Lab 9: Sockets

CSE/IT 107

NMT Computer Science

"Today, most software exists not to solve a problem, but to interface with other software."

— Ian O. Angell

"Computer Science is no more about computers than astronomy is about telescopes."

— Edsger W. Dijkstra

"If people never did silly things, nothing intelligent would ever get done."

— Ludwig Wittgenstein

---

1 Introduction

This lab will be two small fun projects involving what computer scientists call sockets. Really, sockets are network connections: For a computer to be able to receive a connection, it would open a socket and wait for incoming connections on that socket (this program would be a server). Another computer (or, perhaps, the same) may open another socket to connect to the "remote" socket (this program would be the client). Once that has happened, the two machines are connected.

Today, you will implement two small programs that connect to a server and solve a problem. First, you will play a game of Rock Paper Scissor. When you connect to the server, you will be matched with another program that connected – this may take a little while, because you have to wait for someone else to connect. Once you connect, your program will play Rock Paper Scissors with the other program.

In order to communicate, we give you a small "protocol." Generally, a protocol is just a convention or standard that determines how two programs communicate with each other. Our protocol will have things like this: If you receive "wait" from the server, you have to wait for another player to connect and just try to keep receiving messages until another one arrives.

2 Sockets

Sockets are really just simple network connections. To use sockets in Python, you have to `import socket`. We will use the following vocabulary here: the program accepting network connections from others is called a *server*, while the program trying to conncet to that is called a *client*.

In this section, we will only talk about programming a client. The server program will already be running on a remote computer.

To initiate a network connection, use `socket.socket()`:

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

Do not mind the arguments to this function. For now, you have to use them and assume they are black necessary magic.

Just like with files and `open`, the variable `s` is now your reference to this network connection. You have to use it every time you are trying to receive or send something on that network connection.

To then actually connect to a server, use `.connect()` and give it a tuple of hostname or IP *and* port.

```
1 s.connect( ('arctem.com', 50000) )
```

Now this is weird. What are those things?

Generally, we can say that every computer on the internet has its own IP. An IP consists of four numbers joined by periods which can go between 0 and 255. For example, Chris's laptop has the IP `129.138.6.38` when he is connected to the WiFi in the computer science department. A lab computer in the last row of Speare 23 has the IP `129.138.47.83`. The computer ("server") that hosts Russell's website and some of this lab has the IP `104.131.56.87`.

A hostname, such as `arctem.com` (you have seen this in your web browser!), is just something that is more legible and that translates to an IP. For example, `arctem.com` translates to `104.131.56.87`. `paul.kochris.com` translates to `178.62.37.44`.

Now given this information, you can connect to any computer that is actually online. But there could be many many programs running on that computer that accept network connections. If now you connected to that computer, it has to figure out which program you wanted to connect to. But just the IP does not give you that information – the remote computer has no idea.

This is where the port comes in: A server on a remote computer will say "I listen to *port X*," where *X* is any number between 0 and 65535. Then, when a client tries to connect to a computer, that machine will look at the port and "route" your connection to the correct program.

For example, when you open a web browser and go to a web page, you are connecting to the hostname you typed in (for example, `google.com`) at port 80, because that is the standard for HTTP connections. At port 80 there is a program that handles the messages for requesting a web page. If you are using secure browsing, you are connecting to 443, because that is the standard for HTTPS connections.

```
1 data = s.recv(1024)
```

This is how we read data from a socket. `s.recv()` is a function that returns a byte array of the data received at socket `s`, with the argument being the maximum number of bytes received at once. Now, we don't want a byte array. Byte arrays are weird and scary things (not really, but just pretend for now – it's about being in the Halloween spirit!). Instead, we want a string version of `data`. We do this using:

```
data = data.decode()
```

This converts `data` into a string. It is important, however, that we don't immediately do this after reading from the socket. First we want to make sure we actually got something from the socket – if the other end of the socket closed and we try to use the data from it, our program will crash!

```
if data:
  data = data.decode()
else:
  print('Remote socket closed.')
```

If we want to send our own message, we need to do the reverse of decoding it. Instead, we need to encode our string into a byte array!

```
msg = "Stuff"
msg = msg.encode()
s.sendall(msg)
```

This snippet of code takes our string `msg`, encodes it as a byte array, and then uses `s.sendall()` to send it to the socket we're connected to! Simple!

