Wazzap: A Chat Application

Core Idea

A simple chatroom is, at its heart, a place where people can type messages and see what others say in real time, sharing one shared stream of conversation. In this project, that place is not a web page or a mobile app, but the terminal itself. Every person runs a small Python program called a client, and all of those clients talk to another Python program called the server. The server sits in the middle, quietly listening. Whenever one client sends a message, the server takes that message and delivers it to every other connected client. To each user, it feels like everyone is sitting in the same room, even though they are really just sending lines of text over the network.

The core idea of this application is very simple: many clients, one server, one shared conversation. There are no private rooms, no message history, no usernames stored in a database. A user connects, types something, sees what others type, and can leave at any time. This simplicity is helpful. It allows us to focus on understanding how network communication works, how data moves between machines, and how to manage several users at once without making the program too complicated. Even with this small feature set, the chatroom already feels alive: when two or more terminals start sending messages and you see them appear in real time, you can immediately see that your code is doing something real.

Features of the Chat Application

The chat application is designed to show how real-time communication works across a network in a simple, terminal-based environment. Even though it runs entirely in the console, it includes several useful and enjoyable features that make chatting interactive and easy to follow. Each feature is created to teach an important concept about networking or user interaction while keeping the code clear and readable. Together, these features turn a simple text exchange into a dynamic shared conversation.

- **Multiple users in one chatroom:**
  The chatroom allows many users to connect at the same time. Each person runs a small client program that connects to the server. The server keeps all these connections open and manages the flow of messages between them. Whenever one user sends a message, it is instantly shared with everyone else. This gives the feeling of being in one shared space, even though each person may be on a different computer.
- **Nicknames for identification:**
  When a user joins the chatroom, they are asked to choose a nickname. This nickname becomes their identity in the chat. Every message they send appears under their chosen name, allowing others to easily recognize who is speaking. This makes the chat more personal and prevents confusion when several users are active at once.

- **Colored text display:**
  To make the conversation easier to read, each user is automatically assigned a random text color when they join. Their nickname is displayed in that color whenever they send a message. System notifications, such as users joining or leaving, are displayed in a different color so they stand out clearly. This color coding brings structure to the chat and adds personality to the otherwise plain terminal text.
- **Private messaging:**
  Users can send private messages to one another by using a simple message format. By typing the “@” symbol followed by the recipient’s nickname, the message is delivered only to that specific user instead of everyone in the chatroom. This feature makes the chatroom feel more complete, as it supports both group discussions and one-to-one communication.
- **Server announcements:**
  The server automatically sends system messages when users join or leave the chat. These messages appear with a special “[Server]” tag and a unique color. They help everyone keep track of who is currently in the room and maintain a clear picture of what is happening.
- **Real-time message handling:**
  Messages are delivered instantly to all participants. The chatroom uses network sockets and non-blocking input to make sure there are no noticeable delays. The conversation feels smooth and immediate, just like any modern chat application, even though it runs in the terminal.
- **Clean and simple design:**
  The code is written to be as easy to understand as possible. Each module, function, and loop serves a clear purpose, and all actions—sending, receiving, joining, and leaving—are handled step by step. This design makes it a perfect project for learning how real-time communication and networking work in Python.
- **Cross-platform color support:**
  Thanks to the Colorama module, the colored output works the same on Windows, macOS, and Linux. This ensures that everyone sees the same clean and consistent display, regardless of which operating system they use.
- **Graceful connection handling:**
  Both the server and the client close their connections cleanly when the user quits or when a problem occurs. This prevents crashes and makes restarting the chatroom simple. The server even cleans up disconnected users automatically to keep everything stable.

Required Modules

To make this work in Python, we rely on some important modules: **socket**, **select**, **sys, colorama,** and **random**. Each of them solves a specific problem that appears as soon as you try to move beyond a single program talking only to itself.

Socket module

The *socket* module is the bridge between the Python code and the network. Without it, the program would have no direct way to send information to another machine across the internet or even across the local computer. A “socket” is like a virtual plug point where data can flow in and out. On the server side, the socket is created, binded to a specific address and port, and listens for new connections. On the client side, another socket is created and connected to the server’s address and port. Once the connection is made, both sides can send and receive bytes through this link. This is what turns a simple line of text inside your program into something that can travel to someone else’s screen. The *socket* module hides many of the low-level details of networking and gives you a clear set of functions to open connections, accept clients, and exchange messages. It is the foundation of the chatroom.

