CS 382 - Assignment 1 - Implementing a Chat Application

Lead TAs: Ayesha Shafique, Hamna, Abubakar, Yahya, Abdullah, Essa, Zaeem

Due Date: 11:55pm, 24th February, 2025

Contents

1 Assignment Overview

In this assignment, you'll explore socket programming by making a simple chat app, similar to WhatsApp or Messenger. Your job is to make users able to send messages and files. **The assignment must be done individually, and you are required to use Python.** Cases of Plagiarism will be flagged and reported.

1.1 Queries

You should post any assignment related queries **ONLY** on Slack. Please do not email the course staff regarding any assignment related queries. This will ensure that everyone benefits from the answers to your queries.

1.2 Grade Distribution

This assignment is worth **7.5%** of your final grade.

It will be graded out of 25 points. The distribution of points is as follows:

- **SingleClientTest:** A single client is able to communicate with the server (5 points)
- **MultipleClientsTest:** Multiple clients can exchange messages with each other (5 points)
- **FileSharingTest:** Multiple clients can share files with each other (5 points)
- **ErrorHandlingTest:** Your code can handle both server and client errors (5 points)
- **Code Quality:** Following good code practices as described in section 1.4 (5 points)

Disclaimer

Please note that your assignments will be tested on Docker containers with a Ubuntu image as part of GitHub Workflows. As such, it is recommended that you run your code on it at least once before submitting it. It is the student's responsibility to ensure that their code works correctly on Docker containers and GitHub Workflows. Additionally, when the code is pushed to GitHub, they should verify the output of the autograder workflow to confirm that there are no errors.

"It works on my machine" is not a valid response. If your code works on some machines but fails elsewhere, it is likely that the code has not been properly written, tested, or designed for compatibility. Well-thought-out, thoroughly tested code should perform consistently across environments, especially within Docker containers, and it is **YOUR** responsibility to ensure that.

1.3 Submission Instructions

You will submit your assignment via GitHub. Follow these instructions carefully:

- A forked repository will be created for each student, where you will do your work.
- Only the submission on the `main` branch will be graded.
  - If you create other branches and do your work there, it **WILL NOT** be considered for grading. This does not mean you are not allowed to create branches, which is a good practice to do; it just means you should merge them back into **main** once your feature is tested out.
- Pushing your work to GitHub correctly is your responsibility.
  - If you do not push your final submission to GitHub, your assignment will **NOT** be graded.

- Ensure that your latest work is visible on GitHub before the deadline.
- Navigate to your repository URL and verify that your latest commit is present in the `main` branch before submission.

1.4 Code Quality

Your code will be evaluated using automated tools as well as a manual review based on the following criteria:

- **Maintainability, Modularity, and Complexity:** Your code will be analyzed using tools like `Radon` and `Pylint` to assess maintainability, modularity, and cyclomatic complexity. The grading will follow the detailed breakdown provided in Section 7.
- **Naming Conventions:** You are expected to follow `PEP 8` naming conventions. You can find the standards here. We will specifically look at:
  - **Variables, Functions, and Methods:** Use lowercase letters and separate words with underscores. For example, `my_variable`, `calculate_sum()`.
  - **Constants:** Use uppercase letters and separate words with underscores. For example, `MAX_SIZE`, `PI`.
  - **Classes:** Use CamelCase (capitalize the first letter of each word without underscores). For example, `MyClass`, `CarModel`.
- **Code Documentation and Comments:** Your code should be well-documented, following the comment ratio guidelines. The comment percentage will be evaluated as described in Section 7, ensuring that your code is understandable to others.
- **Code Modularity and Reusability:** Your implementation should not be monolithic; instead, break down your code into smaller reusable functions. This will be assessed as part of the modularity score in the automated quality check.
- **Readability and Formatting:** Your code should be readable with meaningful variable and function names. Additionally, proper indentation and spacing should be maintained to improve readability.

The final quality score will be calculated using automated scripts. Ensure that your code adheres to these best practices, as they are essential for writing maintainable and efficient software.

2 GitHub Setup and Assignment Submission

The assignment will be managed and graded via GitHub. Follow the steps below to set up your repository and submit your work.

Important Warning

The `.github` folder is a protected path. Modifying any files within this folder, including workflows and configuration files, is strictly prohibited, and GitHub classrooms will flag such repositories for us. Any unauthorized changes to this folder will result in your assignment being discarded. Ensure that you do not tamper with the contents of the `.github` directory.