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect( ('arctem.com', 50000) )

data = s.recv(1024)
if data:
  data = data.decode()
else:
  print('Remote socket closed.')
msg = "Stuff"
msg = msg.encode()
s.sendall(msg)
```

Listing 1: Code that shows usage, but does not actually make sense

Function	Arguments	Purpose
`socket.socket`	`socket.AF_INET, socket.SOCK_STREAM`	Creates a network connection. Don't alter the arguments. Returns a socket reference.
`s.socket()`	`(host, port)`	Connect to `host` at `port`. `s` has to be created by a call to `socket.socket` (see Listing 1)
`msg.encode()`		Convert the string `msg` into a byte array that is compatible with sockets.
`data.encode()`		Convert the byte array `data` into a string.

Table 1: Summary of socket functions & methods

3 Exercises

Boilerplate

Remember that this lab *must* use the boilerplate syntax introduced in Lab 5.

Linux

Please use Linux in this lab. Below, we will tell you to type in commands to the command line – remember that in Lab 1, you had to type `idle3` in the command line to start Python? Use that same command line to type the commands given.

rps.py Write a program that connects to the server running at `arctem.com` port 50001 and plays a game of rock, paper, scissors. The server will send the following messages:

name Next message sent will be your client's display name.

taken The name sent is already in use. Repeat sending a name.

wait Game has not yet been found (waiting for another player). No response required.

opponent <name> A game has been found. The opponent's name will be inserted for "<name>". No response required.

play The next message sent should consist solely of "r", "p", or "s", depending on whether you wish to play rock, paper, or scissors.

tie Your opponent played the same as you, causing a tie. No response required.

win Your play beat your opponent's, so you won. The next message should consist solely of "y" or "n", indicating your desire to play again.

lose Your opponent's play beat yours, so you lost. The next message should consist solely of "y" or "n", indicating your desire to play again.

disconnect Your opponent disconnected at an unexpected time. The next message should consist solely of "y" or "n", indicating your desire to find a new opponent.

again The next message should consist solely of "y" or "n", indicating your desire to play again. This message will only occur if invalid input is given for "win", "lose", or "disconnect", so if you can be sure that will not occur you do not necessarily need this case.

Every message sent will be terminated with a "\n". Every message you send should be terminated with a "\n". It is possible when you read from a socket that you receive multiple commands at once, or less than a full command. Check for "\n" to know where each command ends. If you receive part of a command, you will have to store that partial command until you get the rest of it.

The valid messages you can send:

<username> Your desired username, in response to `name` or `taken`.

r A play of "rock", in response to `play`.

p A play of "paper", in response to `play`.

s A play of "scissors", in response to `play`.

y Indication of desire to play again, in response to `win`, `lose`, `disconnect`, or `again`.

n Indication of desire to not play again, in response to `win`, `lose`, `disconnect`, or `again`.

For each of these server responses, you need to display an appropriate message with a prompt for input if appropriate. For example, a "win" message might output "Congratulations, you beat <otherplayer>! Do you want to play again?"

maze.py Write a program that connects to the server running at `arctem.com` port 50002 and automatically navigates a maze generated by the server. The first message you send should be your display name (used for the leaderboard). The server will then randomly generate a 10x10 maze that you must navigate. You will always begin in the top left corner of the maze, facing down. You must navigate to the bottom right corner.

The server will send the following messages:

<left> <forward> <right> Three space-separated numbers. The first is the number of open spaces to your left, the second is the number of open spaces in front of you, and the last is the number of open spaces to your right.

invalid Sent if the last message you sent was not a valid message or the movement you attempted was not possible (such as attempting to move forward into a wall).

win <steps> <time> The word "win" followed by an integer, then a floating point, all space-separated. This indicates you have successfully reached the bottom right corner of the maze. After this message is sent, the server will close its end of the socket.

The valid messages you can send (after the initial message indicating your name):

forward Attempt to move one space in the direction you are currently facing.

backward Attempt to move one space opposite the direction you are currently facing.

left Turn left 90°.

right Turn right 90°.

Your program's only input should be the desired display name (though you are free to hard-code that if you wish). Otherwise, your program should navigate the maze on its own. You are free to have any output that you wish to help you get a sense of how your program is doing in navigating the maze, though you MUST print out the stats it receives on successful completion of the maze (total steps and time taken).

If you connect to the server on port 50003, it will send you the leaderboard of fastest times measured in steps. A simple way to do this in the cmd line is to run `nc arctem.com 50003`.

If you connect to the server on port 50004, it will send you the leaderboard of fastest times measured in seconds. A simple way to do this in the cmd line is to run `nc arctem.com 50004`.

4 Submitting

Files to submit:

- rps.py (Section 3)
- maze.py (Section 3)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab9.zip` or `cse107_firstname_lastname_lab9.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.