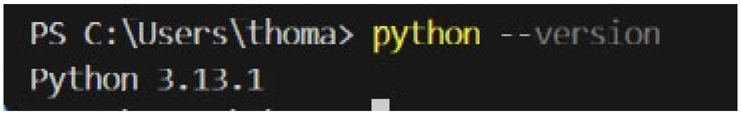
Task 1: Environment Verification

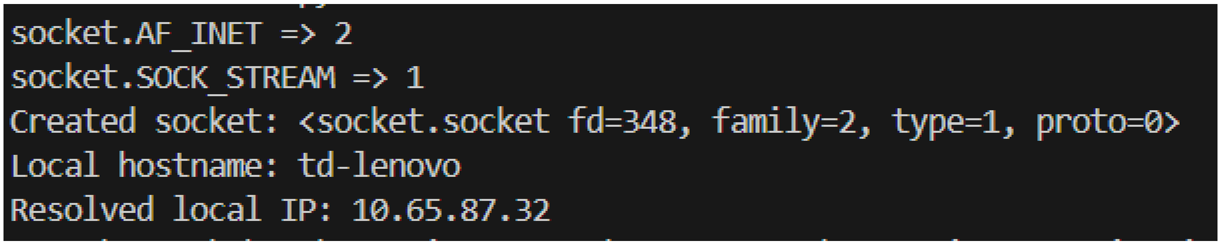
- Verify Python installation using python --version
- Import the socket module without errors



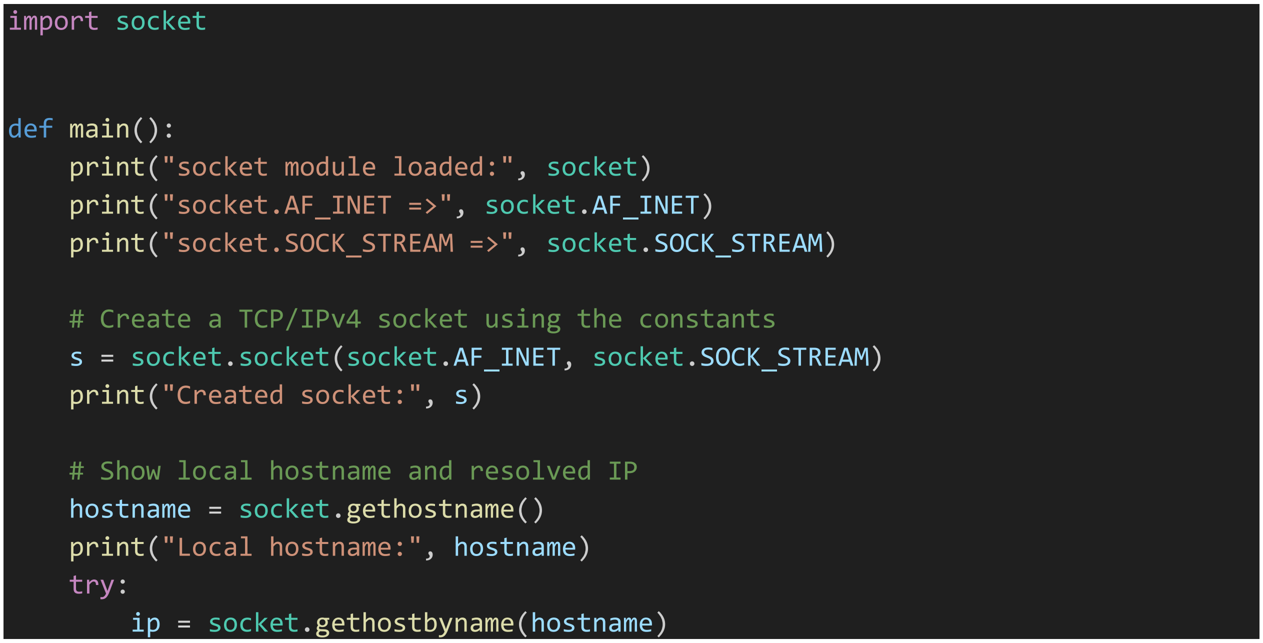
```
PS C:\Users\thoma> python --version
Python 3.13.1
```

