Networking in Python

Introduction

- A **socket** is one endpoint of a two-way communication link across a network. It's essentially the programming interface for network communication.
- In Python, the **socket module** exposes the standard BSD socket interface, allowing you to create programs (client and server) that communicate over the network.
- The term **BSD socket** refers to the **Application Programming Interface (API)** for network communication
  - Berkeley Software Distribution (BSD)
  - It is the **de facto standard API** for networking programming on the internet

Introduction

- The BSD socket API provides a set of function calls and structures that allow applications to create, connect, bind, listen, send, and receive data over networks using protocols like **TCP** and **UDP**.
- A BSD socket is characterized by three key parameters used in the initial socket() function call:

Component	**Description**	**Example Value**
Address Family	Specifies the protocol addresses that can be used.	AF_INET (for IPv4), AF_INET6 (for IPv6)
Socket Type	Specifies the semantics of communication (transport protocol).	SOCK_STREAM (for TCP), SOCK_DGRAM (for UDP)
Protocol	Usually set to 0, which lets the system choose the best protocol based on the family and type.	0

Introduction

- Every socket interaction starts with creating a socket object and involves a sequence of standard method calls.
- Creating the Socket
  - The socket.socket() function is the first step. It requires two main parameters: **s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)**

Parameter	**Value (TCP)**	**Value (UDP)**	**Description**
Address Family	socket.AF_INET	socket.AF_INET	Specifies the network address type, typically **IPv4**.
Socket Type	socket.SOCK_STREAM	socket.SOCK_DGRAM	Specifies the transport protocol: **TCP** (stream-oriented) or **UDP** (datagram-oriented).

TCP Methods (SOCK_STREAM)

- TCP is a **connection-oriented** protocol, meaning a dedicated, reliable channel is established before data transfer. The server and client use distinct sets of methods.
- **Server-Side Flow**
- **s.bind((host, port))**
  - Associates the socket with a specific **local IP address and port number**. The port is the "door" through which the server will listen.
  - **Example:** s.bind(('127.0.0.1', 65432))

TCP Methods (SOCK_STREAM)

- **s.listen()**
  - Puts the server socket into **listening mode**, waiting for incoming client connection requests
  - **Example:** s.listen()
- **conn, addr = s.accept()**
  - This is a **blocking** call that waits until a client connects. When a connection is successfully established (the "three-way handshake" completes), it returns:
  - conn: A **new socket object** representing the actual connection to the client. All communication happens on this new socket.
  - addr: The client's address (IP and port).

TCP Methods (SOCK_STREAM)

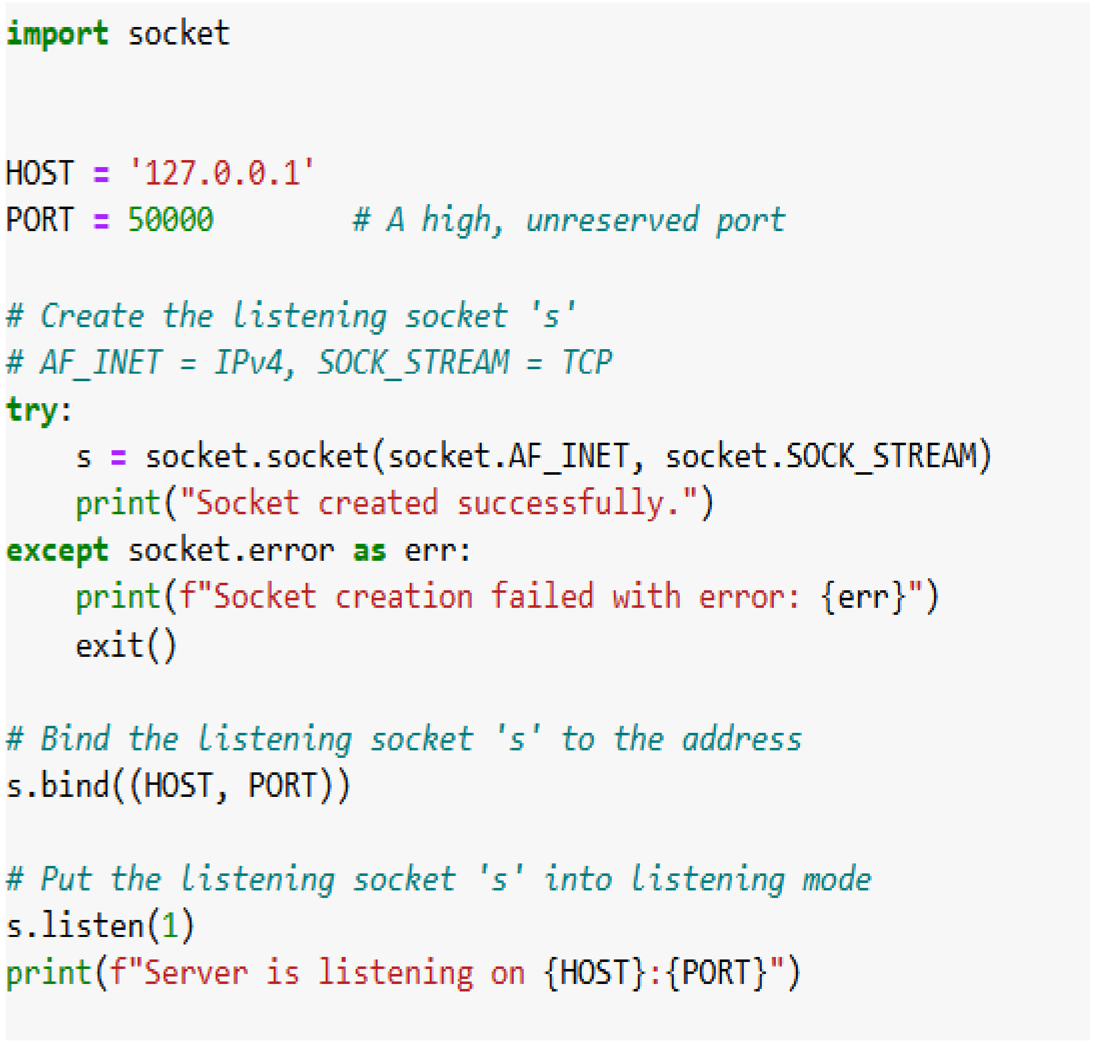
- **conn.recv(bufsize)**
  - Receives TCP data from the connected client. It returns the data as a byte string. The bufsize is the maximum amount of data to be received at once.
- **conn.sendall(data)**
  - Sends the byte string data to the connected client. It continues sending data until all bytes have been transmitted.

