1. Code and docs

1.1 Working with JSON in python

Working with JSON in python is described fairly good here (http://pymotw.com/2/json/) and here http://docs.python.org/2/tutorial/datastructures.html#dictionaries,, and it is also included some working examples in the project code folder. Working with JSON in python is very easy; if you are familiar with *dictionaries* in python (roughly the same as a HashMap in Java), there really is not much of a difference. Using the built-in *json* library in python, you can easily serialize and deserialize dictionary objects.

1.2 TCP Socket programming in python

Before getting started on the project, it is highly recommended that you read and understand the concepts of socket programming. A thorough theoretical description with some simple code examples is given in section 2.7 and 2.7.2 (p. 182-183 and 189 - 193, Kurose & Ross 6th edition) in the textbook.

This section will list the most useful classes and methods used for implementing a server that can handle multiple connected clients via TCP sockets at once. Python has this covered in a very elegant way that makes it rather easy, and with a nice and clean implementation.

1.2.1 The python socket object

This section lists the most important low-level functionality of a python TCP socket. To use it, just import the socket package in your python file, and instantiate a socket object by calling socket.socket().

If something is unclear or wrong in this guide, notify the course staff so that it can be corrected, preferably on the it's learning forum so that everybody can benefit from the discussion. In addition you might check out the documentation at http://docs.python.org/2/library/socket.html as well.

connect(address)
Connect to a remote socket at address. The address is a pair tuple (IP_ADDRESS, PORT).
It will throw an exception if the socket in the other does not exist or accept connections to the specified address.

send(string_data)
Send *string_data* to the socket. The socket must be connected to a remote socket.

NOTE:

Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. The send method returns the number of bytes sent, so one could easily implement this functionality. However, it is rarely necessary to do this, and for our use-case it should not be considered necessary to any checks of this kind.

sendall(string_data)
Send data to the socket. The socket must be connected to a remote socket. Unlike send(), this method continues to send data from *string_data* until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

recv(buffer_size)
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by buffer_size. This method will block, i.e. it will stop all further execution of code until data is received.

NOTE:
For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for example, 4096.

It is possible to have non-blocking sockets where you specify a timeout for the socket recv() call, but this is not the recommended approach for this project, as it will only add complexity.

close()
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

NOTE:
Close() releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call shutdown() before close(). This knowledge can be useful to have have if something is not working or is behaving in an unexpected fashion.

shutdown(how)
Shut down one or both halves of the connection. If how is SHUT_RD, further receives are disallowed. If *how* is SHUT_WR, further sends are disallowed. If how is SHUT_RDWR, further sends and receives are disallowed. Depending on the platform, shutting down one half of the connection can also close the opposite half (e.g. on Mac OS X, shutdown(SHUT_WR) does not allow further reads on the other end of the connection).

NOTE:

the SHUT_WR, SHUT_RD and SHUT_RDWR constants are in the socket package that needs to be imported in order to use the socket object (as described in the introduction).

bind(address)
Bind the socket to an *address* on the host machine. The address is a pair tuple (IP_ADDRESS, PORT). This is usually what you must do on a server prior to accepting incoming connections on the socket. The socket must not already be bound, or else an exception will be thrown.
.
accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

1.2.2 SocketServer
The SocketServer package contains a lot of useful tools for working with TCP (and UDP as well) sockets in python. In fact, using the classes and methods contained in the SocketServer package, you don't have to use the server-side socket methods such as accept() and bind(); this is handled automatically. Therefore, it is really recommended that you understand and use the concepts described in this section, because it will save you a lot of work.

1.2.2.2 SocketServer.BaseRequestHandler

The BaserequestHandler must be subclassed and the handle() method must be overriden in order to define your custom actions and communiction pattern that should be performed upon and after a succesful connection to the server.

When a remote socket connects to the server the request handler is instantiated (once per connection) and the handle() method is called. Check the code skeleton for a very simple example on how to do this.

This is a very powerful pattern, because we don't have to worry about handling connecting from mulitple clients ourself, it is all executed in a multi-threaded environment automatically in the provided code. Further, all data associated with a connection can be stored in the handler object for the connection, instead of having to manage this manually.

