if choice == '1': # Localhost
42                  return '127.0.0.1'
43              elif choice == '2': # Local IPv4
44                  return get_local_ip()
45              elif choice == 'exit()':
46                  raise SystemExit
47              else:
48                  print("Invalid input! Please try again...")
49          except SystemExit:
50              print(f"{reset}Sad to see you leaving...")
51              exit()
```

‘get_local_ip()’ Function

This function is pretty smart! It does its best to automatically get the **local IPv4** address of the device where the server is running on by using a temporary UDP socket and connecting it to a known IP address (8.8.8.8) on port (80).
This operation requires the operating system to assign a **local IPv4** address to the socket, which is then retrieved as the **local IPv4** address of the device.
If there is an error, the function will give the user to input the address manually.

```
53  # Get the local IP address of the device you running the server on:
54  import ipaddress
55
56  def get_local_ip():
57      try:
58          # Get the local IP address automatically:
59          # Create a temporary UDP socket to get the desired IP address:
60          temp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
61          # connect to a known IP address:
62          temp_socket.connect(('8.8.8.8', 80))
63          # To get the local IP address to which the temp socket is bound:
64          local_ip = temp_socket.getsockname()[0]
65          return local_ip
66      except:
67          # Get the IP address manually from the user:
68          print(f"{r}[ERROR]{reset} I didn't managed to get your local IP address by myself.")
69          print("I need your help. Check it manually.")
```

The **‘choose_address()’** function returns the returned value of the **‘get_local_ip’** function and stores it in the **server_ip** variable.

```
 99    server_ip = choose_address()
100    server_port = 23791
```

SERVER_ADDRESS will store a tuple of the **chat server** connection details.

```
109    # Server address definition:
110    SERVER_ADDRESS = (server_ip, server_port)
```

Client Code Overview

Welcome to the client side to our amazing chat server! The **client.py** code is responsible for connecting the user (you, if you wish) to the **server** and enables smooth communication with others. Let's go through it together!

Dependencies:

The client code relies on some Python modules:

```
2   import socket
3   import select
4   import threading
5   import os # Used the 'os.exit(0)' to force close of the entire program.
```

Similar to the **server**, the **client** also requires the **'colorama'** module that makes the terminal look better. The **'check_module'** function ensures it is installed just like in the server code.

```
 8   # To save time - ensure the 'colorama' module is installed:
 9   import subprocess
10
11   def check_module(module_name):
12       try:
13           # Checks if the 'colorama' module is installed:
14           __import__(module_name)
15       except ImportError:
16           # Installs the module:
17           subprocess.check_call(['pip', 'install', module_name])
18
19   module_name = 'colorama'
20   check_module(module_name)
```

The Main Function - 'main()'

This is the main function of the **client.py** program. It calls the **'welcome()'** function to handle **sign-up** or **log-in**, and after the authentication was successful, it starts the sending and receiving threads.

```
360  def main(): # The socket definition and the first steps of the connection are in the log_in and sign_up functions.
361      try:
362          # Welcome / log_in / sign_up:
363          name, user_name, password, client_socket = welcome()
364          if client_socket == False:
365              raise SystemExit
366          # CREATE thread for each operation to handle both at same time:
367          _recv = threading.Thread(target=(receive_data), args=(client_socket,))
368          _send = threading.Thread(target=(send_data), args=(name, user_name, password, client_socket))
369          # START the threading:
370          _recv.start()
371          _send.start()
372          # WAIT for both threading to complete:
373          _send.join()
374          _recv.join()
375          # CLOSE:
376          client_socket.close()
377      except SystemExit:
378          print("Exit program...")
379          exit()
```

‘configuration()’ Function

This is the first step when running the user side. This function enables you to configure the server’s **IP** address and **port** before connecting to the server. It asks the user for the above details and returns it as a tuple.

```
89          # Connection details (server address):
90          return (server_ip, server_port)
```

```
97              SERVER_ADDRESS = configuration()
```

In the **‘configuration()’** function I also imported the **‘ipaddress’** that will check if the **IP** address is valid or not.

