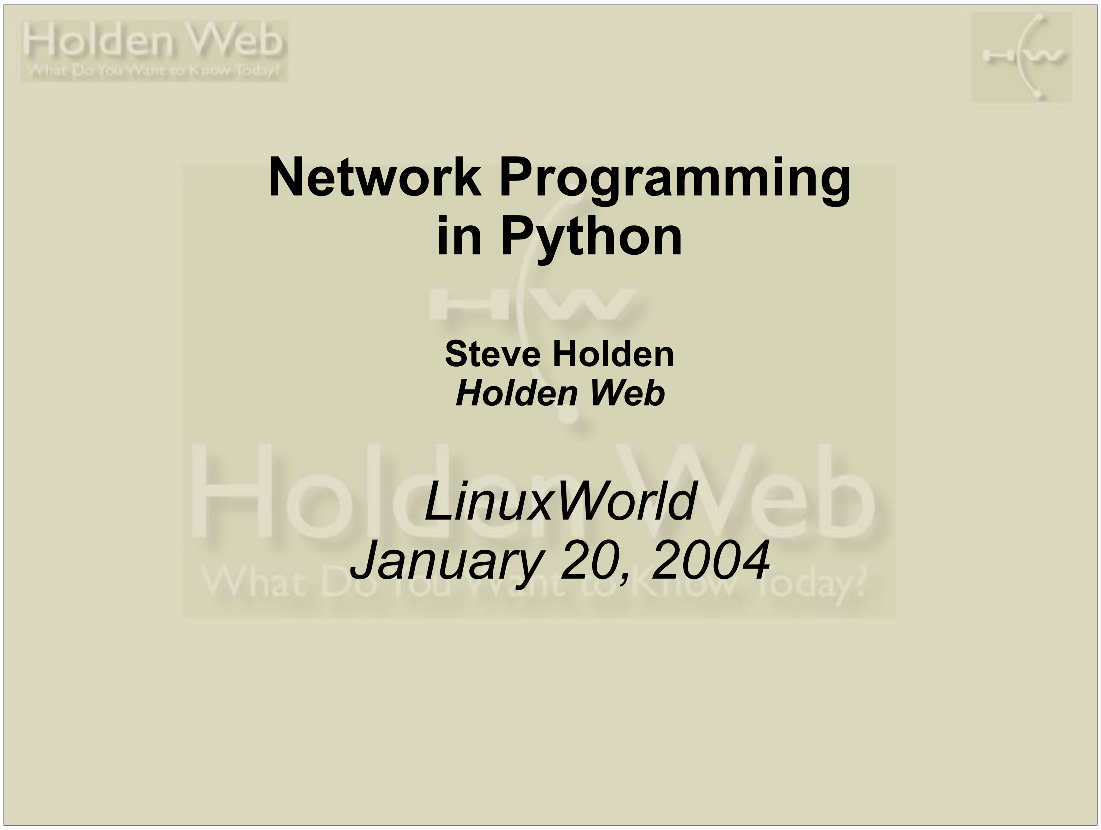


Network Programming in Python

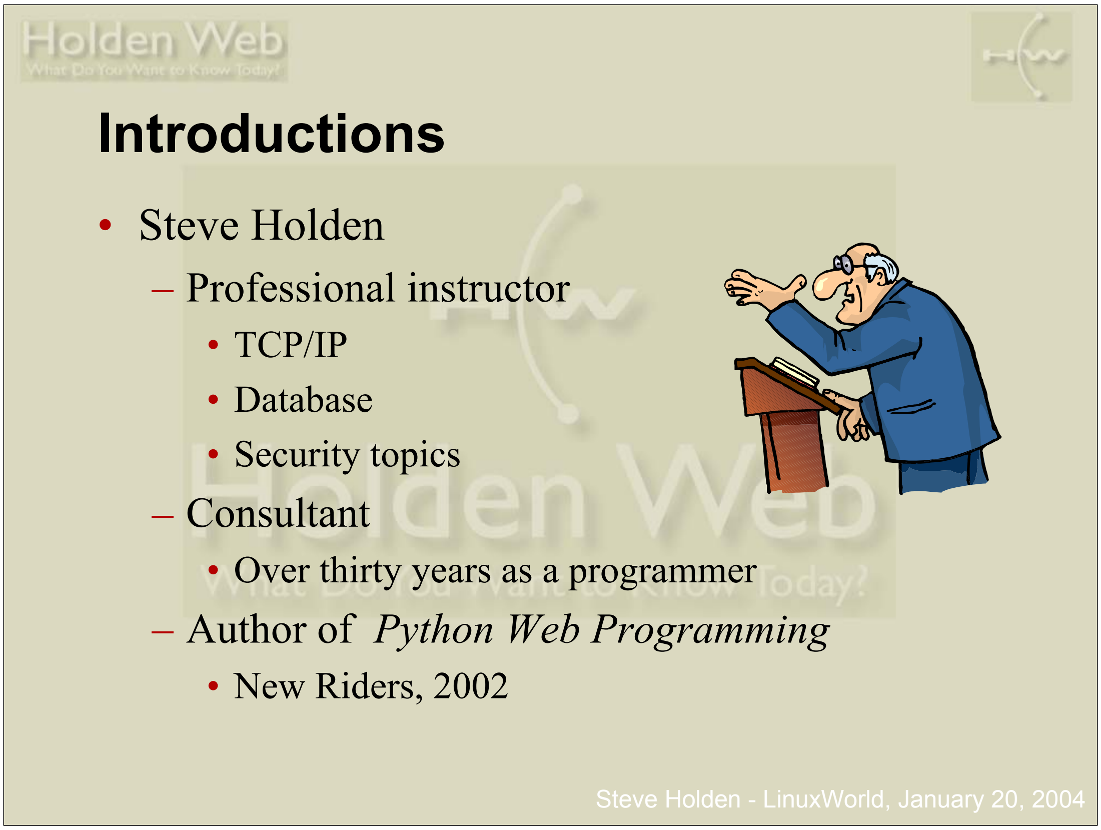
Steve Holden
Holden Web

LinuxWorld
January 20, 2004

Here’s the plan for the three-hour session:

0:10	Introductions
0:20	Network Layering and Client/Server Services Put you at ease with the communications concepts involved and focus on the task ahead. You will probably be familiar with at least some of this material.
0:10	Connectionless programming. Up through Mr. Creosote, a useful remote debugging paradigm.
0:10	Connectionless programming demonstration.
0:30	Connection-oriented programming. The bind/accept paradigm. How clients and servers communicate.
0:15	"Write a client and a server" demonstrations.
0:25	Some client libraries: the ftp, smtp and pop modules.
0:15	Mail or ftp client demonstration
0:25	HTTP and HTML libraries
0:15	Working with the web
0:05	Pointers for future network programming

There will be a 15-minute break at approximately 10:15. We can also take a couple of short comfort breaks as necessary. But you have to let me know that they're necessary!



Introductions

- Steve Holden
  - Professional instructor
    - TCP/IP
    - Database
    - Security topics
  - Consultant
    - Over thirty years as a programmer
  - Author of *Python Web Programming*
    - New Riders, 2002

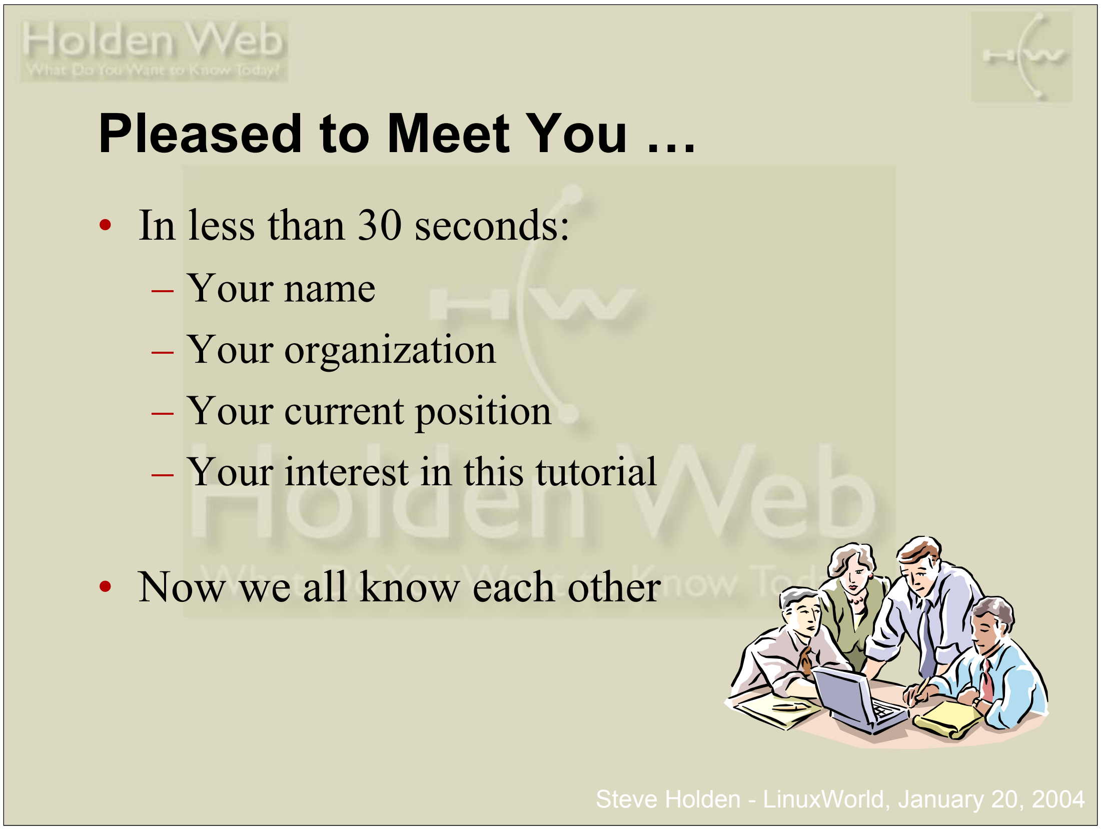
Steve Holden - LinuxWorld, January 20, 2004

I've been involved in teaching for a long time, both as an academic and as a commercial instructor. This slide is just to let you know a bit about me (you turn will come on the next slide).

I first became involved with TCP/IP in 1981, when I met my first BSD VAX 11/750. In 1985 I was appointed Senior Technical Support Specialist for Sun Microsystems in the UK, and a part of my duties was helping to manage the Sun global wide-area network.

When I moved to the USA in the 1990s I had to manage a web server farm controlling about three hundred domains, and developing the associated sites. This stretched my TCP/IP knowledge considerably, and got me deeply involved in web programming. I had done some socket programming in C as a part of my duties at Sun, but hadn't really got very involved with it, as C wasn't my favorite language. The Perl that many sites were written in didn't appeal much more than C. Perl is great when it's used in a disciplined way, but many early web authors took a "program with a shovel" approach.

When I met Python, however, it was a whole different story. Python was everything I wanted in a language – easy to write, easy to read, object-oriented and with a really useful set of libraries, many of which provided networking functionality. When New Riders asked me to write *Python Web Programming* I jumped at the chance.

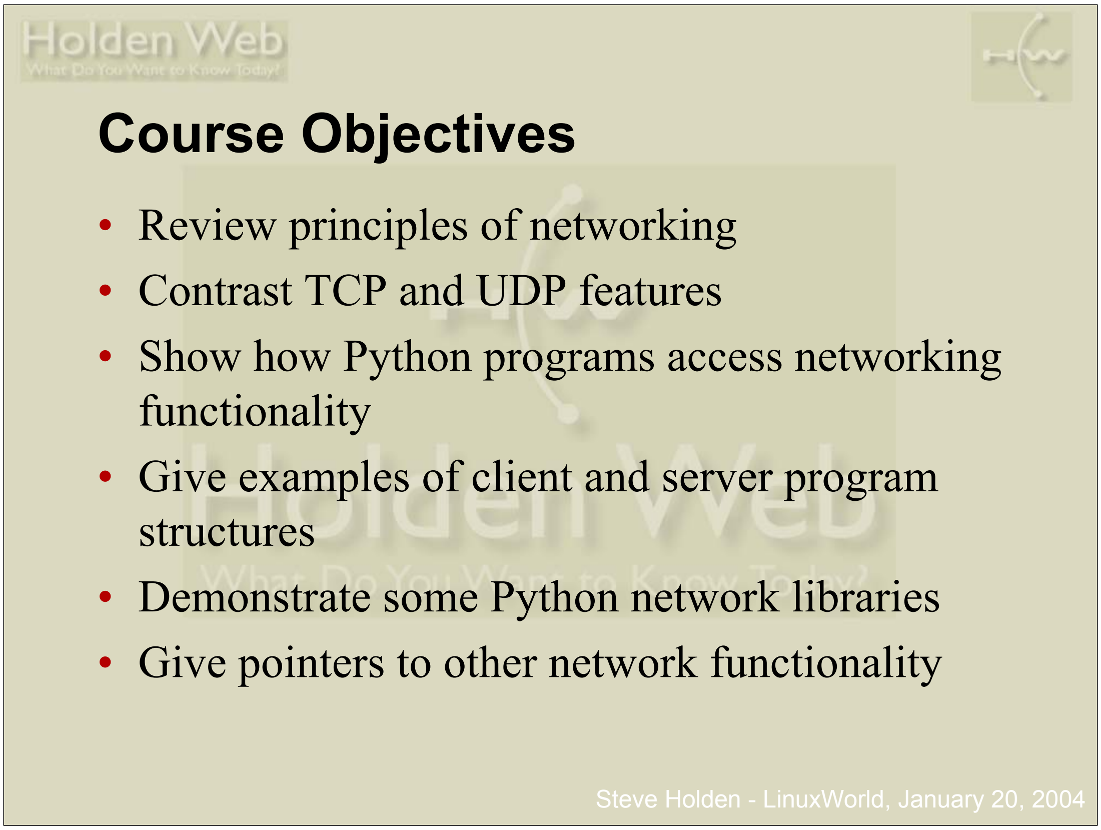


Pleased to Meet You …

- In less than 30 seconds:
  - Your name
  - Your organization
  - Your current position
  - Your interest in this tutorial
- Now we all know each other