Select module

When only one client is connected, life is easy: the server can read from that one socket, then write to it, and everything is fine. The problem appears the moment there are several clients at once. The server now has many sockets open, one for each client, and it must be ready to read data from any of them whenever a user sends a message. It cannot simply “block” and wait forever on one client, because then it would ignore everyone else. This is where the ***select*** module becomes important. The ***select*** module allows the server to watch several sockets at the same time and ask a simple question: “Which of these are ready right now?” Instead of getting stuck waiting on one connection, the server uses ***select*** to get a list of all sockets that have data waiting. It can then quickly loop over them, read the new messages, and broadcast them. This technique is called I/O multiplexing. It lets us build a responsive multi-user chat without using multiple threads or complex frameworks, and it keeps the code small and easier to understand.

sys module

The ***sys*** module plays a quieter but still useful role. It connects your Python program to the environment it is running in. Through ***sys.argv***, you can read command-line arguments, such as the server address or port number, so that you do not have to hard-code them inside the script. Through ***sys.stdin*** and ***sys.stdout***, your program can work directly with input and output streams. In a terminal chat client, we often need to read what the user types and send it over the network while also listening for incoming messages from the server. By combining sys.stdin with select, the client can check both: if the user has typed something, it reads from standard input and sends the message; if the server has sent something, it reads from the socket and prints the message. The sys module makes this coordination between the user, the terminal, and the network smooth and flexible.

Together, these three modules support the main features of our simple chatroom. The server can accept many clients, handle all of them from a single loop, and forward messages to everyone. Each client can type messages and see others' messages appear almost immediately in the same terminal window. There is no complex interface, no graphics, and no extra decoration. This is deliberate. By stripping away everything unnecessary, the project lets you see clearly how real-time communication is built: open sockets, check who is talking, move bytes from one place to another, and display them as readable text.

Once this foundation is solid, it becomes easy to imagine extensions. You can add nicknames, timestamps, colored output, simple commands, or even private messages. But those are steps for later. For now, the goal is a clean, working, many-to-one-to-many conversation loop. With socket to connect, select to manage many connections at once, and sys to talk to the terminal and read arguments, you have everything you need to bring this small chatroom to life.

Colorama module

In a terminal application, text normally appears in one plain color, which makes it difficult to tell who is speaking or which messages come from the server. The colorama module makes it possible to print colored and styled text on any operating system, including Windows, macOS, and Linux. It works by providing simple color codes that can be added around parts of a string. When the terminal reads those codes, it displays the enclosed text in a different color. The module is also careful to reset the color after each line, so the terminal does not remain colored after a message has been printed. With colorama, nicknames can appear in one color, server messages in another, and private messages in yet another. This creates a clearer and more pleasant reading experience. The module is initialized at the start of both the client and server programs so that the color codes work properly everywhere. Without colorama, the colored output would either fail or look broken on some systems, but with it, the color formatting becomes consistent and reliable.

random module

This module provides the ability to make simple random choices, which can be used for a variety of small tasks. In the chatroom, it is used to assign a color to each user automatically when they join. A list of colors is prepared, and the random module is asked to pick one color from that list. Because the choice is random, each user is likely to receive a different color. This helps make the conversation visually distinct, even when many people are talking at once. The random module is simple but very helpful here. It saves the need for the user to choose a color manually and ensures that the server can easily provide unique visual identifiers without complex logic.

Understanding `server.py`

At the top of the file, the server imports several modules that make its features work smoothly. The `socket` module gives the program access to the network so it can send and receive messages. The `select` module allows the server to handle multiple clients at the same time without freezing or blocking on one connection. The `sys` module is used to manage command-line arguments and to exit cleanly when needed. The `colorama` module is used to print colored text in the terminal, allowing nicknames, server messages, and system notices to appear in different colors for better readability. The `random` module helps assign a random color to each new user when they join, so their messages can be recognized easily in the conversation.