```
37      # Get the servers IP ADDRESS manually from the user:
38      import ipaddress
39
40      while True:
41          try:
42              server_ip = input(f"[{b}Server{reset} {g}HOST{reset}/{g}IP{reset}] > {b}")
43              print(reset, end='\r') # Reset color
44
45              # Check if 'exit()':
46              if server_ip == 'exit()':
47                  raise SystemExit
48
49              # Check if 'localhost':
50              elif server_ip == 'localhost':
51                  break
52
53              # Check if valid:
54              ipaddress.IPv4Address(server_ip)
55              break
56
57          except ipaddress.AddressValueError:
58              print(f"{r}[INVALID]{reset} Please anter a valid IP address.")
59          except SystemExit:
60              print("Closing program...")
61              exit()
```

‘welcome()’ Function

This function acts as the entry point for the client program. It provides you the option to choose between **sign-up** and **log-in** (if this is your first time).

```
324 # WELCOME:
325 def welcome():
326     try:
327         print(f'\n{b}WELCOME! To the - CHAT SERVER OF ALL TIME!{reset}')
328         print(f'{p}[ATTENSION]{reset} By default, there are no existing accounts.')
329
330         while True:
331             print('Choose one of the following options:')
332             print(f'[{g}1{reset}] >{g} Sign Up {reset}to create a new account')
333             print(f'[{g}2{reset}] >{g} Log In {reset}to an existing account')
334             choice = input(f"> {b}")
335             print(f'{reset}', end='\r')
336             if choice == '1': # sign up
337                 client_socket = sign_up()
338                 if not client_socket:
339                     raise SystemExit
340
341             elif choice == '2': # log in
342                 name, user_name, password, client_socket = log_in()
343                 if not client_socket:
344                     raise SystemExit
345                 return name, user_name, password, client_socket
346
347             elif choice == 'exit()':
348                 raise SystemExit
349
350             else:
351                 print("Invalid input! Please try again...")
```

‘sign_up()’ Function

If you choose to create a new account, the **‘sign_up()’** function will provide you the ability to do that. It will ask you to provide your name, user_name, password, and to type the password again to confirm it. If all the input is valid, it sends the details to the server for account creation.

```python
# SIGN-UP
def sign_up():
    try:
        while True:
            print(f'- {p}SIGN-UP{reset} -')
            # Define client:
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

            # ENTER:
            # NAME:
            name = input(f"{reset}Enter your {g}name{reset}: {b}")
            print(f'{reset}', end='\r')
            if name == 'exit()':
                raise SystemExit
            elif ':' in name:
                print(f"{r}[INVALID]{reset} Invalid details.\nTry again...")
                continue

            # USER-NAME
            user_name = input(F"{reset}Create a {g}user name{reset}: {b}")
            print(f'{reset}', end='\r')
            if user_name == 'exit()':
                raise SystemExit
            elif ':' in user_name:
                print(f"{r}[INVALID]{reset} Invalid details.\nTry again...")
                continue
            elif len(user_name) < 4:
                print(f"{r}[INVALID]{reset} User-Name is too short! Must contain at least 4 letters.\nTry again...")
                continue

            # PASSWORD:
            while True:
```

As we covered before:

(‘[NEW]:signup:name:user_name:password:client_socket’)

```python
            # SEND NEW CLIENT DATA TO SERVER:
            print(f"{r}[{reset} Creating account...       ]", end='\r')
            # Send new details to server
            sign_up_data = f'[NEW]:signup:{name}:{user_name}:{password}:{client_socket}'

            print(f"{r}[{reset} Sending data to server... ]", end='\r')
            # Add client information to the client:
            client_socket.connect(SERVER_ADDRESS)
            client_socket.send(sign_up_data.encode('utf-8'))

            created = client_socket.recv(1024).decode('utf-8')
            if not created:
                print(f"{r}[ERROR]{reset} An error occurred! Account not created.")
                raise SystemExit
            else:
                print(f"{g}[ ACCOUNT CREATED!          ]{reset}")
                client_socket.close()
                return True
```

This text will be received at the server, which will use the **‘sign_up_operation()’** method of the **‘Client_info’** class to create the desired account. Then, a confirmation text will be sent back to you (the user) and then, an indication banner will pop on the server side.

'log_in()' Function

This function will handle the process of connecting a user to an existing account. It will ask you to provide your name, user_name, and your password. Then, the **client_socket** will send your login details to the **server_socket** that will use the **'log_in_operation()'** method in order to authenticate your login details.

