**11/11/2014**

Do Something Cool with Python

“An Seo!”

by

Darren Dowdall

A00202053

Network Management Year 3

Athlone Institute of Technology

# Table of Contents

[Do Something Cool with Python 4](#__RefHeading__1588_9281728)

[Purpose of the project 4](#__RefHeading__1590_9281728)

[Background to the project 4](#__RefHeading__1592_9281728)

[Initial Plan 4](#__RefHeading__1594_9281728)

[Plan B](#__RefHeading__1596_9281728)

[Choosing A Layout Manager](#__RefHeading__1598_9281728)

[Approach](#__RefHeading__1600_9281728)

[Asynchronous I / Woe 6](#__RefHeading__1602_9281728)

[Locks, Queues, Semaphores and the Select() Method](#__RefHeading__1604_9281728)

[Twisted Framework](#__RefHeading__1606_9281728)

[8Back to the Client 8](#__RefHeading__3251_471238570)

[User Manual](#__RefHeading__1610_9281728)

[System Prerequisites](#__RefHeading__1612_9281728)

[Server](#__RefHeading__1614_9281728)

[Client](#__RefHeading__1616_9281728)

[Login :](#__RefHeading__1618_9281728)

[Main GUI : 10](#__RefHeading__1620_9281728)

[Specification Tables](#__RefHeading__1622_9281728)

[Client Side](#__RefHeading__1624_9281728)

[ChatApp Class : Init Function()](#__RefHeading__1626_9281728)

[ChatApp Class : Initial-UI()](#__RefHeading__1628_9281728)

[ChatApp Class : Close-Event()](#__RefHeading__1630_9281728)

[ChatApp : Continue-Event() 14](#__RefHeading__1632_9281728)

[ChatClient Class : Init Function() 14](#__RefHeading__1634_9281728)

[ChatClient Class : ClientUI Function()](#__RefHeading__1636_9281728)

[ChatClient Class : Threaded Receive Function()](#__RefHeading__1638_9281728)

[ChatClient Class : GrabText Function() 16](#__RefHeading__1640_9281728)

[ChatClient Class : AppendText Function() 16](#__RefHeading__3253_471238570)

[ChatClient Class : KeyNow Function()](#__RefHeading__3255_471238570)

[MainLine](#__RefHeading__1642_9281728)

[Server Side 18](#__RefHeading__1644_9281728)

[ChatProtocol Class : Init Function() 18](#__RefHeading__1646_9281728)

[ChatProtocol Class : ConnectionMade Function() 18](#__RefHeading__1648_9281728)

[ChatProtocol Class : ConnectionLost Function() 18](#__RefHeading__1650_9281728)

[ChatProtocol Class : LineReceived Function()](#__RefHeading__1652_9281728)

[ChatProtocol Class : HandleRegister Function()](#__RefHeading__1654_9281728)

[ChatProtocol Class : HandleChat Function() 20](#__RefHeading__1656_9281728)

[ChatProtocol Class : BroadcastMessage Function() 20](#__RefHeading__1658_9281728)

[ChatFactory Class : Init Function() 20](#__RefHeading__1660_9281728)

[ChatFactory Class : BuildProtocol Function()](#__RefHeading__1662_9281728)

[MainLine](#__RefHeading__1664_9281728)

[Client Application : Login GUI 22](#__RefHeading__1666_9281728)

[Chat Application : Main GUI](#__RefHeading__1668_9281728)

[Twisted Chat Server 24](#__RefHeading__1670_9281728)

# Do Something Cool with Python

## Purpose of the project

The purpose of this project was to create a working chat like application using Python ( 2 or 3 ). For this to be acheived the project would require both a client and server side script to be written. The client script would be presented to the user in a styled GUI whereas the server would be solely command line based.

The client application would be responsible for :

* presenting the user with a GUI login box
* presenting the user with the “An Seo!” chat GUI
* passing data to the server
* receiving data from the server
* displaying new messages to the client interface

The server would be responsible for :

* controlling the connections to multiple clients
* passing messages between clients
* authenticating users

## Background to the project

### Initial Plan

The initial plan for this project was to build a voice recognition program using Python.

