Python Network Programming

David M. Beazley
http://www.dabeaz.com

Edition: Thu Jun 17 19:49:58 2010

Python Network Programming : Table of Contents

Edition: Thu Jun 17 19:49:58 2010

Section 0

Introduction

Support Files

- Course exercises:

http://www.dabeaz.com/python/pythonnetwork.zip

- This zip file should be downloaded and extracted someplace on your machine
- All of your work will take place in the the "PythonNetwork" folder

Python Networking

- Network programming is a major use of Python
- Python standard library has wide support for network protocols, data encoding/decoding, and other things you need to make it work
- Writing network programs in Python tends to be substantially easier than in C/C++

This Course

- This course focuses on the essential details of network programming that all Python programmers should probably know
    - Low-level programming with sockets
    - High-level client modules
    - How to deal with common data encodings
    - Simple web programming (HTTP)
    - Simple distributed computing

Standard Library

- We will only cover modules supported by the Python standard library
- These come with Python by default
- Keep in mind, much more functionality can be found in third-party modules
- Will give links to notable third-party libraries as appropriate

Prerequisites

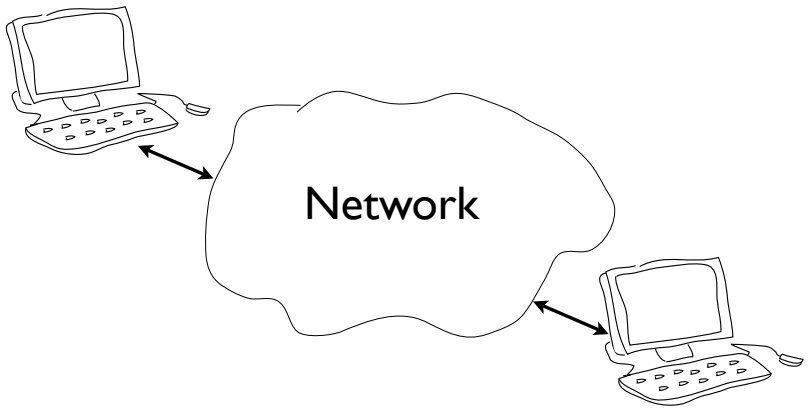
- You should already know Python basics
- However, you don't need to be an expert on all of its advanced features (in fact, none of the code to be written is highly sophisticated)
- You should have some prior knowledge of systems programming and network concepts

Section 1

Network Fundamentals

The Problem

- Communication between computers



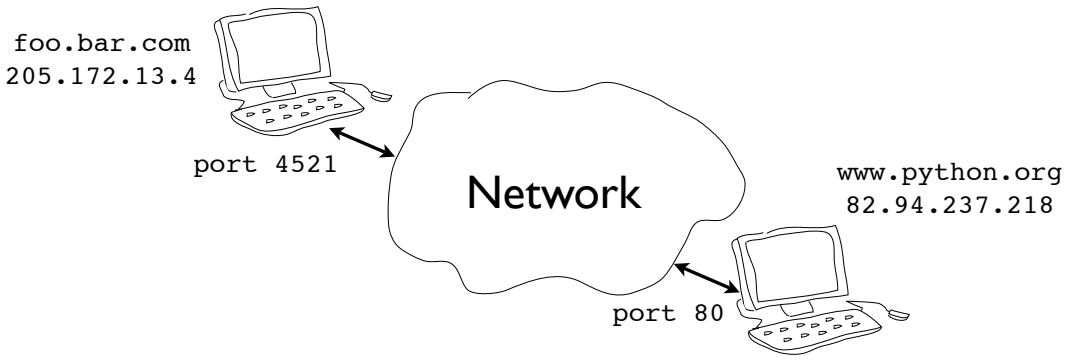
- It's just sending/receiving bits

Two Main Issues

- Addressing
  - Specifying a remote computer and service
- Data transport
  - Moving bits back and forth

Network Addressing

- Machines have a hostname and IP address
- Programs/services have port numbers



Standard Ports

- Ports for common services are preassigned

```
21          FTP
22          SSH
23          Telnet
25          SMTP (Mail)
80          HTTP (Web)
110         POP3 (Mail)
119         NNTP (News)
443         HTTPS (web)
```

- Other port numbers may just be randomly assigned to programs by the operating system

Using netstat

- Use 'netstat' to view active network connections

```
shell % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 *:imaps                 *:*
tcp        0      0 *:pop3s                 *:*
tcp        0      0 localhost:mysql         *:*
tcp        0      0 *:pop3                  *:*
tcp        0      0 *:imap2                 *:*
tcp        0      0 *:8880                  *:*
tcp        0      0 *:www                   *:*
tcp        0      0 192.168.119.139:domain  *:*
tcp        0      0 localhost:domain        *:*
tcp        0      0 *:ssh                   *:*
...
```

