Day 5 - Session 2:
Socket Programming

This notebook introduces **Socket Programming** in Python. Sockets are a fundamental way for programs to communicate over a network. Think of them as the "phone lines" that connect two programs on different computers (or even on the same computer) to send and receive data. This is crucial for real-time data or custom device communication.

1. Introduction to Sockets and TCP/IP

Imagine two programs want to talk to each other over a network. They need a way to connect and send messages.

- **Socket:** This is the "endpoint" or "doorway" that a program uses to send and receive data. It's like one end of a phone connection.
- **TCP/IP (Transmission Control Protocol / Internet Protocol):** These are the rules (protocols) that make sure network communication happens reliably.
    - **IP:** Helps find the right computer (like an address).
    - **TCP: Makes sure data arrives correctly** and in the right order, like a guaranteed mail service. If a piece of data is lost, TCP resends it.

Why Use Sockets (TCP) in Engineering/Science?

- **Direct Communication:** For devices that need to talk directly, without a complex web API.
- **Real-time Data:** Sending continuous streams of sensor data very efficiently.
- **Custom Needs:** When you need a very specific way for devices to communicate.

2. Basic TCP Client-Server: Sending a Simple Message

A common way programs communicate with sockets is a client-server model.

- The **Server** program waits for incoming connections (like a phone waiting for a call).
- The **Client** program starts a connection to the server (like making a phone call).

We will build the simplest possible server and client to send a "Hello" message.

How to Run (Important!):

You **must** save the server and client code into two separate .py files and run them in two separate terminal windows. You cannot run both parts as live servers/clients directly in a single Jupyter notebook.

Bash Commands to Run (Requires 2 separate terminal windows):

1. **Terminal 1 (for Server):**
   - Save the tcp_simple_server.py code.
   - Navigate to its directory.
   - Run: python tcp_simple_server.py
   - You should see: Server listening on 127.0.0.1:65432
2. **Terminal 2 (for Client - while Server is running):**
   - Save the tcp_simple_client.py code.
   - Navigate to its directory.
   - Run: python tcp_simple_client.py
   - You will see client messages and server responses.

2.1 Server Side: Waiting and Responding

The server opens a socket, sets an address (IP and Port), waits for a client to connect, receives a message, sends a reply, and then closes the connection.

File: tcp_simple_server.py

Python
```python
# Save this code as tcp_simple_server.py

import socket  # Import Python's built-in socket module for networking

# ========================
# --- Server Settings ---
# ========================
HOST = '127.0.0.1'   # Loopback address (localhost). Only accessible from the same machine.
PORT = 65432         # Arbitrary non-privileged port (>1023). Avoid ports already in use.
```

```
print("--- TCP Server: Simple Example ---")

# ========================================
# 1. Create a TCP/IP socket
# ========================================
# AF_INET      : Address family for IPv4
# SOCK_STREAM  : Socket type for TCP (connection-oriented, reliable)
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("✅ Server socket created.")

# ========================================
# 2. Bind the socket to a specific IP and port
# ========================================
# This tells the operating system that this socket will listen on HOST:PORT
server_socket.bind((HOST, PORT))
print(f"✅ Socket bound to {HOST}:{PORT}")

# ========================================
# 3. Start listening for incoming connections
# ========================================
# The argument (1) specifies the max number of queued connections (backlog)
server_socket.listen(1)
print("👂 Server listening for incoming connections...")

# ========================================
# 4. Accept a connection (Blocking call)
# ========================================
# This call blocks until a client connects.
# Returns a new socket `conn` to communicate with the client,
# and `addr` contains the client's address.
conn, addr = server_socket.accept()
print(f"🔗 Connection accepted from client: {addr}")

# ========================================
# 5. Receive data from the client
# ========================================
# We expect to receive up to 1024 bytes.
# Data is received as bytes, so we decode to string using UTF-8.
data_bytes = conn.recv(1024)
client_message = data_bytes.decode('utf-8')
print(f"📨 Received from client: '{client_message}'")
```

```python
# ========================================
# 6. Send a response back to the client
# ========================================
# Send a confirmation message.
# We encode the string to bytes before sending.
response = "Hello from server! I got your message."
conn.sendall(response.encode('utf-8'))
print("📤 Response sent to client.")

# ========================================
# 7. Close the client socket
# ========================================
conn.close()
print("❌ Client connection closed.")

# ========================================
# 8. Close the main server socket
# ========================================
server_socket.close()
print("🛑 Server stopped. Socket closed.")
```

2.2 Client Side: Connecting and Sending

The client creates a socket, connects to the server's address, sends a message, receives a reply, and then closes its connection.

