ICS 32 Fall 2013
Code Example: Sockets, Part 1

Background

In previous coursework, and earlier this quarter, you've seen one way that programs can read external input: by opening and reading from files. But compared to the programs we use every day — web browsers, email clients, mobile applications, multiplayer games, and the like — that interact not just with our file system but with other computers and other people, if you're limited to the use of files, you can feel as though you've been put into a very unrealistic box. Fortunately, the Python Standard Library includes a number of tools that we can use to help us write Python programs that can do many of the same things that our favorite Internet-connected programs do.

Sockets

As a first step, we'll start with the tool atop which most of the others are built: *sockets*. In Python, sockets are objects that encapsulate and hide many of the underlying details of how a program can connect directly to another program; usually, this connection is made via a network such as the Internet, though it should be pointed out that you can also use sockets to send data back and forth between programs running on the same machine, and that's where our story begins.

Sockets provide an abstraction of a connection between a program and some other program. Sockets can be used to represent many different kinds of network connections that behave quite differently from one another, but we'll be using them in a particular way. There are variations on what is described here, but these assumptions will serve us well.

When two programs are connected via sockets, each program has a socket representing its end of the connection between them, with each socket having two *streams* available:

- An *input stream*, which receives all of the data sent by the other program, in the order the other program sent it.
- An *output stream*, which takes any data written to it and sends it to the other program, in the order it was written.

Sockets (the way we'll be using them) guarantee that if the data makes it across the network successfully — note that it won't always make it, for a variety of reasons! — it wil be placed into the receiver's input stream in the order it was placed into the sender's output stream. So, for example, if the machine on one side sends three messages — M1, M2, and M3 — the machine on the other side will receive the messages in that order. If M1 fails to make it across the network, neither M2 nor M3 will ever be seen, either. The code underlying Python's sockets does a variety of things like attempting to re-send lost information periodically and holding information received out of order until the information preceding it is received, so we don't have to worry about these kinds of details; we can just think of the two streams and leave the details to the implementation.

One reason why seeing a network connection as two streams is handy for us is that it feels quite a lot like what we're used to doing with files. So far, we've read from files sequentially and written to them sequentially; we'll be able to do the same with sockets. The main difference is that networks are less reliable than files, because so many more things can go wrong in a connection between your computer here in Irvine and one in a faraway place like Korea, so we'll have to be more cognizant of the ways that things can fail; as with most failures in Python programs, these failures will usually arise as exceptions.

There are a number of issues that you have to be aware of when you want to write a robust program that communicates using sockets; this code example ignores most of those issues in the interest of simplicitly. Future examples will begin to explore those details.

Clients and servers

In the context of a socket-based conversation between two programs, we can think of each of the two programs as playing a role. One program was waiting for another program to connect to it and responded to the request; the other initiated the connection. (You can think of this relationship as being the same as the relationship between two people in the midst of a phone call. Someone initiated the call, while someone else answered it.) For the purposes of such a scenario, we say the program that initiated the conversation is a *client*, while the program that responded to that initiation is a *server*.

Sometimes, as in the example below, a program plays one role and never plays the other; in other instances, programs play different roles at different times. In this course, we'll focus our efforts on writing clients, as we're predominantly interested in consuming information and services on the Internet.

Some technical information about the Internet

Writing programs that can communicate via the Internet requires some knowledge of how the Internet works. The Internet is a complex, many-layered combination of hardware and software, but you actually need to know surprisingly little about the underlying technology in order to write programs that use it. Still, there are issues that you will need to be aware of, especially if you want to do some or all of your work on your own computer.

Loads of useful information is available online about all of the topics summarized below; few of the problems I discuss are insurmountable. (Of course, the challenge, as always, is to separate the useful information from the noise.) But all of these issues will have at least some effect on whether you can get your programs to connect to programs running on other machines, even if your programs are completely correct, so it's best to be aware of these issues before you get started.

IP addresses

In general, every machine connected to the Internet has an *IP address*. An IP address is akin to a telephone number; by specifying that a message is to be sent to a particular IP address, the network will be able to determine who should receive the message and how the message should get there, hiding these details from the machines on either end.

An IP address is generally displayed as a sequence of four numbers separated by dots; each of the numbers has a value in the range 0-255 (a range chosen because values in this range can be stored in eight bits, or one byte). For example, as of this writing, the IP address of one of the machines that act as a "server" for the ICS web site has the address 128.195.1.76.

If you want a program of yours to connect to another program running on another machine, you'll have to know the IP address where that other program is running. Note, too, that many machines have a different IP address as often as every time they connect to the Internet, so this isn't the kind of information you can necessarily bookmark and reuse forever.