It would utilise one of Google's APIs to translate the speech received by the microphone (usually in a “.wav” or “.flac” file format) into a workable string of text that could be passed into the terminal to perform any basic terminal command.

However, problems with my build environment along with restrictions on data transmission size with Google’s voice recognition servers (only allowing for a few kbs to be processed at any given time) made this a lot tougher than it should have been.

Through some online research I managed to come across another speech API created by the people at CMU (Carnegie Mellon University) named CMUSphinx. It should have provided me with the tools I needed to translate the audio file into a string of text. But again it eluded me ; some evenings I would make progress with the problem from the evening before, only to be met with another issue almost immediately, and so, after three weeks of perseverance, I decided to switch to plan b.

### Plan B

My second idea for “Do Something Cool with Python” was to create a network based chat application.

My first step on the road to my goal was to create the GUIs that would be required. For this I leveraged the power of Jython (rather than the traditional CPython) coupled with the Java Swing GUI Toolkit. After many hours of experimentation with Swing, I felt happy enough to create the basic look of the client GUI.

### Choosing A Layout Manager

Java Swing comes with many different layout managers available to the developer i.e. GridLayout, BorderLayout, GridBagLayout etc., all of which have their own pros and cons. With some previous experience using a geometry based manager (providing x/y coordinates to define placement of GUI widgets) I decided I wanted to try something a little different. In the end I choose the “GroupLayout” manager to satisfy my needs. It works by adding widget components into both a Horizontal & Vertical grouping scheme in either sequential or parallel groups. Once the widgets have been added, the layout manager is able to calculate the desired position of the components based on their position within these groups.

After some time experimenting with the GUI tools, I was able to finalise my first draft of the client GUI. This left me with the simple task of adding the functional code to my project with the aim of tidying the GUI up once I had a working program.

### Approach

From my perspective, there were two ways to approach this type of application:

1. Peer to Peer ; where the application serves as both a client and a server

2. Client / Server ; where the server acts as the central connection point for all clients and passes info between them

The peer to peer approach would require the coding of just one script that provided both client and server functionality between instances of the chat application, passing messages directly from one to the other. This seemed like it would be simple to code and to get working, but it lacked the “Cool” factor that a multiway chat application would not.

The final decision to choose the client/server approach enabled me to progress with further research into the inner workings of the program and how to approach some of the more intricate tasks involved within.

### Synchronous I / Woe

For both the client and server scripts to work as they were intended there had to be a way for them to both receive & send data (almost) simultaneously. This would require the use of “Multi-Threading”; a separate process that splits from the main thread of the program to perform a given task.

On the client side, the program needed to be able to listen for incoming messages from the server (on a separate receiving thread) as well as being able to send messages through an event call-back ( button click ) from the GUI.

The server side was where most of the issues arose during the development stage. The original implementation was designed to continuously listen on a default port for new incoming connections. When a new connection was established, the main thread would register the new user, provide them with a new unique port and then initialise a thread to deal with the new socket. The main thread then returned to listening for any new incoming connections. \*It is important to note that there are two ways to implement threading in Python: the first is to just call the Thread method on an existing function, this is a good, functional starting off point but it is not as elegant as the second option which is to subclass the threading module and tinker with it's run method. Coupled with it's \_\_init\_\_ method this gives the developer much more control over how the thread should function. This was the route that I originally chose to take.

With time and perseverance I managed to get the code to function as desired and I can safely say that during this period of playing around with threading I have gained more knowledge about Python, and programming in general, than I ever thought possible.

Once I had the handover functionality up and running, I was able to shift my focus to the sharing of messages between clients.

Each time a new message was received from a client, the server would identify the sender, attach the senders name to the message and then append the new chunk of data to a list for the other threaded instances to read from. This list variable had to be visible to all instances and it took quite some time to figure out how to make this happen. Eventually I managed to get this to work (without the use of global variables I might add) and now I had the functionality for not only creating new connections and threads, but also having these threads share the same variable state.