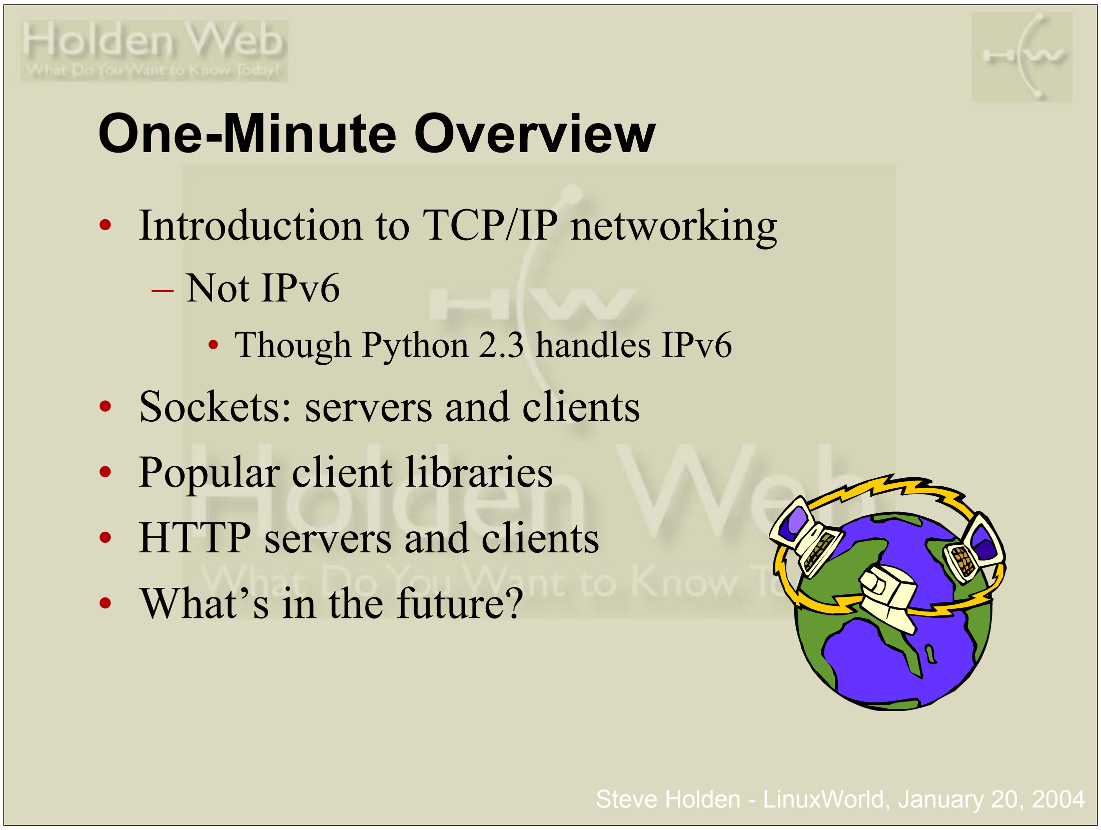
It helps to know why everyone is in the room. You may even find someone else who'd been working on the same problems as you, so pay attention!

Mentally divide ten minutes by the number of students in the class, and try to make sure your introductions don't overrun!



Course Objectives

- Review principles of networking
- Contrast TCP and UDP features
- Show how Python programs access networking functionality
- Give examples of client and server program structures
- Demonstrate some Python network libraries
- Give pointers to other network functionality



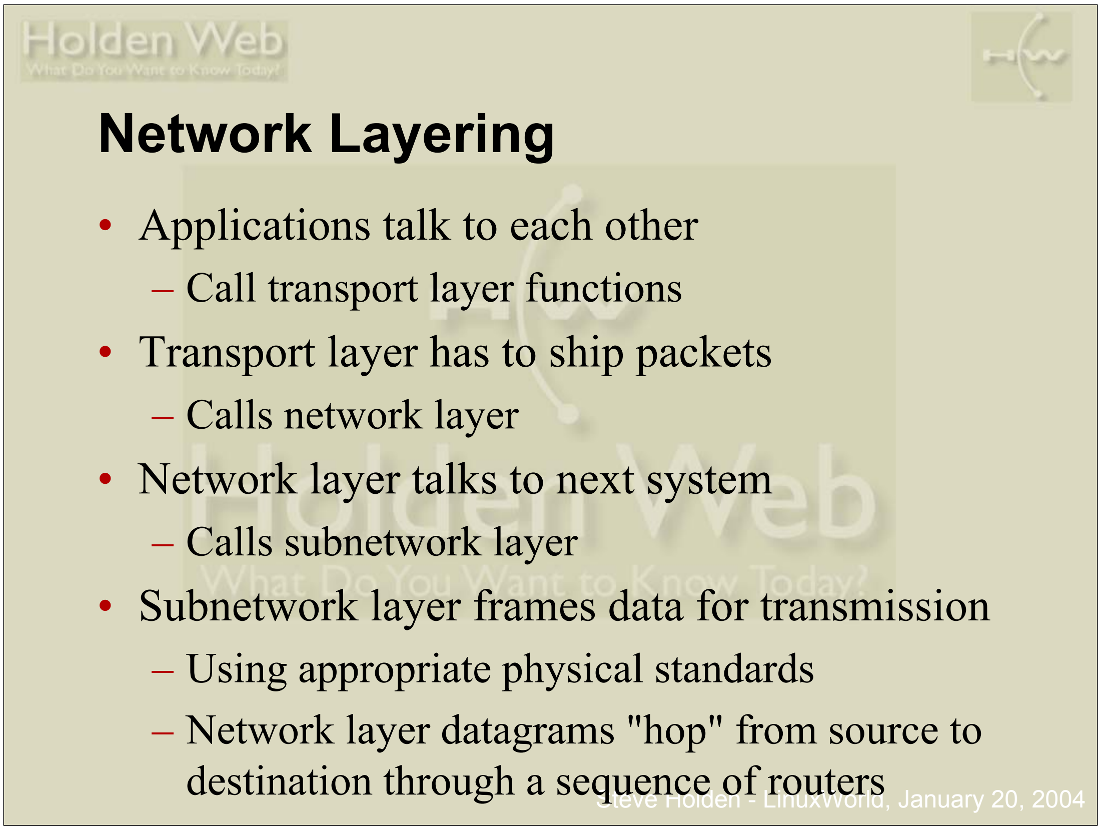
One-Minute Overview

- Introduction to TCP/IP networking
  - Not IPv6
    - Though Python 2.3 handles IPv6
- Sockets: servers and clients
- Popular client libraries
- HTTP servers and clients
- What’s in the future?

The idea behind this workshop is to quickly get you to the stage where you can write useful network-based Python applications.

We start out by looking at an overview of networking in the TCP/IP world, and then examine the basic functionality Python offers for network programming, through the so-called sockets interface.

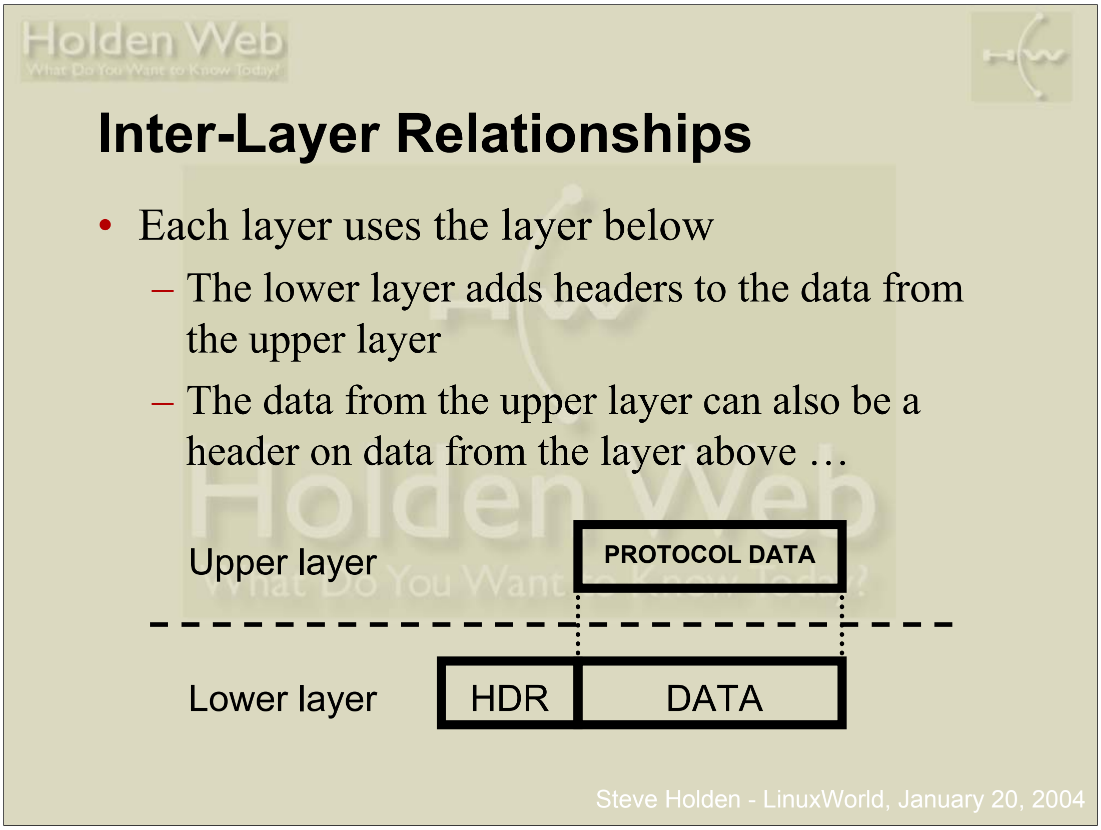
Then we move on to looking at some of the canned libraries you can use for specific purposes.



Network Layering

- Applications talk to each other
  - Call transport layer functions
- Transport layer has to ship packets
  - Calls network layer
- Network layer talks to next system
  - Calls subnetwork layer
- Subnetwork layer frames data for transmission
  - Using appropriate physical standards
  - Network layer datagrams "hop" from source to destination through a sequence of routers

The ISO spent a long time developing a standardized open systems interconnection (OSI) model, based on seven layers. This model was the basis of early networking systems using X.25 as the network layer. As we will see later, TCP/IP uses a simplified model more appropriate to modern transmission systems. This reduced protocol overhead is one of the major reasons for TCP/IP's success in the Open Systems world.



Inter-Layer Relationships

- Each layer uses the layer below
  - The lower layer adds headers to the data from the upper layer
  - The data from the upper layer can also be a header on data from the layer above …

Upper layer

PROTOCOL DATA

Lower layer

HDR

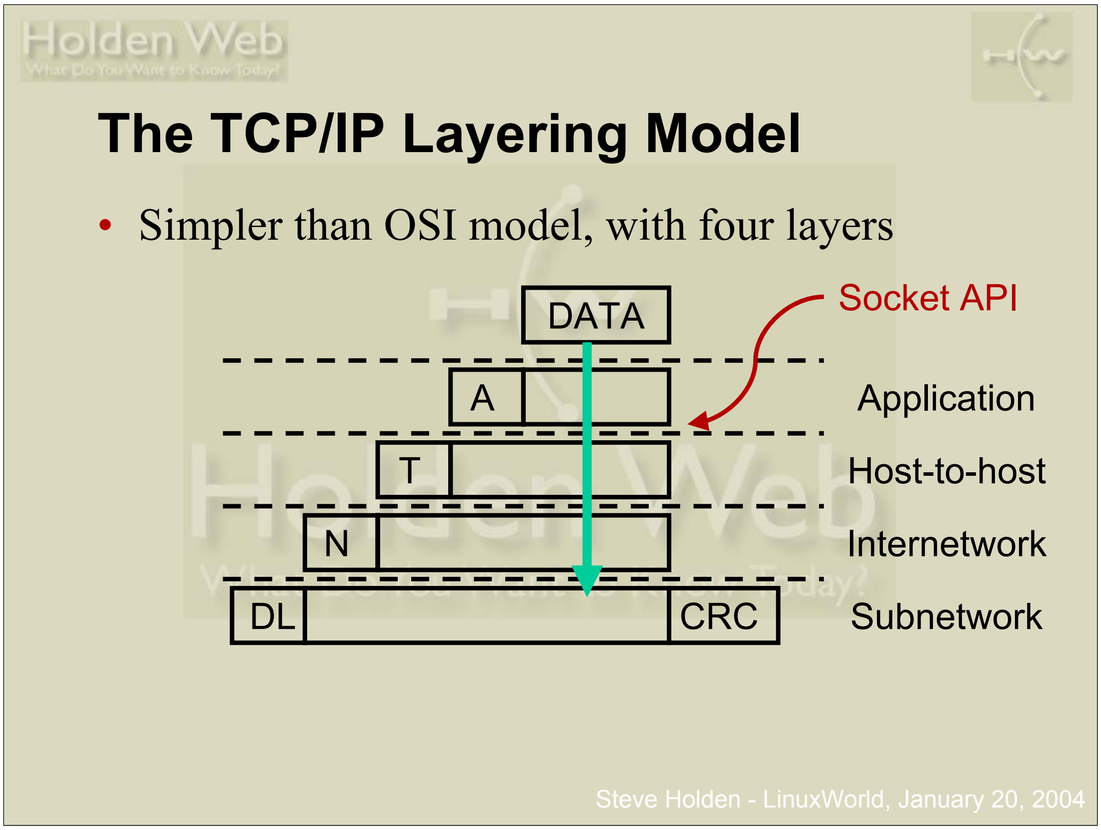
DATA

Steve Holden - LinuxWorld, January 20, 2004

Protocol layering is really quite easy to understand. The advantage is that different layers can cope with different functions.

It would be a very difficult world if a file transfer application had to know whether it was running over an Etherenet or a token ring network and take different actions accordingly. Fortunately the physical network is several layers down from the application code, which is more or less completely isolated from such issues.

We usually think of the application data as traveling "down the stack" to emerge, at the bottom, as a transmission across some network interface. When the transmission reaches its destination it flows "up the stack" again until it is delievered, with all lower-level headers stripped off, to the receiving application process.



