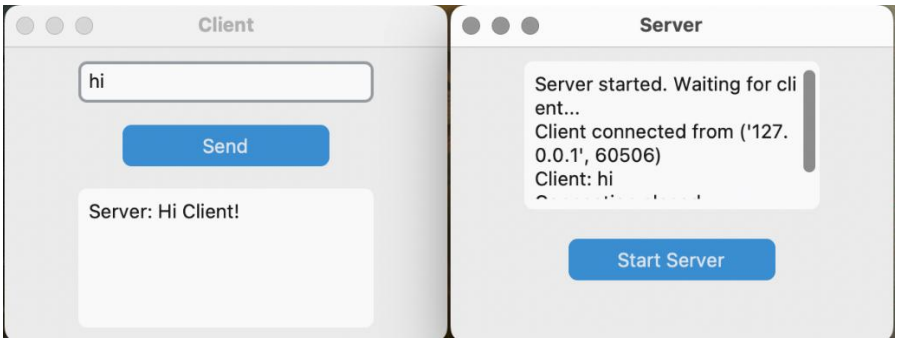
InstaNet Project: Phase 1

Spring 2025

Deadline: Monday 5th May 11:59PM

Introduction: Phase 1

This project aims to provide hands-on experience in designing and developing network applications. You will use Python's socket module to create a simple, functional application that emulates a platform for sharing text and images both publicly (as a broadcast) and privately (via direct messages), like Instagram (in Phase 2). In the current phase (Phase 1), you will be introduced to socket programming and will design a basic client-server communication application, as shown in the image below.



Most of the application elements—such as the interface and logic—have already been designed for you. Your task in both phases is to incorporate the networking components that enable communication between the client and server.

The project is divided into two phases with the following deadlines:

Project Phase	Deadline
Phase 1: Socket Programing Introduction and programming environment setup	Monday 5th May 11:59 PM
Phase 2: Completing the development of a Networked Application (InstaNet)	Thursday 15th May **4 PM**

It is essential that every member of the group fully contributes to achieving the objectives of the phase. The group must write all the code independently, without copying from external sources or other groups.

What is Socket Programming?

Socket programming is a method of connecting two nodes on a network to enable communication between them. One socket (node) listens on a specific port at an IP address, while the other socket initiates a connection to the listening socket. The server typically creates the listening socket, while the client connects to the server.

A socket is created using *socket.socket()*. As follows:

```
server_socket = socket.socket()
```

This establishes the server's ability to send and receive data over a network. The default socket type is TCP, which ensures that data is delivered reliably and in the correct order.

We can bind the server socket to a specific IP address and source port as follows:

```
server_socket.bind(('localhost', 12345))
```

Here, 'localhost' means the server will only accept connections from the same machine, which is typically used for testing purposes. Port 12345 acts as the communication channel that the server listens on.

The server is set to listen for incoming connections with the following code

```
server_socket.listen(1)
```

The number 1 indicates that the server can handle one client connection at a time.

A client also creates a socket, and like the server, it uses TCP by default. The client uses the *connect()* method to establish a connection to the server. The address 'localhost' and destination port 12345 must match the server's *bind()* settings in order for the connection to be successful. Note the client source port is randomly chosen by the OS.

```
client_socket.connect(('localhost', 12345))
```

The server waits for a client to attempt a connection. When a client connects, the *accept()* method returns two values:

- **conn:** A new socket object through which communication will take place with the client
- **addr:** The client's address (source IP and source port)

At this point, the server has successfully established a connection with the client. The server uses the *accept()* method to accept the incoming connection:

```
conn, addr = server_socket.accept()
```

The client sends a message to the server using the *send()* method. The message is encoded into bytes, as sockets work with bytes, not strings:

```
client_socket.send("Hello from Client!".encode())
```

On the server side, the recv(1024) method is used to receive data from the client, up to 1024 bytes. The received bytes are then decoded into a string so that we can read the client's message:

```
data = conn.recv(1024).decode()
```

The server sends a response back to the client using the *send()* method. The message "Hello from Server!" is encoded into bytes before being sent:

```
conn.send("Hello from Server!".encode())
```

After the communication is complete, it's important to close the connection to free up resources:

```
conn.close()
```

conn.close() initiates a graceful TCP connection termination (FIN)

Feel free to explore online resources on socket programming for this project. Here are some valuable resources:

- DataCamp: A Complete Guide to Socket Programming in Python
- TutorialsPoint: Python Socket Programming
- GeeksforGeeks: File Transfer Using TCP Socket in Python

Phase 1 Tasks:

In this phase, you will design a simple chat system between a Client and a Server. The client sends a message to the server, and upon receiving it, the server automatically responds with the fixed reply: 'Hi Client!'. The client then displays the server's response.