The final hurdle was to pass new messages to the client side as they came in. Each server thread would have to identify messages that were meant for them (messages not created by their client) in the list, it would then retrieve the message and send it across the socket to the client application which would then present it to the user. The issues that arose at this stage were ones of blocking sockets. \*Looking back now, I think that I should have threaded the receive function inside each connection thread, in a similar manner to the way it is implemented in the finished client application, to handle incoming messages.

### Locks, Queues, Semaphores and the Select() Method

After spending some time researching the ins and outs multi-threaded network programming (shout out to stack overflow) I came across a few different approaches to dealing with my blocking sockets issue. Locks, queues and semaphores fall into a category known as 'Synchronisation Primitives'. Once called and assigned to a variable, these methods produce an object that can be acquired for a short period of time by each thread, this in turn initiates some processing before releasing the object and signalling the other threads to take their turn.

The 'select.select(socket\_list)' method tries to make this a little easier by treating each socket connection as a file. Once called, with a parameter in the shape of a list of available sockets, it pops out three individual lists for working with; a read\_list, write\_list and exception\_list. The concept behind this is that each thread locks onto each list(s) in turn, checks it for whatever details it requires and then, like the synchronisation primitives mentioned earlier, releases it for the next thread to process. This should have worked ( and it kind of did work in the end ) but the issue with the blocking sockets prevailed.

After being stuck at this point for a decent amount of time, I decided to look into alternative methods for creating a server to handle my connections. It was by chance that I stumbled onto the 'Twisted Python' API, which was the pivotal point of the project.

### Twisted Framework

Twisted is an event driven network programming framework that supports many different network protocols. It employs instances of “Factories” to build, manage and maintain the input/output features required in such networking programs as the Chat Server.

The resulting code is implemented using call-back functions which are called by the framework once a given event has occurred. To pick up on these events a reactor object is created. The reactor is then attached to the server object and told to listen for events that the program would be interested in.

Under the hood, the reactor uses a polling mechanism to produce the desired I/O features. Each time an event happens on the transport (which is basically a twisted term for a live readable/writable connection) the reactor calculates which function the event is associated with as well as what data it should then provide to the function.

After a brief introduction to Twisted (utilising some amazing online tutorials coupled with a great book on the subject) I was able to construct a basic TCP based server application which was capable of connecting multiple telnet clients In a chat like scenario.

### Back to the Client

With the twisted server up and running (accepting user instances over telnet) the only real work left to do was to figure out how to connect the client into the new implementation of the server. At first I tried to connect directly in using standard streaming sockets as in the applications previous iteration, however this did not work as expected ( that's not to say that it is impossible, but with time ticking away I decided to approach this in a different manner).

Next step was to try build my client application using twisted, but alas it was not meant to be as Jython would not accept the importing of some of the twisted modules (even though it was installed on Python 2 which Jython uses).

This got me to thinking; If I can make telnet connections from the terminal, why not make them also from the client application. After a quick search online I discovered a python module called “telnetlib”.

A short time later “An Seo” was born ( and I did a funky dance around my kitchen ).

# User Manual

## **System Prerequisites**

### Server

The server application requires the package “python-twisted” to be installed before it can executed.

The server is written for python 2.7 and can be executed from the terminal using the command *“python path/to/file.py”*

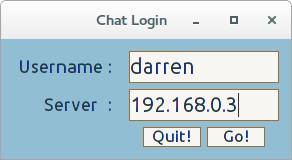
*\* To shut down the server simply close the terminal window or ctrl+z. Once started, the server requires no interaction on behalf of the user.*

### Client

The Client Chat application requires that both Java and Jython are installed before it can execute. It also requires the “telnetlib” module to be installed, which is provided by default in most Linux operating systems.