1.2.2.2 SocketServer.TCPServer

The TCPserver is very straightforward. in order to start it, it must be passed a binding address (a touple as described in the socket section), and the handler class to implement for each connection. This should be the handler class that you have subclassed in the section above. Then you simply call the serve_forever() method, and everything should automagically work.

4.3 Thread programming

As mentioned in 4.2.2.2, the connection handler is executed in a multi-threaded environment automatically. However, you should conceptually try to understand what is really going on.

Consider these points:

- Listening for new connections
- Listenening for new data from all (individual) live connections

At every given time, the server must be able to do this - simultaneously. That means that we cannot have everything run in a single process, we need to execute different tasks in different *threads*.

The textbook approach to achieve this is to:

- Create a main thread for the server
- Create a new thread for listening for new connections to the server socket.
  - i.e a thread that calls accept() on the server socket in an infinite loop
- When a new connection is returned, create a new thread for *this* socket that listens for incoming data on the socket
  - i.e a thread that calls recv() in an infinite loop

As said, you do not need to consider these issues, as the BaseRequestHandler and the TCPServer handles this for you.

The tricky part is that the client side needs to be able to send data to the server at any given time - while also receiving data. This must be implemented manually, and is one of the critical tasks of this project. Perhaps this section gave you a few hints on how to implement this?

1.3.1 Creating and executing a thread in python

Creating and executing a thread in python is fairly straightforward. You need to 1) subclass the Thread class, and 2) override the run() method to give the thread some work to do. Then instantiate the Thread and call the start() method on the object.

NOTE:
DO NOT call the run() method directly, as this will cause the thread to block and suspend the entire calling thread's stack until the run() is finished. it is the start() method that is responsible for actually executing the run() method in a new thread. Therefore, implement the tasks to be done in run(), and call start() on your Thread object after instantiating it.

If your python script still "hangs" after quitting it with control-C in the terminal, bear in mind that if a thread that the script started is in a blocking call or an infinite loop, the thread will not terminate. Thread objects have a boolean property called *deamon*, which states wheter or not the thread should be terminated when it's parent thread is terminated (which is usually the case). In the code examples, the thread is set with this property to be true.

1.4 Tips and tricks

1.4.1 Initial steps

The code skeleton implements a server that allows clients to connect, and echoes back whatever the client sends in uppercase to the client. The client side just sends “hello” and then finishes. You should start by implementing the functionality to keep the client and server connected. For instance you could start by altering lines 14-16 in *client.py* to reading input from the user in an infinite loop and sending the user input to the server. Then you would have a persistent connection to the server where you can send data to the server and receive data back after each time you send something.

The next logical step would be to implement the functionality to enable the client to send and receive data at the same time (ref 4.3). Sockets are able to be read from and written at the same time, so that should not be a concern. When this in place, the you will have a platform to build further upon.

1.4.2 Architecture

Since this is a group project, it might be smart to split up to different working packages. A common way to structure applications is to split it (just as the IP-stack) into different layers where different functionality is performed. A suggestion for dividing the responsibility in the layers for this project is to have a:

- Networking layer
  - Sending and receving data (strings) as described in the above sections
- Parsing layer
  - Parse strings that is sent and received (for our case, this will typically be serializing python dictionaries to JSON strings and vice versa)
- Application layer
  - Application logic

When talking about “layers”, it will usually mean that you make a class (possibly using other classes) that handles the responsibilities in a given layer. Then, the “layering” happens through the objects calling methods on each other. The essence is then that the application layer will never call a method in the network layer, it will call a method in the parsing layer, which will then call a method in the parsing layer. It might not be possible to do it in an entirely clean-cut way, but if you discover that methods are growing out of bounds, consider making utility classes for such methods.

Layering your application like this will enable you to divide the work and let you progress on different parts without being too dependent on other parts. It should be mentioned that it does not have be done this way, it is just a suggestion to help you along the way.

Further, having a working networking layer might enable you to reuse it for other projects of your own or the common project.

1.4.3 Parsing incoming data

When deciding what to do with the response objects, a hot tip is to first check for errors (invalid name, not logged in, etc). An error is only present if it actually happened, else it will not be there (thus the request was successful). Therefore you can deserialize the incoming string and check if the ‘error’ key is present in the resulting dict object and handle the error in a timely fashion.