After the imports, the `colorama` module is initialized with a simple call to make sure colors display correctly on every operating system. A short list of colors is defined, such as green, yellow, blue, and magenta. These colors will be used to make each user's nickname stand out. A dictionary named `client_colors` is created to store which color belongs to which client. Another dictionary called `nick_to_socket` is also created, which will later help the server find a user's connection when a private message is sent.

The next important function is `broadcast`. It is defined with four parameters: the message to send, the socket of the sender, the list of all sockets, and the dictionary of connected clients. The purpose of this function is to send a message to everyone in the chatroom except the sender and the server itself. Inside the function, there is a simple loop over all sockets being watched. For each socket, the server checks that it is not the sender and not the main server socket. If those conditions are met, the server tries to send the message. If sending fails, it means that client is no longer connected. In that case, the socket is closed and removed from the list and from the clients' dictionary. The broadcast function keeps the main loop simple by handling all the details of message delivery in one place. Before the server starts running, it checks whether the user has given the correct command-line argument. When a Python program is launched from the terminal, everything typed after the program's name is stored in a list called `sys.argv`. The first item in this list is always the program name itself, and the following items are any arguments the user provided. The server expects exactly one extra argument: the port number. To verify this, the program checks the length of `sys.argv`. If the length is not two, it means the user forgot to type the port number or typed something extra. When that happens, the program prints a helpful message explaining the correct way to start the server, for example, "Usage: python server.py <port>", and then exits gracefully using `sys.exit(1)`. The number one means the program stopped because of a setup problem. If the correct number of arguments is given, the port number is read from `sys.argv[1]`, converted to an integer, and stored in a variable named `PORT`. This is the port where the server will listen for new clients.

Next, the main server socket is created. The command that builds it tells Python to use an IPv4 TCP connection, which keeps a reliable link open between the server and each client. To make restarting the server easier, an option is set so the same port can be reused immediately after a shutdown. The socket is then bound to the chosen port and instructed to start listening for incoming connections. A message is printed to the terminal to show that the server is running and waiting for clients.

The server needs to keep track of several connections at once, so it creates a list called `sockets_list`. At the beginning, this list only contains the server's listening socket, but every new client connection will be added to it later. It also creates two dictionaries: `clients`, which connects each socket to its nickname, and `nick_to_socket`, which connects each nickname to its socket. These dictionaries make it easy for the server to look up users when sending messages.

The core of the server lives inside a `while True` loop that runs continuously as long as the server is active. Inside this loop, the `select` function is called with the list of sockets to watch. This command tells the operating system, "Wait until one or more of these sockets has something to read or has an error." The function returns two important lists: one for sockets ready to read and one for sockets that have errors.

When the main server socket appears in the readable list, it means a new client wants to join. The server accepts this connection, creating a new socket for that client. Immediately after the connection is accepted, the server expects the client to send its chosen nickname. That nickname is received, cleaned up to remove extra spaces, and stored in the clients' dictionary. A random color is selected from the color list and assigned to this new client in the client_colors dictionary. The nickname and color are also linked in the nick_to_socket mapping. Then the server prints a message in a bright color showing that this user has joined the chatroom and uses the broadcast function to tell everyone else.

When one of the connected clients appears in the readable list, it means that client has sent a message or closed the connection. The server tries to read from the socket. If the message is empty, it means the client has disconnected. The server looks up the nickname of that client, removes it from the active lists and dictionaries, closes the socket, and broadcasts a message to let others know that the user has left.

If the message contains text, the server first decodes it from bytes into a readable string. It then checks whether this message is intended as a private message or a public one. A private message is recognized when the text starts with an at symbol (@) followed by a nickname. The server splits the message into two parts: the target nickname and the actual message text. It uses the nick_to_socket dictionary to find the socket belonging to that nickname. If the target is found, the server sends the private message only to that user and also sends a confirmation back to the sender. If the nickname does not exist, the server notifies the sender that the target was not found.

When the message does not begin with the @ symbol, it is treated as a public chat message. The server retrieves the sender's nickname and color, builds a formatted string with the colored nickname and message text, prints it in its own terminal, and broadcasts it to everyone else. This makes all users see the colored name of the speaker followed by the message.