Task 2: Understanding Sockets

- Create a simple Python script that imports socket
- Identify and explain the purpose of socket.AF_INET and socket.SOCK_STREAM
  - **socket.AF_INET**: It is for selects the IPv4 address family for the socket; when used the address format is a tuple (host, port) like 0.0.0.0 (host), 8000(port). For IPv6 use AF_INET6.
  - **socket.SOCK_STREAM**: requests a TCP (stream) socket: connection-oriented, reliable, ordered byte stream. The common alternative is socket.SOCK_DGRAM for UDP.



```
socket.AF_INET => 2
socket.SOCK_STREAM => 1
Created socket: <socket.socket fd=348, family=2, type=1, proto=0>
Local hostname: td-lenovo
Resolved local IP: 10.65.87.32
```

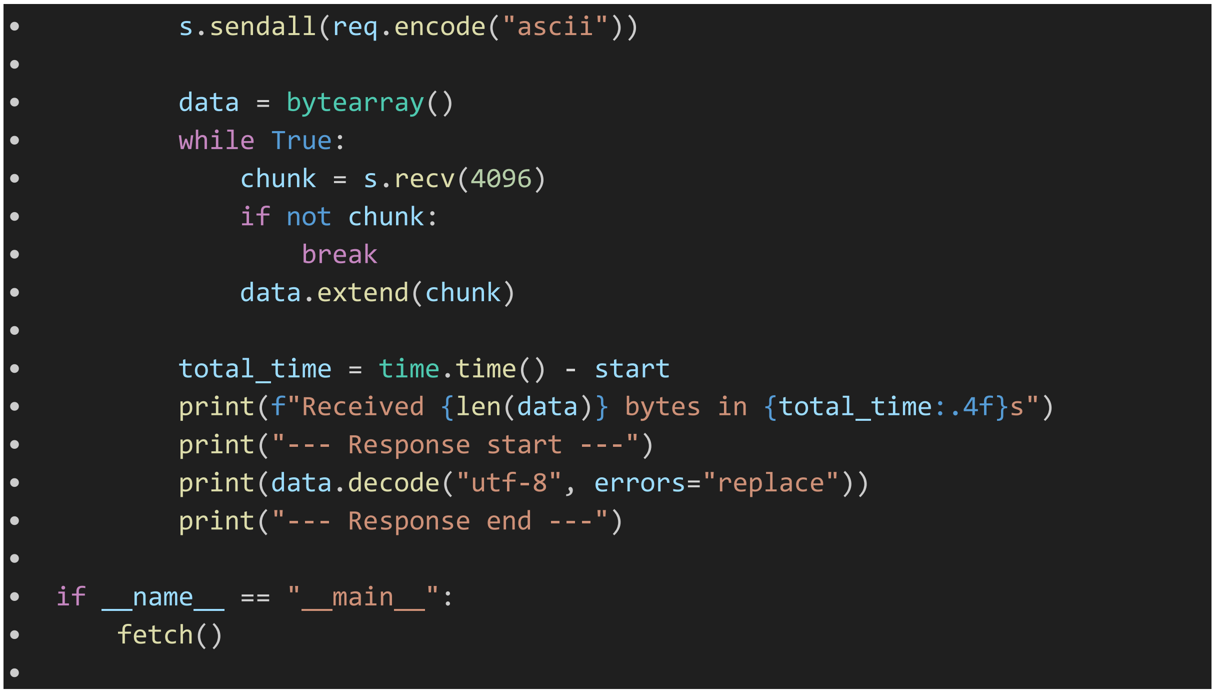


```python
import socket


def main():
    print("socket module loaded:", socket)
    print("socket.AF_INET =>", socket.AF_INET)
    print("socket.SOCK_STREAM =>", socket.SOCK_STREAM)

    # Create a TCP/IPv4 socket using the constants
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("Created socket:", s)

    # Show local hostname and resolved IP
    hostname = socket.gethostname()
    print("Local hostname:", hostname)
    try:
        ip = socket.gethostbyname(hostname)
        print("Resolved local IP:", ip)
    except Exception as e:
        print("Could not resolve local IP:", e)

    s.close()


if __name__ == "__main__":
    main()
```