The "loopback" address

There is a special range of IP addresses that can always be used to connect a computer to itself, regardless of what its IP address is. These are called "loopback" addresses, the most common of which is **127.0.0.1**. So if you want to test connecting two programs on your own machine, you can use **127.0.0.1** to do that. (This also explains T-shirts and bumper stickers that you may have seen that say "There's no place like 127.0.0.1.")

Ports

When you want to connect a program to another program running on another machine, it's not enough to know the IP address of the other machine. Multiple programs on the same machine are likely to be connected to the Internet at any given time. So there needs to be a way to identify not only what machine you'd like to connect to, but also which program on that machine you'd like to connect to.

The mechanism used for this on the Internet is called a *port*. A program acting as a server will register its interest in a particular port by *binding* to it; the operating system will generally only allow one program to be bound to a particular port at a time. Once a program binds to a port on a machine, when a connection is made to that port on that machine, the connection is routed to the program that bound to the port.

Ports have numbers that range from 0-65535 (a range chosen because values in this range can be stored in sixteen bits, or two bytes). It's generally a good idea not to use ports with numbers below 1024, becuase these tend to be reserved for common uses (e.g., web traffic, FTP traffic, email traffic). Beyond that, you may discover some ports at or above 1024 in use — depending on what programs are running on your machine — but most should be available.

The important thing to realize here is this: In order for a client to initiate a connection to a server, the client will need to use not only the IP address of the machine where the server is running, but also the port that the server is listening on.

The Domain Name System and DNS lookups

Though every machine connected to the Internet has an IP address, users don't typically use IP addresses on an everyday basis. Just as IP addresses are akin to telephone numbers, there is an Internet service called the Domain Name System (DNS) that acts as a kind of phone book; given the name of a computer, DNS can tell you its IP address, so long as that name has been registered with DNS previously.

So, for example, when you brought up this web page in your browser, your browser first had to know the address of **www.ics.uci.edu**; it found this out by doing a *DNS lookup*, by sending a message to a Domain Name Server and asking "What is the IP address of www.ics.uci.edu?" In return, the browser received a message that said "It's 128.195.1.76," at which time your browser could connect to that address (on port 80, since that's the port used for web traffic when not otherwise specified) and download this web page.

DNS is unlikely to affect your connections to programs on other machines significantly in this course, other than the fact that you may want to use a hostname instead of an IP address when you know one. But we'll certainly use it later this quarter.

Note that the "loopback" address has its own name: **localhost**. The name **localhost** always resolves to a "loopback" address.

Firewalls

The Internet offers a certain amount of anonymity — it's hard to know who's contacting you or what their motives are if all you can see is their IP address and what port they're connecting to. This kind of anonymity has its benefits, though it also has its serious downsides; when you can't know who's contacting you and can't know what they're trying to accomplish, and when you can't always trust your operating system and other software not to provide outsiders access to information they shouldn't have, the wise solution is to restrict incoming traffic. The theory is that if no one can connect to you, no one can take advantage of you (without you having "asked for it," in some sense, by connecting to them). This is the theory behind *firewalls*, which are software or hardware that restrict other computers' access to computers behind them.

It was once the case that firewalls were mostly used in businesses, as they were the primary targets of online crime and mischief. Nowadays, though, most computers come with some kind of firewall software built into them. This may make it more difficult for programs on other mahines to connect to yours when you want them to, because your computer may be configured to disallow incoming connections. Some firewall software also allows you to disallow certain kinds of outgoing connections, which might also affect your ability to connect to programs running on other machines. There are usually ways to "open a port," which means that you've told our firewall to allow traffic bound for a certain port to move into or out of your machine, while traffic on other ports will still be forbidden. Details of how to do this vary from one context to another, but there is a fair amount of documentation online if you want to learn how to open ports using your particular combination of hardware and/or software firewalls.

I should point out, also, that some Internet service provides have their own firewalls and traffic limitations in plcae, so if you're working from home, your experience — especially in terms of being able to have others connect to you — may vary considerably depending on your provider. This will be of little consequence to your work in this course, as this problem affects servers much moreso than clients, but it's something to be aware of if you want to take your work further than what is assigned.

Routers and network address translation (NAT)

In general, every machine connected to the Internet has an IP address. However, many of us are not connected directly to the Internet at all. For example, I have several computers in my home, but I have only one Internet connection: a cable modem. In order to use more than one of my computers at a time online, it's necessary for me to have some way of sharing that connection.

In order to do that, I do what most people do in this situation: I use a device called a *router*. The router is connected to my one Internet connection. Whenever it's connected, it has an IP address. (Most home Internet users, me included, don't get the same IP address every time they connect, though, which is one reason why it's often hard to run a server from your home.) My computers don't connect directly to the Internet; instead, they connect to the router. The router's job is to forward outgoing traffic from each computer to the single Internet connection, and to take the incoming traffic and route it to the appropriate computer.