The TCP/IP Layering Model

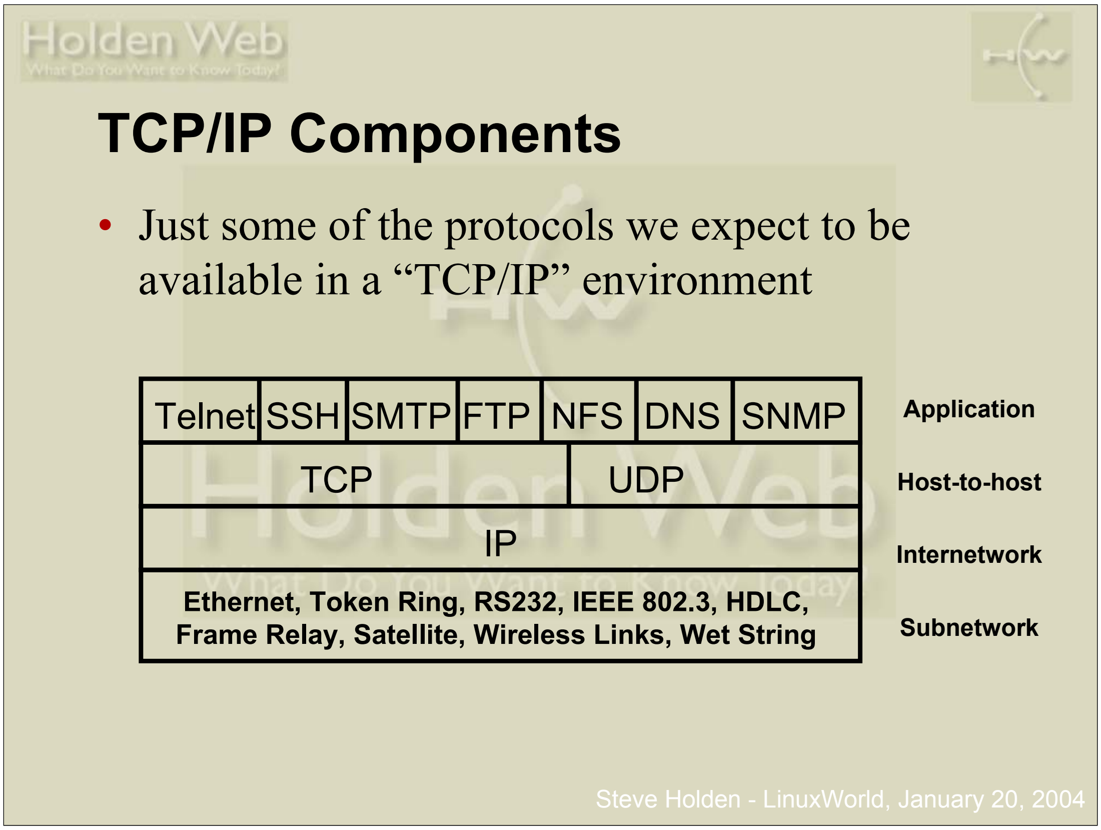
- Simpler than OSI model, with four layers

Steve Holden - LinuxWorld, January 20, 2004

A TCP/IP application must provide its own presentation- and session-layer services if it needs to use them. Many times the data that are interchanged are simple enough not to require presentation layer services, and the interactions are short so no session-layer services are needed. The TCP/IP application layer therefore corresponds to the top three layers of the OSI model. From an application point of view the distinction between datalink and physical layer is irrelevant. They are both collapsed into a single subnetwork layer.

The really important point is that the application can treat the transport-layer API as its way to communicate with remote processes. The layered complexity is essentially invisible to the application code, and is called on by the transport layer libraries without any action being required by the application..

The subnetwork layer *data* is actually an IP datagram, whose data is a TCP or UDP segment, whose data is an application protocol data unit.



TCP/IP Components

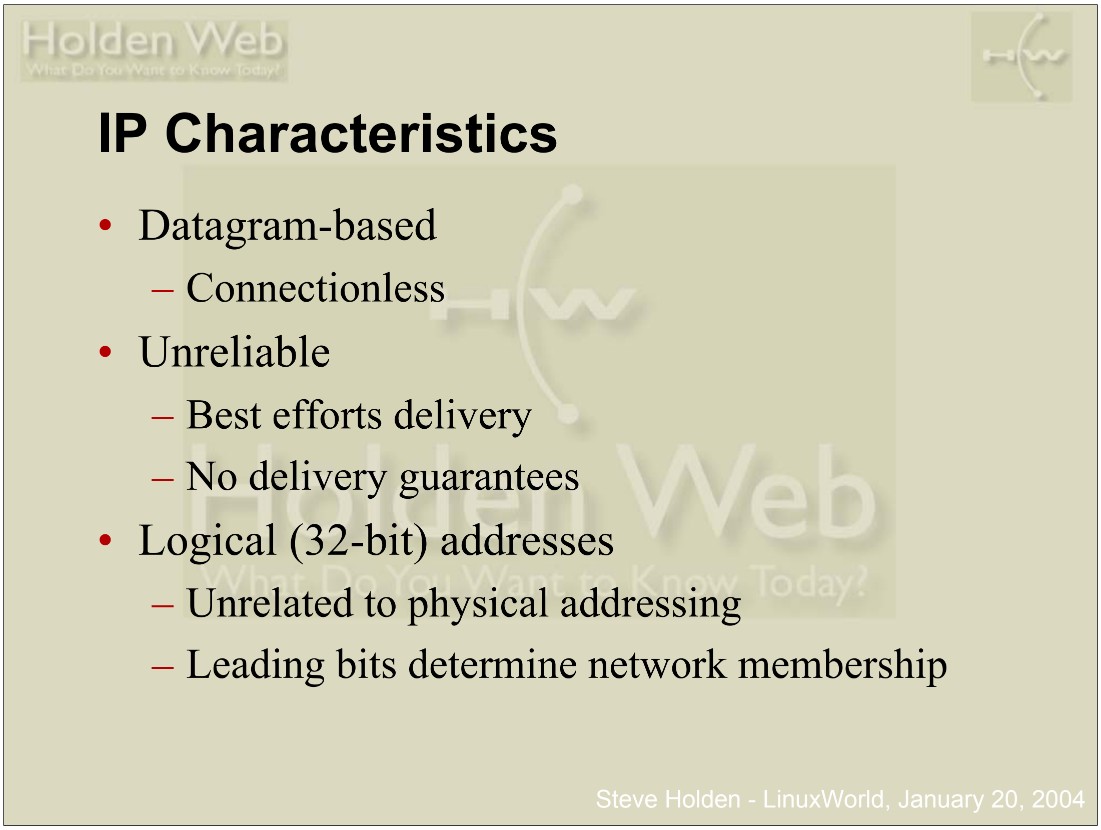
- Just some of the protocols we expect to be available in a "TCP/IP" environment

Telnet	SSH	SMTP	FTP	NFS	DNS	SNMP	**Application**
TCP				UDP			**Host-to-host**
IP							**Internetwork**
Ethernet, Token Ring, RS232, IEEE 802.3, HDLC, Frame Relay, Satellite, Wireless Links, Wet String							**Subnetwork**

Another advantage of TCP/IP is the incredible adaptability of the network layer. It's hard to think of a transmission medium that hasn't been used to carry IP datagrams in some network or other! Two major transport layer protocols (TCP and UDP) are used, though IP can carry many others, including ISO protocols in some cases.

Most applications are designed to run over one or the other of these two major transport protocols, but certain applications such as NFS (the Network File System) and DNS (the Domain Name System) are designed to use either transport layer. Usually TCP is used over wide-area links, and UDP is used over LANs, which are typically rather more reliable.

When we talk about "TCP/IP" we actually assume the presence of all the components shown above, and more besides.



IP Characteristics

- Datagram-based
  - Connectionless
- Unreliable
  - Best efforts delivery
  - No delivery guarantees
- Logical (32-bit) addresses
  - Unrelated to physical addressing
  - Leading bits determine network membership

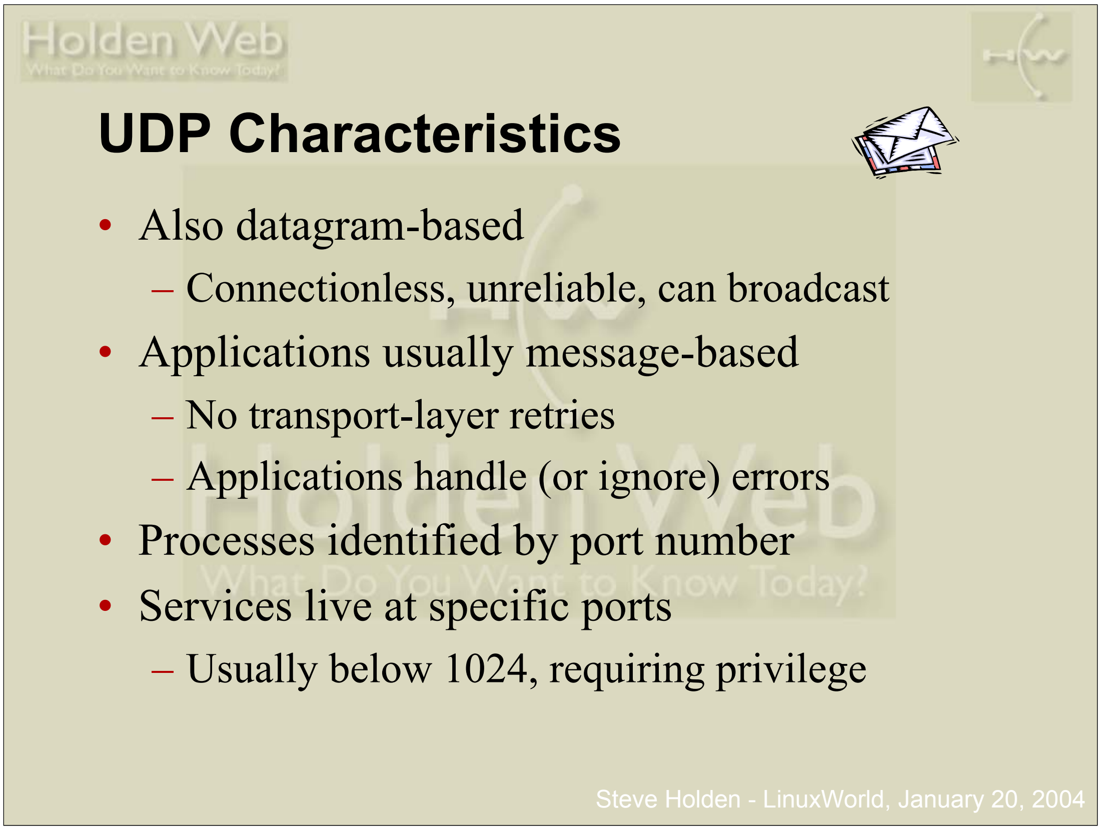
Steve Holden - LinuxWorld, January 20, 2004

The network layer of TCP/IP makes no guarantees of delivery: at any hop along the way a packet may be discarded, either because transmission errors have been detected or simply because the receiving equipment does not have the capacity to process it.

In such circumstances the originating host may or may not receive an ICMP (Internet Control Message Protocol) message detailing the reason for non-delivery. ICMP messages can be useful in helping hosts to change to more appropriate behavior, but are not always actioned even when raised.

Since different physical network use different addressing schemes it's important that IP provides a unified scheme independent of the underlying hardware. This is crucial when building large internets (an "internet" with a small "I" is any interconnected collection of IP networks). You can find a detailed description of IP addressing at

http://www.holdenweb.com/students/3comip.pdf



UDP Characteristics

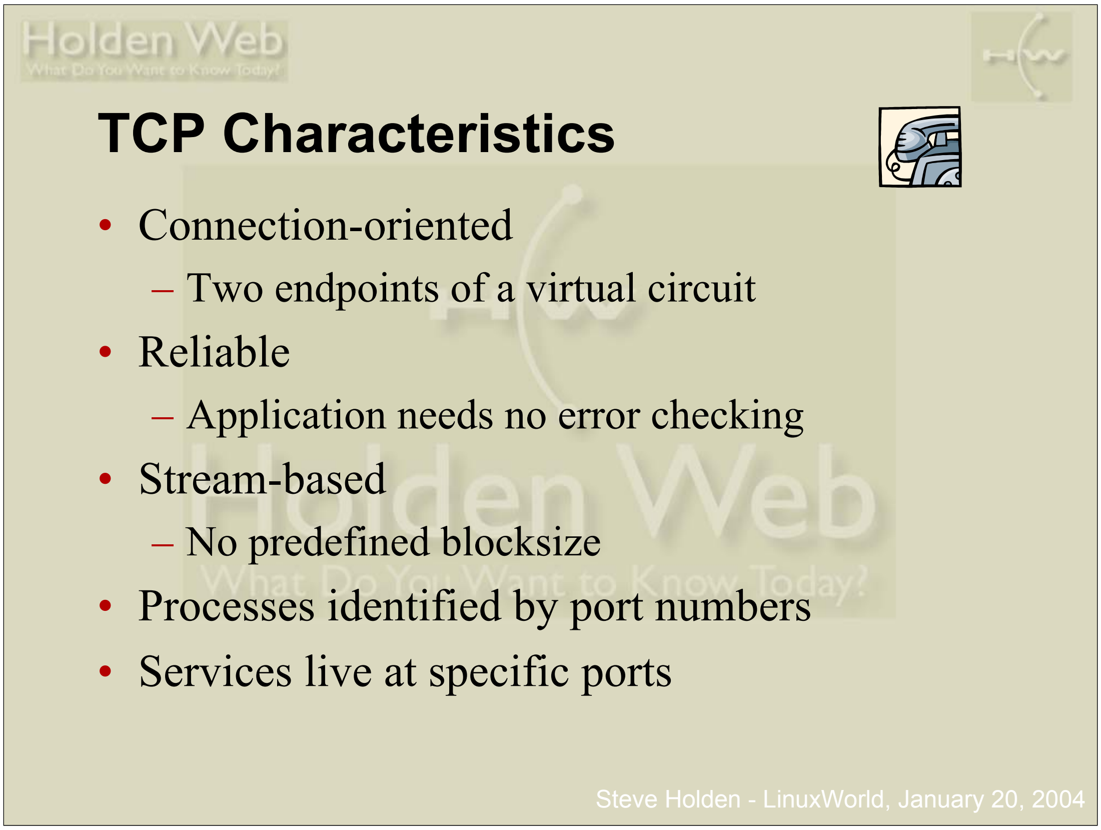
- Also datagram-based
  - Connectionless, unreliable, can broadcast
- Applications usually message-based
  - No transport-layer retries
  - Applications handle (or ignore) errors
- Processes identified by port number
- Services live at specific ports
  - Usually below 1024, requiring privilege

Steve Holden - LinuxWorld, January 20, 2004

You can think of UDP as similar to the postal service – packets go out, they can be lost along the way if an intermediate router runs out of resources or if they are mangled by noise, no acknowledgements are issued. UDP applications are therefore usually simpler messaging-style applications, where the only recovery action will be a fixed number of retries.

DNS is a good example -- although it can use TCP, most DNS traffic is carried by UDP. If a server does not reply, or if the reply (or the request) gets lost, the consequences are usually not tragic. If you fail to resolve a host name because of such an error you (as a user) will usually be quite happy to try again, and this failure will not have been significant.