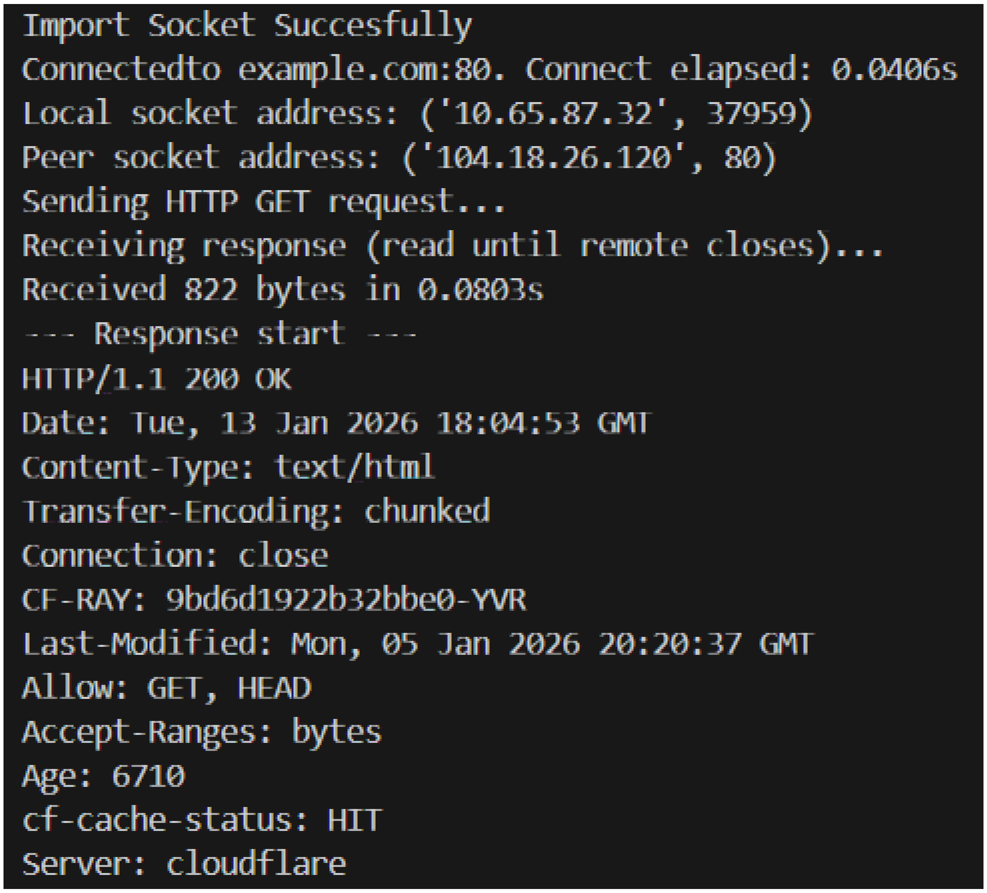
Task 3: Simple Client Connection

- Write a Python script that creates a TCP socket
- Connect to a known public server (example: example.com on port 80)
- Send a simple HTTP GET request and print the response
- Capture and explain the connection behavior
  - The connection is successfully connected. It shows a normal TCP client lifecycle: the client created a TCP/IPv4 socket and connect() completed the TCP three-way handshake in about 0.04. The script sent an HTTP/1.1 GET over the established connection and the server replied with a chunked 200 OK response. Because I used Connection: close, the server closed the TCP connection after sending the final chunk (0), and the client detected EOF when recv() returned empty, then closed the socket; the settimeout(10) prevents the client from blocking indefinitely.

```
•  import socket
•  import time
•
•  def fetch(host: str = "example.com", port: int = 80, path: str = "/",
   timeout: int = 10) -> None:
•      """Simple TCP HTTP GET client. Prints connection info and the full
   response."""
•      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
•          s.settimeout(timeout)
•          start = time.time()
•          s.connect((host, port))
•          connect_time = time.time() - start