2.1 Accepting the Assignment

1. You will receive a GitHub Classroom link to accept the assignment.
2. Click on the link, and GitHub Classroom will create a private repository for you.

3. You can clone the repository using one of the following methods:
    - Using the **Command Line**: Run the following commands in your terminal:

```
git clone <repository-url>
cd <repository-directory>
```

4. Using GitHub Desktop:
    - Open GitHub Desktop.
    - Click on File > Clone Repository.
    - Select the GitHub Classroom repository assigned to you.
    - Choose a local directory and click Clone.

2.2 Working on the Assignment

1. Implement the required functionality by modifying the provided files.
2. Regularly commit your changes using git with the following commands or by Github Desktop:

```
git add .
git commit -m "Implemented feature XYZ"
git push origin main
```

Ensure that your latest work is always pushed to the repository before the deadline.

2.3 Running Tests on GitHub Workflows

GitHub Actions is set up to automatically test your code upon every push. To check your test results:

1. Navigate to your repository on GitHub.
2. Click on the **Actions** tab.
3. Look for the latest workflow run.
4. Click on it to view detailed logs of the test results.

If your tests fail, fix the issues locally and push your changes again.

2.4 Submitting the Assignment

1. Ensure all your code is pushed to GitHub before the deadline.
2. No separate submission is required; your latest pushed commit will be considered for grading.
3. The instructor will evaluate the code using the test suite and automated scripts.

3 Dependencies and Requirements

Ensure the following dependencies are installed. These are also frozen into `requirements.txt`.

- Python 3.10+
- `radon`, `pylint`

3.1 Installing Dependencies

To install all required dependencies, run:

```
pip install radon pylint
pip install -r requirements.txt
```

For documentation:

- Radon Documentation
- Pylint Documentation

4 Introduction

4.1 The Client Server Architecture

Have you ever wondered how an application like Whatsapp, Messenger or Signal works? How is it that you type a message on your phone in let's say Lahore, and your friend in Karachi receives it at nearly the same time? How can you share pictures over that distance? If you have, then this course will answer these questions for you. If you haven't, don't worry. This course is designed to make you not only ask these questions, but also to teach you how to answer them. Roughly speaking, this is what the core architecture of a messaging application looks like:

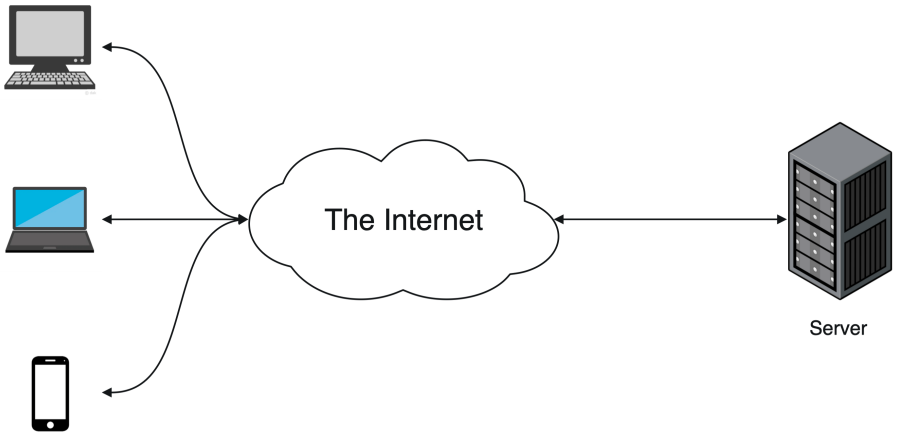

Figure 1: Devices connected to the internet communicating with one central server

Here we can see that multiple devices and the server are connected to each other via the internet. The devices on the left are called `clients`. These `clients` are the ones that you use to send messages and files.

The device on the right is called the `server`. It has many responsibilities. For example, it keeps track of:

- which `clients` are currently active (can send or receive messages)
- where each `client` is located (e.g. TCP connection, IP address)

The beauty of this client-server architecture is that it does not require you to know the location of the `client` you want to send a message to. All you need to know is the `client's` name (e.g. username, phone number, email address). The `server` will take care of the rest. **No two clients talk directly to each other.** All communication is done through the `server`, whether you are sending a message or a file.