As you will see later, a server binds to a specific port when it starts up. Originally a client had to "just know" what port the server would be listening on, and so default port numbers were allocated: 80 for HTTP, 25 for SMTP and so on. Later on, services like the PortMapper were introduced so that a server could use any port, and register it with the Portmapper – the clients would start with a PortMapper enquiry to find out which port their server was listening on.



TCP Characteristics

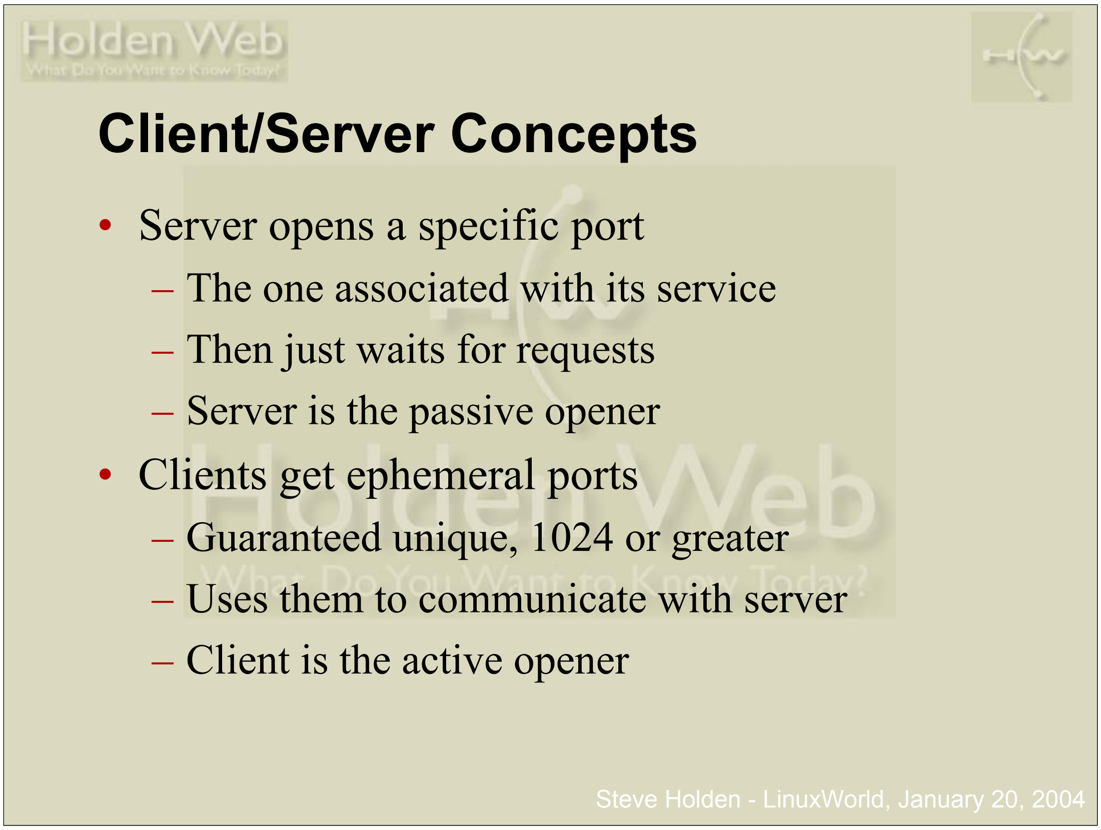
- Connection-oriented
  – Two endpoints of a virtual circuit
- Reliable
  – Application needs no error checking
- Stream-based
  – No predefined blocksize
- Processes identified by port numbers
- Services live at specific ports

Steve Holden - LinuxWorld, January 20, 2004

A TCP conversation requires specific endpoints, so you can think of it as more like a telephone call: one system calls another, and has to get a reply before communication can take place.

Because of the need for defined endpoints, TCP has no broadcast capability, unlike UDP. Because TCP is a reliable protocol the possibility of error is rather low (though not zero) – errors are detected and corrected in the transport layer. This makes it relatively easy to use, since the applications can assume the data they send will, eventually, be received in the absence of a complete connectivity failure.

TCP is a much more sophisticated protocol than UDP and includes both error detection and correction, relieving applications of housekeeping tasks which would otherwise quickly become burdensome. There is no need to transmit data is any particular block size, and an application is free to send as few as one byte or as much as several megabytes at a time. The transport layer will take responsibility for buffering this data and sending it out as a stream of packets of an appropriate size.



Holden Web
What Do You Want to Know Today?

Client/Server Concepts

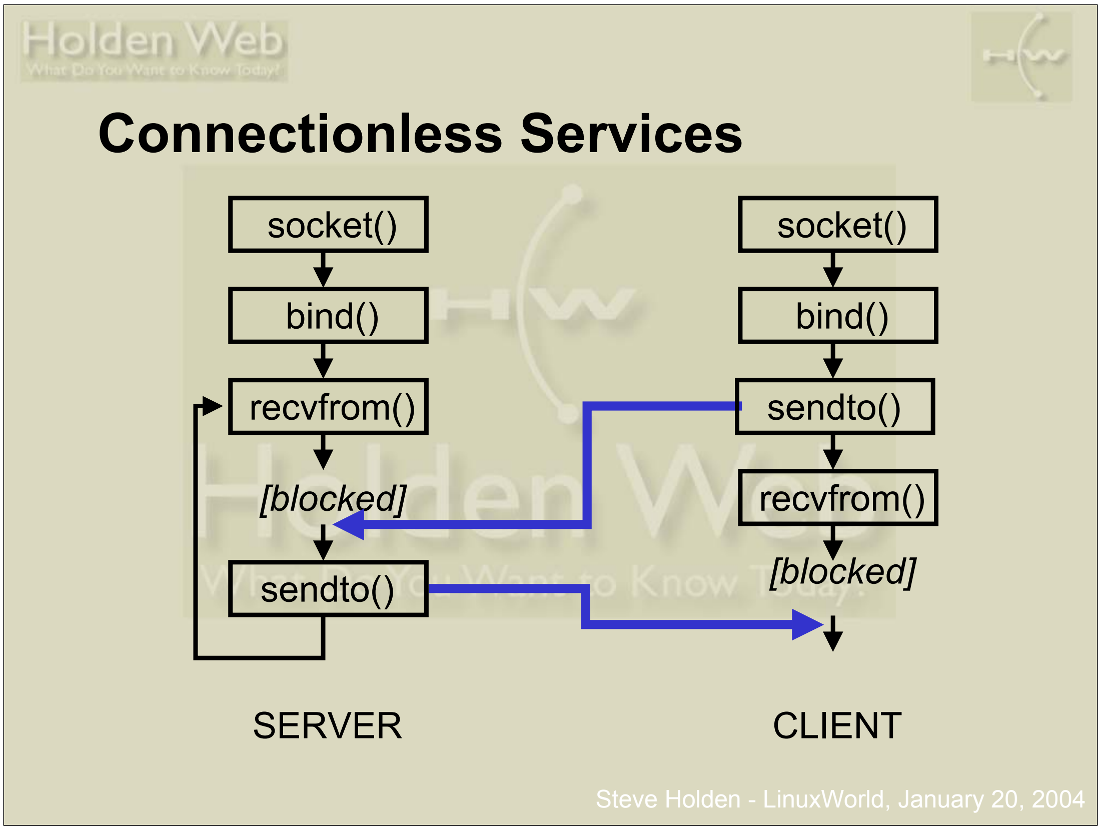
- Server opens a specific port
  - The one associated with its service
  - Then just waits for requests
  - Server is the passive opener
- Clients get ephemeral ports
  - Guaranteed unique, 1024 or greater
  - Uses them to communicate with server
  - Client is the active opener

Steve Holden - LinuxWorld, January 20, 2004

Although not all client/server systems are based around the idea of a passive and an active opener this is currently the dominant paradigm for networking, used by FTP, Telnet, DNS, SMTP, SNMP and many others.

The client and the server will typically use the socket API, originally devised as part of the bsd Unix implementation, to interact with the networking features of their host operating system. A socket gives the application code access to the network interface using convenient calls to transmit and receive data. Python standardizes access to sockets to eliminate the need to worry about platform differences -- network programming is remarkably consistent among Linux, UNIX and Windows.

The fact that a client port number is guaranteed unique on its host allows traffic to be multiplexed and demultiplexed over shared network paths. Even when two telnet clients on the same host are in touch with telnet servers on the same host the client address/port number combination is unique, allowing the traffic to be delivered to the correct process.



Connectionless Services

socket()
bind()
recvfrom()
[blocked]
sendto()

SERVER

socket()
bind()
sendto()
recvfrom()
[blocked]

CLIENT

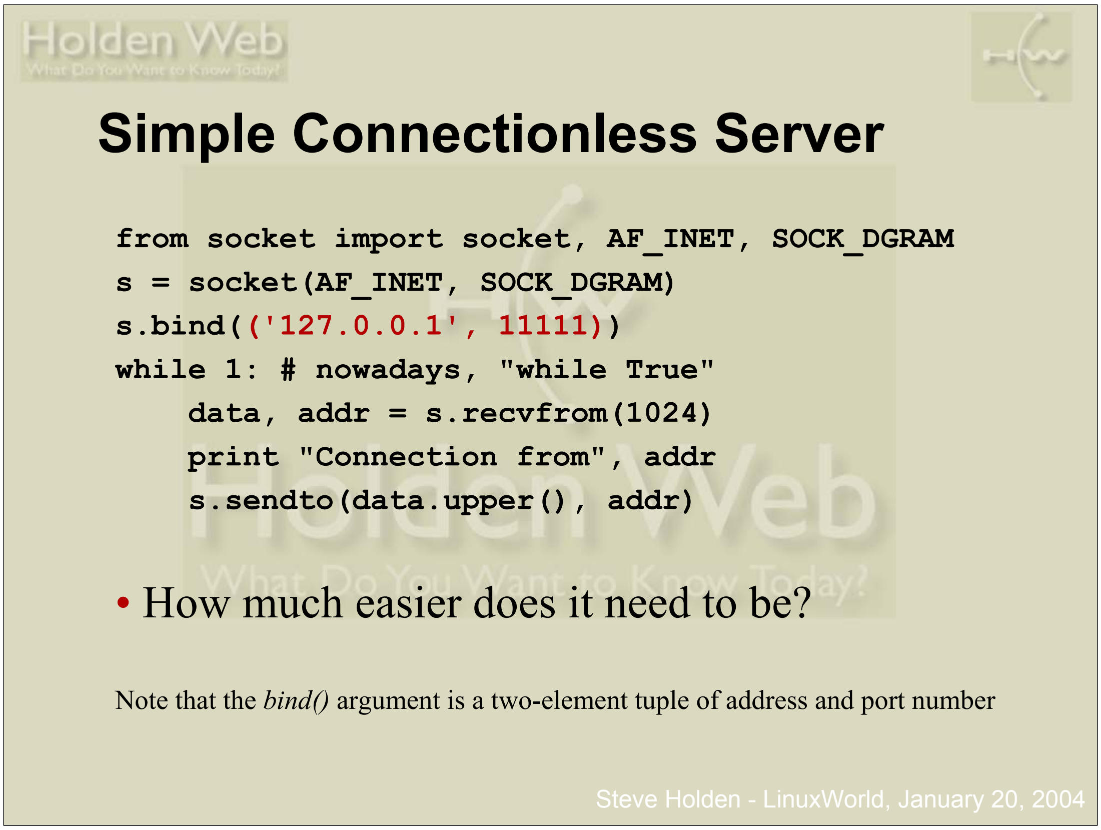
When UDP is used, the server creates a socket and binds address(es) and a port number to it. The server then waits for incoming data (remember: UDP is connectionless).

The clients also create a socket, then they bind it to the appropriate interface – typically allowing the transport layer to choose an *epehemeral* (short-lived) port number rather than specifying a particular port.

The client sends data to the server, which awakes from its blocked state and starts to compute its response. Meantime the client has issued a **`recvfrom()`** using the same address it sent the data to, and is blocked awaiting the response which should eventually arrive from the server.

When the server sends its result back, it goes to the address and port number the incoming data was received from. The server then loops around to wait for another request. The arrival of the server's data unblocks the client, which can then continue.

This is something of a simplification: using a library based on the **`select()`** system call it is possible to use sockets in a non-blocking fashion. This does complicate the code somewhat, however.



