ICS 32 Winter 2014
Code Example: Sockets, Part 2

Background

Sending and receiving text via sockets

Many programs that communicate over a network do so by sending text back and forth; quite often, textual communication protocols are based around the concept of *lines* of text terminated by a special sequence of characters called an *end-of-line sequence*. We've previously seen how to read lines of text from files; unfortunately, the mechanisms we've seen for reading data from a socket are a lot harder to work with, allowing us to retrieve a given number of *bytes* from a socket's input stream, which leaves us with two difficulties:

- Decoding the bytes we receive, turning them from bytes into strings.
- Finding where one line of ends and another begins. And since we have to ask for a certain number of bytes at a time, but are never necessarily sure ahead of time how many bytes make up a line (since different lines may contain different numbers of bytes), we'll have to go to a lot of trouble: if we read too few characters, we'll need to read more; if we read too many, we'll need to hang on to the extra ones so we can use them as part of the next line we read.

It seems a shame that we can't just treat a socket the way we do a text file. Files contain bytes, too, but when we read from text files in Python, the file object converts between bytes and strings and finds where one line ends and another begins automatically; we just have to call **readline()** and the right things happen.

As is often the case in a programming language library, when you dig a little bit further, you find the tool you were looking for. If you want to treat a socket similarly to how you treat a text file — reading lines and getting back strings, writing strings — you can do it. You just have to ask for a "middleman" of sorts, an object that behaves outwardly the way a file object does, but that reads and writes via a socket rather than a file.

Sockets in Python provide a **makefile()** method that can give you such a "middleman." The arguments you pass to this method are the same as the arguments you pass to the built-in **open()** function that opens files, and the object returned to you appears a lot like a file object — it supports methods like **readline()** and **write()**.

This code example makes use of the **makefile()** method to simplify how we read from and write to our sockets.

The code

In lecture, we rewrote the echo client program from the previous code example using the socket's **makefile()** method to let us read and write lines of text instead of bytes. That example follows.

- The **echo_client** module

Trying out the example client

An echo server like the one we connected to during lecture is now running on the same machine that the Connect Four server for Project #2 is running. (See a previously-sent email for an indication of where that is.) The echo server is listening on port 5151.

For fun, try connecting to port 5150 or 5152 instead, where there are servers that do something different — they still respond with a line of text every time you send them one, but the text they send is not just a copy of what you send them. See if you can figure out what these servers do.