Any sockets that appear in the error list are handled carefully: the server removes them from all tracking structures and closes them to avoid keeping broken connections. The main loop continues running, ready to handle new messages or new clients at any moment.

The entire loop is wrapped in a try-except block. If the server operator presses Ctrl+C to stop the program, the KeyboardInterrupt exception is caught. The server then prints a short shutdown message, closes all sockets, and exits cleanly. This ensures that all resources are released properly and that the server can be restarted without any leftover issues.

In this updated version of the server, every user can be recognized by a nickname, every message is displayed with clear color formatting, and users can send private messages directly to each other. These features make the chatroom easier to follow, more personal, and closer to the behavior of a real online chat service, while the underlying structure of the program remains clear and easy to understand.

Understanding `client.py`

The client script acts as the doorway through which users enter the chatroom. While it is smaller and simpler than the server, it performs one important task that makes real-time chatting possible: it listens for incoming messages from the server while also waiting for the user to type new messages. To make this work smoothly, the client uses the same fundamental modules as the server—`socket`, `select`, and `sys`—along with the additional `colorama` module to support colored text in the terminal. Each of these modules plays a specific role that allows the client to communicate easily and display messages clearly.

The `socket` module is used for making the actual connection between the client and the server. It provides the tools needed to send and receive data over the network. The `select` module allows the client to pay attention to two things at once: what the user types and what the server sends back. Without it, the client would have to wait for one action to finish before handling the other, making the chat unresponsive. The `sys` module manages command-line arguments, which specify the address and port of the server that the client should connect to. Finally, the `colorama` module ensures that any colored or styled text sent from the server is displayed correctly on the user's terminal, no matter which operating system they are using.

When the program starts, it checks whether the user has provided the correct information to connect to the chatroom. The client needs two pieces of information: the host address and the port number of the server. When the program is run from the terminal, anything typed after the script name is stored in a list called `sys.argv`. The first element of this list is always the program name itself, while the following elements are the arguments entered by the user. The line that checks `len(sys.argv) != 3` simply verifies that there are three items in the list—the program name, the host, and the port. If this condition is not met, it means the user has forgotten something. When that happens, the program prints a short message explaining how to run it properly, such as "Usage: python client.py <host> <port>." After displaying this reminder, the program stops using `sys.exit(1)` to avoid running without the required information. If the correct input is given, the host and port are read from `sys.argv[1]` and `sys.argv[2]` and stored in the variables `HOST` and `PORT`. The client now knows exactly where to connect.

Next, the client creates its network socket using `socket.AF_INET` and `socket.SOCK_STREAM`. This setup means it will use IPv4 and the TCP protocol, which provides reliable communication between the client and server. Inside a `try` block, the client attempts to connect to the server using these two pieces of information. If the server is offline, unreachable, or the address is incorrect, an error is displayed, and the client exits gracefully. If the connection succeeds, a short welcome message is printed in the terminal to inform the user that the chat connection has been established. This message usually explains that the user can type messages freely and use keyboard shortcuts like `Ctrl+C` or `Ctrl+D` to leave the chatroom.

After successfully connecting, the client immediately asks the user to choose a nickname. This nickname is how other participants will recognize the user in the chat. The chosen name is typed in the terminal and sent to the server right away. Once the server receives it, it assigns the user a random color and announces their arrival to the chatroom. From this point on, any messages sent by this user will appear under that nickname, in their unique color.

The main body of the client program is a continuous loop that keeps the chat alive. It runs inside a `try` block using `while True`, meaning it continues operating until either the user quits or the connection is closed. Inside the loop, the client uses the `select` module to watch two sources at the same time: the user's keyboard input and the network socket connected to the server. These are placed together in a small list. The `select` function waits until one of these sources becomes active. If the user types something, it becomes ready; if the server sends something, the socket becomes ready. The function then returns which of the two has activity, allowing the program to react immediately without delay.

When the server sends new data, the client reads it using the socket's `recv` method. If the data is empty, it means the server has closed the connection, so the client displays a message saying that it has been disconnected, closes its own socket, and exits the program. If the data is not empty, the client decodes it from bytes to readable text. Because the server is responsible for all message formatting and color labeling, the client does not need to process the message any further—it simply prints it to the terminal. Messages are written directly to standard output and flushed so that they appear instantly. The result is a live chat feed where each message appears the moment it is received, with nicknames shown in color and special notices from the server highlighted for visibility.