Simple Connectionless Server

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 11111))
while 1: # nowadays, "while True"
    data, addr = s.recvfrom(1024)
    print "Connection from", addr
    s.sendto(data.upper(), addr)
```

- How much easier does it need to be?

Note that the *bind()* argument is a two-element tuple of address and port number

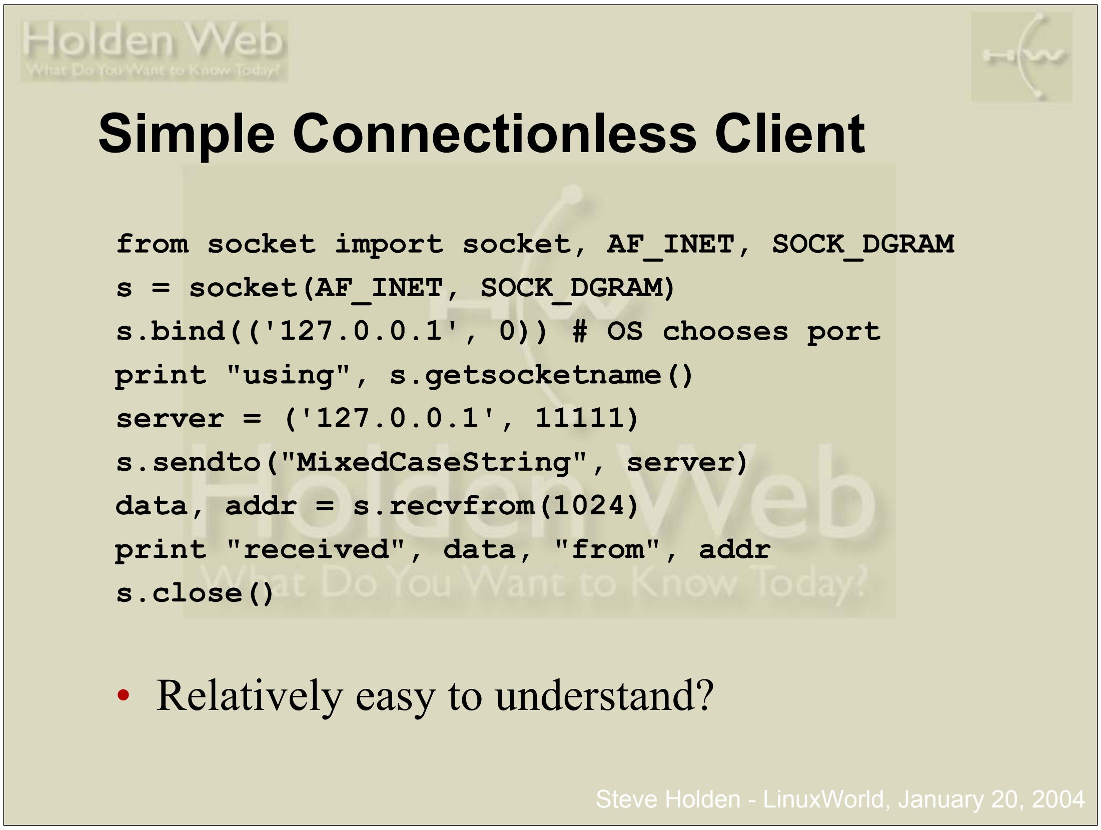
It's more usual to import the whole socket library and use qualified names, but the **`from`** statement is a convenient way to access only specific names from a module. I did this to keep code lines on the slide shorter, and hence more readable. The following code is an equivalent but rather more conventional way to create the socket:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

This achieves exactly the same ends. Which style you use is largely a matter of taste and readability. The remainder of the code will run unchanged since the other features it uses are attributes of the socket, and are accessed the same way no matter how the socket was created.

Note carefully that the **`bind()`** call takes a single argument, a tuple containing an IP address string and a port number. If the IP address is the empty string then the code will bind to all interfaces, which is how most servers actually start up. The example above is a little more secure, since only local processes can connect via the local loopback interface.

The code on this slide is on your CD as **`udpserv1.py`** if you want to run it now.



Simple Connectionless Client

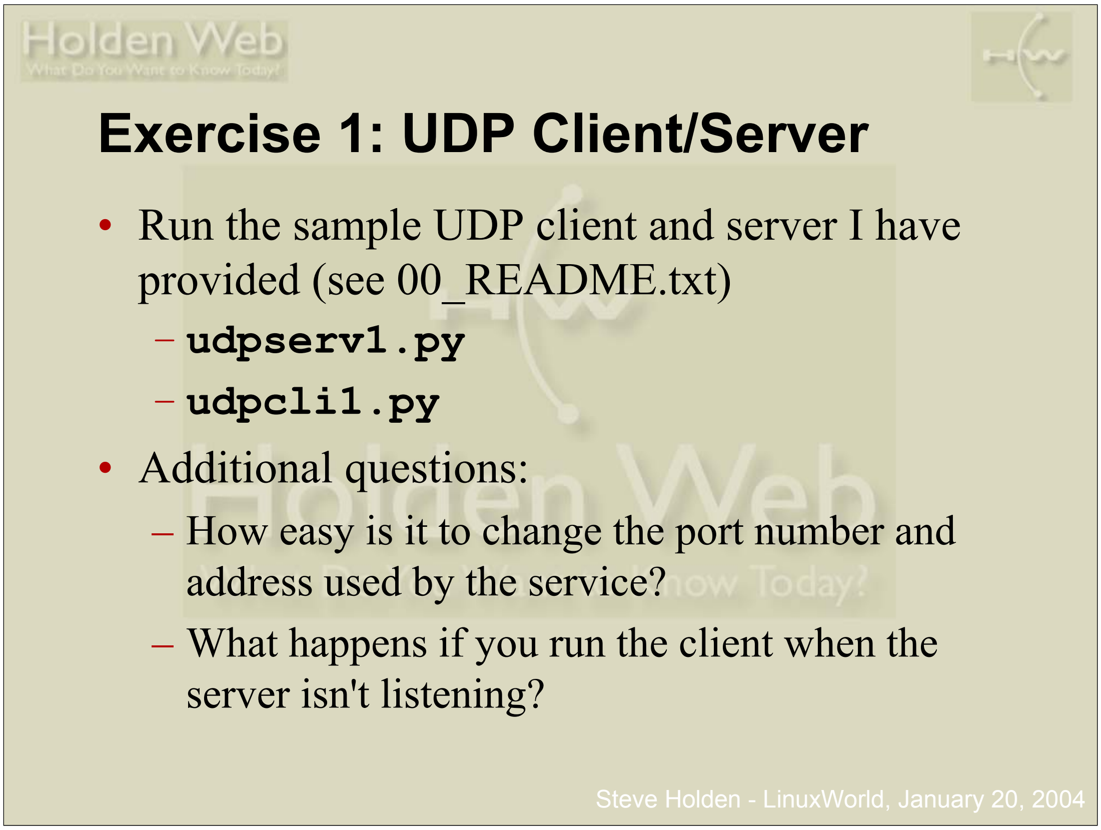
```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 0)) # OS chooses port
print "using", s.getsocketname()
server = ('127.0.0.1', 11111)
s.sendto("MixedCaseString", server)
data, addr = s.recvfrom(1024)
print "received", data, "from", addr
s.close()
```

- Relatively easy to understand?

The client specifies port number zero to indicate that it simply wants an ephemeral port – this is more efficient than attempting to use a specific port number because the port requested might already be in use and then the **`bind()`** call would fail.

The **`getsocketname()`** call tells the user the address and port number being used for the client end of the communication. While this isn't an essential part of the program it's useful debugging data.

The client simply sends the data and (usually) receives a reply from the server. This particular program is somewhat inadequate in terms of error checking: if the server's response is somehow lost then the client will hang forever. The socket library was recently (2.3) updated to include timeout features that previously had only been available in third-party additions.



Holden Web

What Do You Want to Know Today?

Exercise 1: UDP Client/Server

- Run the sample UDP client and server I have provided (see 00_README.txt)
  - `udpserv1.py`
  - `udpcli1.py`
- Additional questions:
  - How easy is it to change the port number and address used by the service?
  - What happens if you run the client when the server isn't listening?

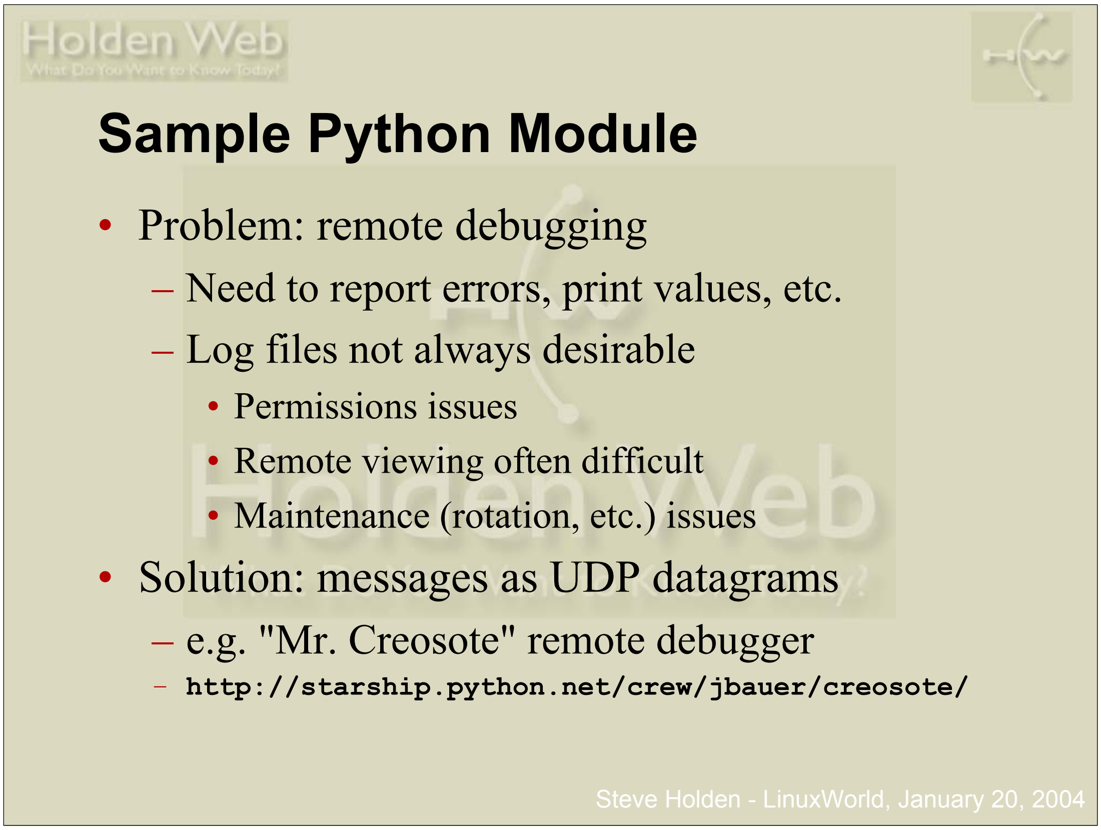
Steve Holden - LinuxWorld, January 20, 2004

This simple exercise allows you to verify that your networking setup is OK and that Python is making things happen as it should.

The programs are *very* simple, with no error checking at all. When you run the client with no server listening it will hang, waiting forever for a non-existent server to reply. Clearly this situation isn’t ideal when everything is supposed to run unattended ☺

Note that these programs will run on almost an platform that Python supports. The socket layer is very robust, and considerably eases portability problems in networked applications. Once I found out how easy Ptyhon was to use in this way I entirely abandoned any idea of using C or C++. No hair shirts for me …

You could also try to run two copies of the server. What happens then?



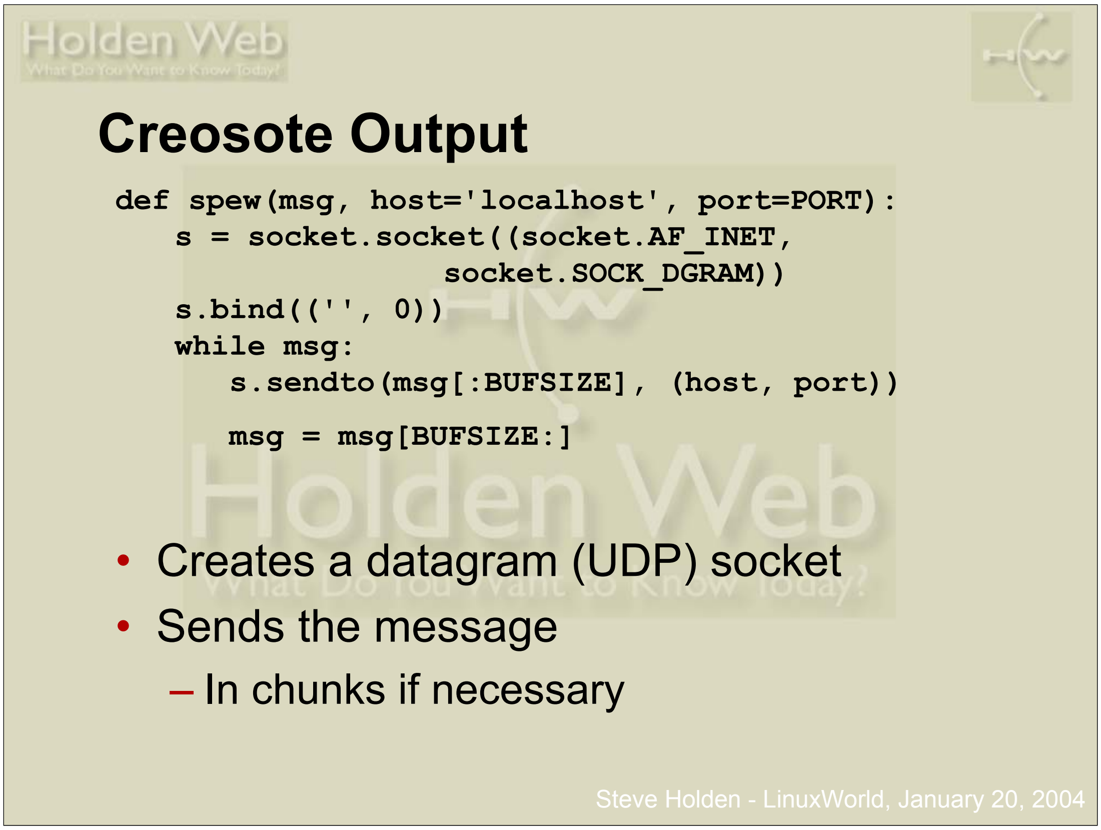
Sample Python Module

- Problem: remote debugging
  - Need to report errors, print values, etc.
  - Log files not always desirable
    - Permissions issues
    - Remote viewing often difficult
    - Maintenance (rotation, etc.) issues
- Solution: messages as UDP datagrams
  - e.g. "Mr. Creosote" remote debugger
  - `http://starship.python.net/crew/jbauer/creosote/`

Please try to overlook the fact that certain inhabitants of Pythonia appear to find it necessary to drag in obscure references to Monty Python's Flying Circus. The name is in no way meaningful.

The creosote module shows how useful UDP can be for messaging-style applications. Debugging messages can be emitted by a program, and it doesn't matter whether anything is listening or not. You could even use IP address 255.255.255.255 (the local broadcast address) as the destination for reports, or the directed broadcast address of your local subnet.

In either case, any machine on the LAN that was running a creosote debugger would pick uo the broadcasts and display them!



Creosote Output

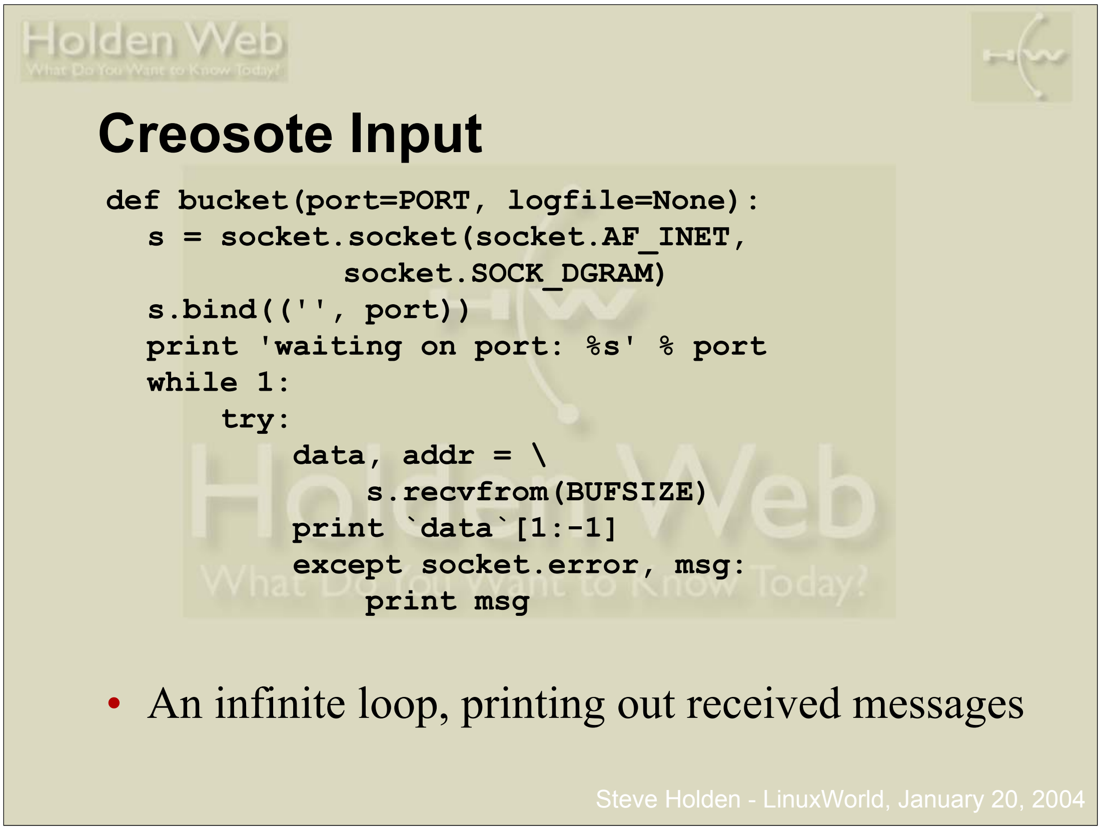
```
def spew(msg, host='localhost', port=PORT):
    s = socket.socket((socket.AF_INET,
                       socket.SOCK_DGRAM))
    s.bind(('', 0))
    while msg:
       s.sendto(msg[:BUFSIZE], (host, port))
       msg = msg[BUFSIZE:]
```

- Creates a datagram (UDP) socket
- Sends the message
  - In chunks if necessary

This is pretty much the output routine from the Mr. Creosote module.

The definition line shows how Python allows you to define default values. In this case the (destination) host defaults to the machine the code is running on, and the (destination) port number to a value previously defined globally in the module.

The **`bind()`** call requests an ephemeral port usable on any interface. The while loop sends chunks (whose size, **`BUFSIZE`**, is also a module parameter) which are successively deleted from the message. The loop terminates when the message is reduced to an empty string.



Creosote Input