- Note: Must execute from the command shell on both Unix and Windows

Connections

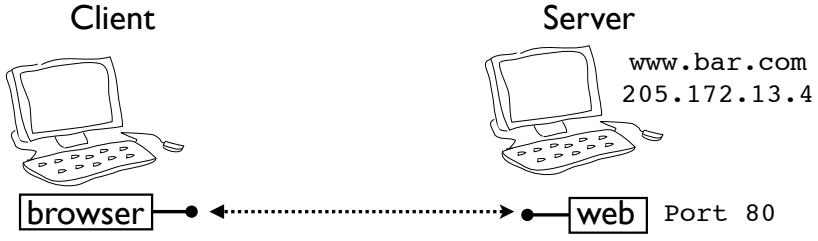
- Each endpoint of a network connection is always represented by a host and port #
- In Python you write it out as a tuple (host,port)

```
("www.python.org",80)
("205.172.13.4",443)
```

- In almost all of the network programs you'll write, you use this convention to specify a network address

Client/Server Concept

- Each endpoint is a running program
- Servers wait for incoming connections and provide a service (e.g., web, mail, etc.)
- Clients make connections to servers



Request/Response Cycle

- Most network programs use a request/ response model based on messages
- Client sends a request message (e.g., HTTP)

```
GET /index.html HTTP/1.0
```

- Server sends back a response message

```
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 48823

<HTML>
...
```

- The exact format depends on the application

Using Telnet

- As a debugging aid, telnet can be used to directly communicate with many services

```
telnet hostname portnum
```

- Example:

type this and press return a few times →

```
shell % telnet www.python.org 80
Trying 82.94.237.218...
Connected to www.python.org.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 31 Mar 2008 13:34:03 GMT
Server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2
mod_ssl/2.2.3 OpenSSL/0.9.8c
...
```

Try info.cern.ch instead

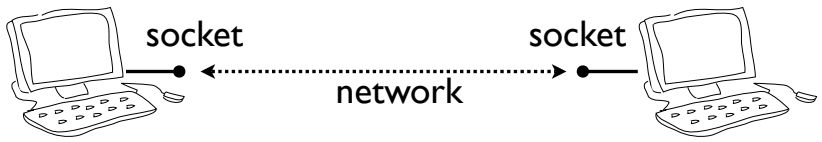
GET / HTTP/1.1
Host: info.cern.ch\n\n

Data Transport

- There are two basic types of communication
- Streams (TCP): Computers establish a connection with each other and read/write data in a continuous stream of bytes---like a file. This is the most common.
- Datagrams (UDP): Computers send discrete packets (or messages) to each other. Each packet contains a collection of bytes, but each packet is separate and self-contained.

Sockets

- Programming abstraction for network code
- Socket: A communication endpoint



- Supported by socket library module
- Allows connections to be made and data to be transmitted in either direction

Socket Basics

- To create a socket

```
import socket
s = socket.socket(addr_family, type)
```

- Address families

```
socket.AF_INET      Internet protocol (IPv4)
socket.AF_INET6     Internet protocol (IPv6)
```

- Socket types

```
socket.SOCK_STREAM  Connection based stream (TCP)
socket.SOCK_DGRAM   Datagrams (UDP)
```

- Example:

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
```


Socket Types

- Almost all code will use one of following

```
from socket import *

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

- Most common case: TCP connection

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

Using a Socket

- Creating a socket is only the first step

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

- Further use depends on application
- Server
  - Listen for incoming connections
- Client
  - Make an outgoing connection

TCP Client

- How to make an outgoing connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("www.python.org",80))          # Connect
s.send("GET /index.html HTTP/1.0\n\n")    # Send request
data = s.recv(10000)                      # Get response
s.close()
```

use info.cern.ch instead

- s.connect(addr) makes a connection

```
s.connect(("www.python.org",80))
```

addr must be a tuple (host,port)

- Once connected, use send(),recv() to transmit and receive data
- close() shuts down the connection

Exercise 1.1

Time : 10 Minutes

Try the above example using Python, but don't try it interactively using Python shell. Create a file "client.py", then run it with "python client.py".

Otherwise, you may get a socket error 54!

Server Implementation

- Network servers are a bit more tricky
- Must listen for incoming connections on a well-known port number
- Typically run forever in a server-loop
- May have to service multiple clients

TCP Server

- A simple server

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

- Send a message back to a client

```
% telnet localhost 9000
Connected to localhost.
Escape character is '^]'.
Hello 127.0.0.1
Connection closed by foreign host.
%
```

Server message

TCP Server

- Address binding

```
from socket import *
s = socket(AF_INET,SOCK_STR
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

binds the socket to a specific address

- Addressing

binds to localhost

```
s.bind(("",9000))
s.bind(("localhost",9000))
s.bind(("192.168.2.1",9000))
s.bind(("104.21.4.2",9000))
```

If system has multiple IP addresses, can bind to a specific address

TCP Server

- Start listening for connections

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Tells operating system to start listening for connections on the socket

- s.listen(*backlog*)
- backlog is # of pending connections to allow
- Note: not related to max number of clients

TCP Server

- Accepting a new connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Accept a new client connection

- s.accept() blocks until connection received
- Server sleeps if nothing is happening

TCP Server

- Client socket and address

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Accept returns a pair (client_socket,addr)

`<socket._socketobject object at 0x3be30>`

This is a new socket that's used for data

`("104.23.11.4",27743)`

This is the network/port address of the client that connected

TCP Server

- Sending data

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Send data to client

Note: Use the client socket for transmitting data. The server socket is only used for accepting new connections.

