Basic Exercises:

1. **Error Handling (Client):**
   - **Hint:** Use try...except blocks to catch exceptions like socket.error (or its subclasses like ConnectionRefusedError or ConnectionResetError). Print informative messages to the user indicating the nature of the error.
2. **File Size Limit (Server):**
   - **Hint:** Add a MAX_FILE_SIZE constant to your server code. Before writing the file, check the received data's length. If it exceeds the limit, send an error message to the client and close the connection.
3. **Duplicate File Handling (Server):**
   - **Hint:** Before saving the file, check if a file with the same name already exists in the upload_directory. If it does, either reject the upload or generate a unique filename (e.g., append a counter like "_1", "_2", etc.). Use os.path.exists() to check for file existence.
4. **Improved User Interface (Client):**
   - **Hint:** Use clearer and more informative prompts when asking the user for input. Provide detailed explanations of errors and give feedback on successful uploads. Consider using formatted strings (f-strings) for better output presentation.

Intermediate Exercises:

5. **Server-Side File Listing:**
   - **Hint:** On the server-side, create a function that lists all files in the upload_directory using os.listdir(). Send the list to the client as a string, separated by newlines. The client should then receive and display this list. Add a command, e.g., 'LIST', that the client sends to trigger a list request.
6. **File Downloading:**
   - **Hint:** Add a command (e.g., 'DOWNLOAD filename') to the client. When the server receives this command, it should open the specified file in binary read mode ('rb'), send the file's contents in chunks to the client, and close the file. The client should receive the file contents and save them to a new file locally. The server could also send the filesize in advance.
7. **Progress Indication:**
   - **Hint:** Use a library like tqdm to easily create progress bars. You'll need to track the amount of data transferred and the total size of the file.
8. **Multiple Client Support (Server):**
   - **Hint:** Use Python's threading module to handle multiple clients concurrently. Create a new thread for each client connection, allowing the server to handle many uploads simultaneously.

Advanced Exercises:

9. **Secure Communication:**
   - **Hint:** This is complex, but a simple starting point might be using a shared secret (password) to authenticate the client before allowing uploads. Both client and server would need to perform this authentication check. You'd

be better served using a proper secure protocol like TLS/SSL in a real-world application, not just a basic password.

10. **Robust Error Handling:**
    - **Hint:** Use comprehensive try...except blocks to catch a variety of exceptions. Implement logging using the logging module to record errors and other important events.
11. **Asynchronous Programming (Client):**
    - **Hint:** Use the asyncio library. Use await and async keywords appropriately. You'll need to adapt your socket operations (e.g., client_socket.send(), client_socket.recv()) to work with asyncio. This requires a significantly different programming paradigm.
12. **More Sophisticated File Management:**
    - **Hint:** Add commands to the client (e.g., 'DELETE filename', 'RENAME oldname newname') and implement the corresponding logic on the server. Thorough error handling and user feedback are essential.