Lab 2: Sockets with Python 3

CSC 335 – Fall 2022

Welcome to a tutorial on sockets with Python 3. We have a lot to cover, so let's just jump right in. The socket library is a part of the standard library, so you already have it.

```
import socket

# create the socket
# AF_INET == ipv4
# SOCK_STREAM == TCP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

The s variable is our TCP/IP socket. The AF_INET is in reference to th family or domain, it means ipv4, as opposed to ipv6 with AF_INET6. The SOCK_STREAM means it will be a TCP socket, which is our type of socket. TCP means it will be connection-oriented, as opposed to connectionless.

Okay, so what is a socket? The socket itself is just one of the endpoints in a communication between programs on some network.

A socket will be tied to some port on some host. In general, you will have either a client or a server type of entity or program.

In the case of the server, you will bind a socket to some port on the server (localhost). In the case of a client, you will connect a socket to that server, on the same port that the server-side code is using.

Let's make this code so far our server-side:

```
s.bind((socket.gethostname(), 1234))
```

For IP sockets, the address that we bind to is a tuple of the hostname and the port number.

Now that we've done that, let's listen for incoming connections. We can only handle one connection at a given time, so we want to allow for some sort of a queue, just incase we get a slight burst. If someone attempts to connect while the queue is full, they will be denied.

Let's make a queue of 5:

```
s.listen(5)
```

And now, we just listen!

```
while True:
    # now our endpoint knows about the OTHER endpoint.
    clientsocket, address = s.accept()
    print(f"Connection from {address} has been established.")
```

Full code for server.py:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 1234))
s.listen(5)

while True:
    # now our endpoint knows about the OTHER endpoint.
    clientsocket, address = s.accept()
    print(f"Connection from {address} has been established.")
```

Now we need to make our client's code!

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Now, since this is the client, rather than binding, we are going to connect.

```
s.connect((socket.gethostname(), 1234))
```

In the more traditional sense of client and server, you wouldnt actually have the client and server on the same machine. If you wanted to have two programs talking to eachother locally, you could do this, but typically your client will more likely connect to some external server, using its public IP address, not socket.gethostname(). You will pass the string of the IP instead.

Full client.py code up to this point:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))
```

Okay, now we just run both. First, let's run our server:

python3 server.py

python3 client.py

On our server, we should see:

Connection from ('192.168.86.34', 54276) has been established.

Our client, however, just exits after that, because it has completed its job.

So we've made a connection, and that's cool, but we really want to send messages and/or data back and forth. How do we do that?

Our sockets can send and recv data. These methods of handling data deal in buffers. Buffers happen in chunks of data of some fixed size. Let's see that in action:

Inside server.py, let's add:

```
    clientsocket.send(bytes("Hey there!!!","utf-8"))
```

Into our while loop, so our full code for server.py becomes:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 1234))
s.listen(5)

while True:
    # now our endpoint knows about the OTHER endpoint.
    clientsocket, address = s.accept()
    print(f"Connection from {address} has been established.")
    clientsocket.send(bytes("Hey there!!!","utf-8"))
```

So we've sent some data, now we want to receive it. So, in our client.py, we'll do:

```
msg = s.recv(1024)
```

This means our socket is going to attempt to receive data, in a buffer size of 1024 bytes at a time.

Then, let's just do something basic with the data we get, like print it out!

```
print(msg.decode("utf-8"))
```

Cool, our full client.py code is now:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))
```

```
msg = s.recv(1024)
print(msg.decode("utf-8"))
```

Now, run both server.py and then client.py. Our server.py shows:

Connection from ('192.168.86.34', 55300) has been established.

While our client.py now shows:

Hey there!!!

And it exits. Okay, so let's adjust that buffer slightly, changing the client.py recv to be in 8 bytes at a time.

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))

msg = s.recv(8)
print(msg.decode("utf-8"))
```

Now, re-run the client.py, and you will instead see something like:

Hey ther

Not lookin so hot! So you might realize that literally adds up to 8 characters, so each byte is a character. Why not... go back to 1024? or some massive number. Why work in buffers at all?

At some point, no matter what number you set, many applications that use sockets will eventually desire to send some amount of bytes far over the buffer size. Instead, we need to probably build our program from the ground up to actually accept the entirety of the messages in chunks of the buffer, even if there's usually only one chunk. We do this mainly for memory management. The calculations depending on application can vary, and you're free to play with the buffer size later. The only thing I can for sure promise is: you need to plan from the beginning to handle communications in chunks.

For our client, how might we do this? A while loop sounds like it could fit the bill. Data will come in as a stream, so really, handling for this is as simple as changing our client.py file to:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))
```

```
while True:
    msg = s.recv(8)
    print(msg.decode("utf-8"))
```

So, at the moment, we will receive this data and print it in chunks. If we run client.py now, we see:

Hey ther
e!!!

You should also take note that our client.py no longer exits. This connection now is remaining open. This is due to our while loop. We can use .close() on a socket to close it if we wish. We can do this either on the server, or on the client...or both. It's probably a good idea to be prepared for either connection to drop or be closed for whatever reason. For example, we could close the connection after we send our message on the server:

server.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 1234))
s.listen(5)

while True:
    # now our endpoint knows about the OTHER endpoint.
    clientsocket, address = s.accept()
    print(f"Connection from {address} has been established.")
    clientsocket.send(bytes("Hey there!!!","utf-8"))
    clientsocket.close()
```

If we run this, however, we will see our client.py then spams out a bunch of nothingness, because the data it's receiving, is, well, nothing. It's empty. 0 bytes, but we are still asking it to print out what it receives, even if that's nothing! We could fix that:

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))

full_msg = ''
while True:
    msg = s.recv(8)
    if len(msg) <= 0:
        break
```

```
    full_msg += msg.decode("utf-8")

print(full_msg)
```

So, now we are buffering through the full message. When we reach the end, which we're noting by getting 0 bytes, we break, and then return the message. This then concludes client.py. Now, client probably wants to also maintain the connection. How might we do that? Another while loop could do the trick.

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))


while True:
    full_msg = ''
    while True:
        msg = s.recv(8)
        if len(msg) <= 0:
            break
        full_msg += msg.decode("utf-8")

    print(full_msg)
```

Of course, we probably should yet again make sure the full_msg has something of substance before we print it out:

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))


while True:
    full_msg = ''
    while True:
        msg = s.recv(8)
        if len(msg) <= 0:
            break
        full_msg += msg.decode("utf-8")

    if len(full_msg) > 0:
```

```
    print(full_msg)
```

This works, but we have issues. What happens when we stop closing the client socket on the server's side? We never actually get a message! Why's this?

TCP is a communication *stream*...so how do we actually know when a message is actually happening? Generally, we need sort of way to notify the receiving socket about the message and how big it will be. There are many ways that we can do this. One popular way is to use a sort of header that always leads our message. We could also use some sort of footer, but this could cause trouble should someone learn about our methods.

Please turn on your code on Moodle for both the files

1. Server.py
2. Client.py