```
def bucket(port=PORT, logfile=None):
  s = socket.socket(socket.AF_INET,
             socket.SOCK_DGRAM)
  s.bind(('', port))
  print 'waiting on port: %s' % port
  while 1:
      try:
          data, addr = \
              s.recvfrom(BUFSIZE)
          print `data`[1:-1]
          except socket.error, msg:
              print msg
```

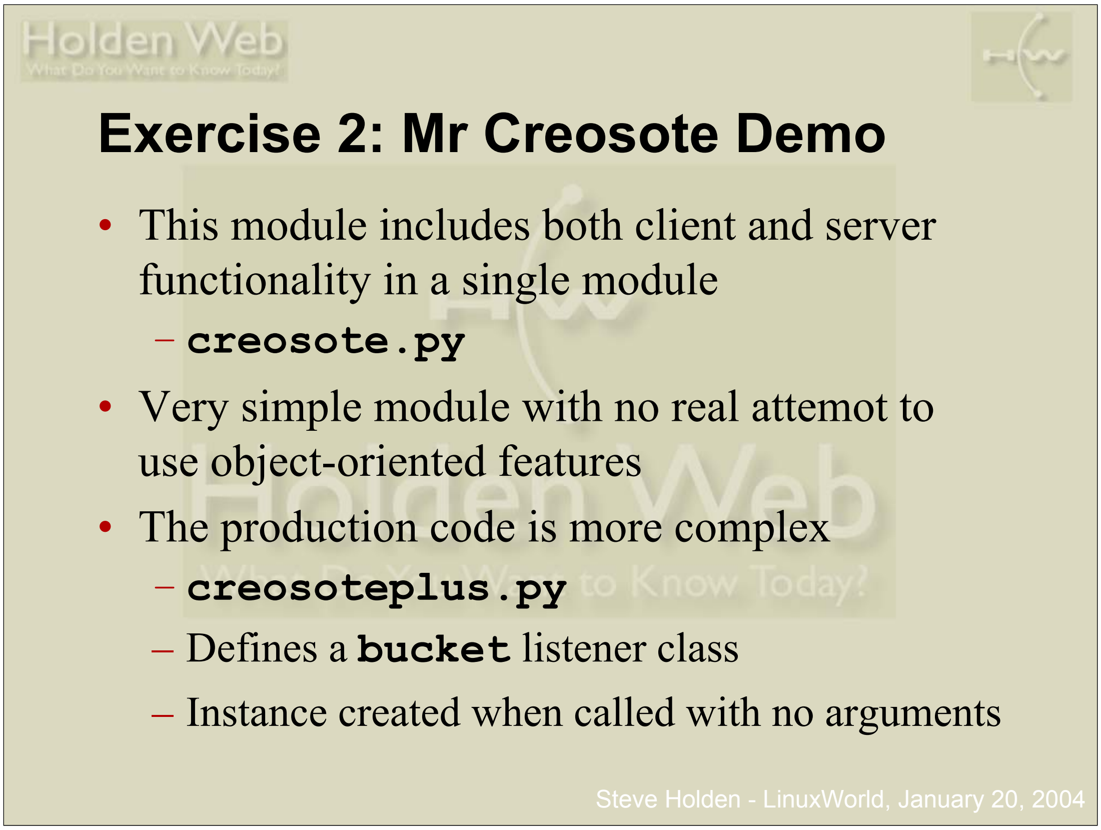
- An infinite loop, printing out received messages

This slide is a simplification of the real code, which has logging abilities that I have omitted clarity.

Again the port defaults to a value preset elsewhere in the module. The function binds to all local addresses on the given port number (the empty string) and proceeds to receive messages on that socket.

The `print` statement uses backticks to produce a readable representation of the message even if it contains binary characters, and strips the quotes off with the `[1:-1]` slicing.

The **`while 1`** loop is typical of server code: a good server will run forever. In the case of the **`recvfrom()`** call the argument sets the maximum message size than can be received.



Exercise 2: Mr Creosote Demo

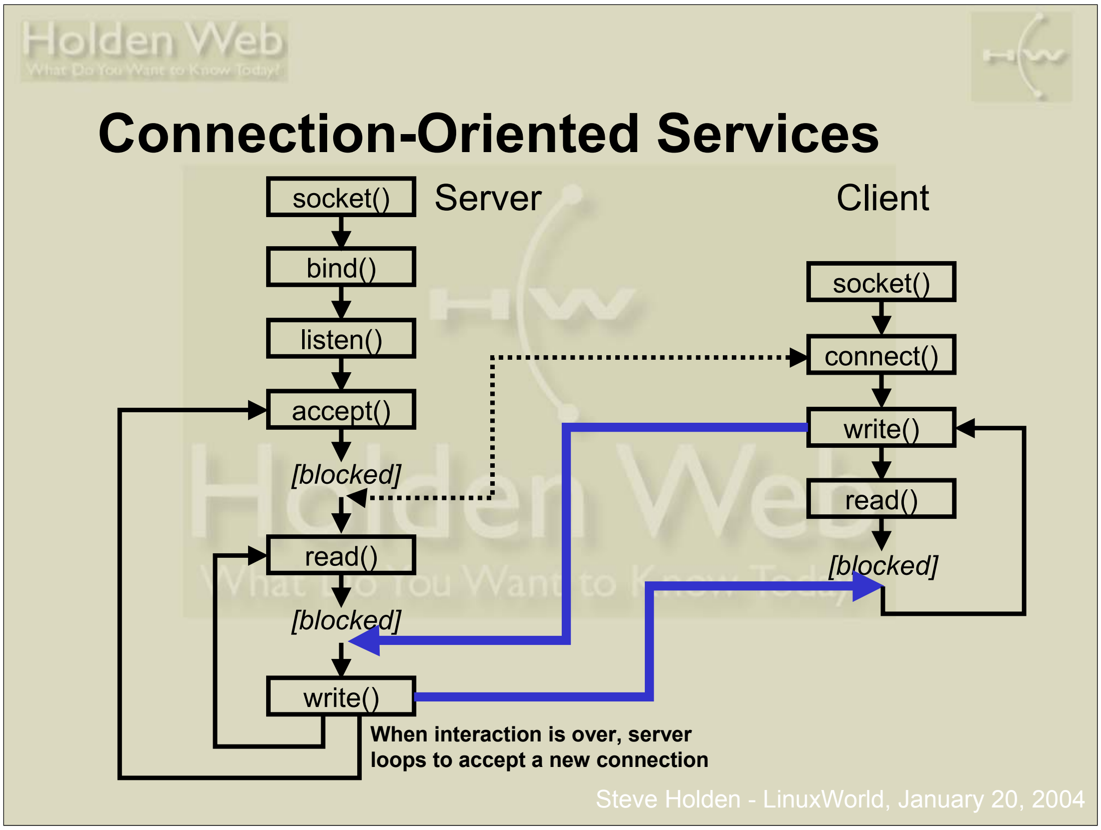
- This module includes both client and server functionality in a single module
  - `creosote.py`
- Very simple module with no real attemot to use object-oriented features
- The production code is more complex
  - `creosoteplus.py`
  - Defines a **`bucket`** listener class
  - Instance created when called with no arguments

Steve Holden - LinuxWorld, January 20, 2004

You may not necessarily have a laptop with you today, but I am hoping that enough of you will have brought them with you to make exercises practical. All the code is available on CD or via the web, so you will be able to run the exercises when you get back home anyway.

To assist those who aren't able to work hands-on I will demonstrate how code can be written and run on my own laptop, and field questions interactively.

Because a current focus is integration of Windows and open source technologies I will be running the exercise code on a Windows 2000 laptop, sometimes in a command window and sometimes under Cygwin. The exercises have all been carefully tested on Linux, Windows and Cygwin, and so you shouldn't have any portability problems. That's one of Python's strengths!



Holden Web

What Do You Want to Know Today?

Connection-Oriented Services

Server

socket()

bind()

listen()

accept()

[blocked]

read()

[blocked]

write()

Client

socket()

connect()

write()

read()

[blocked]

When interaction is over, server loops to accept a new connection

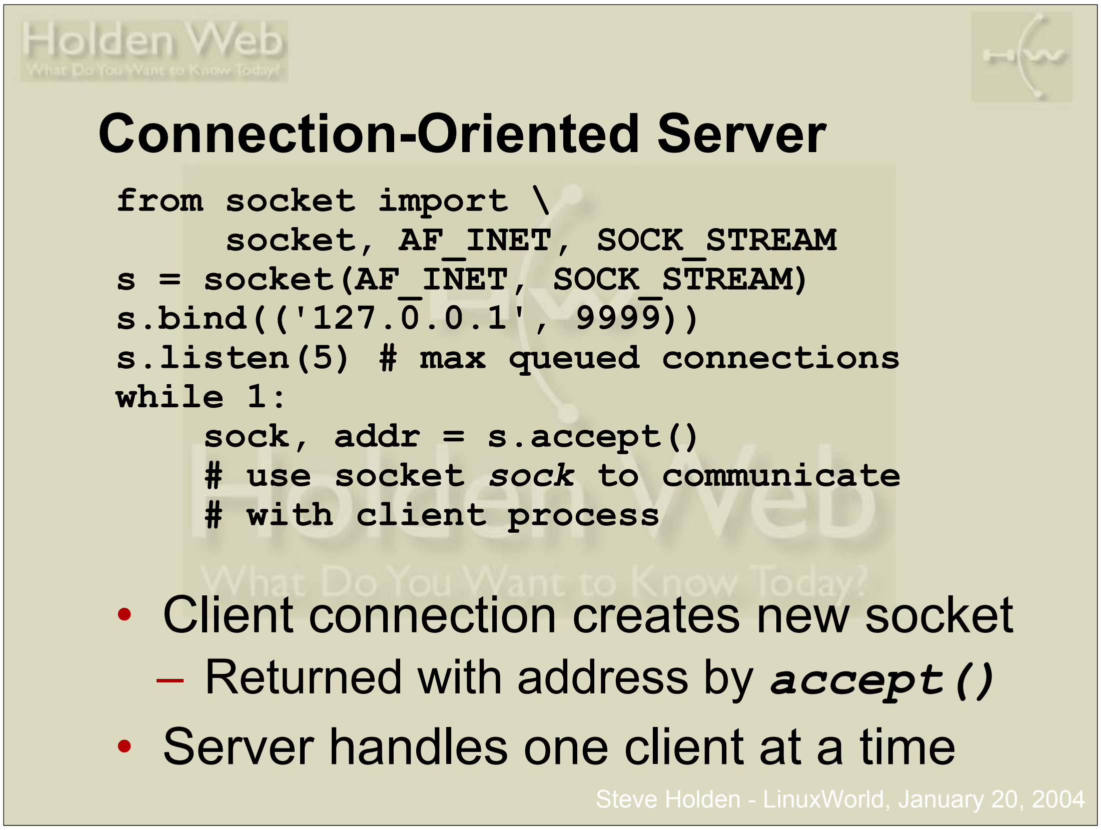
Steve Holden - LinuxWorld, January 20, 2004

A connection-oriented server creates a socket, binds it to one or more local ports on which it will listen for connections, and then puts the socket into the listening state to wait for incoming connections. At this point the server process blocks until a connection request arrives.

A client creates its own socket, usually without specifying any particular port number, and then connects to the endpoint the server is listening on. The server’s **`accept()`** call returns a ***new*** socket that the server can use to send data across this particular connection..

The two parties then exchange data using **`read()`** and **`write()`** calls.

The major limitation of this structure is the non-overlapped nature of the request handling in the server. Theoretically it's possible for the server to use its original socket to listen for further requests while the current request is being handled, but that isn't shown here. You will learn how to overcome this limitation using standard library classes.



Connection-Oriented Server