•          print(f"Connected to {host}:{port} (connect {connect_time:.4f}s)")
•          print("Local:", s.getsockname(), "Peer:", s.getpeername())
•
•          req = f"GET {path} HTTP/1.1\r\nHost: {host}\r\nConnection:
   close\r\n\r\n"
```



```
        s.sendall(req.encode("ascii"))

        data = bytearray()
        while True:
            chunk = s.recv(4096)
            if not chunk:
                break
            data.extend(chunk)

        total_time = time.time() - start
        print(f"Received {len(data)} bytes in {total_time:.4f}s")
        print("--- Response start ---")
        print(data.decode("utf-8", errors="replace"))
        print("--- Response end ---")

if __name__ == "__main__":
    fetch()
```



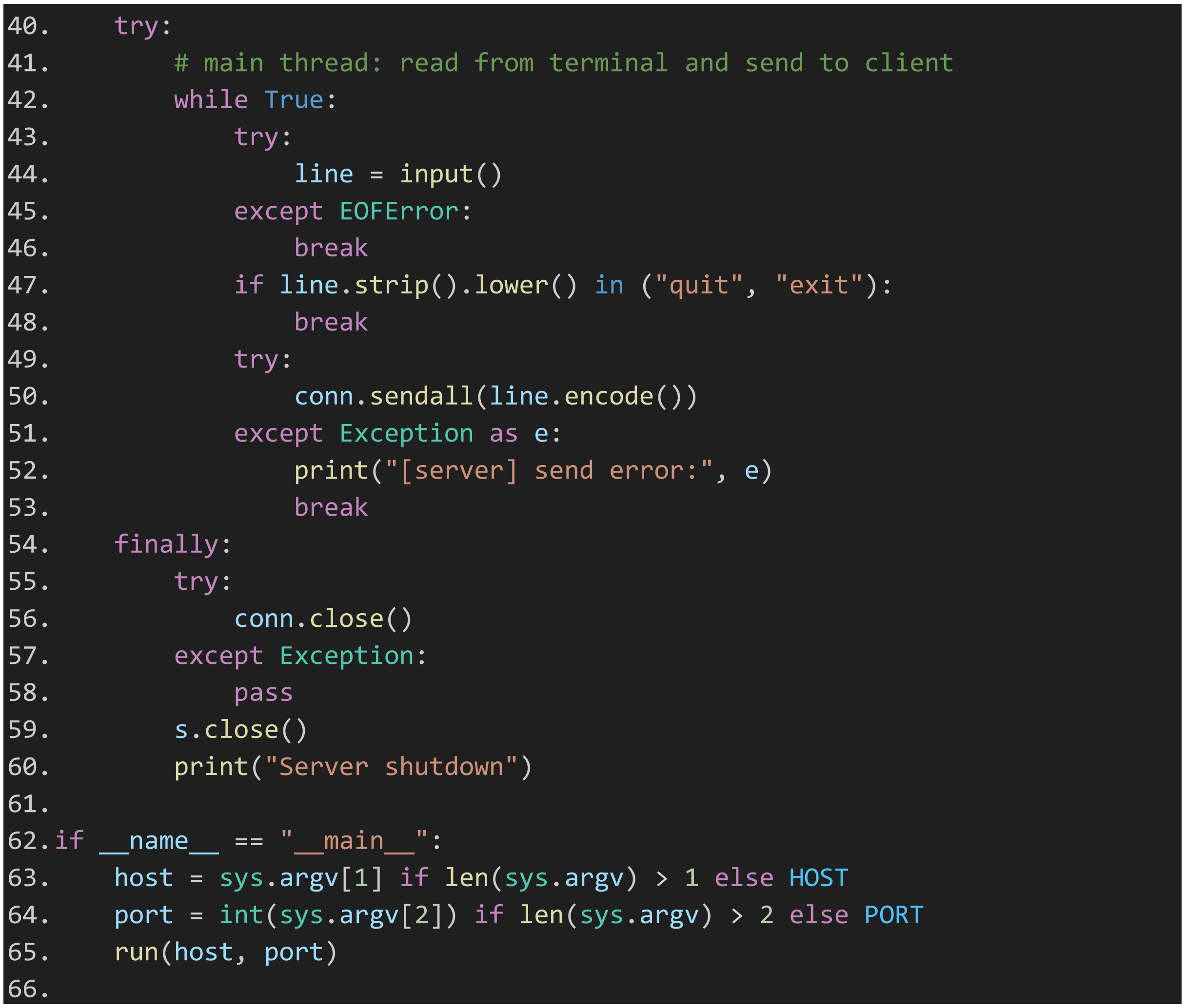
```
Import Socket Succesfully
Connectedto example.com:80. Connect elapsed: 0.0406s
Local socket address: ('10.65.87.32', 37959)
Peer socket address: ('104.18.26.120', 80)
Sending HTTP GET request...
Receiving response (read until remote closes)...
Received 822 bytes in 0.0803s
--- Response start ---
HTTP/1.1 200 OK
Date: Tue, 13 Jan 2026 18:04:53 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: close
CF-RAY: 9bd6d1922b32bbe0-YVR
Last-Modified: Mon, 05 Jan 2026 20:20:37 GMT
Allow: GET, HEAD
Accept-Ranges: bytes
Age: 6710
cf-cache-status: HIT
Server: cloudflare
```

