Computer Network Project1

Name : Seongkyu Lee

Student ID : 2012129008

Environment

Host OS : Mac OS 10.13.6

Guest OS : Ubuntu 18.04.1

Virtual Machine : Virtual Box 5.2.18

Language : Python 3.6.6

Function Explanations

1. `socket(socket.AF_INET, socket.SOCK_STREAM)`
   The function returns *socket object* which is AF_INET address family and TCP protocol type. *socket object* has methods implementing socket system calls like `connect()`, `close()`. First argument `socket.AF_INET` represents the address (and protocol) families. Addresses for AF_INET sockets are IP addresses and port numbers. Second argument `soocket.SOCK_STREAM` represents the socket types. SCOK_STREAM is a connection-based protocol like TCP. The connection is established until the connection is terminated by a network error or by one of the connected member.

2. `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`

The function sets socket option to restart my server quickly when it terminates. First argument `socket.SOL_SOCKET` represents the level. The constant is used for `getsockopt()` or `setsockopt()` to manipulate the socket-level options like `SO_REUSEADDR`, `SO_DEBUG`. Second argument `socket.SO_REUSEADDR` allow `bind()` to reuse of local addresses for socket. That is why `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` must be called before binding. Third argument 1 represents the value of `socket.SO_REUSEADDR`. Nonzero value means “yes”.

3. `select([], [], [])`
   The function is used for I/O multiplexing. It permits a program to monitor multiple file descriptors. The three arguments are sequences of ‘waitable objects’ which are waiting until ready for reading, waiting until ready for writing, and waiting for an exceptional condition in order. The function returns a triple of lists of objects that are ready which are readable list, writable list, exceptional list. The more details will be discussed in below section. Server-side usually uses this function.

4. `bind(("0.0.0.0", 7777))`
   The function binds the socket to address. Binding means assigning an address to socket. First argument is address, which consists of IP address and port number. In this case, IP address is `"0.0.0.0"` and port number is `7777`. Server-side usually uses this function.

5. `listen(5)`

   The function enable a server to accept connections. First argument is backlog, which means the number of pending connections the queue will hold. If multiple clients connect to the server, the server holds the connecting requests in a queue. The server connects the clients one by one as and when queue-member proceeds. In this case, the server will accept clients up to 5. Server-side usually uses this function. Server-side usually uses this function.

6. `connect((host, port))`

   The function enable a client to connect to a server that is bound to the address specified argument. First argument is address of server, which consists of IP address and port number. Client-side usually uses this function.

7. `accept()`

   The function enable a server socket to accept a connection from client sockets. The server socket must be bound to an address and listening for connections, which means calling `bind((host, port))` and `listen()` before calling this function. Server-side usually uses this function

8. `close()`

   The function mark the socket closed. The related system resource like file descriptor is also closed when the socket is closed. The sockets are automatically closed when the are

garbage-collected, but closing the socket explicitly is better.

Select Function

Explanation

`select()` function is one way to handle concurrency. As I said above, it is used for I/O multiplexing. It can monitor multiple file descriptors. Not only socket, but also file object can be monitored by the function if the file object has file descriptor.

The function gets three arguments which are monitored by it. The first one is about reading, the second one is about writing, and the last one is about error. The function returns subset of arguments, readable lists, writable lists, exceptional lists in order.

`select()` is not an only way to handle concurrency. There are several ways like Asynchronous I/O, or multi-threads. Using `select()` has some pros and cons

Pros

1. The function can check for I/O completion on more than one socket(even for file object).
2. The function only uses single core and single thread, so it does not need to consider sync problem
3. `select()` is implemented in many different languages. If the