4.2 Task Summary

In this assignment, you will be implementing a simple chat application based on the Client-Server architecture. You have to ensure that you follow the specification in section 6. More information about the server and client can be found in section 5.

5 What to Implement?

5.1 Application Server

The application server is a **multi-threaded** server. It listens for new connections from clients at a given **host** and **port**, accepts them, and handles the messages sent from each client. The server can concurrently connect up to `MAX_NUM_CLIENTS` clients. This means that if `MAX_NUM_CLIENTS` is 10, at max only 10 clients can use the service at a time, and for a 11^{th} client to connect, one of the previous 10 clients will need to be disconnected. When a client requests to join a server that already has `MAX_NUM_CLIENTS`, its request will be rejected. Furthermore, the server does not store information of disconnected clients. The server must handle all possible errors and remain alive until explicitly terminated.

5.2 Application Client

The application client represents the interface for your chat application, that connects to the application server with a unique username i.e. no two clients connected to the server at the same time can have the same username. It does the following tasks:

- Connects to the server at a given host and port
- Read user input from the standard input stream and send messages to the server, accordingly.
- Receive and handle messages from the server
- End the connection with the server and shutdown on receiving an error from the server. More details about this in section 6. Before shutting down, it also shows a message to the user on why the connection is being terminated.

For the application client and application server to communicate with each other over the Internet, we will use socket programming.

6 Protocol Description and API Specification

Before describing the API in detail, we will begin by listing down the sequence of events that will take place for this application to work. This will help you understand the API better. The sequence of events is as follows:

1. The server creates a TCP socket and starts listening on a fixed port number, (e.g. localhost:5000).
2. The client creates a TCP socket and connects to the server.
3. The client sends a `join` message to the server.
4. The server should add the new client to the list of clients and stores its TCP connection details (IP address and port number), so that it can communicate with the client later on.

Using this protocol, the server can communicate (send and recieve messages) to clients. The clients can also communicate with the server. The client can send a message to the server, which will then forward the message to the intended recipient. The server can also send a message to all the clients.

A typical interaction between a server and a connected client can be summarized as follows:

- User inputs a specific command string in the predefined formats into the Application Client input. This is a simple standard input (stdin).
- The client evaluates the input based on predefined formats, reformats it according to formats required for processing at the server (if needed), and sends the formatted message to the server.
- On receiving a message from a client, the server processes it, again based on predefined formats (refer to section 6.1).
- Based on what command was processed, the server forwards a formatted message back to the client (for example, if a list of connected clients was requested) or forwards a message from the client to other clients (i.e. a text message or a file). In case of errors, appropriate error messages are sent back.
- A client receiving a server message decodes it according to the predefined formats and then outputs it to the Application Client in the formats specified.

Example interaction

- Three clients with usernames `client1`, `client2`, and `client3` connect to the server and `client3` inputs the following string: `msg 2 client1 client2 Hello my friends!`
- The client application should interpret this as a message to be sent to clients `client1` and `client2`, with the message body `Hello my friends!`.
- The client application should then send this message to the server.
- The server should recieve the message, understand that it is coming from `client3`, and that the intended recipients are `client1` and `client2`.
- The recipients should then recieve the message from the server and display it on their respective client applications. It should read: `msg: client3: Hello my friends!`

This describes the basic crux of the chat application; your implementation will build up on this to include the required functionality and exception handling. This will require implementing application APIs and protocols.

Application API are the actions that a user at a client can perform using the application i.e. the functionality that an application supports. For your app, this includes client joining and quitting, sending and receiving messages and files, getting a list of connected users, and a basic help function. For implementation purposes, each of these APIs (other than join which is automatically called on starting a client) are called on by the user at a client using string inputs of specific formats so the application client understands commands.

Application protocols define how an application client and application server communicate with each other. As with user inputs, these need to follow specific conventions and formats so the client and server can understand messages sent and received and do appropriate processing. For this assignment, these constitute a set of protocols defined by message types (e.g. `send_message`, `forward_file`), which also include error messages for exception handling, and 4 fixed message string formats. The message exchanged for each protocol belongs to one of the 4 string formats. Since the messages are sent/received as strings, the different fields in a message will be separated by a single space character (‘ ’). The 4 formats are as follows:

Type 1:

Type	Username

Type 2:

Type

Type 3:

Type	List of Usernames

Type 4:

Type	List of Usernames	Message

List of usernames in Type 3 and 4 will be formatted as:

Num of Users	User 1	User 2	...	User N

We describe the API and protocols in detail in the following sections.