Task 4: Create a TCP Server

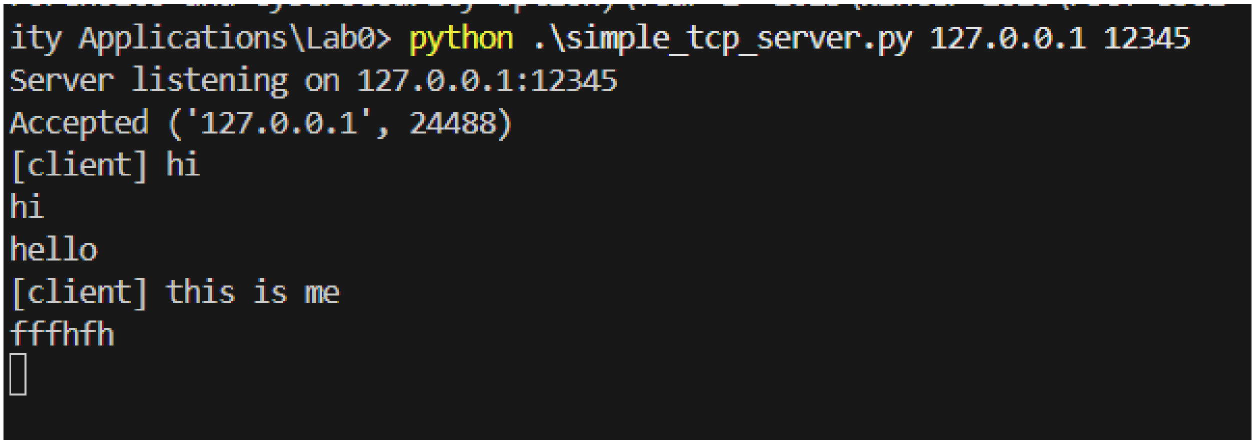
Write a Python script that:

1. Binds to localhost on a chosen port (e.g., 12345)
2. Listens for incoming connections
3. Accepts a connection and receives a message from the client
4. Sends a response back

```python
import socket
import threading
import sys

HOST = "127.0.0.1"
PORT = 12345

def _recv_loop(conn):
    try:
        while True:
            data = conn.recv(4096)
            if not data:
                print("[server] client disconnected")
                break
            print("[client]", data.decode(errors="replace"))
    except Exception as e:
        print("[server] recv error:", e)
    finally:
        try:
            conn.shutdown(socket.SHUT_RDWR)
        except Exception:
            pass
        conn.close()

def run(host=HOST, port=PORT):
    s = socket.socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((host, port))
    s.listen(1)
    print(f"Server listening on {host}:{port}")
    conn, addr = s.accept()
    print("Accepted", addr)
    t = threading.Thread(target=_recv_loop, args=(conn,), daemon=True)
    t.start()

