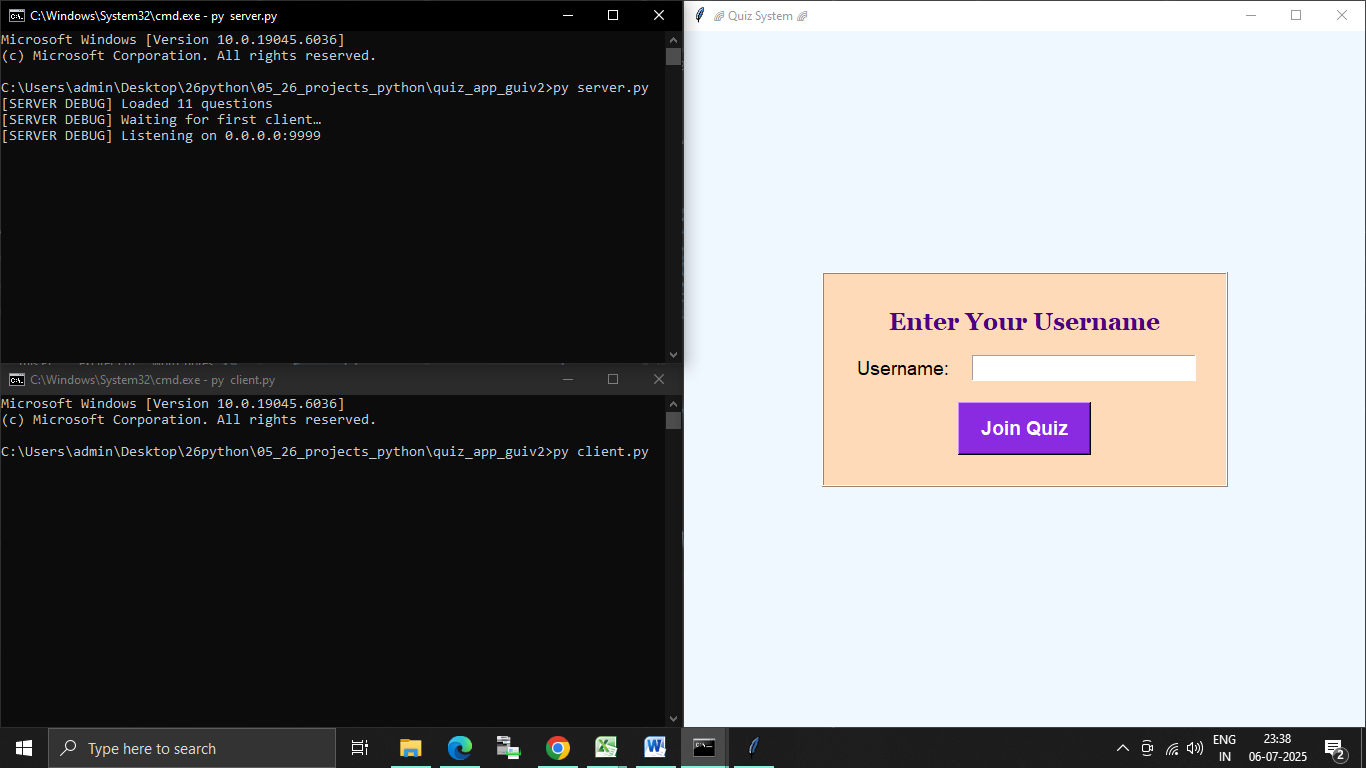
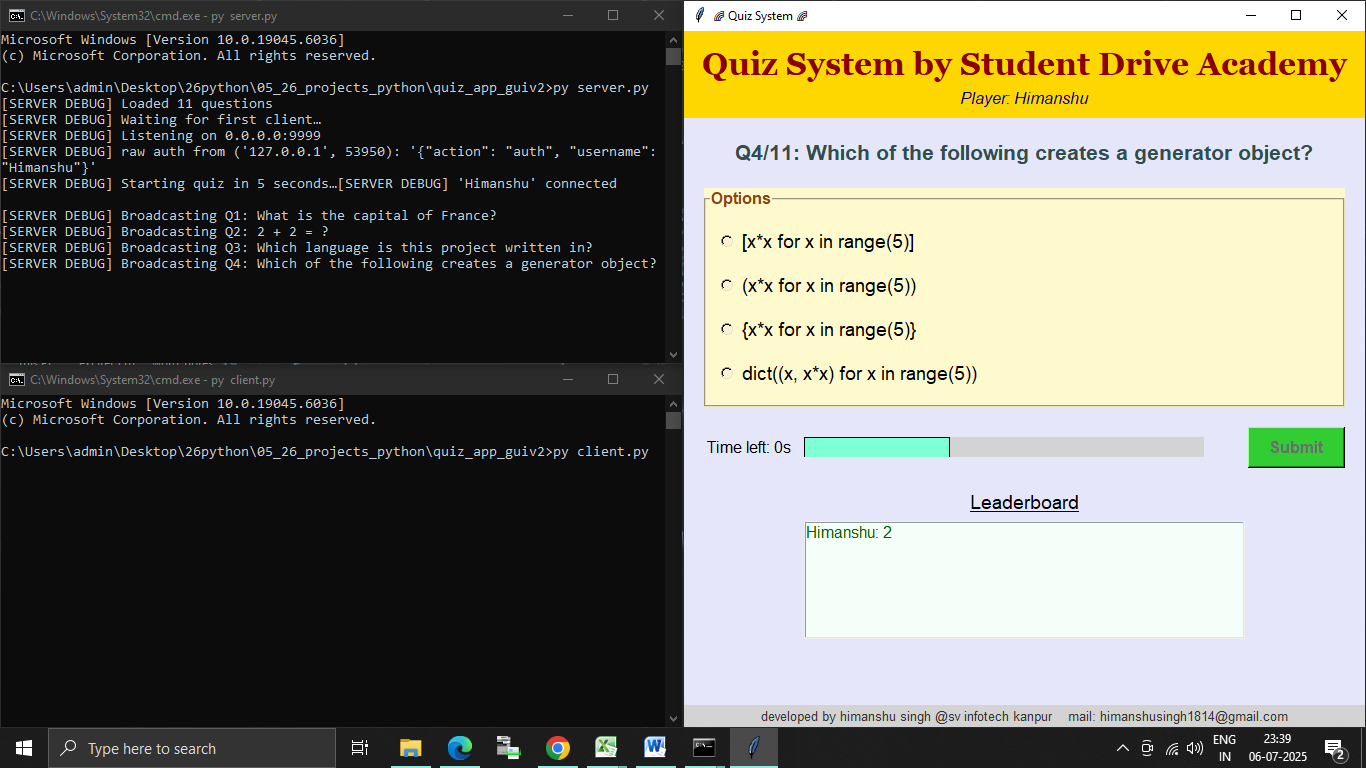