Both apps use a simple GUI that is already designed for you, so your task is to implement the underlying logic and facilitate the communication between the two apps.

You are provided with two Python files:

- server.py — The GUI for the server.
- client.py — The GUI for the client.

Each file includes:

- A functional GUI built using CustomTkinter.
- Clear # TODO comments where you will write your code.

Step 1: Setting Up the Environment

In this preliminary phase of the project, you are required to prepare the development environment and familiarize yourself with socket programming.

Installing Python

You will need to install Python 3 on your machine. To do this, you can download and install the Python package or any Python IDE (e.g., PyCharm).

- For the official Python 3 download, visit https://www.python.org/downloads/
- For PyCharm, download the IDE from https://www.jetbrains.com/pycharm/download/

Installation guide:

- Windows: https://www.educademy.co.uk/how-to-install-python-and-pycharm-on-windows
- Mac: https://www.geeksforgeeks.org/how-to-install-python-and-pycharm-on-mac/

Testing the Development Environment

You should already be familiar with how to run Python applications from an IDE (e.g., PyCharm). However, it is also important to know how to run Python code from the terminal. To test your environment, follow the steps below

Unix-Based systems (Mac, Linux):

1. Open the Terminal and navigate to the directory where the test code is saved. You can do this using the cd (change directory) command.

   For example:

   ```
   cd /path/to/your/code/folder
   ```

   Replace path/to/your/code/folder with the actual path where your test code is located.

2. Run the code: python3 test.py
3. You will be able to see the result printed on the window

Windows systems:

1. Open the Command Prompt and navigate to the directory where the test code is saved. You can do this using the cd (change directory) command.

   For example:

   ```
   cd /path/to/your/code/ folder
   ```

   Replace /path/to/your/code/folder with the actual path where your test code is located.

2. Run the code: python test.py
3. You will be able to see the result printed on the window

Follow these references in case you faced an issue:

- https://vteams.com/blog/how-to-run-a-python-script-in-terminal/
- https://www.geeksforgeeks.org/how-to-use-cmd-for-python-in-windows-10/

Step 2. Open the project in your IDE.

Using your IDE (e.g., PyCharm), open the Phase 1 folder of the project.

Typically, in your IDE, go to **File > Open** (or Open Project), then browse to the Phase 1 folder (not the .zip file) and select it.

Step 3. Install the Dependencies:

A dependency is any external library or package that your Python project requires to run correctly. These dependencies are not part of Python's standard library, so you must install them separately.

For example, Phase 1 uses CustomTkinter, so you will need to install it as a dependency:

```
customtkinter==5.2.2
```

CustomTkinter is used to create modern, customizable graphical user interfaces (GUIs).

There are two ways to install the dependencies:

1) Using pip with requirements.txt:
   - Open your terminal or command prompt.
   - Navigate to the folder where your requirements.txt file is located.
   - Run: pip install -r requirements.txt

2) Installing them individually (if needed)
   - Run: pip install customtkinter==5.2.2

If you completed these tasks, your environment is ready, and you are prepared to move to socket programming.

Step 4. Start Coding

Complete the missing server-side and client-side functions in server.py and client.py to enable proper communication between the two.

Use the **TODO** comments as guidelines to fill in the necessary functionality.

Requirements

Part A – Server (server.py):

Use Python socket to:

- Bind the server to local host and port 12345
- Wait for one client to connect.
- Accept a connection
- Receive a message.
- Display it in the GUI.
- Send back a fixed reply: "Hi Client!".
- Closes connection

Part B – Client (client.py):

Use Python socket to:

- Connect to the server on localhost and port 12345.
- Send the user-typed message.
- Receive the reply and display it in the textbox.
- Closes connection

Note: Technically, one side initiating a FIN (closing the connection) is enough to begin the connection termination, but having both sides explicitly close the connection is good practice.

Testing

To test or use the chat system properly:

Open two separate terminals (or two separate IDEs).

In one terminal, run the server:

```
python server.py
```

Note: You must start the server by clicking the "Start" button in the GUI before sending any messages.

In the second terminal, run the client:

```
python client.py
```

Note: For mac use python3

Submission

Submit the source code for both the server and the client. You are also required to include a screenshot of your GUI output, similar to the example shown in this document. In the screenshot, make sure that the message sent from the client includes your group name(s). Additionally, ensure your code is commented with your name(s).

Your solution should:

- Work on a single computer (localhost).
- Show one message from client to server and one reply.
- Be written entirely inside the provided TODO blocks.

Sample output (your names instead of "hi"):