6.1 API Description

Each Application Client should be able to perform the tasks given below. For each function, the client must get the user input from standard input (stdin) and process the input according to the below-mentioned formats. If the input does not match with any format, the client should print on stdout:

`incorrect user input format`

Your chat application should support the following API:

6.1.1 Message

Sends a message from this client to other clients connected to the server using the names specified in the user input. The application server must ensure that the client (whom the message is sent to) will only receive the message **once** even if his/her username appears **more than once** in the user input.

User Input: `msg <recipient_count> <recipient_1> <recipient_2> ...<recipient_n> <Message>`

6.1.2 Available Users

Lists all the usernames of clients connected to the application-server (including itself).

User Input: `list`

6.1.3 File Transfer

Sends a file to other clients connected to the server. The other clients should save the file with the same filename (as specified by the sender, includes the file extension i.e. .txt or .py) prefixed with their username (e.g. client1_solution.py).

User Input: `file <recipient_count> <recipient_1> ...<recipient_n> <file_name>`

6.1.4 Help

Lists all the commands supported by the application client and the syntax.

User Input: `help`

6.1.5 Quit

Let the application server know that the client is disconnecting, close the TCP connection with the application server, print the following message to stdout, and shutdown gracefully. It should show `quitting`

User Input: `quit`

6.2 Application Protocols

A message is a basic unit of data transfer between client and server. The different types of messages that can be exchanged between a client and a server are as follows (the message formats mentioned are given below):

6.2.1 Join

Direction: Client → Server

Format: `join <username>`

Sender Action: This message serves as a request to join the chat application. Whenever a new client is spawned, it will send this message to the server.

Receiver Action: When a server receives this message, three things can happen at the server:

1. The server already has `MAX_NUM_CLIENTS`, so it will reply with an error message 'err_server_full' and print the following message to stdout: `disconnected: server full`
2. The username is already taken by another client. In this case, the server will reply with an `err_username_unavailable` error message and print the following message to stdout: `disconnected: username not available`
3. The server allows the user to join the chat. In this case, it will not reply but will print: `join: <username>`.

6.2.2 Request User List

Direction: Client → Server

Format: `list`

Sender Action: A client sends this message to the server when it reads a list as user input.

Receiver Action: The server will reply with `RESPONSE_USERS_LIST` message and will print: `request_users_list: <username>` to stdout.

6.2.3 Response User List

Direction: Server → Client

Format: `list <username_1> <username_2> ...<username_n>`

Sender Action: The server will send the list of all usernames (including the one that has requested this list) to the client.

Receiver Action: Upon receiving this message, the client will print: `response_users_list: <username_1> <username_2> ...<username_n>` to stdout, where the usernames are sorted in ascending order.

6.2.4 Send Message

Direction: Client → Server

Format: `msg <recipient_count> <recipient_1> <recipient_2> ...<recipient_n> <Message>`

Sender Action: A client sends this message to the server.

Receiver Action: The server forwards this message to each user whose name is specified in the request. It will also print: `msg: <sender_username>` to stdout `<recipient_count>` times. For each username that does not correspond to any client, the server will print: `msg: <sender_username> to non-existent user <recv. username>` to stdout.

6.2.5 Foward Message

Direction: Server → Client

Format: `msg <sender_username> <Message>`

Sender Action: The server will forward the messages it receives from clients. It will specify the username of the sender in the message in the List of Usernames field.

Receiver Action: The client, upon receiving this message, will print: `msg: <sender_username>: <Message>` to stdout.

6.2.6 Send File

Direction: Client → Server

Type: 4

Format: `file <recipient_count> <recipient_1> ...<recipient_n> <file_name> <file_contents>`

Sender Action: A client sends this message to the server. In the place of the message, it will place the file (the filename will be placed before the actual file content, separated by space). Receiver Action: The server forwards this file to each user whose name is specified in the request. It will also print: `file: <sender_username>` to stdout. For each username that does not correspond to any client, the server will print: `file: <sender_username> to non-existent user <receiver username>` to stdout.

6.2.7 Forward File

Direction: Server → Client

Format: `file <sender_username> <file_name> <file_contents>`

Sender Action: The server will forward the files it receives from clients. It will specify the username of the sender in the message in the List of Usernames field.