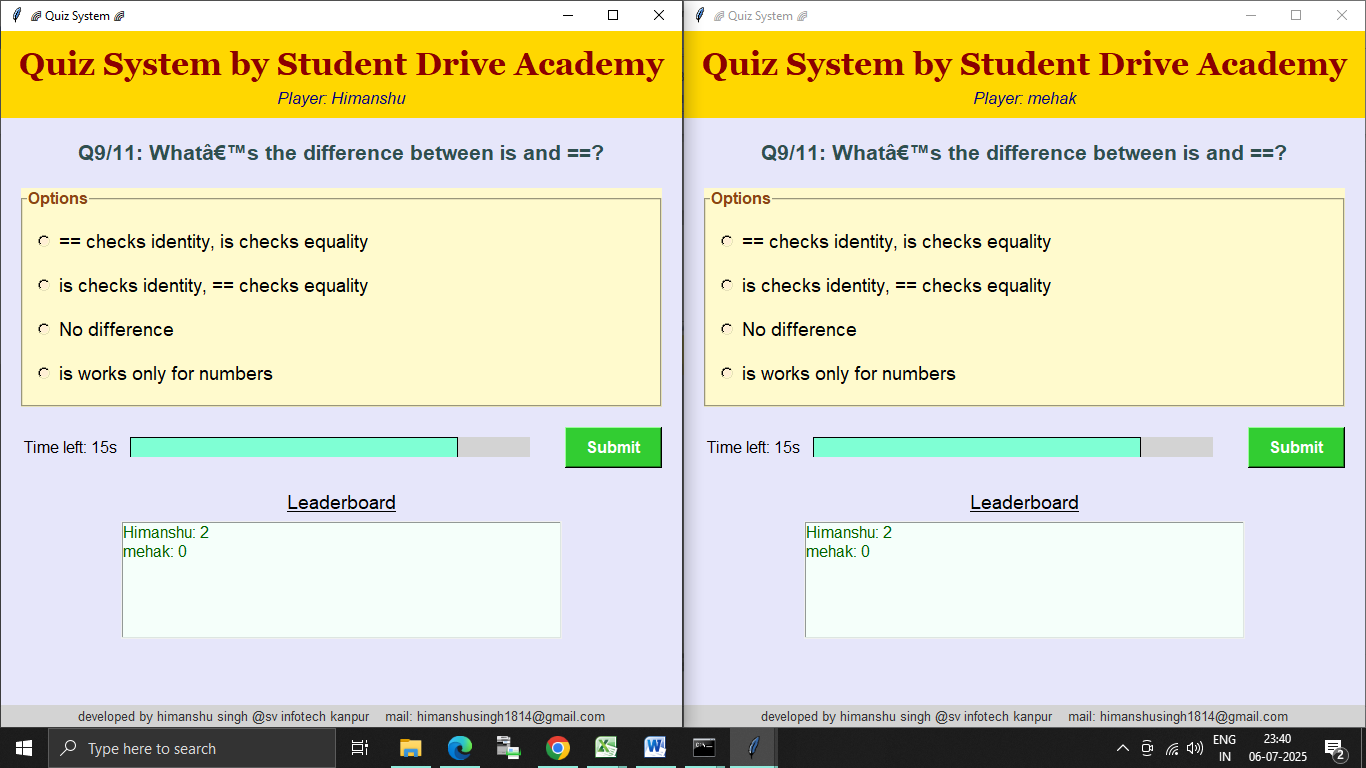
**NETWORK PROGRAMMNG IN PYTHON**