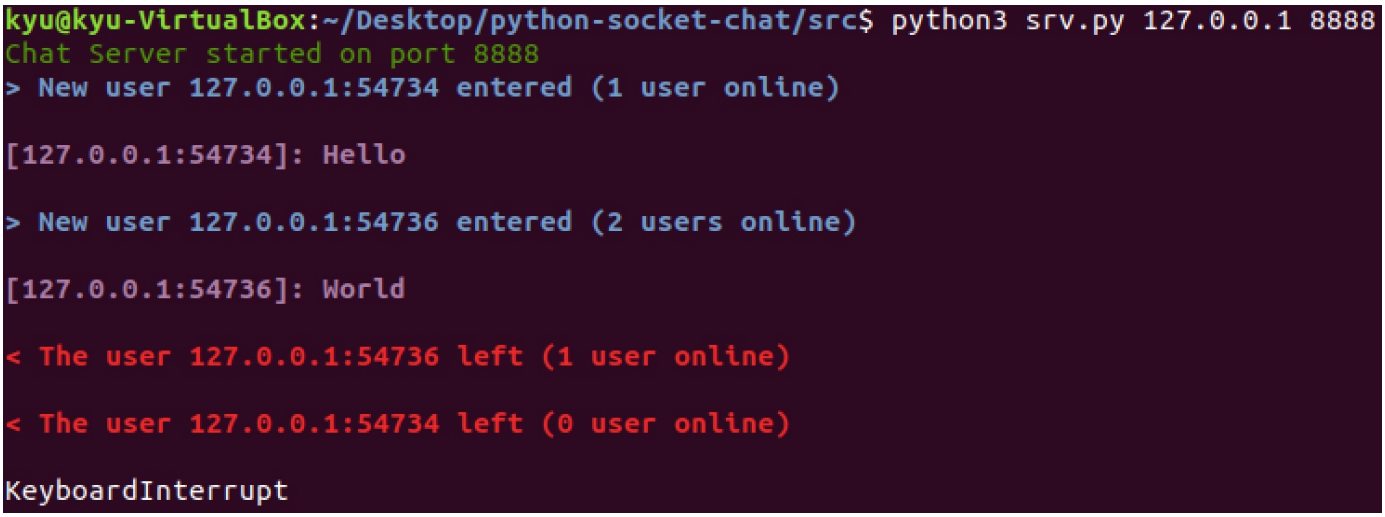
user understand once, user can use it in any languages.

Cons

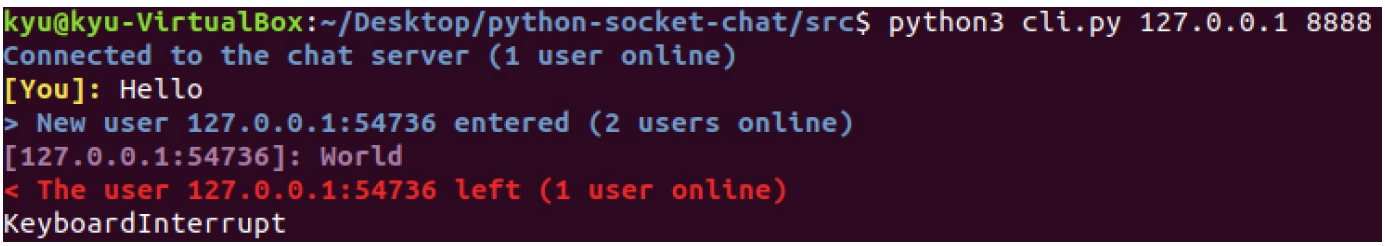
1. The function is impossible to run concurrently. If request to server increases, it may get some problem.
2. Compared to use other library, the user has to implement all event loop and handle errors