If the user types something in the terminal and presses Enter, the client reads that line from standard input. If the input is empty, it usually means the user has ended the session using a special key like `Ctrl+D`. The client then prints a short goodbye message, closes the socket, and exits. If the line contains text, the client sends it to the server in encoded form. The server will handle everything else—checking whether the message is public or private, formatting it with colors, and sending it to the correct people. From the user's point of view, chatting is simple: type a line, press Enter, and see the results appear instantly across all connected users.

The loop that handles these operations is surrounded by a `try-except` structure. This ensures that if the user interrupts the program by pressing `Ctrl+C`, the client will not crash or leave the connection hanging. Instead, it prints a short "Closing connection" message, shuts down the socket cleanly, and exits. This makes sure that the program always ends gracefully, no matter how the user decides to leave the chatroom.

Through this design, the client becomes an efficient and responsive participant in the chat system. It keeps communication flowing smoothly in both directions—receiving and sending messages in real time. With the addition of nicknames, colors, and private message support, the chat feels organized, personal, and interactive. The client's role is simple but vital: it translates the user's input into network data, displays the incoming conversation beautifully, and ensures that the experience feels alive and seamless in the terminal.

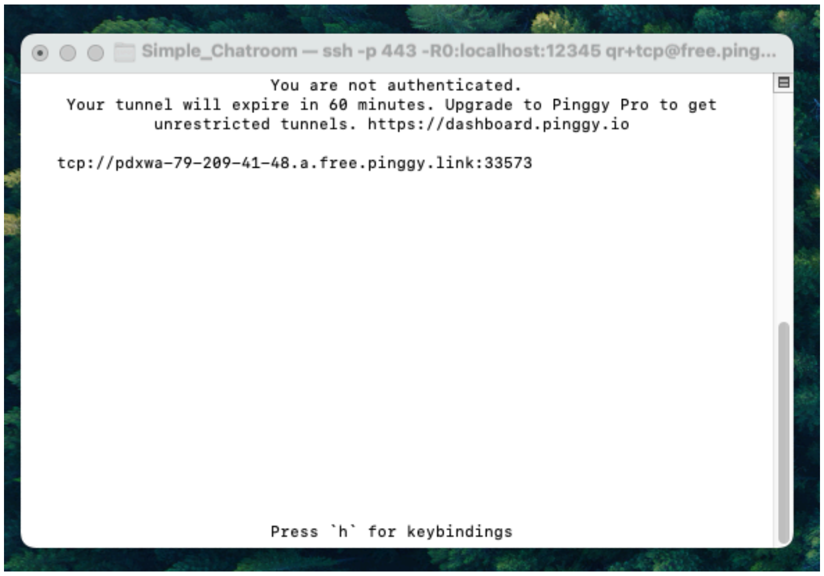
Pinggy

When the chatroom is first created, both the server and the clients are designed to work only within the same computer or local network. Connections are made through the address "localhost" or "127.0.0.1," which means that all communication happens internally on the same machine. This setup is ideal for testing, but it prevents users from other locations from joining the chat, because the computer running the server is usually hidden behind a router and does not have a public address that can be reached directly. To make the local server accessible from the internet, a tunneling service such as Pinggy can be used. Through this service, a secure tunnel is created between the local machine and a public server operated by Pinggy.

By using Pinggy, a connection is established that allows all incoming traffic from the internet to be forwarded to the local server. This is done through a short SSH command, which tells Pinggy which local port should be linked to the public address. In this project, the server is set to listen on port 12345, so the command is written in a way that any incoming request to Pinggy's public address is redirected to `localhost:12345` on the user's machine. Once the tunnel is successfully created, a special public address is provided by Pinggy. It appears in the terminal in a form similar to `tcp://gbwmz-79-209-41-48.a.free.pinggy.link:46229`. This address serves as the entry point through which users on the internet can reach the chat server running locally.

