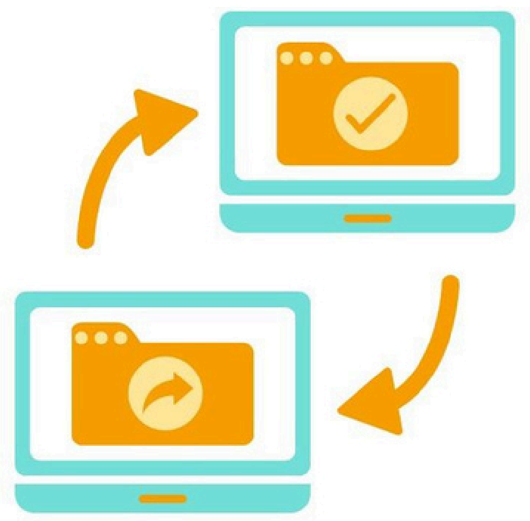
FILE TRANSFER SYSTEM

What is a File Transferring System?

- **Definition:** A File Transferring System enables the movement of digital files (e.g., documents, images, videos) between devices.
- **Functionality**: It allows efficient and secure file sharing over networks like the Internet or LAN.
- **Usage**: Applicable for both personal file sharing and large-scale organizational tasks like software distribution.

File Sharing Transfer Protocol Colored Icon



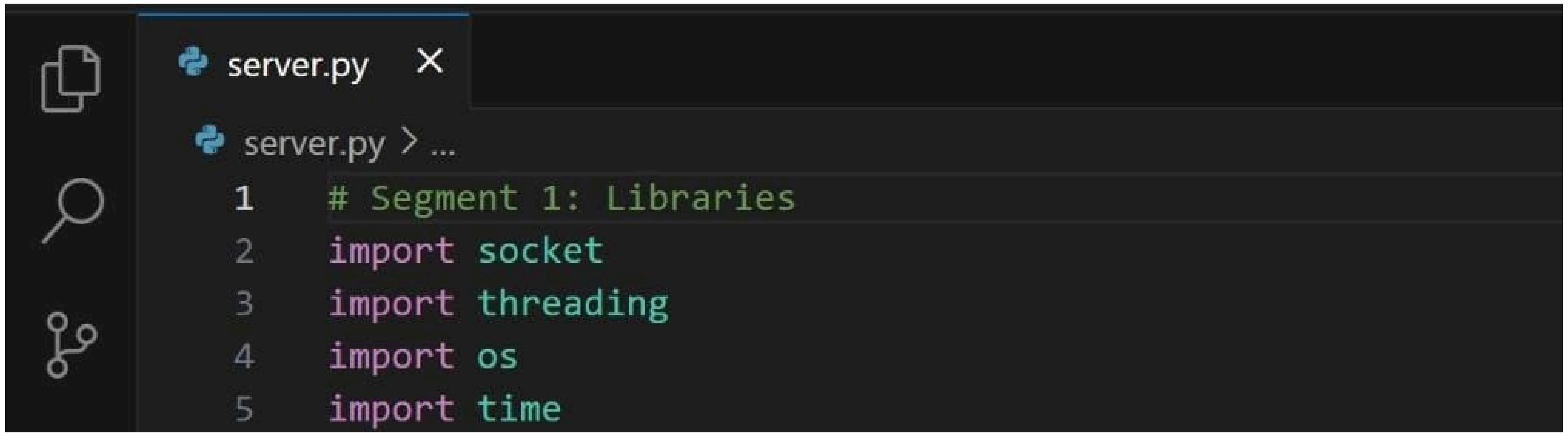
PROJECT GOALS

- Built a client-server based file transfer system using Python.
- Supports upload and download functionality.
- Includes basic user authentication to restrict access.
- Uses TCP sockets for reliable data transfer.
- Libraries used: socket, threading, os, time.

TECH STACK

- Language: Python
- Socket Type: TCP (SOCK_STREAM)
- IDE: Visual Studio Code
- Libraries: socket, threading, os

FILE HANDLING



```python
# Segment 1: Libraries
import socket
import threading
import os
import time
```

These are the modules we’ve been using for this project.

SIGNIFICANCE’S

socket Module:

- The socket module in Python is used for network programming.
- It allows communication between computers over a network using protocols like TCP or UDP.
- You can create servers or clients using sockets.

threading Module:

- The threading module is used to run multiple tasks at the same time (concurrently).
- It is helpful when you want, for example, a server to handle multiple clients at once.