TCP Methods (SOCK_STREAM)

Client-Side Flow

- **s.connect((host, port))**
  - Actively attempts to establish a **TCP connection** with the server located at the specified remote host and port. This initiates the three-way handshake.
  - **Example:** s.connect(('127.0.0.1', 65432))
- **s.sendall(data)**
  - Sends data to the connected server.
- **s.recv(bufsize)**
  - Receives data from the connected server.

Simple TCP Server Example (server.py)



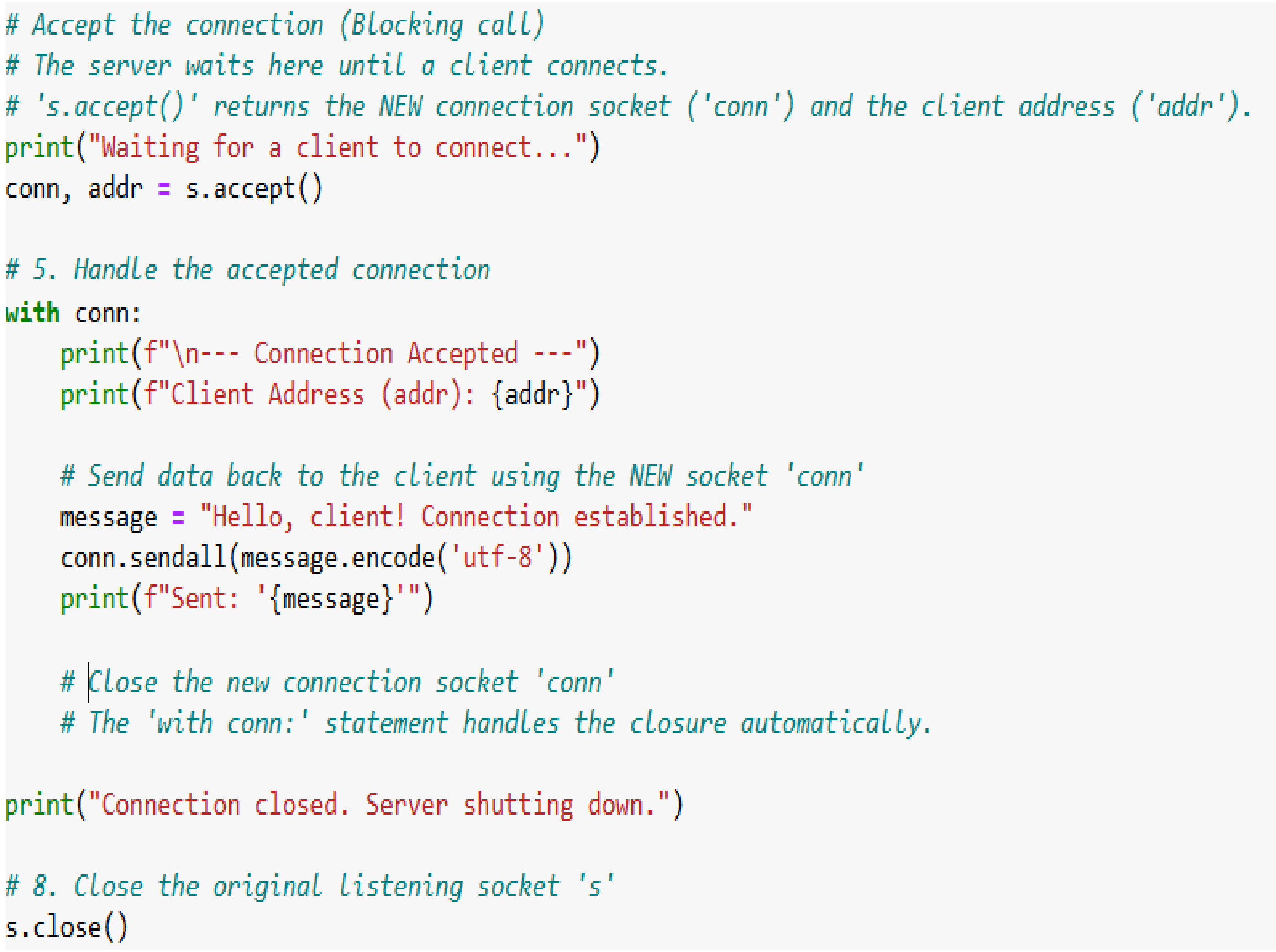
```python
import socket


HOST = '127.0.0.1'
PORT = 50000        # A high, unreserved port

# Create the listening socket 's'
# AF_INET = IPv4, SOCK_STREAM = TCP
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("Socket created successfully.")
except socket.error as err:
    print(f"Socket creation failed with error: {err}")
    exit()

# Bind the listening socket 's' to the address
s.bind((HOST, PORT))

# Put the listening socket 's' into listening mode
s.listen(1)
print(f"Server is listening on {HOST}:{PORT}")
```



```python
# Accept the connection (Blocking call)
# The server waits here until a client connects.
# 's.accept()' returns the NEW connection socket ('conn') and the client address ('addr').
print("Waiting for a client to connect...")
conn, addr = s.accept()

# 5. Handle the accepted connection
with conn:
    print(f"\n--- Connection Accepted ---")
    print(f"Client Address (addr): {addr}")

    # Send data back to the client using the NEW socket 'conn'
    message = "Hello, client! Connection established."
    conn.sendall(message.encode('utf-8'))
    print(f"Sent: '{message}'")

    # Close the new connection socket 'conn'
    # The 'with conn:' statement handles the closure automatically.

print("Connection closed. Server shutting down.")

# 8. Close the original listening socket 's'
s.close()
```