Snapshot

Server



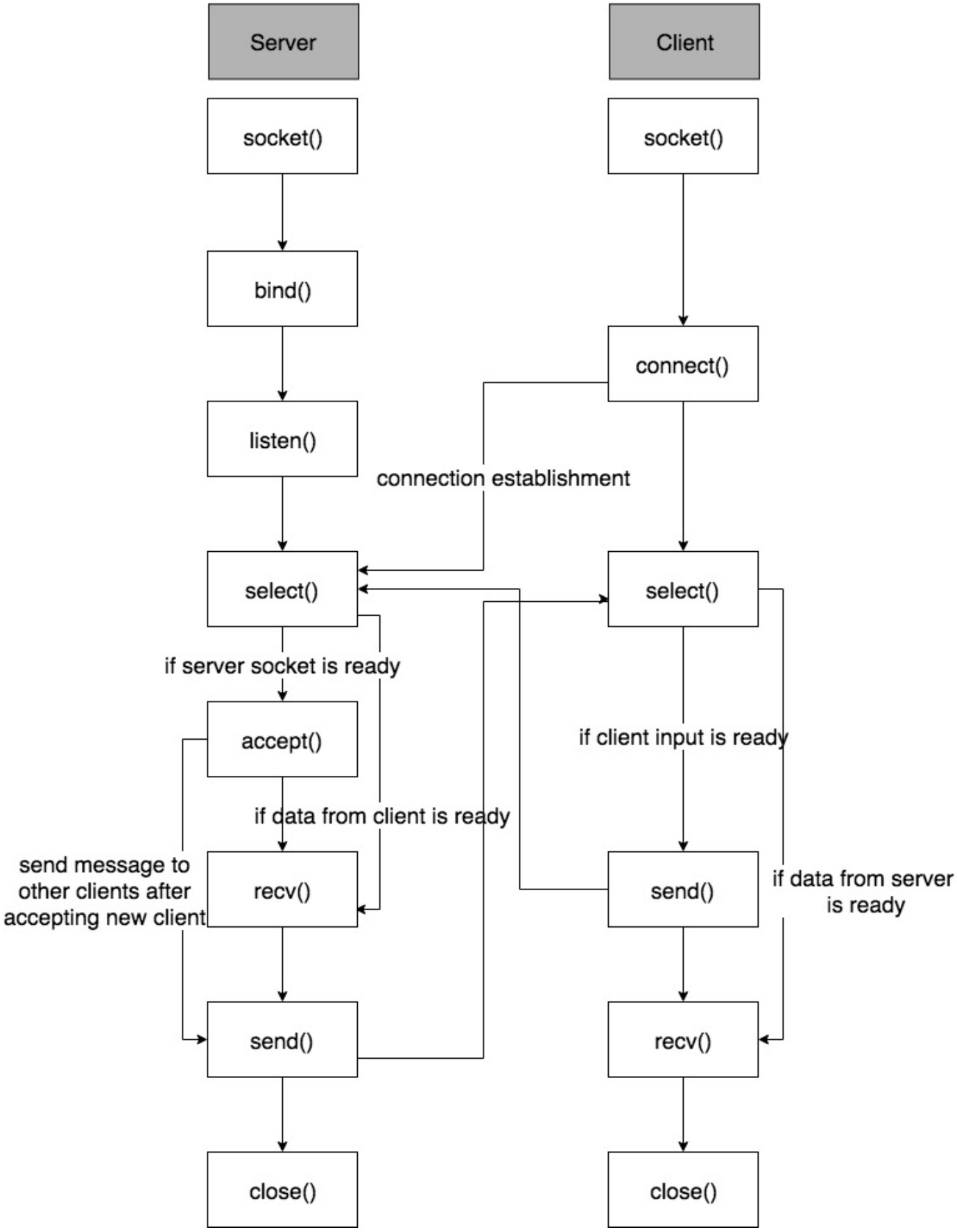
Client1



Client2

```
kyu@kyu-VirtualBox:~/Desktop/python-socket-chat/src$ python3 cli.py 127.0.0.1 8888
Connected to the chat server (2 users online)
[You]: World
[You]:
< You have been disconnected
```

Flow chart



The flow chart of this application is very similar to TCP connection. Because it is based on TCP connection. The only difference is that the

application has `select()` flow. It allows server to connect with many different clients. In addition, it allows client to handle both sys.stdin from client and data from server.

Code

cli.py

`main()` function

```
def main():
  # Get host and port number from arguments
  if len(sys.argv) < 3:
    print("usage: python3 ", sys.argv[0], "<host> <port>")
    sys.exit()
  else:
    host = sys.argv[1]
    port = int(sys.argv[2])
```

`main()` function first parse command line arguments to get `host` address of server and `port` number of server.

```
    # Create TCP socket and set timeout
    connSock = socket.socket(socket.AF_INET, socket.SOCK_S
TREAM)
    connSock.settimeout(Constants.TIMEOUT_TIME)

    # Connect to server with host and port number
```

```
    connect(connSock, host, port)

    while True:
        # Receive message from other clients
        # Send sys.stdin to other clients
        run_client(connSock)
```

Next, create TCP socket with AF_INET address family. The socket has blocking operations at default. By setting timeout with `settimeout()`, the socket has non-blocking operations now. Any socket operations raise a timeout exception when the operation time is longer than the timeout period value.

After creating socket, connect `connSock` to server socket whose address consists of `host` address and `port` number by using `connect()` function.

At last, run `run_client()` function until the connection is closed.