SIGNIFICANCE’S

os Module:

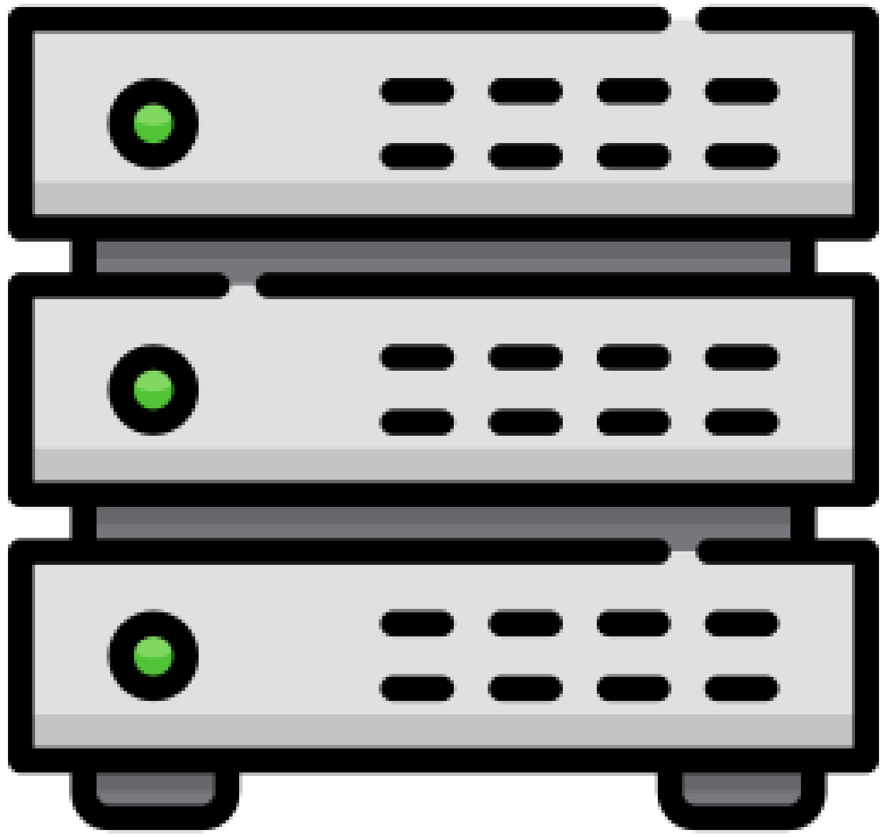
- The os module provides functions to interact with the operating system.
- You can use it to access file paths, environment variables, create/remove directories, and more.

time Module:

- The time module is used to work with time-related functions.
- You can pause the program, measure execution time, or get the current time.

SERVER

1. **TCP Socket Setup**: Initializes a TCP socket and listens on port 8080.
2. **Client Handling**: Accepts client connections and spawns a new thread for each client.
3. **Authentication & Commands**: Performs basic (dummy) username/password authentication and supports upload, download, and exit commands.
4. **File Management:** Saves uploaded files and serves requested files for download.