File: tcp_simple_client.py

Python

```python
# Save this code as tcp_simple_client.py

import socket  # Import Python's built-in socket module

# ========================
# --- Client Settings ---
```

```python
# ========================
HOST = '127.0.0.1'  # IP address of the server (localhost = same machine)
PORT = 65432        # Port number the server is listening on

print("--- TCP Client: Simple Example ---")

# ========================================
# 1. Create a TCP/IP socket
# ========================================
# AF_INET      : Address family for IPv4
# SOCK_STREAM  : Socket type for TCP (reliable stream-based connection)
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("✅ Client socket created.")

try:
    # ========================================
    # 2. Connect to the TCP server
    # ========================================
    # Attempts to make a connection to the server at HOST:PORT
    client_socket.connect((HOST, PORT))
    print(f"🔌 Connected to server at {HOST}:{PORT}")

    # ========================================
    # 3. Send data to the server
    # ========================================
    # All data sent over TCP must be bytes, so we encode the string
    message_to_send = "Hello server, this is the client!"
    client_socket.sendall(message_to_send.encode('utf-8'))
    print(f"📤 Client sent: '{message_to_send}'")

    # ========================================
    # 4. Receive a response from the server
    # ========================================
    # Try to receive up to 1024 bytes of response
    data_bytes = client_socket.recv(1024)
    server_response = data_bytes.decode('utf-8')  # Decode from bytes to string
    print(f"📨 Client received: '{server_response}'")

except ConnectionRefusedError:
    print("❗ Connection refused: Is the server running? Check address/port.")
except Exception as e:
    print(f"❗ An unexpected error occurred: {e}")
```

```
finally:
    # ========================================
    # 5. Close the client socket
    # ========================================
    client_socket.close()
    print("❌ Client socket closed.")
```

3. Structured Message Exchange (Sending/Receiving Dictionaries)

Sending just plain text is often not enough. In engineering, you need to send structured data like sensor readings with an ID, value, and unit. **JSON** is perfect for this because it's human-readable and easily converts to/from Python dictionaries.

Conceptual Approach:

- **Before Sending (Client):** Convert your Python dictionary -> JSON string (json.dumps()) -> bytes (.encode()).
- **After Receiving (Server):** Convert bytes -> JSON string (.decode()) -> Python dictionary (json.loads()).

Key Idea: Sockets only understand bytes. JSON helps us put structured data into those bytes.

```python
# Save this file as tcp_json_server.py

import socket
import json  # For encoding/decoding structured JSON data

# ========================
# --- Server Settings ---
# ========================
HOST = '127.0.0.1'  # Localhost (your machine)
PORT = 65434        # Use a different port than previous examples
```

```python
# 1. Create the server socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))        # Bind to address
server_socket.listen(1)                 # Wait for 1 client
print(f"📡 JSON Server listening on {HOST}:{PORT}...")

# 2. Accept a connection
conn, addr = server_socket.accept()
print(f"🔗 Connection accepted from: {addr}")

# 3. Receive data
data_bytes = conn.recv(1024)
received_json_str = data_bytes.decode('utf-8')  # Convert bytes to string
print(f"📨 Received raw JSON string: {received_json_str}")

# 4. Parse and respond
try:
    received_data_dict = json.loads(received_json_str)  # Parse string to
dictionary
    print("✅ Parsed Data:")
    print(f"   Sensor ID : {received_data_dict.get('sensor_id')}")
    print(f"   Value     : {received_data_dict.get('value')}")
    print(f"   Timestamp : {received_data_dict.get('timestamp')}")

    # Respond with confirmation
    response_dict = {"status": "success", "message": "Data received and
parsed!"}
    conn.sendall(json.dumps(response_dict).encode('utf-8'))

except json.JSONDecodeError:
    print("❌ Error: Received invalid JSON.")
    error_response = {"status": "error", "message": "Invalid JSON format."}
    conn.sendall(json.dumps(error_response).encode('utf-8'))

# 5. Clean up
conn.close()
server_socket.close()
print("🔚 Server stopped.")
```