`connect(sock, host, port)` function

```
def connect(sock, host, port):
    # Connect to server
    try:
        sock.connect((host, port))
    except:
        print(Texts.CONNECT_ERROR)
        sys.exit()
```

`connect(sock, host, port)` gets socket, host address, and port number as an argument. The function tries to connect to server socket whose address consists of host and port. If it fails to connect, it prints the connection error message and exits the process.

run_client(connSock) function

```
def run_client(connSock):
    # Get the lists of sockets with sys.stdin and connecte
d socket as an inputs
    try:
        readableList, writableList, errorList = select.sel
ect([sys.stdin, connSock], [], [])

        for sock in readableList:
            # Receive message from server if readable sock
et is connected socket
            # Send message to server if readable socket is
 stdin
            recv_msg(connSock) if sock == connSock else se
nd_msg(connSock)


    except KeyboardInterrupt:
        # Handle Ctrl + C
        connSock.send(b"\n")
        connSock.close()
```

```
        print(Texts.KEY_INTER)
        sys.exit()
```

`run_client(connSock)` gets readable list which is a subset of sys.stdin (client input) and connSock (server). Readable socket will be selected by `select()` function. If connSock is ready to read, it receive the message by using `recv_messge(connSock)`. If sys.stdin is ready to read, it reads the message from sys.stdin and sends the message to server by using `send_msg(connSock)`.

The client process is terminated by three situations.

1. Terminated by SIGINT (Ctrl + C keyboard input)
   First case is handled by except clause. Client sends terminal word (\n) to server, closes connection, and exits the process.
2. Terminated by terminal word (\n)
3. Terminated by server
   Second and third case will be discussed in `recv_msg(connScok)` function.

recv_msg(connSock)

```
def recv_msg(sock):
    # Recieve message from server
    data = sock.recv(Constants.RECV_BUFSIZE)
    if not data:
        print(Texts.DISCON_MSG)
```

```
            if sock:
                sock.close()
            sys.exit()
        else:
            sys.stdout.write(data.decode('utf-8'))
            display_you()
```

`recv_msg(sock)` receive message from server. The maximum amount of data to be received at once is defined by `Constants.RECV_BUFSIZE`. When client sends terminal word(\n), server will close the connection, so client can not get data. Client closes the socket and exit the process. When server is terminated, client can not get data. For this case, client again closes the socket and exit the process.

If the client accepts proper data, client will write the data on console. Data is message from other clients or server. After that, display '[YOU]' on console by using `display_you()`.

send_msg(sock)

```
def send_msg(sock):
    # Send message from client
    message = sys.stdin.readline()
    sock.send(message.encode('utf-8'))
    display_you()
```

`send_msg(sock)` sends message to server. It reads the message from

standard input, and sends it to server. Before sending the message, the str type message is encoded to byte. Because socket sends and receives byte. After sending message, display '[You]' on console by using `display_you()`.

display_you()

```
def display_you():
    # Display 'YOU' on console
    sys.stdout.write(Texts.SHOW_YOU)
    sys.stdout.flush()
```

`display_you()` displays the '[YOU]' on console by using `sys.stdout.write(Texts.SHOW_YOU)`. After writing it, flushing out standard output.

Texts and Constants

```
class TextColors:
# ANSI escape sequences for colored text
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    RED = '\33[31m'
    YELLOW = '\33[33m'
    BLUE = '\33[34m'

class Texts:
    SHOW_YOU = TextColors.YELLOW + TextColors.BOLD + "[You
```

```
]: " + TextColors.ENDC
    CONNECT_ERROR = TextColors.RED + TextColors.BOLD + "\r
 You can not connect to the server, check IP and host \n"
+ TextColors.ENDC
    DISCON_MSG = TextColors.RED + TextColors.BOLD + "\r< Y
ou have been disconnected \n" + TextColors.ENDC
    KEY_INTER = "\rKeyboardInterrupt\n"

class Constants:
    TIMEOUT_TIME = 2
    RECV_BUFSIZE = 2 ** 12
```

Three class defines texts and constants. `TextColors` defines ANSI escape sequences, which is used for colored message on console.

---

srv.py

main()

```
def main():
    # Get host and port number from arguments
    if len(sys.argv) < 3:
        print(Texts.USAGE % sys.argv[0])
        sys.exit()
    else:
        host = sys.argv[1]
        port = int(sys.argv[2])
```

```
    # Open server socket and bind it to input ip, port
    servSock = socket.socket(socket.AF_INET, socket.SOCK_S
TREAM)
    servSock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSE
ADDR, 1)
    servSock.bind((host, port))
    servSock.listen(Constants.LISTEN_BACKLOG)
```

`main()` function first parse command line arguments to get `host` address of server and `port` number of server.