The router and my computers form their own *local-area network*, or *LAN*. The router assigns a "fake" IP address to each of my computers, using a range of addresses that is never assigned to computers on the Internet. As traffic flows into and out of the router, it performs a task called *network address translation*, or *NAT*, which means that it converts the internal, "fake" IP addresses used by my computers to its own IP address for traffic going out, and converts its own IP address back to the "fake" IP addresses on the way back in. As far as the outside world is concerned, I don't have many computers; I just have one: the router.

Many routers also act as firewalls, disallowing incoming traffic in most cases unless you specifically configure them to allow it. Most home computer users only ever initiate connections, so this is a safe and relatively painless restriction for most people.

Why this will affect you when you work from home is that your router will make it much more difficult for programs on other machines to connect to yours. They'll need to have your router's IP address, not your computer's IP address. (That's not hard to give them, since Googling *What is my IP address?* will show you the router's IP address, since that's the only address the outside world ever sees.) You'll also need to configure your router to allow incoming traffic on at least one port, and to send incoming traffic on those ports to a particular one of your computers. Details of how you set this up vary considerably from one router to another, but are generally available online if you know what model of router you have.

Wait... I'm getting overwhelmed!

If most of these details are new to you, you might be feeling overwhelmed. I present these details as useful background information, though they're not the focus of our work here. (They are details that are worth learning more about if you want to be in the technology field, but it's fine to acquire them gradually.) Other than the use of IP addresses and ports, none of these details is likely to afect your work all that much when you're working on the machines in the ICS labs, but you'll need to be aware of some of these issues if you want to use your own machine.

A simple Python client program

In lecture, we wrote a simple Python client program, whose role was to connect to a server program that I called an *echo server*, which I had written before lecture and made available on the ICS network. The echo server has a simple job: when a client connects to it, it reads a line of input from the client and then sends a copy of it back to the client, then reads another and sends a copy of it back, and so on, until the client disconnects.

The socket module

The Python Standard Library includes a module called **socket**, which is a very good entry point into the world of Internet-connected Python programs. We'll explore that module some here, and will continue to see new functionality in that module as time goes on.

The first thing to know about the **socket** module is that it includes a type called **socket**, which is the kind of object we'll need to interact with. Recall, from the *Background* section above, that a socket object represents the endpoint of a connection between a Python program and another program; the other program need not be running on the same computer (though it can be) and need not be written in Python (though, again, it can be).

Before we can proceed, we need to know how to get one of these socket objects. This is simple: we call its *constructor*, a special function that constructs a socket object for us. To do that, we'll first need to import the **socket** module, so we'd include this line in our module:

```
import socket
```

Then we can say this to create our socket object and assign it into a local variable *s*:

```
s = socket.socket()
```

Sockets can be used, fundamentally, for two things: *listening* and *connecting*. Listening means that we want to wait for another program to contact us; connecting means that we want to connect to another program. In this example, since we're writing a client, we'll connect.

Connecting requires that we ask the socket to connect to a particular port on a particular *host* — we identify the host either by an IP address, such as **128.195.1.83**, or by a name, such as **www.ics.uci.edu**. The **socket** module represents host/port commbinations as two-element Python tuples, with the first element of the tuple being a string identifying the host and the second element of the tuple being an integer specifying the port number.

Once we have our address tuple, we can *connect* the socket to it:

```
connect_address = ('128.195.1.83', 5151)
s.connect(connect_address)
```

If the call to the **connect** method succeeds, our socket is connected successfully; if the attempt to connect fails, **connect** will raise an exception instead.

Once connected successfully, we can use our socket to send and receive data between our program and the other program that we're connected to. The one tricky part is that the data that is sent and received is made up of what are called *bytes*, represented by a Python object with the type **bytes**. If what we intend to send back and forth is actually text, we'll need to perform conversion between the **bytes** and **str** types.

Sending bytes to the other program is done by calling the **send()** method on the socket, passing it the bytes we want to send. Our program will block (i.e., make no further progress) until the bytes are sent.

Receiving bytes from the other program is done by calling the **recv()** method on the socket, passing it the number of bytes (at maximum) that we want. Our program will block until we've received some bytes, though we won't necessarily get as many as we asked for; we'll instead get back one of the following:

- An empty **bytes** object if the other program has specified that it will not send any more bytes (e.g., it has closed the connection).
- A **bytes** object containing somewhere between one byte and the number of bytes we asked for. In general, we're never guaranteed we'll get as many as we asked for, but we'll always get back something unless the other program has disconnected.

The Python documentation suggests a somewhat small power of 2 for the number of bytes, so we'll go with 4,096 bytes.

```
incoming_bytes = s.recv(4096)
```

We could use the **len()** function to check how many bytes we got back and act accordingly.

Finally, when we're done with the socket, we'll want to close it, to let the other program know that we've finished our conversation with it.

```
s.close()
```

The code

The final code example from lecture is below.

- 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.