```



```
    try:
        # main thread: read from terminal and send to client
        while True:
            try:
                line = input()
            except EOFError:
                break
            if line.strip().lower() in ("quit", "exit"):
                break
            try:
                conn.sendall(line.encode())
            except Exception as e:
                print("[server] send error:", e)
                break
    finally:
        try:
            conn.close()
        except Exception:
            pass
        s.close()
        print("Server shutdown")

if __name__ == "__main__":
    host = sys.argv[1] if len(sys.argv) > 1 else HOST
    port = int(sys.argv[2]) if len(sys.argv) > 2 else PORT
    run(host, port)

```



```
ity Applications\Lab0> python .\simple_tcp_server.py 127.0.0.1 12345
Server listening on 127.0.0.1:12345
Accepted ('127.0.0.1', 24488)
[client] hi
hi
hello
[client] this is me
fffhfh
```

Task 5: Create a TCP Client

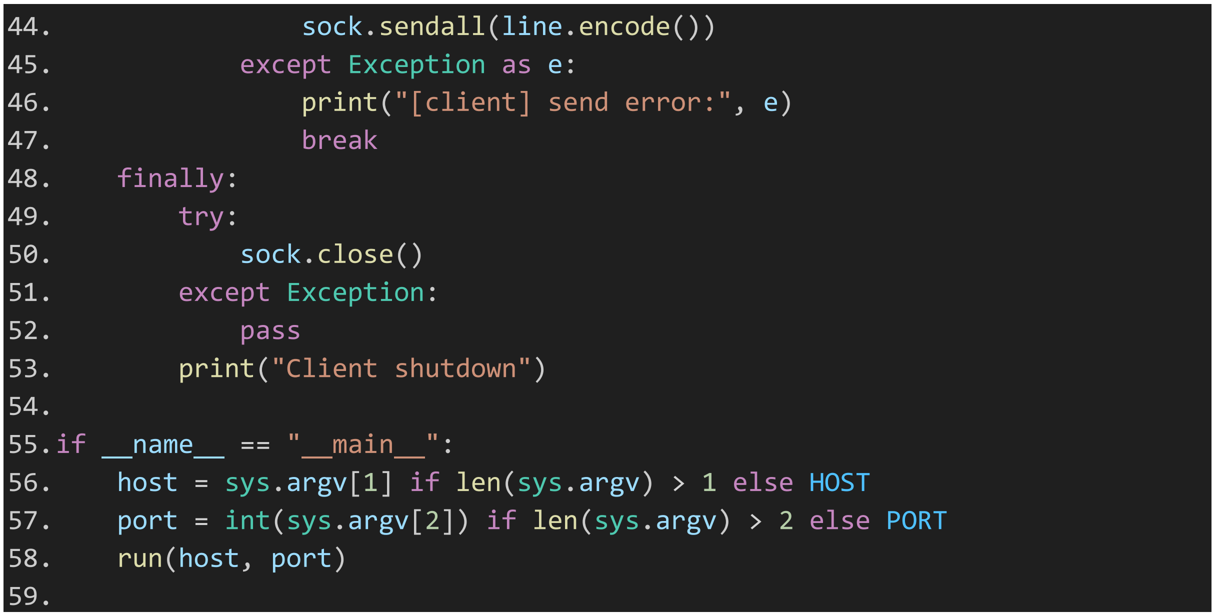
Write a Python script that:

1. Connects to your server on localhost:12345

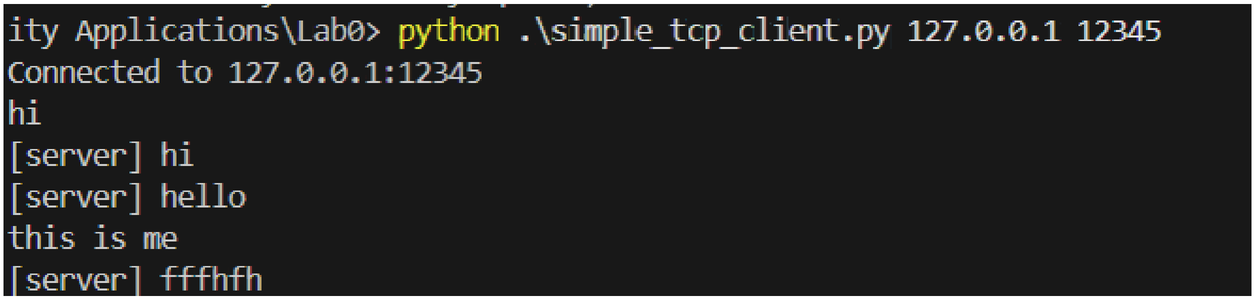
2. Sends a custom message

3. Receives and prints the server’s response

```python
import socket
import threading
import sys

HOST = "127.0.0.1"
PORT = 12345

def _recv_loop(sock):
    try:
        while True:
            data = sock.recv(4096)
            if not data:
                print("[client] server disconnected")
                break
            print("[server]", data.decode(errors="replace"))
    except Exception as e:
        print("[client] recv error:", e)
    finally:
        try:
            sock.shutdown(socket.SHUT_RDWR)
        except Exception:
            pass
        sock.close()

def run(host=HOST, port=PORT):
    sock = socket.socket()
    sock.connect((host, port))
    print(f"Connected to {host}:{port}")
    t = threading.Thread(target=_recv_loop, args=(sock,), daemon=True)
    t.start()

    try:
        while True:
            try:
                line = input()
            except EOFError:
                break
            if line.strip().lower() in ("quit", "exit"):
                break
            try:
```



```python
                sock.sendall(line.encode())
            except Exception as e:
                print("[client] send error:", e)
                break
    finally:
        try:
            sock.close()
        except Exception:
            pass
        print("Client shutdown")

if __name__ == "__main__":
    host = sys.argv[1] if len(sys.argv) > 1 else HOST
    port = int(sys.argv[2]) if len(sys.argv) > 2 else PORT
    run(host, port)
```



```
ity Applications\Lab0> python .\simple_tcp_client.py 127.0.0.1 12345
Connected to 127.0.0.1:12345
hi
[server] hi
[server] hello
this is me
[server] fffhfh
```

Task 5: Security Reflection

- Identify at least three security risks related to raw socket programming.
  - Raw sockets can be put into "promiscuous mode," allowing an application to intercept and read all traffic passing through a network interface, even if it isn't addressed to that specific host. If sensitive data is not encrypted, it can be easily captured.
  - It can be manually constructed the IP header, they can easily forge the source IP address. This is a primary technique used in DoS attacks, where an attacker hides their identity or tricks a server into responding to a victim's IP
  - Improperly crafted raw packets can bypass the operating system's built-in sanity checks. Sending malformed packets to a target can trigger buffer overflows or cause the target's network stack to crash, leading to system instability or remote code execution.
- Explain why input validation, access control, and protocol awareness matter at the network level.
  - **Input Validation:** the website need the validation to prevent the bot to attach the website and without it, the attacker might try to attempt overflow the packet.

    - **Access Control:** It is to ensure that only authorized entities can send or receive specific types of traffic.

    - **Protocol Awareness:** it is the state of connection that the protocolrecognize the packet to prevent the random packet hijack the session

- Suggest ways to secure your simple client-server application

    - Use TLS/SSL: Wrap your sockets in Transport Layer Security (TLS). This provides encryption, data integrity, and authentication, preventing man-in-the-middle attacks.

    - Enforce Timeouts: Set strict timeouts for connections and data reads. This prevents "Slowloris" type attacks where a client holds a connection open indefinitely to exhaust server resources.