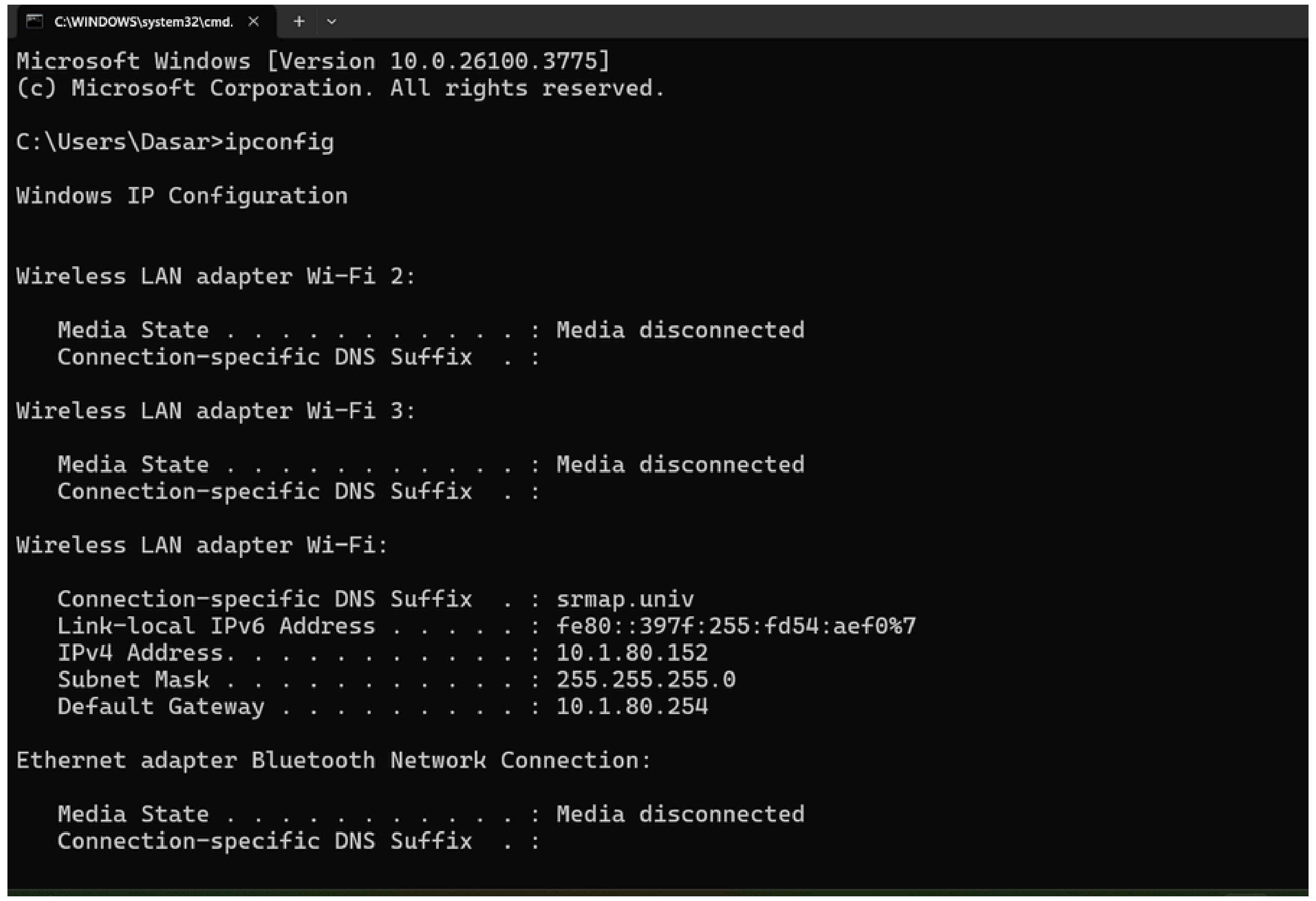


CLIENT

- Creates a TCP socket and connects to the server using IP and port.
- Prompts user for username and password, sends to server.
- Waits for server authentication response.
- After login, accepts commands from user:
    - Upload: Reads file, sends to server with <EOF> marker.
    - Download: Requests file, saves incoming data as a local file.
    - Exit: Cleanly closes connection with the server.
- Handles network errors and invalid responses gracefully.
- Runs as a simple command-line application.