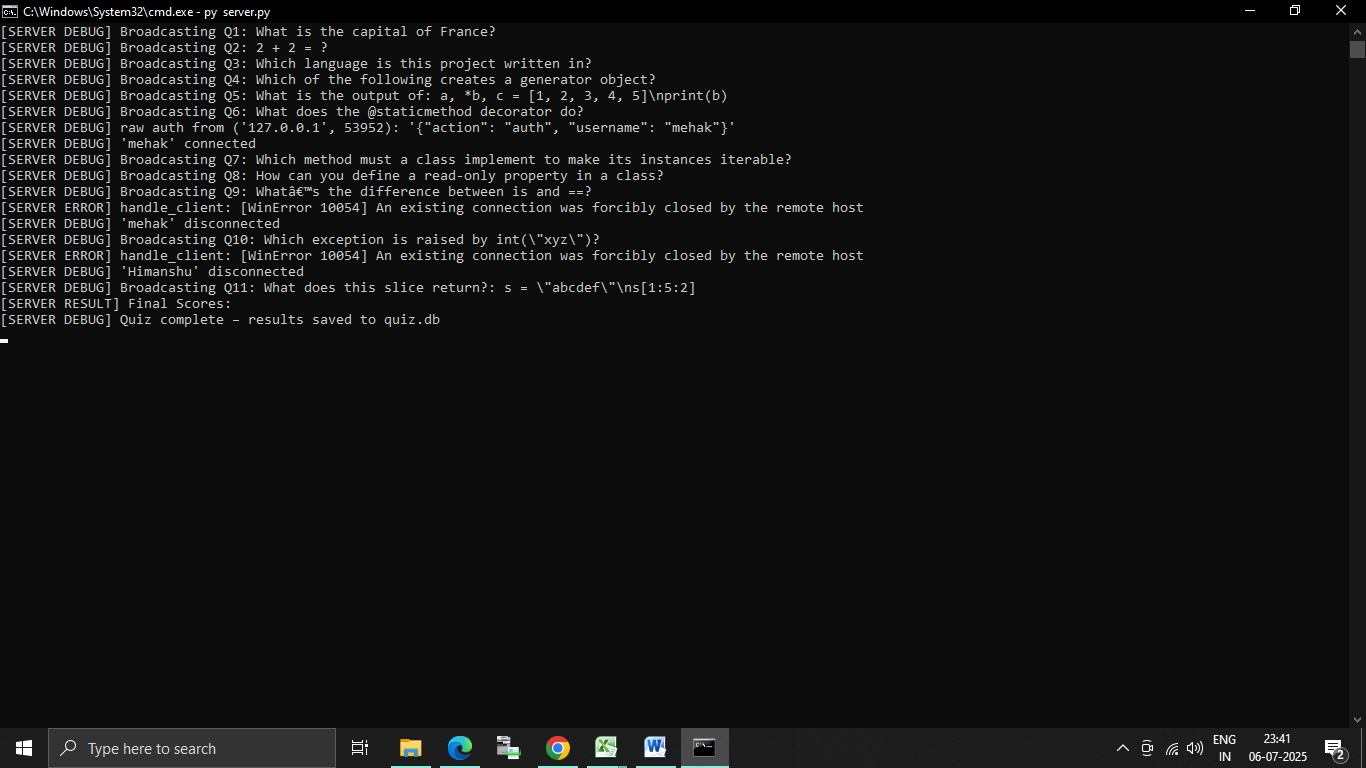
**SOCKET PROGRAMMING IN PYTHON**

**PROJECT QUIZ SYSTEM IN PYTHON (MULLTIPLE CLIENT ))**









**Source code:**

**py server.py**

**import socket**

**import threading**

**import json**

**import sqlite3**

**import time**

**from datetime import datetime**

**HOST, PORT = '0.0.0.0', 9999**

**# 1) Load quiz data immediately**

**with open('questions.json','r') as f:**

**questions = json.load(f)**

**print(f"[SERVER DEBUG] Loaded {len(questions)} questions")**

**# 2) Track connected clients**

**clients = [] # each is {'conn','username','score','event','last\_ans'}**

**clients\_lock = threading.Lock()**

**client\_connected = threading.Event()**

**# 3) SQLite persistence**

**db = sqlite3.connect('quiz.db', check\_same\_thread=False)**

**cur = db.cursor()**

**cur.execute('''**

**CREATE TABLE IF NOT EXISTS results (**

**id INTEGER PRIMARY KEY,**

**username TEXT,**

**score INTEGER,**

**taken\_at TEXT**

**)**

**''')**

**db.commit()**

**def broadcast(msg: dict):**

**data = (json.dumps(msg) + '\n').encode()**

**with clients\_lock:**

**for c in clients:**

**try:**

**c['conn'].sendall(data)**

**except Exception as e:**

**print(f"[SERVER ERROR] to {c['username']}: {e}")**

**def handle\_client(conn, addr):**

**f = conn.makefile('r')**

**username = None**

**try:**

**# first message must be auth**

**line = f.readline().strip()**

**print(f"[SERVER DEBUG] raw auth from {addr}: {repr(line)}")**

**req = json.loads(line)**

**if req.get('action')!='auth':**

**conn.close(); return**

**username = req.get('username') or f"guest\_{addr[1]}"**

**client = {**

**'conn': conn,**

**'username': username,**

**'score': 0,**

**'event': threading.Event(),**

**'last\_ans': None**

**}**

**with clients\_lock:**

**clients.append(client)**

**client\_connected.set()**

**conn.sendall(b'{"status":"ok"}\n')**

**print(f"[SERVER DEBUG] '{username}' connected")**

**# then accept answer messages**

**for line in f:**

**msg = json.loads(line)**

**if msg.get('action')=='answer':**

**client['last\_ans'] = (msg['question\_index'], msg['choice'])**

**client['event'].set()**

**except Exception as e:**

**print(f"[SERVER ERROR] handle\_client: {e}")**

**finally:**

**conn.close()**

**with clients\_lock:**

**clients[:] = [c for c in clients if c['conn'] is not conn]**

**print(f"[SERVER DEBUG] '{username}' disconnected")**

**def quiz\_manager():**

**# wait for someone to join**

**print("[SERVER DEBUG] Waiting for first client…")**

**client\_connected.wait()**

**# give a 5s buffer to join more clients**

**print("[SERVER DEBUG] Starting quiz in 5 seconds…")**

**time.sleep(5)**

**total = len(questions)**

**for qi, q in enumerate(questions):**

**# clear previous answers**

**with clients\_lock:**

**for c in clients:**

**c['event'].clear()**

**c['last\_ans'] = None**

**# broadcast question**

**print(f"[SERVER DEBUG] Broadcasting Q{qi+1}: {q['question']}")**

**broadcast({**

**'action': 'question',**

**'question\_index': qi,**

**'total\_questions': total,**

**'question': q['question'],**

**'options': q['options'],**

**'time\_limit': q.get('time\_limit', 20)**

**})**

**# wait up to time\_limit or until all answered**

**deadline = time.time() + q.get('time\_limit', 20)**

**while time.time() < deadline:**

**with clients\_lock:**

**if all(c['event'].is\_set() for c in clients):**

**break**

**time.sleep(0.1)**

**# score the answers**

**with clients\_lock:**

**for c in clients:**

**ans = c['last\_ans']**

**if ans and ans[0] == qi and ans[1] == q['answer']:**

**c['score'] += 1**

**# broadcast interim leaderboard**

**board = sorted([(c['username'], c['score']) for c in clients],**

**key=lambda x: -x[1])**

**broadcast({'action':'leaderboard','scores':board})**

**# final results → broadcast + persist + console print**

**final = [(c['username'], c['score']) for c in clients]**

**broadcast({'action':'final','scores': final})**

**print("[SERVER RESULT] Final Scores:")**

**for user, score in final:**

**print(f" • {user}: {score}")**

**cur.execute(**

**"INSERT INTO results(username,score,taken\_at) VALUES(?,?,?)",**

**(user, score, datetime.now().isoformat())**

**)**

**db.commit()**