TCP Server

- Closing the connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Close client connection

- Note: Server can keep client connection alive as long as it wants
- Can repeatedly receive/send data

TCP Server

- Waiting for the next connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print "Received connection from", a
    c.send("Hello %s\n" % a[0])
    c.close()
```

Wait for next connection

- Original server socket is reused to listen for more connections
- Server runs forever in a loop like this

Exercise 1.2

Time : 20 Minutes

Advanced Sockets

- Socket programming is often a mess
- Huge number of options
- Many corner cases
- Many failure modes/reliability issues
- Will briefly cover a few critical issues

Partial Reads/Writes

- Be aware that reading/writing to a socket may involve partial data transfer
- send() returns actual bytes sent
- recv() length is only a maximum limit

```
>>> len(data)
1000000
>>> s.send(data)
37722
>>>

>>> data = s.recv(10000)
>>> len(data)
6420
>>>
```

Sent partial data

Received less than max

Partial Reads/Writes

- Be aware that for TCP, the data stream is continuous---no concept of records, etc.

```
# Client
...
s.send(data)
s.send(moredata)
...


# Server
...
data = s.recv(maxsize)
...
```

#send's not equal to #recv's

but, len(#bytes sent) = len(#bytes recv'd)

This recv() may return data from both of the sends combined or less data than even the first send

- A lot depends on OS buffers, network bandwidth, congestion, etc.

Sending All Data

- To wait until all data is sent, use sendall()

```
s.sendall(data)
```

- Blocks until all data is transmitted
- For most normal applications, this is what you should use
- Exception : You don't use this if networking is mixed in with other kinds of processing (e.g., screen updates, multitasking, etc.)

End of Data

- How to tell if there is no more data?
- recv() will return empty string

```
>>> s.recv(1000)
''
>>>
```

- This means that the other end of the connection has been closed (no more sends)

Data Reassembly

- Receivers often need to reassemble messages from a series of small chunks
- Here is a programming template for that

```
fragments = []                # List of chunks
while not done:
     chunk = s.recv(maxsize)  # Get a chunk
     if not chunk:
         break                # EOF. No more data
     fragments.append(chunk)

# Reassemble the message
message = "".join(fragments)
```

- Don't use string concat (+=). It's slow.

Timeouts

- Most socket operations block indefinitely
- Can set an optional timeout

```
s = socket(AF_INET, SOCK_STREAM)
...
s.settimeout(5.0)    # Timeout of 5 seconds
...
```

- Will get a timeout exception

```
>>> s.recv(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
socket.timeout: timed out
>>>
```

- Disabling timeouts

```
s.settimeout(None)
```

Exercise 1.3

Time : 15 Minutes

Odds and Ends

- Other supported socket types
  - Datagram (UDP) sockets
  - Unix domain sockets
  - Raw sockets/Packets
- Sockets and concurrency
- Useful utility functions

UDP : Datagrams

- Data sent in discrete packets (Datagrams)
- No concept of a "connection"
- No reliability, no ordering of data
- Datagrams may be lost, arrive in any order
- Higher performance (used in games, etc.)

UDP Server

- A simple datagram server

```
from socket import *
s = socket(AF_INET,SOCK_DGRAM)

s.bind(("",10000))

while True:
    data, addr = s.recvfrom(maxsize)

    resp = "Get off my lawn!"
    s.sendto(resp,addr)
```

Create datagram socket

Bind to a specific port

Wait for a message

Send response (optional)

- No "connection" is established
- It just sends and receives packets

UDP Client

- Sending a datagram to a server

```
from socket import *
s = socket(AF_INET,SOCK_DGRAM)

msg = "Hello World"
s.sendto(msg,("server.com",10000))

data, addr = s.recvfrom(maxsize)
```

Create datagram socket

Send a message

Wait for a response (optional)

returned data

remote address

- Key concept: No "connection"
- You just send a data packet

Unix Domain Sockets

- Available on Unix based systems. Sometimes used for fast IPC or pipes between processes
- Creation:

```
s = socket(AF_UNIX, SOCK_STREAM)
s = socket(AF_UNIX, SOCK_DGRAM)
```

- Address is just a "filename"

```
s.bind("/tmp/foo")          # Server binding
s.connect("/tmp/foo")       # Client connection
```

- Rest of the programming interface is the same