Python
```python
# Save this file as tcp_json_client.py

import socket
import json  # For structured data format

# ========================
# --- Client Settings ---
# ========================
HOST = '127.0.0.1'  # Server IP (same as server)
PORT = 65434        # Same port as server

# 1. Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # 2. Connect to the server
    client_socket.connect((HOST, PORT))
    print(f"✅ Connected to JSON server at {HOST}:{PORT}")

    # 3. Prepare structured sensor data
    sensor_data = {
        "sensor_id": "TEMP_001",
        "value": 28.5,
        "unit": "Celsius",
        "timestamp": "2025-07-01T14:30:00"
    }

    # Convert to JSON string and send
    json_message_str = json.dumps(sensor_data)
    client_socket.sendall(json_message_str.encode('utf-8'))
    print(f"📤 Client sent structured data: {json_message_str}")

    # 4. Receive and decode response
    response_bytes = client_socket.recv(1024)
    response_json_str = response_bytes.decode('utf-8')
    response_dict = json.loads(response_json_str)
    print(f"📥 Client received structured response: {response_dict}")

except ConnectionRefusedError:
    print("❗ Connection refused: Is the JSON server running?")
except Exception as e:
    print(f"❗ Unexpected error: {e}")
```

```
finally:
    client_socket.close()
    print("❌ Client socket closed.")
```

4. Protocol Design (Binary vs. JSON)

When you design how programs talk over sockets, you choose a "protocol" (the rules for sending data).

4.1 JSON Protocol (Human-Friendly Text)

- **How it works:** Data is sent as human-readable text (JSON strings).
- **Good for:** When it's easy for people to read, or when data structures change often. APIs often use JSON.
- **Think of it:** Sending a detailed, labeled message in plain English.
- **Example (recap):** {"temp": 25.5, "unit": "C"}

4.2 Binary Protocol (Computer-Friendly Bytes)

- **How it works:** Data is sent as raw bytes (numbers in their most compact computer form). Not human-readable without a special tool.
- **Good for:** Very high-speed data, very small messages, or devices with limited memory/power.
- **Think of it:** Sending data as raw electrical signals or highly compressed codes.
- **Python Tool:** The struct module helps convert Python numbers to/from raw bytes.

Python
```python
import struct
import json

print("\n--- Protocol Design: Binary vs. JSON ---")
```

```python
# ---------------------------
# JSON Example (Human-readable)
# ---------------------------
# This is how data is usually exchanged on the web — readable but slightly heavy.

json_data = {
    "id": "S1",        # String ID (2 characters)
    "val": 25.5        # Temperature value
}

# Convert the dictionary into a UTF-8 encoded byte string
json_bytes = json.dumps(json_data).encode('utf-8')

print(f"📝 JSON Data: {json_data}")
print(f"📦 JSON Bytes: {json_bytes} (Length: {len(json_bytes)} bytes)")

# ---------------------------
# Binary Protocol Example (Compact, not human-readable)
# ---------------------------
# Use struct to pack raw numbers efficiently into bytes.

sensor_id_bin = 101         # Integer (short)
temperature_bin = 28.75     # Float

# Struct format: '<h f' means:
# <  : Little-endian (byte order)
# h  : Short integer (2 bytes)
# f  : Float (4 bytes)
binary_packet = struct.pack('<h f', sensor_id_bin, temperature_bin)

print(f"\n🔧 Binary Data: Sensor ID = {sensor_id_bin}, Temperature = {temperature_bin}")
print(f"📦 Binary Bytes: {binary_packet} (Length: {len(binary_packet)} bytes)")

# Unpack the bytes back to Python values
unpacked_id, unpacked_temp = struct.unpack('<h f', binary_packet)

print(f"🔓 Unpacked Binary: ID = {unpacked_id}, Temp = {unpacked_temp}")

# ---------------------------
# Size Comparison
```

```
# ---------------------------
print("\n📏 Comparing data sizes:")
print(f"JSON Payload Length  : {len(json_bytes)} bytes")
print(f"Binary Payload Length: {len(binary_packet)} bytes")
```