Simple TCP Client Example (client.py)

```python
import socket

# --- Configuration ---
HOST = '127.0.0.1'  # Server's address (must match what the server is bound to)
PORT = 50000        # Server's port (must match what the server is bound to)

# Create the client socket 'c'
# AF_INET = IPv4, SOCK_STREAM = TCP
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as c:

    # Connect to the server's address (Blocking call)
    print(f"Attempting to connect to {HOST}:{PORT}...")
    try:
        c.connect((HOST, PORT))
    except ConnectionRefusedError:
        print("Error: Connection refused. Is the server running?")
        exit()

    print("Connection successful.")

    # Receive data from the server
    # We use the client socket 'c' for receiving.
    data = c.recv(1024)

    print("\n--- Message Received ---")
    print(f"Received (decoded): {data.decode('utf-8')}")

#  The client socket 'c' automatically closes here.
```

Example:

How to Run:

1. Save the two files as `server.py` and `client.py`.
2. Open **two separate terminal windows**.
3. In the first terminal, run the server: `python server.py`
4. In the second terminal, run the client: `python client.py`

```python
import socket

# --- Configuration ---
HOST = '127.0.0.1'     # Listen only on the local machine
PORT = 50000           # The same port the client will send to
BUFFER_SIZE = 1024     # Max number of bytes to receive at once

# Create the UDP socket 's'
# AF_INET = IPv4, SOCK_DGRAM = UDP
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:

    # Bind the socket 's' to the address
    s.bind((HOST, PORT))
    print(f"UDP Server up and listening on {HOST}:{PORT}")

    while True:
        print("\nWaiting to receive message...")

        # Receive a datagram (Blocking call)
        # s.recvfrom() returns the data and the sender's address (addr)
        data, addr = s.recvfrom(BUFFER_SIZE)

        # Process received data
        message = data.decode()
        print(f"Received message: '{message}'")
        print(f"From address (addr): {addr}")

        # Send a reply back to the sender's address
        reply = f"Echo: {message}".encode()
        s.sendto(reply, addr)
        print("Reply sent.")
```

```python
import socket

# --- Configuration ---
SERVER_HOST = '127.0.0.1'  # Server's address
SERVER_PORT = 50000        # Server's port
SERVER_ADDRESS = (SERVER_HOST, SERVER_PORT)
BUFFER_SIZE = 1024

# Create the UDP socket 'c'
# AF_INET = IPv4, SOCK_DGRAM = UDP
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as c:

    message = "Hello, UDP World!"

    # Send the message
    # In UDP, we send directly to the server's address without a 'connect' step.
    c.sendto(message.encode(), SERVER_ADDRESS)
    print(f"Sent message: '{message}'")

    # Wait for a response datagram (Blocking call)
    # c.recvfrom() returns the data and the server's address (addr)
    data, addr = c.recvfrom(BUFFER_SIZE)

    print("\n--- Echo Received ---")
    print(f"Received from {addr}: {data.decode()}")

# The client socket 'c' automatically closes here.
```