My First Sock..

Server Side

Concept

It wouldn't make much sense to try any make a phone call to someone who doesn't have a phone..

Same with sockets.. We first need an actual socket server with which to connect to.

Imports

luckily for us, socket should be pre-installed (Cause it's actually needed for pip..)

So let's start:

first_tcp_server.py

```
import socket
```

[NOTE] : If you're used to vs-code / code-oss and have the relevant extensions, then you should see the suggestion pop-up..

The Class

It is perceived that if you have been following allong in this series, that I need not explain the need for OOP…

def init

INIT :: # ? [SEG-01]

In the init function, we will define and configure our sockets..

For this particular example we will be using `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`.

AF_INET -> This address family provides interprocess communication between processes that run on the same system or on different systems.

SOCK_STREAM -> In the Internet domain, the SOCK_STREAM socket type is implemented on the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol. A stream socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries.

[NOTE] : So in a NutShell; we're saying we want to use TCP/IP..

For a "Client" to 'call', us we will need to 'registered' our "Number",

```
self.ServSock   = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.Host       = '127.0.0.1'
self.Port       = 8000  # Google says to use port 80, but this is ok
self.ADDR       = (self.Host, self.Port)
```

def main(self)

MAIN :: # ? [SEG-02]

In order to use this bag of socket codes as a *Server*, we must bind the socket.

```
self.sock.bind(('', self.port))
```

In my experience, you're not going to be heard if no-one is listening..

```
max_connections = 50
self.sock.listen(max_connections)
```

Surely, every "Company Building" has a 'front-door' ? Our 'front-door' will be having an endless 'revolving door'.. Better know as a `while true` loop. Each visitor getting their very own 'thread' which will handle the inbound and outbound traffic between server and that 'visitor'. In order to achieve this **crochet**, we will need to:

```
import threading as thr
from thr import Thread as Thr
```

From here on, (within the `while True` Loop), we will be accepting out clients' requests to connect to the server. To put it mordernly; "the clerk/secretary has now shown our clients to the other room. . .

```
            try:
                conn, addr = self.sock.accept()
            except socket.error as e:
                print("[ERROR_CONNECTING_NEW_CLIENT] :", str(e))
            try:
                new_thread = Thr(group=None, target=self.handle_client, args=(conn, addr))
                new_thread.start()
                self.threads.append(t1)
                self.th_count+=1
            except ThreadError as e:
                print(f'[ERROR]:[SERVER:~:MAIN]: {str(e)}')
```

. . . this other room being:

def handle__clients(self, conn, addr)

HANDLE__CLIENT :: # ? [SEG-03]

To avoid the situation where you walk into a room, and it's like there's so many people talking over each other, so loudly, you couldn't possibly order a cup off coffee.. We will use threading events.

First, we initiate it..

```
self.E = thr.Event()
```

Then in the next `while True:`, we will first wait to recieve the length of a packet which is about to be sent..

```
data_len = int(conn.recv(64).decode())
if not data_len:
    self.E.wait()
if data_len > 0:
    new_data = str(conn.recv(data_len).decode())
```

By first sending the size, we can combat data corruptions and broken pipe errors. Which any online gamer will know, is a pain. But, TCP (unlike UDP) doesn't easily cause 'build-up' requests. But it does depend on what the goal is.. where online games have way too much data which become irrelevant very quickly, UDP is fine. However, if you're making online transactions„ (well, you don't want any issues the: we got enough)

Once the client have secured a connection and the Client has sent the message__len & message__ (packet), the server can now process this data and determine how to further handle the client.. So if eg. the client has send "Hello Server!", we can then choose a response for the server to send back to client, using the same technique/method that client used to send the data..

def reply(self, conn, data)

REPLY :: # ? [SEG-04]

Once it has been determined that the client's connection is following procedure, we can then reply.. So same as with the server side, the client will first be expecting the length of the data, then the actual data.

```
msg_len = len(data)
send_len = str(msg_len).encode()
send_len += b' ' * (64 - len(send_len))
conn.send(send_len)
conn.send(data.encode())
```

FirstSocketServer

Closing Word

Having this fundamental noted, we did not dive very deep, as this is to be an intermediate series.

Next up

We will build the client side program, so we can say a proper “Hello Server!”