```
283             # CHECK:
284             if ':' in name or ':' in user_name or ':' in password:
285                 print(f"{r}[INVALID]{reset} Invalid details.\nTry again...")
286                 continue
287 
288             print(f"{r}[{reset} Checking login data...    ]", end='\r')
289             # Check details at server database:
290             log_in_data = f'[NEW]:login:{name}:{user_name}:{password}:{client_socket}'
291 
292             print(f"{r}[{reset} Sending data to server... ]", end='\r')
293             # Add client information to the client:
294             client_socket.connect(SERVER_ADDRESS)
295             client_socket.send(log_in_data.encode('utf-8'))
296 
297             log_in_success = client_socket.recv(1024).decode('utf-8')
298 
299             if log_in_success == 'True':
300                 print(f"{g}[ CONNECTED successfully!   ]{reset}")
301                 print('\n------------------------------------------------------------')
302                 print(f"{reset}Good to see you {g}{name.capitalize()}{reset}!                                ")
303                 return name, user_name, password, client_socket
304 
305             elif log_in_success == 'False':
306                 print("                                                  ", end='\r')
307                 print(f"{r}ERROR!{reset}")
308                 print("Make sure that the account is not in use on another device and the details you've entered are correct.")
309                 client_socket.close()
310                 continue
```

If the returned value of **'log_in_operation()'** is **'True'**, the server will send a permission back to the **client_socket** that will return all your account data back to the **'welcome()'** function that will then, return the data to the **'main()'** function that will use it in the **'send_data()'** and **'receive_data()'** threads.

```
357 def main(): # The socket definition and the first steps of the connection are in the log_in and sign_up functions.
358     try:
359         # Welcome / log_in / sign_up:
360         name, user_name, password, client_socket = welcome()
361         if client_socket == False:
362             raise SystemExit
363         # CREATE thread for each operation to handle both at same time:
364         _recv = threading.Thread(target=(receive_data), args=(client_socket,))
365         _send = threading.Thread(target=(send_data), args=(name, user_name, password, client_socket))
```

If the login succeeds, the **'log_in()'** will leave the **client_socket** opened, or closed vice versa.

As you can see in the above last code snippet from the **'main()'** function, if the **client_socket** is **'False'**, it means it's closed and will raise the **SystemExit.**

```
374     except SystemExit:
375         print("Exit program...")
376         exit()
```

Multi-threading for Communication

We have reached one of the most important parts of our client project. Without it, we could not send and receive messages in the chat room.
To make it happen smoothly, I created two different functions that will operate as two different threads simultaneously: **'receive_data()'** and **'send_data()'**.

'receive_data()' Function

This function as I mentioned above is running as a separate thread to continuously receive data from the server. It listens for incoming messages and displays them in the chat room.

```python
# RECEIVE:
def receive_data(client_socket):
    while True:
        try:
            # Receive encoded data from the user:
            data = client_socket.recv(1024)
            # Checks if there is no data - server closed the conncetion:
            if not data:
                print(f"{reset}[ {r}SERVER DISCONNECTED{reset} ]")
                break

            # Decode the data into a readable text:
            message = data.decode('utf-8')
            # Check if the message is 'exit_server'. if so, the server is shuting down.
            if message == 'exit_server':
                print(f"{reset}[ {r}SERVER IS SHUTTING DOWN{reset} ] - Disconnecting...")
                # Fore exit the entire program:
                os._exit(0)
                break
            # PRINT received message:
            print(f"{message}")
        except (ConnectionAbortedError, ConnectionResetError):
            break
    client_socket.close()
```

'send_data()' Function

To the side of the **'receive_data()'**, this function will continuously prompt your input and will send it to the server. It allows you to send messages to other clients via the server.

```python
# SEND:
def send_data(name, user_name, password, client_socket):
    while True:
        # Wait for input from the user:
        print(f"{Lb}>", end='\r')
        message = input(f"{reset}")
        if ':' in message:
            print("The use of ':' is not allowed!")
            continue
        elif message == '' or message == None:
            continue

        # Encode the message in order to send the binary data to the server:
        complete_message = f'[ACTIVE]:{name}:{user_name}:{password}:{client_socket}:{message}'
        data = complete_message.encode('utf-8')

        # Send the message to the server:
        client_socket.send(data)
```

We've Reached the End!

Congratulations! You've reached the end of this amazing project overview.
I hope you enjoyed it and that the content was interesting.
You've taken an exciting journey through the world of real-time messaging with this **Chat-Server Python project**.