Client Side

Decrypt and Encrypt functions -> input: data | output: data

```
host = '127.0.0.1'
port = 8080

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connecting with Server
sock.connect((host, port))
```

The codes above, specify a host and port for the connection.

Then, a TCP(SOCK_STREAM) socket is created, and connection is made.

There are 4 valid commands: “fetch”, “para example”, “upload example”, “download example”.

But before that, the client must enter his password which has been hard-coded as 123456.

Specific error handlings have been added.

For example: If an invalid command is given by the client, the error “invalid command: the client must pay attention to acceptable input patterns” appears. Or if the requested file does not exist in the server, errors are responded accordingly.

```
listOfCommands = msg.split(" ")
```

The code above, splits the msg written by the client by space, resulting in a list of commands.

There are many ifs and elseIfs in the code to identify whether the given command is true according to the certain rules mentioned above, with their first one being:

```
if listOfCommands[0] == "upload" and len(listOfCommands) == 2:
```

The codes below starting from line 43, start sending the file to be uploaded to the server. Specific measures must be taken by the server side to handle the sent data which will be mentioned in the server - side documentation.

```
try:
    with open("files/"+requestedFile, 'rb') as file:
        sock.send(bytes(encrypt(msg), 'ascii'))
        file_data = file.read()
    # Send the file contents
    sock.sendall(encrypt(file_data))
```

Same type of computation is available for other commands like “download”, “para” and “fetch”.

Para command:

```
elif listOfCommands[0] == "para" and len(listOfCommands)==2:
    sock.send(bytes(encrypt("para "+listOfCommands[1]),'ascii'))
    paraResponse = decrypt(sock.recv(1024)).decode()
    if paraResponse=="SameFileHere":
        try:
            with open("files/" + listOfCommands[1], 'rb') as file:
                file_data = file.read()
                sock.sendall(encrypt(file_data))
                paraOrNot = decrypt(sock.recv(1024)).decode()
                if paraOrNot=="successfullyParallelized":
                    print(Fore.LIGHTGREEN_EX, "the file has been
parallelized",
                          Fore.RESET)
        except FileNotFoundError:
            print("could not find the file in clientSide to parallelize")

    elif paraResponse=="SameFileNotHere":
        print(Fore.RED,"the server does not have this file",Fore.RESET)
# if the client wants to parallelize a file but the pattern for para command
is False
```

The code above handles the “para” command.

If the first part of listOfCommands is “para” and the length of listOfCommand is 2, then the command given by the client is a valid “para” command.

The “para command” is now sent to the server to be taken care of.

The server responds and its response is assigned to paraResponse variable.

If paraResponse equals “sameFileHere”, a file with same name (as given by client in the command) is available, so parallelization can take place.

Now the file is opened and read, and its data is sent over the TCP link to the server. The server then updates its file with the data in client’s file.

Server Side

The authenticate function checks if the password given by the client is 123456:

If yes: authenticate returns True, and the response “correct” is sent to the client so that he knows he has been accepted.

If no: it returns False, and the response “incorrect” is sent to the client so that he knows he has not been accepted.

The infoSend function sends information about list of files in the server to the client whenever a fetch command is received. The client then prints the info.

Same as in the client side, a TCP socket is created and a connection is made. The socket then starts listening to the link.

The server-side code is suspended until the client is accepted. Otherwise, the code gets stuck in the for loop below.

```
auth = False
while not auth:
    authData = decrypt(conn[0].recv(1024)).decode()
    # the authenticate function returns True if the user is authorized.
Otherwise, false is returned.
    auth = authenticate(authData, conn[0])
```

The command received by the server is now divided into pieces as in the client-side:

```
listOfCommands = data.split(" ")
```

If an upload command is given to the server, the server prepares itself for retrieving the file and writes the data info a new file. That is how, a file is uploaded to a server.

```
with open("files/" + requestedFile, 'wb') as file:
    file.write(fileData)
print(Fore.LIGHTGREEN_EX, f"File '{requestedFile}' uploaded successfully.",
Fore.LIGHTGREEN_EX)
```

Download and fetch commands work somehow the same.