5. Simulated Telemetry Transfer (Continuous Data Flow)

Imagine a sensor constantly sending data. This is a "telemetry transfer."

Conceptual Approach:

- A "Sender" program continuously creates and sends sensor data packets (e.g., JSON).
- A "Receiver" program continuously listens for, receives, and prints these packets.
- This needs more advanced server code (using threading) to handle continuous receiving without stopping.

Python

```python
# ----------------------------------------------
# Conceptual Simulation: Real-Time Telemetry Flow
# ----------------------------------------------

print("\n--- Simulated Telemetry Transfer ---")

# This is a simplified, non-executable example showing how sensor data might be continuously sent.
# In a real system, you'd use sockets + loops + threads/timers to achieve this.

print("📡 Imagine a Temperature Sensor sending data every second:")

# These are examples of telemetry data packets that might be transmitted
print("{'sensor_id': 'Temp001', 'value': 25.5, 'unit': 'C', 'timestamp': '2025-07-01T10:00:01'}")
print("{'sensor_id': 'Temp001', 'value': 25.7, 'unit': 'C', 'timestamp': '2025-07-01T10:00:02'}")
```

```python
print("{'sensor_id':  'Temp001',  'value':  25.6,  'unit':  'C',  'timestamp':
'2025-07-01T10:00:03'}")

print("\n🖥️ On the other side, a receiver logs each packet as it arrives...")

# Explanation of Use Case:
print("🔄 This continuous flow of data is typical in:")
print("   • Real-time sensor systems")
print("   • IoT telemetry streams")
print("   • Remote monitoring (e.g., spacecraft, industrial machinery)")

print("\n💡 In actual implementation, you'd use:")
print("   - A `while True` loop on both sender and receiver")
print("   - `socket.send()` / `recv()` calls")
print("   - Timestamps to track freshness of data")
print("   - Logging or storing incoming values for analytics")

# Note:
# This example is intentionally simple. Real-time data transfer needs:
# - Threading or AsyncIO (to handle continuous read/write)
# - Reliable connection (e.g., TCP or MQTT)
# - Timestamps for synchronization
# - Optional compression or binary protocol for efficiency
```

6. Key Takeaways

This session provided a hands-on introduction to network communication using sockets in Python.

- **What Sockets Are:** Learned that sockets are the software "endpoints" for network communication.
- **TCP/IP Basics:** Understood TCP as the reliable protocol that ensures data arrives correctly, and IP for addressing.
- **Client-Server Model:** Learned the basic roles of a server (listens, accepts) and a client (connects, sends).
- **Basic Socket Steps:** Understood the sequence of creating, binding, listening, accepting, connecting, sending, receiving, and closing sockets.

- **Data as Bytes:** Realized that all data sent over sockets is raw bytes, requiring .encode() and .decode() for strings.
- **Structured Data (JSON):** Learned how to send structured data (Python dictionaries) by converting them to JSON strings (json.dumps()) and back (json.loads()).
- **Protocol Choice:** Understood the trade-offs between JSON (human-readable, flexible) and Binary (compact, fast) protocols for different engineering needs.

By grasping these basic socket concepts, you can start to understand how devices and programs communicate at a fundamental level, which is valuable for real-time systems and custom network applications.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

📩 **For queries:** rajath@analogdata.ai| 📱 (+91) 96633 53992 | 🌐 https://analogdata.ai

---