Multiple Clients Monotone Icon

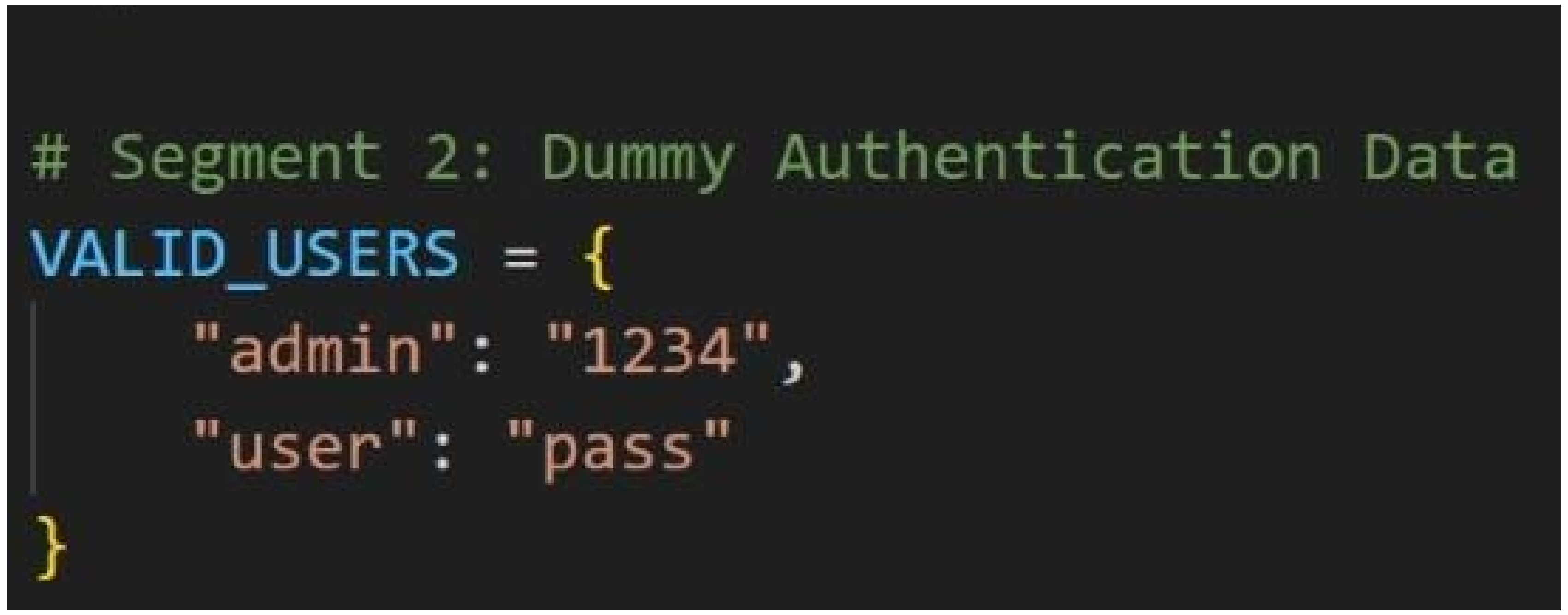
- **Server IP Requirement**: The client must know the server’s IPv4 address (e.g., 192.168.1.5) to connect over the same local network.
- **How to Find IP (Windows):** Use ipconfig in Command Prompt to locate the IPv4 address under Wi-Fi or Ethernet adapter.
- **Connection Conditions:** Both devices must be on the same network, the server must be running, and firewalls/antivirus should not block port 8080.



```
Microsoft Windows [Version 10.0.26100.3775]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Dasar>ipconfig

Windows IP Configuration


Wireless LAN adapter Wi-Fi 2:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi 3:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . : srmap.univ
   Link-local IPv6 Address . . . . . : fe80::397f:255:fd54:aef0%7
   IPv4 Address. . . . . . . . . . . : 10.1.80.152
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.1.80.254

Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
```