Next, create TCP socket with AF_INET address family. The socket is set to reuse address, so server can reuse the same address even though it is terminated by error. After setting the option, the socket is bound to address by host address and port number. It starts to listen connection with backlog `Constants.LISTEN_BACKLOG`.

```
    # Add server socket to the list of socket
    connList = []
    connList.append(servSock)

    # Print server start message
    print(Texts.START_SRV % port )

    while True:
        # connect to new clients
```

```
        # receive message from a client and send it to the
 other clients
        run_server(servSock, connList)

    if servSock:
        servSock.close()
```

Server socket is appended to `connList`, which holds the connected sockets. It prints the start of server. At last, run `run_server()` until the connection is closed. Before finishing the process, it closes the server socket if it still opens.

run_server(servSock, connList)

```
def run_server(servSock, connList):
    try:
        # Get the lists of sockets with sockets from conne
cted sockets
        readableList, writableList, errorList = select.sel
ect(connList, [], [])

        for sock in readableList:
            # Connect to new client if readable socket is
server socket
            # Receive message from a client and send the m
essage to other clients if readable socket is client socke
t
            connect_client(sock, servSock, connList) if so
```

```
ck == servSock else recv_and_send_msg(sock, servSock, conn
List)
```

`run_server(connSock, connList)` gets readable list which is a subset of `connList`. `connList` includes both a server and all clients. Readable socket will be selected by `select()` function. If `servSock` is ready to read, server connects to new client by using `connect_client(sock, servSock, connList)`. If client sockets are ready to read, server receive message from a client and send the message to the other clients by using `recv_and_sen_msg(sock, servSock, connList)`.

```
    except KeyboardInterrupt:
        # Handle Ctrl + C
        # Close socket server before exiting the process
        for sock in connList:
            if sock:
                sock.close()

        print(Texts.KEY_INTER)
        sys.exit()
```

The server is terminated by SIGINT (KeyboardInterrupt). It is handled by except clause. Server closes all the client sockets connected to server, and server socket itself. It prints the interruption message and exits the process.

connect_client(sock, servSock, connList)

```
def connect_client(sock, servSock, connList):
    # Connect to client
    newConnSock, addr = servSock.accept()
    ip, port = addr
    connList.append(newConnSock)

    # Send connection message to other clients
    conn_msg = make_conn_msg(connList)
    join_msg = make_join_msg(ip, port, connList)

    newConnSock.send(conn_msg.encode('utf-8'))
    send_all(newConnSock, servSock, join_msg.encode('utf-8
'), connList)
    print(join_msg)
    return
```

`connect_client(sock, servSock, connList)` accepts the connection from client socket. New client socket is appended to `connList`. Server sends `conn_msg` to newly connected client and `join_msg` to other clients. At last, it prints the `join_msg` to server itself.

send_all(sendSock, servSock, message, connList)

```
def send_all (sendSock, servSock, message, connList):
    # Send message to connected sockets except server and
```

```
sender
    for sock in connList:
        if sock != sendSock and sock != servSock:
            try:
                sock.send(message)
            except:
                if sock:
                    sock.close()
                connList.remove(sock)
```

`send_all (sendSock, servSock, message, connList)` sends the message to sockets in `connList` except server and socket who first sends the message to server. If server fails to send the message to socket, it close the socket and remove the socket from `connList`

recv_and_send_msg(sock, servSock, connList)

```
def recv_and_send_msg(sock, servSock, connList):
    # Receive message from a client and send the message t
o the other clients
    try:
        # Get data
        data = sock.recv(Constants.RECV_BUFF).decode('utf-
8')

        # Get address of client sending the message
        ip, port = sock.getpeername()
```

`recv_and_send_msg(sock, servSock, connList)` tries to receive data from socket, and gets ip address and port number of the socket.

```
        if data == "\n":
            # Client exits the chat room when the data is
new line
            # Send closing connection message to other cli
ents and server
            msg = make_leave_msg(ip, port, connList)
            send_all(sock, servSock, msg.encode('utf-8'),
connList)
            print(msg)

            # Close connection
            connList.remove(sock)
            if sock:
                sock.close()
            return
```

If the received data is “\n”, server sends leaving message to all other clients and server, and close the connection to the socket.

```
        else:
            # Send message to the other clients and server
            data = data.rstrip()
```

```
            msg = Texts.SEND_MSG % (ip, port, data)
            send_all(sock, servSock, msg.encode('utf-8'),
connList)
            print(msg)
            return
```