Receiver Action: The client, upon receiving this message, will save the file (with the name that was received) and print: `file: <sender_username>:<file_name>` to stdout. The client should save the file with the same filename (as specified by the sender, includes the file extension i.e. .txt or .py) prefixed with their username (e.g. client1_solution.py).

6.2.8 Server Full

Direction: Server → Client

Format: `err_server_full`

Sender Action: The server will send this message to the client when the server is full.

Receiver Action: The client, upon receiving this message, will print: `disconnected: server full` to stdout and close the connection with the server.

6.2.9 Unknown Command

Direction: Server → Client

Format: `err_unknown_message`

Sender Action: The server will send this message to the client when the server receives an unknown command. It will also print `disconnected: <username> sent an unknown command`

Receiver Action: The client, upon receiving this message, closes the connection to the server and shuts down with the following message on screen: `disconnected: server received an unknown command`

Note: This is not being tested.

6.2.10 Username Unavailable

Direction: Server → Client

Format: `err_username_unavailable`

Sender Action: The server will send this message to the client when the username is already taken by another client.

Receiver Action: The client, upon receiving this message, will print: `disconnected: username not available` to stdout and close the connection with the server.

6.2.11 Disconnect

Direction: Client → Server

Format: `disconnect`

Sender Action: The client will send this to the server to let it know it's disconnecting

Receiver Action: The server will remove the client from the list of connected clients and close the connection with the client. It will also print: `disconnected: <username>` to stdout.

7 Testing

7.1 Test Files

Hidden Test Cases

We will **not** have hidden test cases for this assignment. The test cases for this assignment are deterministic and will only test the commands already provided in test files.

- **`BasicTest.py`**
  - The base class for all tests.
  - Provides utilities for setting up the environment, handling messages, and verifying results.
- **`SingleClientTest.py`**
  - Tests the behavior of the application with a single client connected.
  - Verifies:
    - Message sending.
    - User list requests.
    - Disconnections.
- **`MultipleClientsTest.py`**
  - Evaluates performance with multiple simultaneous clients.
  - Ensures:
    - Correct message routing.
    - Server handling of multiple connections.

- **`FileSharingTest.py`**
  - Tests file-sharing functionality.
  - Verifies:
    - File transmission between clients.
    - Server handling of file transfers.
- **`ErrorHandlingTest.py`**
  - Evaluates application behavior in error scenarios.
  - Checks:
    - Invalid user input.
    - Username conflicts.
    - Server-full conditions.

7.2 Running the Tests

7.2.1 Quick Start

To run all tests, execute:

```
python3 TestChatApp.py
```

7.2.2 Options

- **`-client`**: Path to the client implementation (default: `client.py`).
- **`-server`**: Path to the server implementation (default: `server.py`).
- **`-port`**: Port number for the server (default: random between 2000-65500).
- **`-test`**: Specify tests to run (e.g., `SingleClient`, `MultipleClients`).
- **`-verbose`**: Enables verbose output for detailed logs.
- **`-help`**: Displays help information.

7.2.3 Examples

Run specific tests:

```
python3 TestChatApp.py --client client.py --server server.py --test SingleClient
    MultipleClients
```

Enable verbose mode:

```
python3 TestChatApp.py --verbose
```

7.3 Clearing Outputs

Each test deletes previous output files by default. If needed, manually clear outputs:

```
python3 -c "from TestChatApp import delete_with_rm_rf; delete_with_rm_rf()"
# or
rm -rf ./client_* ./test_* ./*_test_* ./server_out*
```

7.4 Code Quality Check

7.4.1 Overview

`ReviewCodeQuality.py` ensures that the code adheres to standards of:

- Maintainability
- Modularity
- Cyclomatic Complexity
- Comments Documentation

It evaluates files using tools like **Radon** and **Pylint**, scoring them on various metrics.