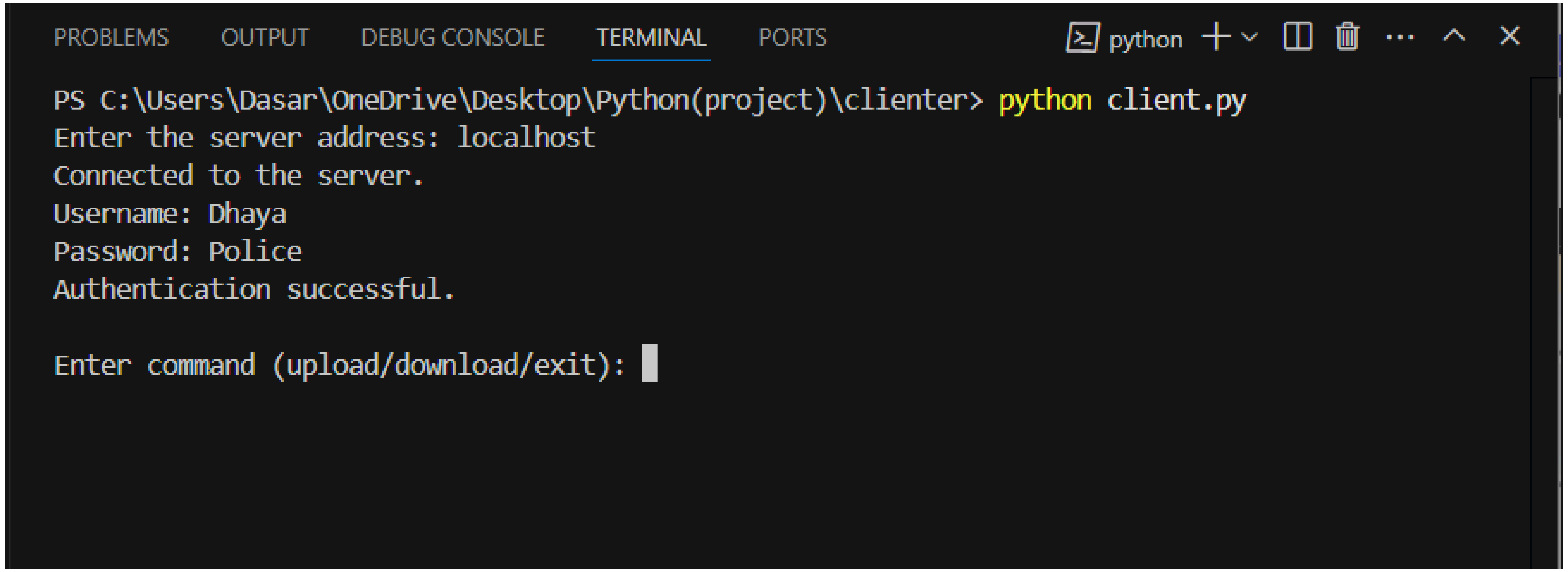
AUTHENTICATION

- Here we are using an dummy authentication . That means there is no sort of Hashing involved .
- The Username and password is compared to the Valid_user dictionary.
- Only if the client enters the valid credentials it gets connects to the server.



```
# Segment 2: Dummy Authentication Data
VALID_USERS = {
    "admin": "1234",
    "user": "pass"
}
```

After the authentication client would send one of the three options that is upload download and exit



```
PS C:\Users\Dasar\OneDrive\Desktop\Python(project)\clienter> python client.py
Enter the server address: localhost
Connected to the server.
Username: Dhaya
Password: Police
Authentication successful.

Enter command (upload/download/exit):
```

UPLOAD (CLIENTSIDE)

- User enters command upload.
- Client prompts for a filename.
- Checks if file exists locally.
- Sends filename and file content to server in chunks.
- Ends the upload with <EOF> marker
- The time library (module) that we imported is used at the time of EOF we create a 0.5 sec of time delay just to make sure the server is knowing the difference between the contents and the EOF.
- client_socket.send(filename.encode())
- Reads file in binary: f.read(4096)
- Sends all chunks, then b"<EOF>"

UPLOAD (SERVERSIDE)

- Server receives the upload command.
- Receives filename and sends "OK" to client.
- Opens a new file and starts writing incoming binary data.
- Stops writing when it detects <EOF> in data.
- Sends confirmation to client.

Behind the scenes:

- with open(filename, "wb") as f:
- if chunk.endswith(b"<EOF>"): → stop
- client_socket.send("Upload complete")

RESULT

- File successfully received by server.
- Saved under same filename on server machine.
- Client receives confirmation message.
- Verified with test files (.txt, .jpg, .py etc).

DOWNLOAD (CLIENTSIDE)

- User enters command download.
- Enters filename to request from server.
- Waits for server response (OK or File not found).
- If OK, receives file in chunks.
- Stops receiving on <EOF>.

Behind the scenes:

- Creates a new file: open(filename, 'wb')
- recv(4096) repeatedly
- Ends when <EOF> detected

DOWNLOAD(SERVERSIDE)

- Receives download command.
- Checks if the requested file exists.
- If yes, opens file in binary and sends it chunk by chunk.
- Appends <EOF> to indicate transfer is complete.

Behind the scenes:

- f.read(4096) and client_socket.send(chunk)
- client_socket.send(b"<EOF>")

RESULT

- Client saves the file successfully.
- Output message: File '<filename>' saved successfully.
- Integrity verified — file matches original.

EXIT

- Cient sends "exit" command to server.
- Server receives it, breaks the loop, and closes the connection.
- Client also closes its socket.
- Server stays active for other clients.

Key Points:

- Clean disconnect.
- No crashes or memory leaks.
- Thread ends safely on server side.

CONCLUSION

- Successfully established a TCP socket connection between client and server.
- File upload and download tested with multiple file types (e.g., .txt, .pdf, .png).
- Authentication system worked correctly with valid and invalid credentials.
- Server handled multiple clients simultaneously using threading.
- Files transferred across devices connected to the same Wi-Fi / hotspot network.
- Verified integrity of transferred files — no data loss or corruption.

THANK YOU

AP23110011573 - M.Narayana
AP23110011587 - M.Preveen
AP23110011591 - D.Lekith Kishore
AP23110011595 - S.Vivek