The client application can be executed from the terminal using the command “*jython path/to/file.py*”

## Login :

Login

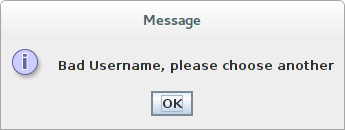
Upon initialising the client application the user is presented with a window consisting of two entry fields :

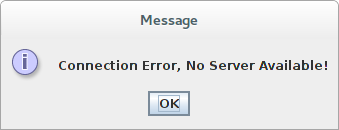
1: requires the input of a username

2: requires the IP Address / Hostname for the server

If either of these details are unacceptable (e.g. either the username is already taken or there is no connection available) then the user is presented with a Dialog Box explaining the error as well as allowing them choose what they would like to do next ( re-enter details or quit ).

If all details are validated, the application proceeds to the main client GUI whilst hiding the login GUI behind the new window.

Bad Username

Bad Host

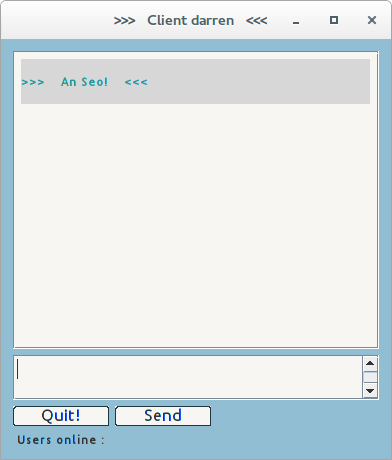
## Main GUI :

After validation the user is presented with the client GUI window. At this point their username is registered with the server application and they can proceed to chat with any other connected clients. If a new message is received by the client program, it is immediately displayed in the main content area, formatted in a way that identifies the client the sent the message as well as a delimiter to show where the message begins and ends.

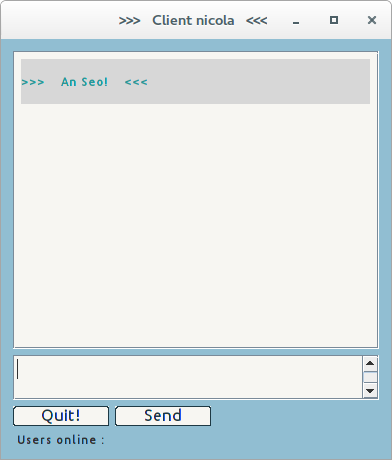
To write a message the user types into the lower text field and clicks the send button. This event relays the message to the server who then passes the new message to all clients except the one that created it. This message is also displayed in the main content area as soon as the button is clicked.

While connected to the chat application, the user will be notified of any new clients who join the chat with a message in the content area. If a user departs the chat a message is broadcast to all current clients notifying them of the departure. This is also displayed in the content area.

To leave the chat, the user can click on the quit button or the default close window button. Once this event happens, the server removes the clients name from its database and broadcasts the departure of the client to all other connected clients.

Main GUI

*Deafault Main Window & Initial message display*

New Client

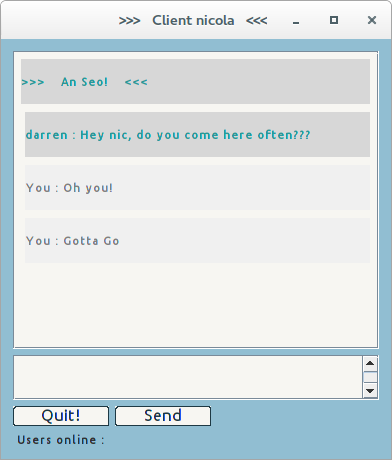
New Client Notify

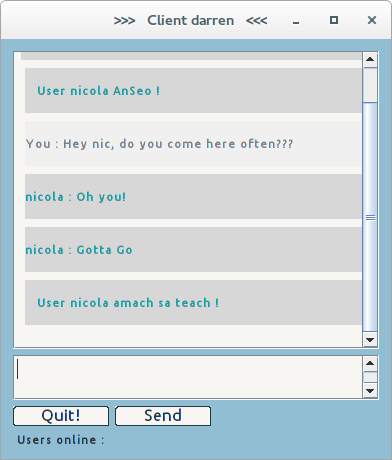
*New Client Joins & User Notification Received*