If the received data is not “\n”, server formats the data and sends formatted data to other clients and server.

```

    except:
        # Handle error
        ip, port = sock.getpeername()

        # Send error message to other clients and server
        msg = make_leave_msg(ip, port, connList) + " (erro
r) "
        send_all(sock, servSock, msg.encode('utf-8'), conn
List)
        print(msg)

        # Close connection
        connList.remove(sock)
        if sock:
            sock.close()
        return
```

If the function fails to receive data or get any other error, it gets ip address and port number and make error message. The server sends the error message to other clients and server. The socket raising error will be removed from `connList` and close its connection.

make_msg() functions

```
def make_conn_msg(connList):
    # Get current user numbers and make connection message
    nUser = len(connList) - Constants.NUM_SERVER
    user = Texts.USER if nUser == 1 else Texts.USERS
    return Texts.CONN_MSG % (nUser, user)

def make_join_msg(ip, port, connList):
    # Get current user numbers and make join message
    nUser = len(connList) - Constants.NUM_SERVER
    user = Texts.USER if nUser == 1 else Texts.USERS
    return Texts.JOIN_MSG % (ip, port, nUser, user)

def make_leave_msg(ip, port, connList):
    # Get current user numbers and make leave message
    nUser = len(connList) - Constants.NUM_SERVER - Constan
ts.NUM_LEAVING_CLIENT
    user = Texts.USER if (nUser == 1 or nUser == 0) else T
exts.USERS
    return Texts.EXIT_MSG % (ip, port, nUser, user)
```

`make_*_msg()` counts current client users and make messages for its purpose. `connList` contains server socket, so it has to exclude `Constraints.NUM_SERVER` when counts client users.

Texts and Constants

```
class TextColors:
    # ANSI escape sequences for colored text
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    RED = '\33[31m'
    GREEN = '\33[32m'
    YELLOW = '\33[33m'
    BLUE = '\33[34m'
    PURPLE = '\33[35m'

class Texts:
    USAGE = "usage: python3 %s <host> <port"
    START_SRV = TextColors.GREEN + "Chat Server started on port %s" + TextColors.ENDC
    NAME_EXIST = TextColors.RED + TextColors.BOLD + "\r Username already exists!\n" + TextColors.ENDC
    WELCOME_MSG = TextColors.GREEN + TextColors.BOLD + "\r Welcome to this chat application. You can exit with Enter ('\\n')\n" + TextColors.ENDC
    CONN_MSG = TextColors.BLUE + TextColors.BOLD + "\rConnected to the chat server (%s %s online)\n" + TextColors.ENDC
```

```
    JOIN_MSG = TextColors.BLUE + TextColors.BOLD + "\r> Ne
w user %s:%s entered (%s %s online)\n" + TextColors.ENDC
    EXIT_MSG = TextColors.RED + TextColors.BOLD + "\r< The
 user %s:%s left (%s %s online)\n" + TextColors.ENDC
    SEND_MSG = TextColors.PURPLE + TextColors.BOLD + "\r[%
s:%s]: %s\n" + TextColors.ENDC
    USER = "user"
    USERS = "users"
    KEY_INTER = "\rKeyboardInterrupt\n"

class Constants:
    RECV_BUFF = 2 ** 12
    LISTEN_BACKLOG = 10
    NUM_SERVER = 1
    NUM_LEAVING_CLIENT = 1
```

Three class defines texts and constants. `TextColors` defines ANSI escape sequences, which is used for colored message on console.

Reference

1. https://docs.python.org/3/library/socket.html#socket.AF_INET
2. https://realpython.com/python-sockets/
3. https://steelkiwi.com/blog/working-tcp-sockets/
4. https://stackoverflow.com/questions/287871/print-in-terminal-with-colors
5. https://gist.github.com/chrisopedia/8754917

6. https://pymotw.com/2/select/
7. https://stackoverflow.com/questions/15260558/python-tcpserver-address-already-in-use-but-i-close-the-server-and-i-use-allow
8. https://serverfault.com/questions/329845/how-to-forcibly-close-a-socket-in-time-wait
9. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_71/rz
10. https://www.gnu.org/software/libc/manual/html_node/Socket_002d
Options.html
11. https://notes.shichao.io/unp/ch6/