**print("[SERVER DEBUG] Quiz complete – results saved to quiz.db")**

**def start\_server():**

**threading.Thread(target=quiz\_manager, daemon=True).start()**

**srv = socket.socket()**

**srv.bind((HOST, PORT))**

**srv.listen()**

**print(f"[SERVER DEBUG] Listening on {HOST}:{PORT}")**

**while True:**

**conn, addr = srv.accept()**

**threading.Thread(target=handle\_client,**

**args=(conn, addr),**

**daemon=True).start()**

**if \_\_name\_\_=='\_\_main\_\_':**

**start\_server()**

**py client.py**

**import socket**

**import json**

**import threading**

**import tkinter as tk**

**from tkinter import messagebox**

**HOST, PORT = '127.0.0.1', 9999**

**class QuizClientApp:**

**def \_\_init\_\_(self, root):**

**self.root = root**

**root.title("🌈 Quiz System 🌈")**

**# — Window size & center —**

**win\_w, win\_h = 900, 740**

**scr\_w = root.winfo\_screenwidth()**

**scr\_h = root.winfo\_screenheight()**

**x = (scr\_w - win\_w) // 2**

**y = (scr\_h - win\_h) // 2**

**root.geometry(f"{win\_w}x{win\_h}+{x}+{y}")**

**root.configure(bg='#F0F8FF') # AliceBlue**

**self.build\_login()**

**def build\_login(self):**

**self.login\_frame = tk.Frame(self.root,**

**bg='#FFDAB9', # PeachPuff**

**bd=2, relief='ridge',**

**padx=30, pady=30)**

**self.login\_frame.place(relx=0.5, rely=0.5, anchor='c')**

**tk.Label(self.login\_frame,**

**text="Enter Your Username",**

**font=('Georgia', 18, 'bold'),**

**bg='#FFDAB9', fg='#4B0082')\**

**.grid(row=0, column=0, columnspan=2, pady=(0,15))**

**tk.Label(self.login\_frame,**

**text="Username:",**

**font=('Arial', 14),**

**bg='#FFDAB9')\**

**.grid(row=1, column=0, sticky='e', padx=(0,10))**

**self.username\_entry = tk.Entry(self.login\_frame,**

**font=('Arial', 14))**

**self.username\_entry.grid(row=1, column=1, padx=(10,0))**

**self.username\_entry.focus()**

**tk.Button(self.login\_frame,**

**text="Join Quiz",**

**font=('Arial', 14, 'bold'),**

**bg='#8A2BE2', fg='white',**

**activebackground='#4B0082',**

**padx=15, pady=8,**

**command=self.on\_join)\**

**.grid(row=2, column=0, columnspan=2, pady=(20,0))**

**def on\_join(self):**

**self.username = self.username\_entry.get().strip()**

**if not self.username:**

**messagebox.showwarning("Input Error", "Please enter a username.")**

**return**

**# Remove login, build full UI**

**self.login\_frame.destroy()**

**self.build\_banner()**

**self.build\_quiz\_frame()**

**self.build\_footer()**

**self.connect\_to\_server()**

**def build\_banner(self):**

**# Top banner**

**self.banner = tk.Frame(self.root, bg='#FFD700', height=80) # Gold**

**self.banner.pack(fill='x')**

**tk.Label(self.banner,**

**text="Quiz System by Student Drive Academy",**

**font=('Georgia', 24, 'bold'),**

**bg='#FFD700', fg='#8B0000')\**

**.pack(pady=(10,0))**

**# Username display**

**tk.Label(self.banner,**

**text=f"Player: {self.username}",**

**font=('Arial', 12, 'italic'),**

**bg='#FFD700', fg='#000080')\**

**.pack(pady=(0,8))**

**def build\_quiz\_frame(self):**

**# Main quiz area**

**self.quiz\_frame = tk.Frame(self.root,**

**bg='#E6E6FA', # Lavender**

**padx=20, pady=20)**

**self.quiz\_frame.pack(fill='both', expand=True)**

**# Question label**

**self.question\_lbl = tk.Label(self.quiz\_frame,**

**text="",**

**font=('Arial', 16, 'bold'),**

**wraplength=800,**

**justify='left',**

**bg='#E6E6FA',**

**fg='#2F4F4F')**

**self.question\_lbl.pack(pady=(0,20))**

**# Options**

**opts\_frame = tk.LabelFrame(self.quiz\_frame,**

**text="Options",**

**font=('Arial', 12, 'bold'),**

**bg='#FFFACD', # LemonChiffon**

**fg='#8B4513', # SaddleBrown**

**padx=10, pady=10)**

**opts\_frame.pack(fill='x', pady=(0,20))**

**self.radio\_var = tk.IntVar(value=-1)**

**self.opts = []**

**for i in range(4):**

**rb = tk.Radiobutton(opts\_frame,**

**text="",**

**variable=self.radio\_var,**

**value=i,**

**font=('Arial', 14),**

**bg='#FFFACD',**

**activebackground='#FAFAD2',**

**selectcolor='#FFEFD5')**

**rb.pack(anchor='w', pady=5)**

**self.opts.append(rb)**

**# Controls (timer, progress, submit)**

**ctrl = tk.Frame(self.quiz\_frame, bg='#E6E6FA')**

**ctrl.pack(fill='x', pady=(0,20))**

**self.timer\_lbl = tk.Label(ctrl,**

**text="Time left: --",**

**font=('Arial', 12),**

**bg='#E6E6FA')**

**self.timer\_lbl.pack(side='left')**

**# Progress bar (canvas)**

**self.prog\_canvas = tk.Canvas(ctrl,**

**width=400, height=20,**

**bg='#D3D3D3',**

**highlightthickness=0)**

**self.prog\_canvas.pack(side='left', padx=10)**

**self.prog\_bar = self.prog\_canvas.create\_rectangle(**

**0,0,0,20, fill='#7FFFD4') # Aquamarine**

**# Submit button**

**self.submit\_btn = tk.Button(ctrl,**

**text="Submit",**

**font=('Arial', 12, 'bold'),**

**bg='#32CD32', fg='white',**

**activebackground='#228B22',**

**state='disabled',**

**padx=15, pady=5,**

**command=self.on\_submit)**

**self.submit\_btn.pack(side='right')**

**# Leaderboard**

**tk.Label(self.quiz\_frame,**

**text="Leaderboard",**

**font=('Arial', 14, 'underline'),**

**bg='#E6E6FA')\**

**.pack()**

**self.lb = tk.Listbox(self.quiz\_frame,**

**font=('Arial', 12),**

**bg='#F5FFFA', # MintCream**

**fg='#006400', # DarkGreen**

**height=6)**

**self.lb.pack(fill='x', padx=100, pady=(5,0))**

**def build\_footer(self):**

**# Bottom footer**

**self.footer = tk.Label(self.root,**

**text=("developed by himanshu singh @sv infotech kanpur "**

**"mail: himanshusingh1814@gmail.com"),**

**font=('Arial', 10),**

**bg='#D3D3D3',**

**fg='#333333')**

**self.footer.pack(side='bottom', fill='x')**

**def connect\_to\_server(self):**

**self.sock = socket.socket()**

**try:**

**self.sock.connect((HOST, PORT))**

**self.sock\_file = self.sock.makefile('r')**

**self.send({'action':'auth','username':self.username})**

**resp = json.loads(self.sock\_file.readline())**

**if resp.get('status')!='ok':**

**raise RuntimeError(resp.get('msg','Auth failed'))**

**except Exception as e:**

**messagebox.showerror("Connection Error", str(e))**

**self.root.destroy()**

**return**

**threading.Thread(target=self.listen, daemon=True).start()**

**def send(self, msg):**

**self.sock.sendall((json.dumps(msg)+'\n').encode())**

**def listen(self):**

**for line in self.sock\_file:**

**m = json.loads(line.strip())**

**act = m.get('action')**

**if act=='question':**

**self.root.after(0, lambda msg=m: self.on\_question(msg))**

**elif act=='leaderboard':**

**self.root.after(0, lambda msg=m: self.on\_leaderboard(msg))**

**elif act=='final':**

**self.root.after(0, lambda msg=m: self.on\_final(msg))**

**break**

**self.sock.close()**

**def on\_question(self, m):**

**self.current = m**

**idx, tot = m['question\_index'], m['total\_questions']**

**self.question\_lbl.config(text=f"Q{idx+1}/{tot}: {m['question']}")**

**for i, opt in enumerate(m['options']):**

**self.opts[i].config(text=opt)**

**self.radio\_var.set(-1)**

**self.submit\_btn.config(state='normal')**

**# reset & start timer/progress**

**self.time\_left = m.get('time\_limit',20)**

**self.update\_progress()**

**self.countdown()**

**def update\_progress(self):**

**idx, tot = self.current['question\_index'], self.current['total\_questions']**

**width = 400 \* (idx+1)/tot**

**self.prog\_canvas.coords(self.prog\_bar, 0,0, width,20)**

**def countdown(self):**

**if self.time\_left >= 0:**

**self.timer\_lbl.config(text=f"Time left: {self.time\_left}s")**

**self.time\_left -= 1**

**self.root.after(1000, self.countdown)**

**else:**

**self.submit\_btn.config(state='disabled')**

**def on\_submit(self):**

**choice = self.radio\_var.get()**

**if choice<0:**

**messagebox.showwarning("No Selection", "Please pick an option.")**

**return**

**self.send({**

**'action':'answer',**

**'question\_index': self.current['question\_index'],**

**'choice': choice**

**})**

**self.submit\_btn.config(state='disabled')**

**def on\_leaderboard(self, m):**

**self.lb.delete(0, 'end')**

**for user, sc in m.get('scores', []):**

**self.lb.insert('end', f"{user}: {sc}")**

**def on\_final(self, m):**

**final = m.get('scores', [])**

**your = next((s for u,s in final if u==self.username), None)**

**board = "\n".join(f"{u}: {s}" for u,s in final)**

**messagebox.showinfo("Quiz Over",**

**f"Your score: {your}\n\nFinal leaderboard:\n{board}")**

**self.root.destroy()**

**if \_\_name\_\_ == '\_\_main\_\_':**

**root = tk.Tk()**

**app = QuizClientApp(root)**

**root.mainloop()**

**py admin\_client.py**

**import sys, socket, json**

**if len(sys.argv)!