7.4.2 Metrics and Scoring

1. **Maintainability Index (MI)**
   - Grade A/B: 1 point
   - Grade C: 0.75 points
   - Grade D: 0.5 points
   - Grade E: 0.25 points
   - Grade F: 0.1 points
2. **Cyclomatic Complexity (CC)**
   - Grade A/B: 1 point
   - Grade C: 0.75 points
   - Grade D: 0.5 points
   - Grade E: 0.25 points
   - Grade F: 0.1 points
3. **Pylint Score** (scaled to 2.5 points)
   - Score 8–10: 2.5 points
   - Score 7–7.9: 2 points
   - Score 5–6.9: 1.5 points
   - Score 3–4.9: 1 point
   - Below 3: 0.5 points
4. **Comments Ratio** (percentage of comments in the code)
   - >= 15%: 0.5 points
   - 10–14.9%: 0.45 points
   - 5–9.9%: 0.35 points
   - 2–4.9%: 0.25 points
   - <2%: 0.1 points

7.4.3 Running Code Quality Checks

To check code quality, run:

```
python3 ReviewCodeQuality.py <file_paths>
```

7.4.4 Example

```
python3 ReviewCodeQuality.py client.py server.py
```

7.4.5 Output Example

```
Processing: client.py
Results for client.py:
  Modularity: 1/1
  Cyclomatic Complexity: 1/1
  Pylint Rating: 2/2.5
  Comments Ratio: 0.45/0.5
  Final Score: 4.95/5

Processing: server.py
Results for server.py:
  Modularity: 0.75/1
  Cyclomatic Complexity: 0.75/1
  Pylint Rating: 1.5/2.5
  Comments Ratio: 0.35/0.5
  Final Score: 3.35/5

Average Score for 2 file(s): 4.15/5
```

8 Starter Code

You have been provided with the following files in the Code directory:

```
PA1
└── Code
    ├── Tests
    │   ├── Outputs
    │   │   ├── ErrorHandling
    │   │   ├── FileSharing
    │   │   ├── MultipleClients
    │   │   └── SingleClient
    │   ├── BasicTest.py
    │   ├── SingleClientTest.py
    │   ├── MultipleClientsTest.py
    │   ├── FileSharingTest.py
    │   └── ErrorHandlingTest.py
    ├── ReviewCodeQuality.py
    ├── TestChatApp.py
    ├── client.py
    ├── requirements.txt
    ├── sample_output.txt
    ├── server.py
    └── util.py
```

You will write code in the following files:

- `client.py`: This file contains the implementation of the client side of the chat application.
- `server.py`: This file contains the implementation of the server side of the chat application.

- `util.py`: This file contains the implementation of the utility functions that can be used by both the client and the server.

9 Running the Code on Docker

A **Dockerfile** is included for consistent testing. Use the following steps:

9.1 Build and Run Container

To build and run the container, use the following command:

```
docker compose run --rm netcen-spring-2025
# or
docker-compose run --rm netcen-spring-2025
```

What this command does:

- `run`: Executes the `netcen-spring-2025` service defined in the `docker-compose.yml` file as a temporary, one-off container.
- `-rm`: Automatically removes the container after the task completes, ensuring no leftover containers clutter the system.
- This command starts the container and runs the service or command specified for `netcen-spring-2025` in `docker-compose.yml`.

9.2 Access Code

- The code is mounted at `/home/netcen_pa1` inside the container.
- This allows you to access and work with your files directly within the container.

10 Tips on Getting Started

Here is some advice for this assignment.

1. First, start early. The assignment is not very difficult but there is a lot of new information that might overwhelm you, so it's important that you give it time. Read this handout carefully. Also, debugging will take a lot of time so make sure you avoid deadline day panic.
2. Unless otherwise stated, each line should end with a newline character. The delimiter between each word of your input and output should be a single space.
3. Google is your friend. Google can give you answers to many questions faster than any TA can. Use these resources effectively. Questions ranging from string manipulation to list trimming or parsing through user input and dictionaries; these can be answered in less than a minute with a few simple keywords being searched for.
4. All the output specified in this document is done to stdout. Since we will be using stdout for testing, you must ensure that only the specified output goes there.
5. You are provided with a helper function in **util.py**, that you are recommended to use.
6. **TestChatApp.py**, invokes instances of `client.py` and `server.py` by themselves so you do not need to have them running when testing. However, we recommend that you start working on your code by testing your `server.py` and `client.py` files manually since that will be a better way to track your progress in the earlier stages.

7. The test cases are very robust and rigorously check your standard output. **Make sure that you are printing your output exactly as specified in section 6 and are not printing extra spaces in between or extra newline characters at the end.** These are very easy to overlook and frustrating to debug, so be extra careful.

8. Your server code **must not crash** at any point unless explicitly being told to do so. Make sure you have done exceptional handling to preclude failures.

9. Your code MUST follow `PEP8` standards.

Good Luck, and Happy Coding!