After the tunnel is created, the server continues to operate on the local computer exactly as before, but all communication from the outside world is automatically passed through Pinggy. When a remote client connects to the public address provided by Pinggy, the data is received by Pinggy's server and then securely forwarded to the chat server running on the user's machine. From the client's perspective, the connection feels no different from connecting to a regular online server. The local computer becomes accessible over the internet without any changes being required in the code.

Once the public address and port are provided by Pinggy, they can be shared with anyone who wishes to join the chatroom. The same `client.py` program is used by the remote participants, but instead of typing `127.0.0.1` or `localhost`, the host and port given by Pinggy are entered. In this example, the host is `gbwmz-79-209-41-48.a.free.pinggy.link`, and the port is `46229`. When the client is started with these values, the connection is routed through Pinggy to the local server, allowing real-time communication between people in different locations.


Simple_Chatroom — ssh -p 443 -R0:localhost:12345 qr+tcp@free.ping...

```
                    You are not authenticated.
  Your tunnel will expire in 60 minutes. Upgrade to Pinggy Pro to get
         unrestricted tunnels. https://dashboard.pinggy.io

tcp://pdxwa-79-209-41-48.a.free.pinggy.link:33573




                    Press `h` for keybindings
```

Figure 1. A screenshot of your Pinggy setup

No modifications to the server or client code are required for this process to work. The server is already configured to listen on all interfaces through the address “0.0.0.0,” and the client is already able to accept any host and port as input. The only important condition is that the same port number used in the server program must also be the one specified in the Pinggy command. When this requirement is met, Pinggy is able to forward traffic correctly to the server.

Through this setup, a local chatroom is effectively transformed into a global one. The server continues to run on the user’s own computer, but by means of the Pinggy tunnel, it is made reachable from anywhere on the internet. Users from different places can connect to the public address, and their messages are transmitted securely to the local server, making the chatroom available to the world without any complex network configuration.

Challenges

Building this chatroom project is not just an exercise in writing code; it is a journey into how communication really works inside computers and across networks. What begins as a simple idea—sending text from one terminal to another—quickly reveals many small but important challenges that teach valuable lessons about programming, problem-solving, and the logic of networked systems.

One of the first challenges faced in this project is understanding how computers connect and exchange data through sockets. It requires learning how a client and server talk to each other using network ports and addresses. This can feel abstract at first, but once the basic connection succeeds, it becomes clear that sockets are like telephone lines between programs: each side must listen, speak, and stay synchronized.

Another challenge appears when several users are connected at the same time. The server cannot simply wait for one client’s message before listening to another, or the whole system would freeze. To solve this, the project introduces the `select` module, which allows the program to watch many connections at once. Through this, we learn about non-blocking input and the importance of handling multiple data sources efficiently—a concept that is central to all real-time communication systems.

Managing clean disconnections and preventing crashes is also a learning experience. Every open connection must be monitored carefully, and when a user leaves, the server needs to remove them from its lists to avoid sending data into empty sockets. This teaches us about the importance of resource management and error handling in long-running programs.

Adding new features such as nicknames, colors, and private messaging brings a different set of lessons. We learn how to build upon existing logic without breaking it, and how small design decisions—like using dictionaries to map nicknames to sockets—make new capabilities easier to add later. Implementing color output with the Colorama module introduces us to the idea of improving usability and readability without changing core functionality. Creating private messaging shows how message routing can be controlled at the server level, teaching the difference between shared and direct communication.

Throughout this process, debugging and testing also play a big role. Because many parts of the chatroom run simultaneously, errors are not always obvious. Learning to observe the flow of data, track messages, and understand how timing affects network programs is one of the most valuable skills gained from this project.

Beyond the technical aspects, the project also teaches patience and incremental design. Each feature—connecting users, broadcasting messages, assigning nicknames, coloring output, and supporting private chats—is built step by step. By testing each part before adding the next, we learn how complex systems can grow from simple, working foundations.

In the end, this project demonstrates more than just how to make a chatroom. It teaches the principles behind all interactive, networked applications. We discover how data travels, how real-time systems coordinate multiple users, and how a clear structure in code can make even complicated features manageable. Through the process, we gain confidence not only in writing Python but also in thinking like a systems designer—someone who can imagine how many small parts work together to create a seamless experience.