=2:**

**print("Usage: python admin\_client.py questions.json")**

**sys.exit(1)**

**with open(sys.argv[1],'r') as f:**

**questions = json.load(f)**

**HOST, PORT = '127.0.0.1', 9999**

**ADMIN\_PASS = 'supersecret'**

**sock = socket.socket()**

**sock.connect((HOST, PORT))**

**f = sock.makefile('r')**

**# Authenticate as admin**

**sock.sendall((json.dumps({**

**'action':'auth','role':'admin','password':ADMIN\_PASS**

**})+'\n').encode())**

**print("Auth:", json.loads(f.readline()))**

**# Load questions**

**sock.sendall((json.dumps({**

**'action':'load','questions': questions**

**})+'\n').encode())**

**print("Load:", json.loads(f.readline()))**

**# Start quiz**

**sock.sendall((json.dumps({**

**'action':'start'**

**})+'\n').encode())**

**print("Start:", json.loads(f.readline()))**

**sock.close()**

**questions.json**

**[**

**{**

**"question": "What is the capital of France?",**

**"options": ["Paris", "London", "Berlin", "Madrid"],**

**"answer": 0,**

**"time\_limit": 15**

**},**

**{**

**"question": "2 + 2 = ?",**

**"options": ["3", "4", "5", "2"],**

**"answer": 1,**

**"time\_limit": 10**

**},**

**{**

**"question": "Which language is this project written in?",**

**"options": ["Java", "C++", "Python", "Ruby"],**

**"answer": 2,**

**"time\_limit": 12**

**},**

**{**

**"question": "Which of the following creates a generator object?",**

**"options": [**

**"[x\*x for x in range(5)]",**

**"(x\*x for x in range(5))",**

**"{x\*x for x in range(5)}",**

**"dict((x, x\*x) for x in range(5))"**

**],**

**"answer": 1,**

**"time\_limit": 15**

**},**

**{**

**"question": "What is the output of: a, \*b, c = [1, 2, 3, 4, 5]\\nprint(b)",**

**"options": ["[1, 2, 3, 4]", "[2, 3, 4]", "(2, 3, 4)", "([2, 3, 4],)"],**

**"answer": 1,**

**"time\_limit": 15**

**},**

**{**

**"question": "What does the @staticmethod decorator do?",**

**"options": [**

**"Binds method to the class, not the instance",**

**"Allows method to access instance data",**

**"Automatically converts method to class method",**

**"Makes method run in a separate thread"**

**],**

**"answer": 0,**

**"time\_limit": 15**

**},**

**{**

**"question": "Which method must a class implement to make its instances iterable?",**

**"options": ["\_\_getitem\_\_", "\_\_iter\_\_", "\_\_call\_\_", "\_\_next\_\_"],**

**"answer": 1,**

**"time\_limit": 15**

**},**

**{**

**"question": "How can you define a read-only property in a class?",**

**"options": [**

**"Use @property without a setter",**

**"Define \_\_get\_\_ only in a descriptor",**

**"Use readonly keyword",**

**"Both A and B"**

**],**

**"answer": 3,**

**"time\_limit": 15**

**},**

**{**

**"question": "What’s the difference between is and ==?",**

**"options": [**

**"== checks identity, is checks equality",**

**"is checks identity, == checks equality",**

**"No difference",**

**"is works only for numbers"**

**],**

**"answer": 1,**

**"time\_limit": 15**

**},**

**{**

**"question": "Which exception is raised by int(\\\"xyz\\\")?",**

**"options": ["ValueError", "TypeError", "SyntaxError", "KeyError"],**

**"answer": 0,**

**"time\_limit": 15**

**},**

**{**

**"question": "What does this slice return?: s = \\\"abcdef\\\"\\ns[1:5:2]",**

**"options": ["bdf", "bd", "ace", "bcde"],**

**"answer": 1,**

**"time\_limit": 15**

**}**

**]**

**quiz.db will initiate automatically**

first run **py server.py**

then **py client.py** in other terminal

**1. What Is Network Programming?**

At its heart, **network programming** means writing code that lets one computer talk to another over a network. You’ll deal with:

* **Protocols** (rules for formatting and exchanging data): TCP, UDP, HTTP, etc.
* **Client–Server architecture**: one side listens (“server”) and one side connects (“client”).
* **Sockets**: the programming abstraction that represents an endpoint for sending/receiving data.

**2. Prerequisites**

* Python 3.x installed
* Basic familiarity with Python syntax
* (Optional) Command-line comfort

**3. The Python socket Module**

Python’s builtin [socket](https://docs.python.org/3/library/socket.html) module gives you low-level access to the BSD socket API.