Nicola Chatting

Illustration 1: Darren Chatting

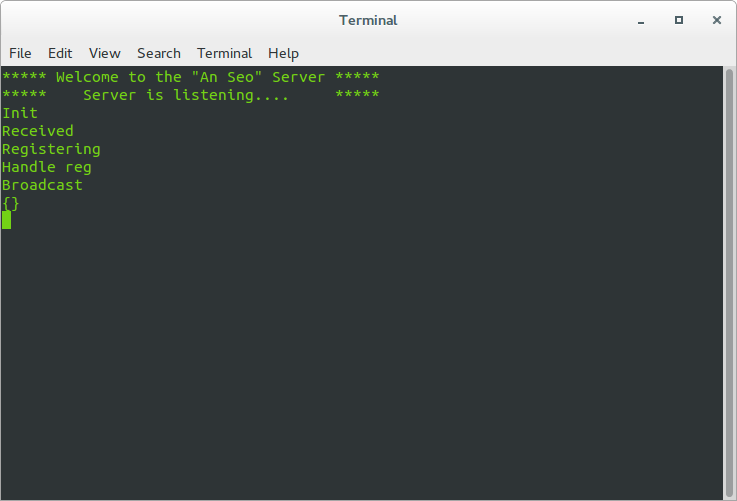
*Client nicola receives the message from client darren & responds*

Client nicola leaves

Departure Notify

*User nicola departs chat & user darren recieves notification*

*Server Receives, Registers new users and handles chat message transportation*

Server Side

# Specification Tables

## Client Side

### ChatApp Class : Init Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance | Call the super class  Initiate the Login GUI | Base Class Instance |

*Psuedocode:*

Call to super method to initialise the base class

Call to Initial GUI function

### ChatApp Class : Initial-UI()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance | Create Layout  Create Widgets  Define widget actions  Component Placement | Login GUI |

*Psuedocode:*

Build the GUI:

Create widgets

Attach callbacks to the buttons

Add widgets to the Content Pane using group layout manager

Set the defaults for the GUI window

### ChatApp Class : Close-Event()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Event (on quit button click) | Display Goodbye message | Exit Application |

*Psuedocode:*

If the users clicks “quit” button, close the application

### ChatApp : Continue-Event()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Event (on go button click)  or  Event (on return keypress)  Strings from Text Fields | Grab strings from entry fields  Try make a telnet connection   * send username * acknowledge connection * Display Dialog if username taken * Call Main GUI   Except when none available   * Display Diaglog | Telnet connection  Dialog displayed  Call to Main GUI |

Psuedocode:

If the user clicks the “go” button:

Grab the text from the entry fields

Try to establish a connection:

Send the username to the server

Process the response

If the response is a bad username:

Present a dialog and ask the user to re-enter the details

Hide the GUI & Call the main App

Except If there is no connection available:

Then present a error dialog box to the user

Reset the text and allow the user choose what to do

### ChatClient Class : Init Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Insance  Username  Host port  telnet connection | Call the super class  Initialise Variables  Thread Receiving function  Call the Main Client UI | Threaded function  Main GUI  Instance variables |

Psuedocode:

Call the super method to initialise the base class

Set the instance variables

Create the threaded function, set to daemon and run

Call the Main UI

### ChatClient Class : ClientUI Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance | Create Layout  Create Widgets  Define widget actions  Component Placement | Main Chat GUI |

Psuedocode:

Build the GUI:

Create widgets

Attach callbacks to the buttons

Add widgets to the Content Pane using group layout manager

Set the defaults for the GUI window

### ChatClient Class : Threaded Receive Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Telnet connection  Incoming data | While connection available:  Try receive message   * Pass to AppendText function   Except if there is no connection   * Pass | Call to AppendText function |

Psuedocode:

While the connection is alive

Try to receive a message from the connection:

If a message is received:

Call the AppendText function to display message

### ChatClient Class : GrabText Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Event (on send button click)  String from text area | Retrieve string from text area  Do not send empty string  Set the field to empty & gain focus  Write the data to the connection | AppendText function call  Outgoing data |

Psuedocode:

On “send” button click:

Retrieve the text entered in the text field

If it is an empty string:

then drop the message

Otherwise:

Format the message

Pass the message to the AppendText function to display message

Reset text and gain focus in the message area

Send the message out the conection

### ChatClient Class : AppendText Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Message  User | Create label & format depending on user  Set label attributes  Display label in content area | New message label  Displayed in window |

Psuedocode:

Create the message label

If user is current user

format appropriately

Otherwise no user name

format appropriately

Set the label attributes

Insert the new label to the content area

repaint the window (just to be sure)

### ChatClient Class : KeyNow Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Event (on return key press)  String from text area | Create a Key class object  Pass the Key object the event  Test the returned value  Call the grab text function | GrabText function call  New class object  Event |

Psuedocode:

On return keypress (inside text area) call KeyNow function

Create a Key class object

Pass the objects KeyPressed function the event and store the returned value to a variable

If the value is equal to 10 (keycode for return key)

Call the GrabText function

### MainLine

Psuedocode:

If the name of this instance is “\_\_main\_\_”

Call the ChatApp class

## Server Side

### ChatProtocol Class : Init Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Chat Factory Object  User Dictionary | Initialise instance variables | Instance variables |

Psuedocode:

Reactor Event : New Incoming Connection :

Initialise the instance variables

Set the initial user name to “None”

Set the initial user state to unregistered

### ChatProtocol Class : ConnectionMade Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Reactor object  Connection object | If New user  Send Acknowledge  Set Client state to New User | Client State  Outgoing data |

Psuedocode:

Reactor Event : Connection Complete

If the state is that of a new user:

Send the client a greeting

### ChatProtocol Class : ConnectionLost Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Reason/Exception variable  Reactor object  Connection object  User Dictionary | Remove the user from the user dictionary  Notify all clients of departure | User removed from user dictionary  Outgoing notification to clients |

Psuedocode:

Reactor Event : Connection Dropped

Check if the username is stored in the user dictionary

Remove it if it is

Broadcast a Departure message

### ChatProtocol Class : LineReceived Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Line/Connection object  Reactor object  State variable | If client state is new user   * pass control to handle register function   Otherwise   * pass control to handle chat function | Handle chat function  or  Handle register function |

Psuedocode:

Reactor Event : Data received on the transport

If this is a new client:

Handle the registration of the new client

Otherwise:

Handle the chatting function of the server

### ChatProtocol Class : HandleRegister Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Line/Connection object  Reactor object  User Dictionary | If name taken, notify client  Else acknowledge   * Send greeting * Set the instance name to username * Add client to user dictionary * Set the client state variable to Chatting | Client Notification  Username added to dictionary  State set to chatting |

Psuedocode:

If the username already exists on the server:

Request a new name from the client

Otherwise:

Send an Acknowledgement & Greeting

Set the instance name to the that of username received

Update the user dictionary

Set the state to that of a registered client

### ChatProtocol Class : HandleChat Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Line/Connection object  Reactor object  Message string | Format the message  Pass control to broadcast message function | Formatted Message  Broadcast Message function |

Psuedocode:

Format the message for transport

Broadcast the message to all clients

### ChatProtocol Class : BroadcastMessage Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Line/Connection object  Reactor object  Formatted Message  User Dictionary | For each item in the dictionary that is not this client   * Send the new message | Outgoing data |

Psuedocode:

For each client in our dictionary:

That is not this client:

Send the new message

### ChatFactory Class : Init Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance | Create the user dictionary object | Display server up message |

Psuedocode :

Initialise the shared state dictionary

Display server greeting

### ChatFactory Class : BuildProtocol Function()

|  |  |  |
| --- | --- | --- |
| Input | Processing | Output |
| Self Instance  Address | Return a ChatProtocol object | ChatProtocol object |

Psuedocode :

Call the protocol class (instance) for each new client

### MainLine

Psuedocode:

Create a Factory instance

Notify the reactor to listen for incoming TCP connections on the default port

Initialise the reactor object

# Client Application : Login GUI

# Chat Application : Main GUI

# Twisted Chat Server