**import socket**

Key classes/functions:

* socket.socket(family, type) — create a socket object
* bind(address) — attach a server socket to a local address (host, port)
* listen(backlog) — start listening for connections (TCP only)
* accept() — accept an incoming TCP connection
* connect(address) — initiate a connection from a client
* send() / recv() — send and receive bytes
* sendto() / recvfrom() — send/receive with a specified address (UDP)
* close() — close the socket

**4. A Simple TCP Echo Server & Client**

**4.1 TCP Server**

**# tcp\_echo\_server.py**

**import socket**

**HOST = '127.0.0.1' # localhost**

**PORT = 65432 # arbitrary non-privileged port**

**with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as serv:**

**serv.bind((HOST, PORT))**

**serv.listen() # backlog defaults to a reasonable value**

**print(f"Server listening on {HOST}:{PORT}")**

**conn, addr = serv.accept() # wait for a client**

**with conn:**

**print('Connected by', addr)**

**while True:**

**data = conn.recv(1024) # receive up to 1024 bytes**

**if not data:**

**break**

**conn.sendall(data) # echo it back**

**4.2 TCP Client**

**# tcp\_echo\_client.py**

**import socket**

**HOST = '127.0.0.1'**

**PORT = 65432**

**with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:**

**sock.connect((HOST, PORT))**

**for msg in [b'Hello', b'World', b'!']:**

**print(f"Sending: {msg!r}")**

**sock.sendall(msg)**

**data = sock.recv(1024)**

**print(f"Received: {data!r}")**

**Run** the server first (python tcp\_echo\_server.py), then the client.

**5. UDP “Echo” Server & Client**

UDP is connection-less and doesn’t guarantee delivery.

**5.1 UDP Server**

**# udp\_echo\_server.py**

**import socket**

**HOST, PORT = '127.0.0.1', 65433**

**with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as serv:**

**serv.bind((HOST, PORT))**

**print(f"UDP server on {HOST}:{PORT}")**

**while True:**

**data, addr = serv.recvfrom(1024)**

**print(f"Received {data!r} from {addr}")**

**serv.sendto(data, addr) # echo back**

**5.2 UDP Client**

**# udp\_echo\_client.py**

**import socket**

**HOST, PORT = '127.0.0.1', 65433**

**with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as sock:**

**for msg in [b'Foo', b'Bar']:**

**sock.sendto(msg, (HOST, PORT))**

**data, \_ = sock.recvfrom(1024)**

**print(f"Echoed back: {data!r}")**

**6. Handling Multiple Clients**

A single-threaded server handles one client at a time. To support many clients:

1. **Threading**: spawn a new threading.Thread for each connection.
2. **selectors module**: multiplex many sockets in a single thread (more scalable).

**6.1 Threaded TCP Server (overview)**

**import socket, threading**

**def handle\_client(conn, addr):**

**with conn:**

**print('Client', addr)**

**while True:**

**data = conn.recv(1024)**

**if not data: break**

**conn.sendall(data)**

**with socket.socket() as serv:**

**serv.bind(('0.0.0.0', 65432))**

**serv.listen()**

**while True:**

**conn, addr = serv.accept()**

**threading.Thread(target=handle\_client, args=(conn, addr), daemon=True).start()**

**7. Higher-Level Helpers**

**7.1 socketserver Module**

**The** [**socketserver**](https://docs.python.org/3/library/socketserver.html) **module provides classes to simplify servers:**

**from socketserver import ThreadingTCPServer, StreamRequestHandler**

**class EchoHandler(StreamRequestHandler):**

**def handle(self):**

**for line in self.rfile:**

**self.wfile.write(line)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**with ThreadingTCPServer(('0.0.0.0', 9999), EchoHandler) as server:**

**server.serve\_forever()**

**7.2 asyncio & Async IO**

Modern async approach (Python 3.7+):

**import asyncio**

**async def handle\_echo(reader, writer):**

**data = await reader.read(100)**

**writer.write(data)**

**await writer.drain()**

**writer.close()**

**async def main():**

**server = await asyncio.start\_server(handle\_echo, '127.0.0.1', 8888)**

**async with server:**

**await server.serve\_forever()**

**asyncio.run(main())**

**8. Practical Tips & Next Steps**

* **Keep data small** (chunk sizes of 1 KB–4 KB often work well).
* **Handle exceptions** (ConnectionResetError, timeouts).
* **Use timeouts** with sock.settimeout() to avoid hangs.
* **Security**: never trust incoming data; sanitize and validate.
* **Explore higher-level protocols**: HTTP with requests/aiohttp, FTP with ftplib, email with smtplib/imaplib.
* **Learn packet-level** programming with scapy for advanced network tasks.

**Summary**

1. Learn basic **socket API**: socket(), bind(), listen(), accept(), connect(), send()/recv().
2. Practice both **TCP** and **UDP** echo servers/clients.
3. Scale to multiple clients with **threads** or **selectors**.
4. Try the **socketserver** and **asyncio** libraries for cleaner code.
5. Move on to real-world protocols (HTTP, FTP, SMTP) and packet crafting.

With these fundamentals, you’ll have the building blocks to create chat servers, file transfer tools, custom network services—and understand how any networked application under the hood moves bytes around. Happy coding

**EXAMPLE:**

Below is a pair of Python scripts—for the server and the client—that together implement a simple, production-quality, CLI-based chat system. They support multiple clients, clean startup/shutdown, logging, argument parsing, and basic error handling.

**server.py**

**#!/usr/bin/env python3**

**"""**

**server.py**

**A multi-client chat server. Accepts TCP connections, broadcasts messages**

**from any client to all others, and handles clean shutdown on SIGINT/SIGTERM.**

**"""**

**import argparse**

**import logging**

**import signal**

**import socket**

**import sys**

**import threading**

**from typing import List, Tuple**

**logging.basicConfig(**

**level=logging.INFO,**

**format='[%(asctime)s] %(levelname)s: %(message)s',**

**datefmt='%Y-%m-%d %H:%M:%S'**

**)**

**class ChatServer:**

**def \_\_init\_\_(self, host: str, port: int):**

**self.address: Tuple[str, int] = (host, port)**

**self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**

**self.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)**

**self.clients: List[socket.socket] = []**

**self.clients\_lock = threading.Lock()**

**self.running = False**

**def start(self) -> None:**

**"""Bind, listen, and start accepting clients."""**

**self.server\_socket.bind(self.address)**

**self.server\_socket.listen()**

**self.running = True**

**logging.info(f"Chat server listening on {self.address[0]}:{self.address[1]}")**

**signal.signal(signal.SIGINT, self.\_shutdown\_signal)**

**signal.signal(signal.SIGTERM, self.\_shutdown\_signal)**

**try:**

**while self.running:**

**conn, addr = self.server\_socket.accept()**

**logging.info(f"Connection from {addr}")**

**with self.clients\_lock:**

**self.clients.append(conn)**

**threading.Thread(**

**target=self.\_handle\_client,**

**args=(conn, addr),**

**daemon=True**

**).start()**

**finally:**

**self.\_cleanup()**

**def \_handle\_client(self, conn: socket.socket, addr: Tuple[str, int]) -> None:**

**"""Receive messages from one client and broadcast them."""**

**try:**

**with conn:**

**while True:**

**data = conn.recv(1024)**

**if not data:**

**break**

**message = f"{addr[0]}:{addr[1]}> {data.decode().strip()}"**

**logging.info(f"Broadcasting: {message}")**

**self.\_broadcast(message, exclude=conn)**

**except ConnectionResetError:**

**logging.warning(f"Connection lost from {addr}")**

**finally:**

**with self.clients\_lock:**

**if conn in self.clients:**

**self.clients.remove(conn)**

**logging.info(f"Client {addr} disconnected")**

**def \_broadcast(self, message: str, exclude: socket.socket) -> None:**

**"""Send message to all clients except the sender."""**

**with self.clients\_lock:**

**for client in list(self.clients):**

**if client is not exclude:**

**try:**

**client.sendall((message + "\n").encode())**

**except Exception:**

**logging.exception("Error sending to client, removing")**

**self.clients.remove(client)**

**def \_shutdown\_signal(self, signum, frame) -> None:**

**"""Signal handler to stop the server loop."""**

**logging.info(f"Shutdown signal ({signum}) received")**

**self.running = False**

**def \_cleanup(self) -> None:**

**"""Close all client connections and the server socket."""**

**logging.info("Shutting down server, closing connections")**

**with self.clients\_lock:**

**for client in self.clients:**

**try:**

**client.shutdown(socket.SHUT\_RDWR)**

**client.close()**

**except Exception:**

**pass**

**self.clients.clear()**

**self.server\_socket.close()**

**logging.info("Server shutdown complete")**

**def parse\_args() -> argparse.Namespace:**

**parser = argparse.ArgumentParser(description="Multi-client chat server")**

**parser.add\_argument(**

**"--host", "-H",**

**default="0.0.0.0",**

**help="Interface to bind (default: all interfaces)"**

**)**

**parser.add\_argument(**

**"--port", "-p",**

**type=int,**

**default=5000,**

**help="Port number to listen on (default: 5000)"**

**)**

**return parser.parse\_args()**

**def main() -> None:**

**args = parse\_args()**

**server = ChatServer(args.host, args.port)**

**server.start()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**client.py**

**#!/usr/bin/env python3**

**"""**

**client.py**

**A CLI chat client. Connects to the chat server, reads user input from stdin,**

**sends it, and concurrently listens for incoming messages to display.**

**"""**

**import argparse**

**import logging**

**import socket**

**import sys**

**import threading**

**from typing import Tuple**

**logging.basicConfig(**

**level=logging.INFO,**

**format='[%(asctime)s] %(levelname)s: %(message)s',**

**datefmt='%Y-%m-%d %H:%M:%S'**

**)**

**def receive\_messages(sock: socket.socket) -> None:**

**"""Continuously read from the socket and print to stdout."""**

**try:**

**while True:**

**data = sock.recv(1024)**

**if not data:**

**logging.info("Server closed the connection")**

**break**

**print(data.decode(), end='')**

**except Exception:**

**logging.exception("Error receiving data")**

**finally:**

**sock.close()**

**sys.exit(0)**

**def parse\_args() -> argparse.Namespace:**

**parser = argparse.ArgumentParser(description="CLI chat client")**

**parser.add\_argument(**

**"host",**

**help="Server hostname or IP address"**

**)**

**parser.add\_argument(**

**"port",**

**type=int,**

**help="Server port number"**

**)**

**return parser.parse\_args()**

**def main() -> None:**

**args = parse\_args()**

**server\_addr: Tuple[str, int] = (args.host, args.port)**

**sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**

**try:**

**sock.connect(server\_addr)**

**except Exception as e:**

**logging.error(f"Could not connect to {server\_addr}: {e}")**

**sys.exit(1)**

**logging.info(f"Connected to chat server at {server\_addr}")**

**threading.Thread(target=receive\_messages, args=(sock,), daemon=True).start()**

**try:**

**for line in sys.stdin:**

**message = line.strip()**

**if message.lower() == "/quit":**

**break**

**sock.sendall((message + "\n").encode())**

**except KeyboardInterrupt:**

**logging.info("Interrupted by user")**

**finally:**

**logging.info("Closing connection")**

**sock.close()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**How to Use**

1. **Start the server** on a machine (or container) accessible to clients:

bash

CopyEdit

python3 server.py --host 0.0.0.0 --port 5000

1. **Run one or more clients** from any terminal:

bash

CopyEdit

python3 client.py 192.168.1.100 5000

1. **Chat** by typing messages and pressing Enter.
   * Incoming messages appear with the sender’s IP and port.
   * To exit the client, type /quit or press Ctrl+C.

**Production-Level Features**

* **Argument parsing** via argparse
* **Structured logging** (timestamps, levels)
* **Graceful shutdown** with signal handling (server) and clean client exit
* **Thread-based concurrency** to support multiple clients
* **Error handling** around network operations

You can extend this further by adding user nicknames, authentication, encryption (TLS), or a GUI—but this foundation gives you a robust, maintainable CLI chat application ready for production use.