```
from socket import \
     socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.bind(('127.0.0.1', 9999))
s.listen(5) # max queued connections
while 1:
    sock, addr = s.accept()
    # use socket sock to communicate
    # with client process
```

- Client connection creates new socket
  - Returned with address by ***accept()***
- Server handles one client at a time

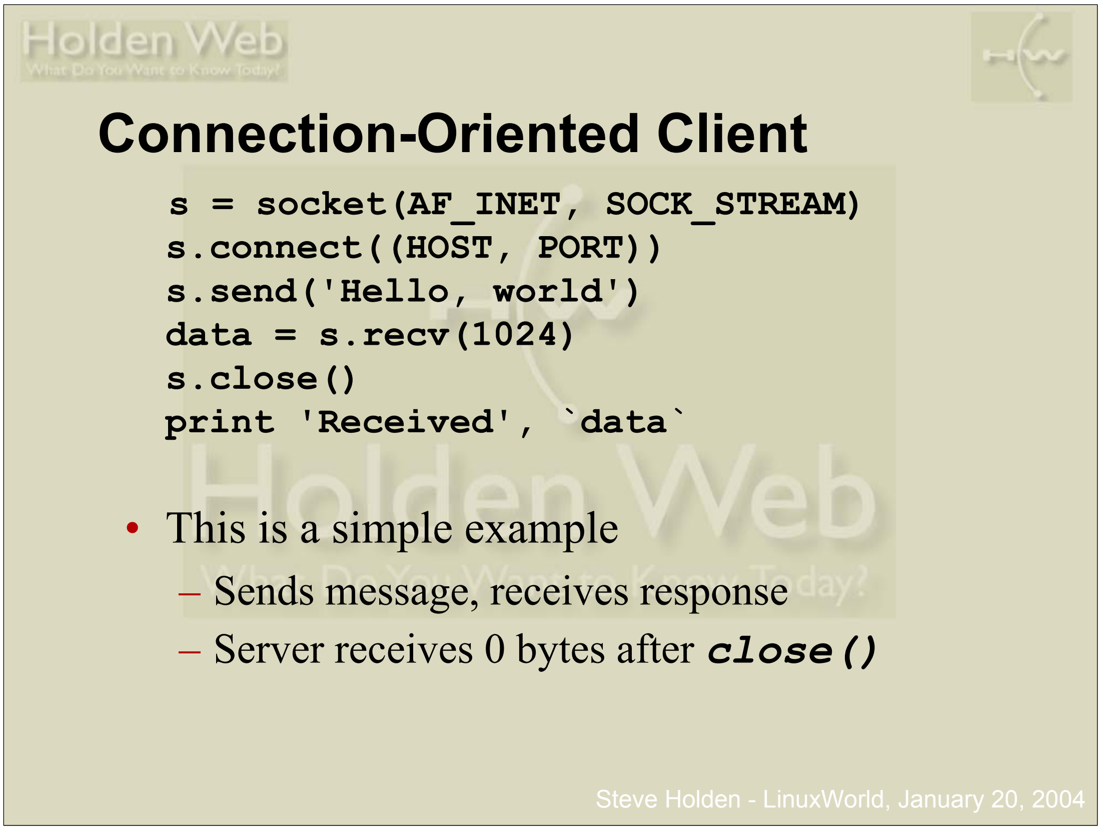
Steve Holden - LinuxWorld, January 20, 2004

Connection-oriented servers are a little more complex because the connection allows clients and servers to interact across multiple **send()** and **recv()** calls.

The server blocks in **accept()** until a server connects. The return value from ***accept()*** is a tuple consisting of a socket and the client address (which is the usual (address, port) tuple).

The server can, if it chooses, use multitasking techniques such as creating a new thread or forking a new process to allow it to handle several concurrent connections. Either solution allows the connection to be processed while the main control loop returns to execute another **accept()** and deal with the next client connection.

Since each connection generates a new server-side socket there is no conflict between the different conversations, and the server can continue to use the **listen()**ing socket to listen for incoming connections while it serves already-connected clients.



Connection-Oriented Client

```
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', `data`
```

- This is a simple example
  - Sends message, receives response
  - Server receives 0 bytes after ***`close()`***

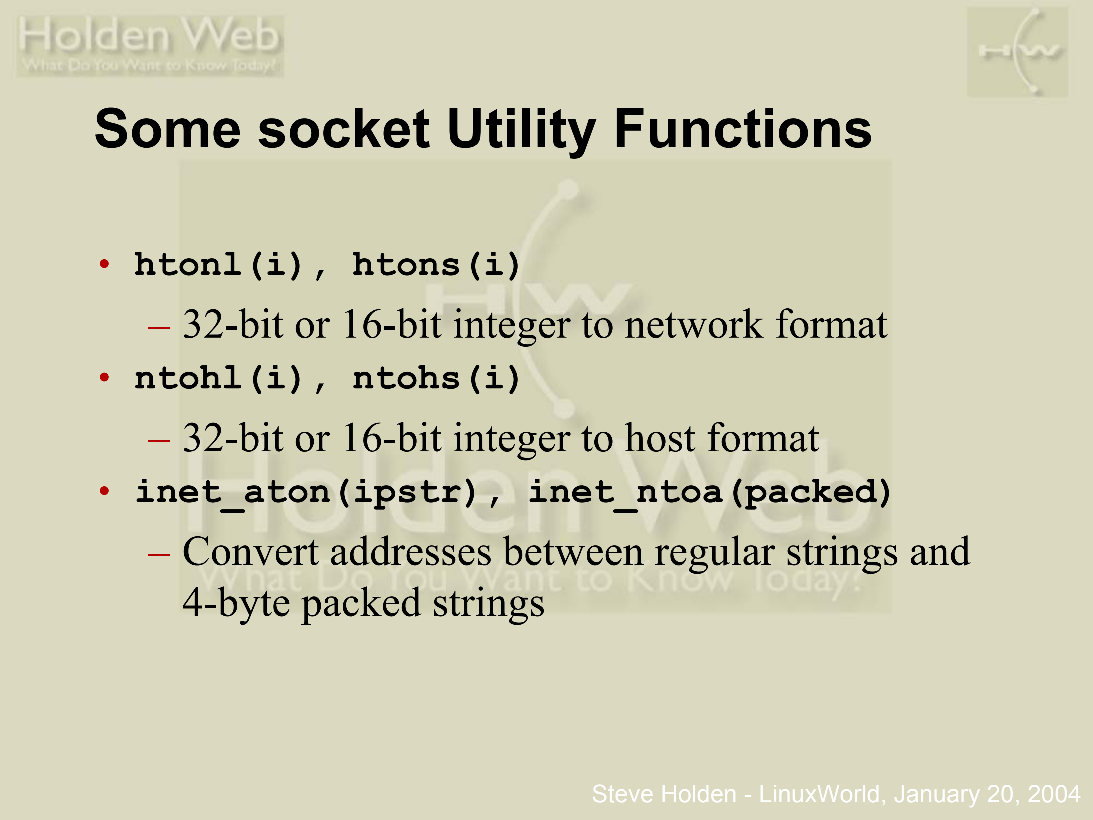
Steve Holden - LinuxWorld, January 20, 2004

This very simple client just sends a single message and receives a single response.

More typical code will send a request and use its understanding of the application protocol to determine when the response to that request has been completed.

Protocols like HTTP 1.0 use a separate connection for each request. Protocols like telnet can exchange thousands of messages before the connection is closed, and the code tends to be more complex iin that case.

Under normal circumstances a **`recv()`** call guarantees that at least one byte of data will be returned. When a program seems the empty string (zero bytes) returned from the **`recv()`** it knows that the other end has terminated the connection by calling **`close()`** on its socket.



Some socket Utility Functions

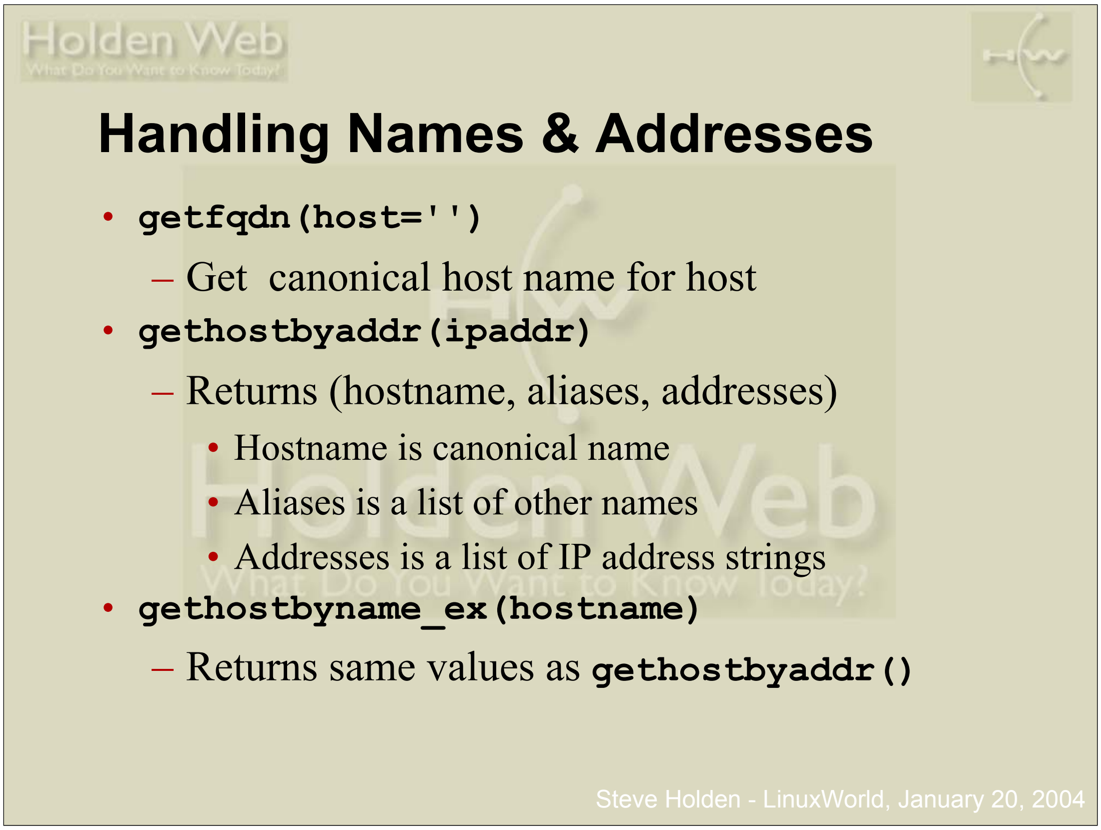
- `htonl(i), htons(i)`
  - 32-bit or 16-bit integer to network format
- `ntohl(i), ntohs(i)`
  - 32-bit or 16-bit integer to host format
- `inet_aton(ipstr), inet_ntoa(packed)`
  - Convert addresses between regular strings and 4-byte packed strings

Steve Holden - LinuxWorld, January 20, 2004

Although there is no formal presentation layer in the TCP/IP suite, certain data types are so commonly interchanged that it is useful to be able to transmit them in network-neutral format. This makes it easy for little-endian and big-endian machines to communicate with each other.

l stands for *long*, a 32-bit value, and *s* signifies a 16-bit *short*.

If you need to communicate IP addresses as a part of your application, the **`inet_*()`** functions allow you to do so efficiently.



Handling Names & Addresses

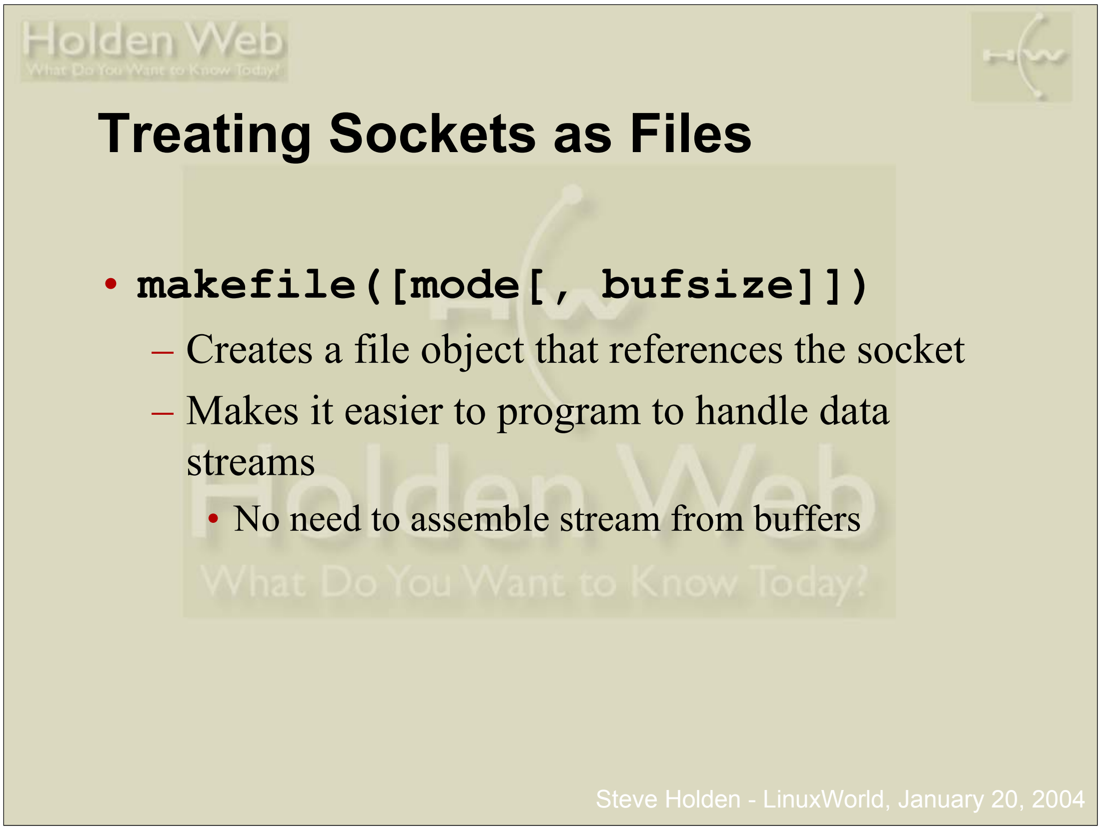
- `getfqdn(host='')`
  - Get canonical host name for host
- `gethostbyaddr(ipaddr)`
  - Returns (hostname, aliases, addresses)
    - Hostname is canonical name
    - Aliases is a list of other names
    - Addresses is a list of IP address strings
- `gethostbyname_ex(hostname)`
  - Returns same values as `gethostbyaddr()`

These utility functions are useful ways to access the domain name of the host your are running on, and other DNS services.

A typical use would be as follows:

```
>>> import socket
>>> socket.gethostbyaddr("10.0.0.10")
('prom01.holdenweb.com', [], ['10.0.0.10'])
```

This shows interactive use of the Python interpreter, which is handy for debugging purposes. If you simply enter an expression, the interpreter prints out the resulting value.



Treating Sockets as Files

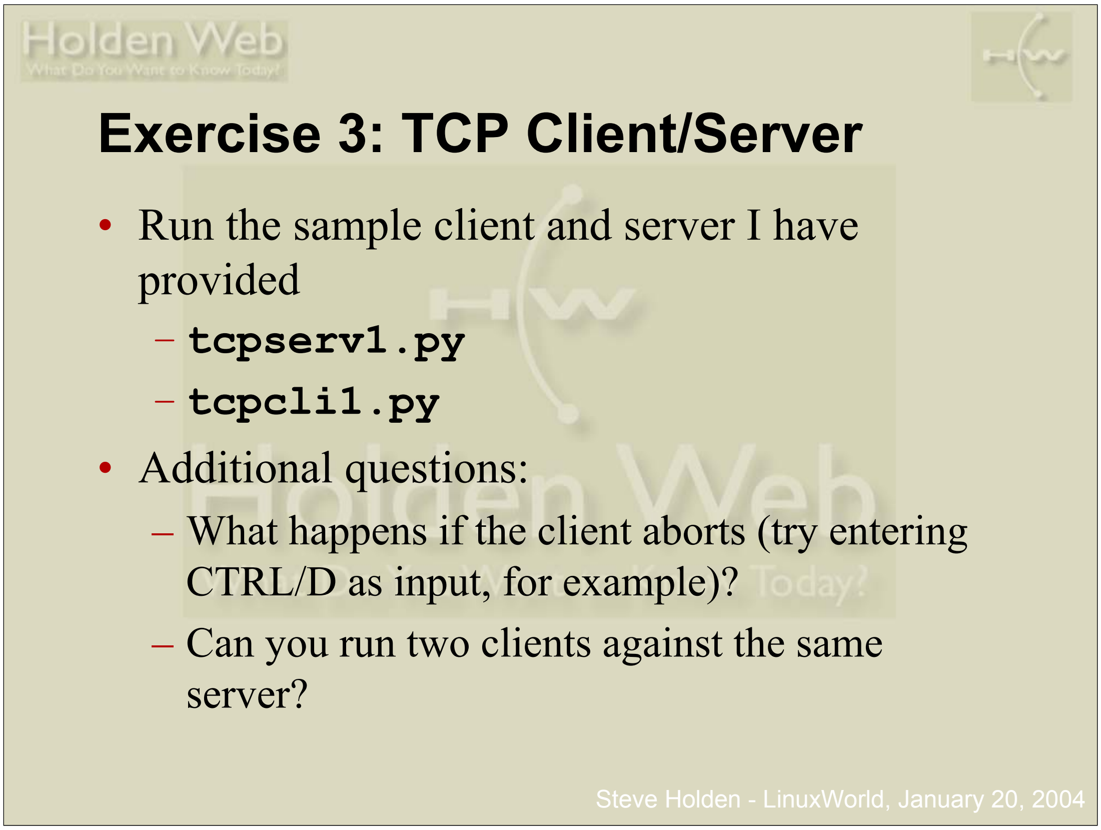
- `makefile([mode[, bufsize]])`
  - Creates a file object that references the socket
  - Makes it easier to program to handle data streams
    - No need to assemble stream from buffers

Sometimes if the protocol involves variable-length strings it's easier to use the file-based functions like **`readline()`** and **`write()`** to handle I/O.

Calling **`s.makefile()`** on a socket **`s`** yields an object sufficiently "file-like" that it allows you to write your programs in this more familiar style. Otherwise it's sometimes necessary to assemble input strings from a sequence of **`recv()`** calls, and add line terminators to the strings you pass to **`send()`**.

The socket-based file object can be closed without closing the underlying socket.

This paradigm also makes it easier to adapt existing code, written to handle files, to network applications.



Exercise 3: TCP Client/Server

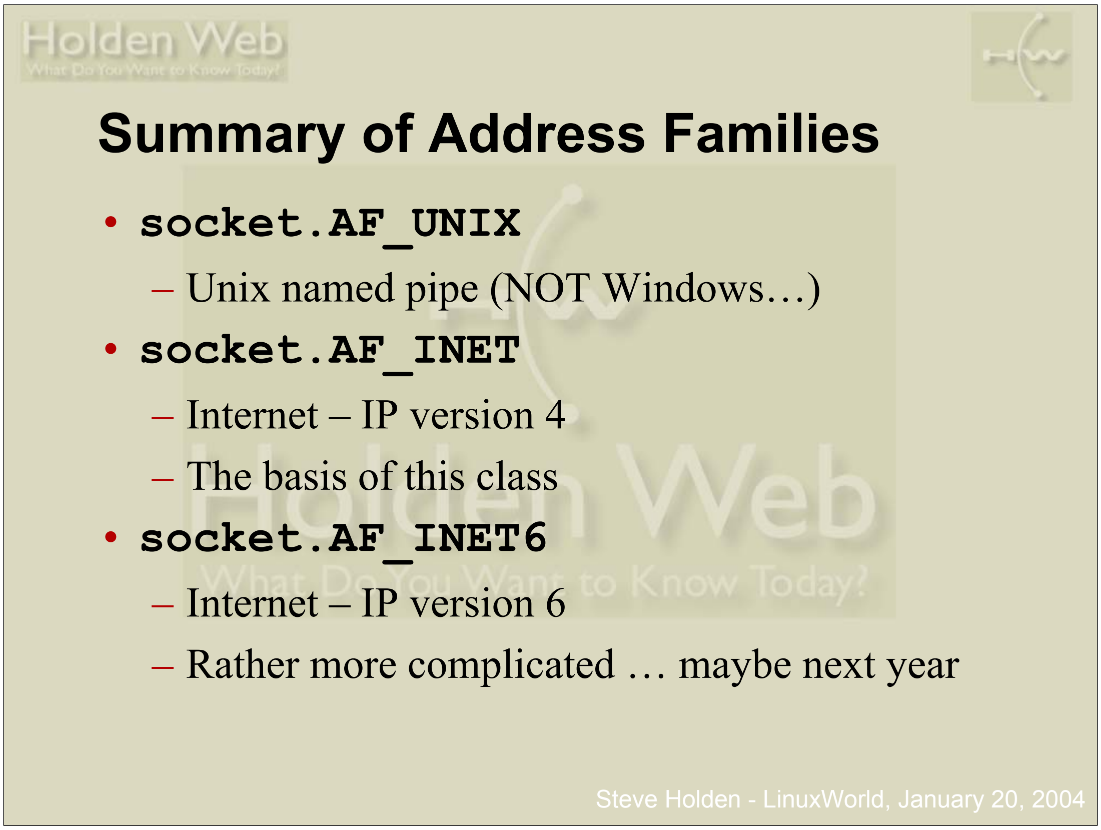
- Run the sample client and server I have provided
  - `tcpserv1.py`
  - `tcpcli1.py`
- Additional questions:
  - What happens if the client aborts (try entering CTRL/D as input, for example)?
  - Can you run two clients against the same server?

Steve Holden - LinuxWorld, January 20, 2004

The idea of this exercise is to show two programs: one acts as a server, and each time it receives a connection it repeatedly reads strings from the client, returning the same string in upper case.

The second simply acts as a client, reading strings from the user, sending them to the server and echoing the server's output back until an empty line is entered, at which time it closes the connection to the server and terminates.

This is a somewhat fragile set-up for a number of reasons. The first is that if a client terminates without properly closing its server connection the server may not detect this situation, and will continue to wait for input. The effect of this fault is magnified by the server's inability to handle multiple connections – it won't accept a second connection until the first one terminates. We'll see how to get around this problem shortly.



Holden Web
What Do You Want to Know Today?

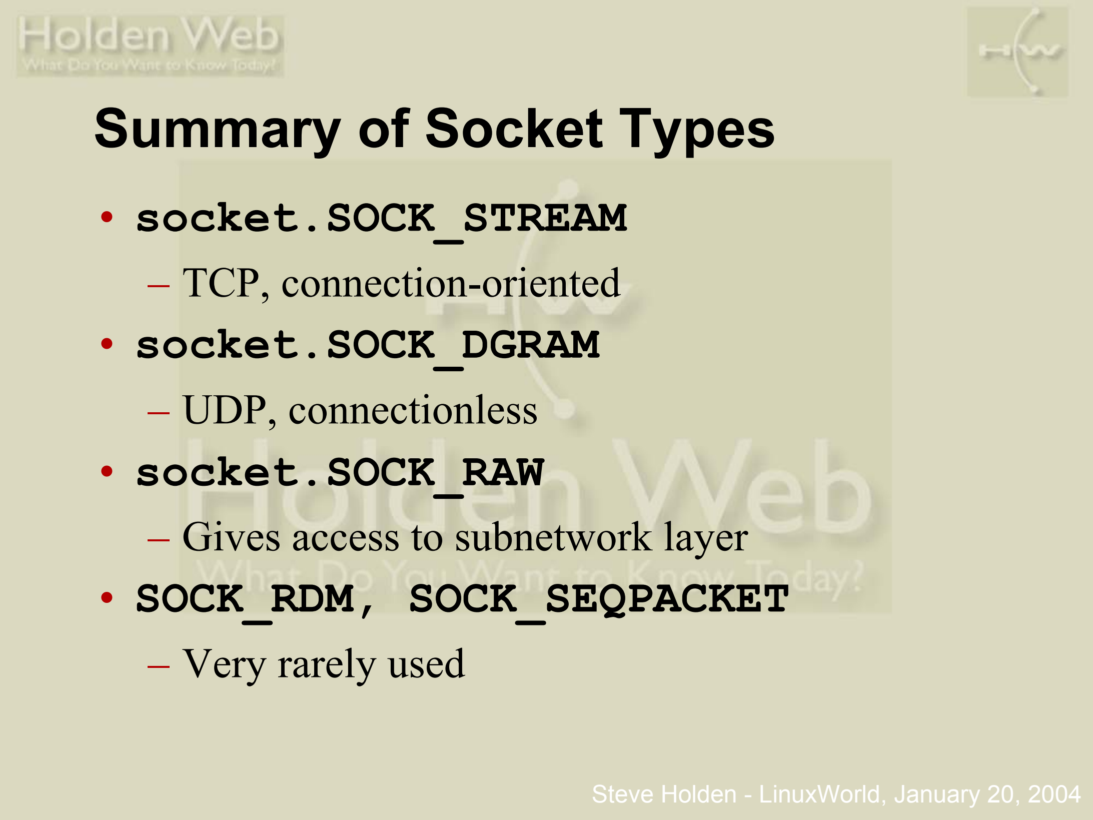
Summary of Address Families

- `socket.AF_UNIX`
  - Unix named pipe (NOT Windows…)
- `socket.AF_INET`
  - Internet – IP version 4
  - The basis of this class
- `socket.AF_INET6`
  - Internet – IP version 6
  - Rather more complicated … maybe next year

Steve Holden - LinuxWorld, January 20, 2004

Mostly we are concerned with IP v4 sockets and the related services, although IPv6 is coming, and Python 2.3 is expanding socket support to include it.

Unix named pipes are available if you need them, and work in more or less the same ways as the IP version 4 sockets we'll be using in the class.

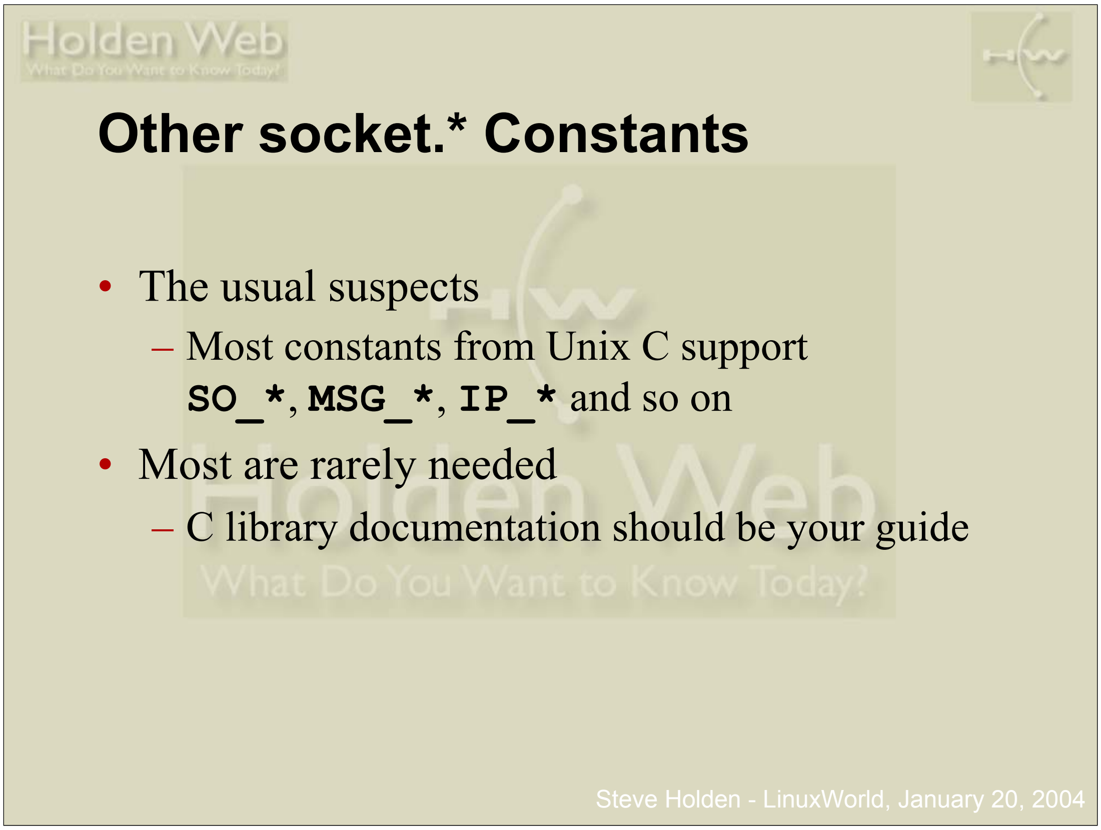


Summary of Socket Types

- `socket.SOCK_STREAM`
  - TCP, connection-oriented
- `socket.SOCK_DGRAM`
  - UDP, connectionless
- `socket.SOCK_RAW`
  - Gives access to subnetwork layer
- `SOCK_RDM, SOCK_SEQPACKET`
  - Very rarely used

Most of the services we are interested in will be TCP-based, and therefore use SOCK_STREAM sockets. We do look briefly at SOCK_DGRAM sockets. SOCK_RAW sockets have been the basis of (for example) **ping** programs to generate and handle ICMP traffic, but this is beyond the scope of this presentation.

Still. It's always useful to know that you can get access to the network hardware if you need it – most times you don't, but typically when you need such access you *really* need it.



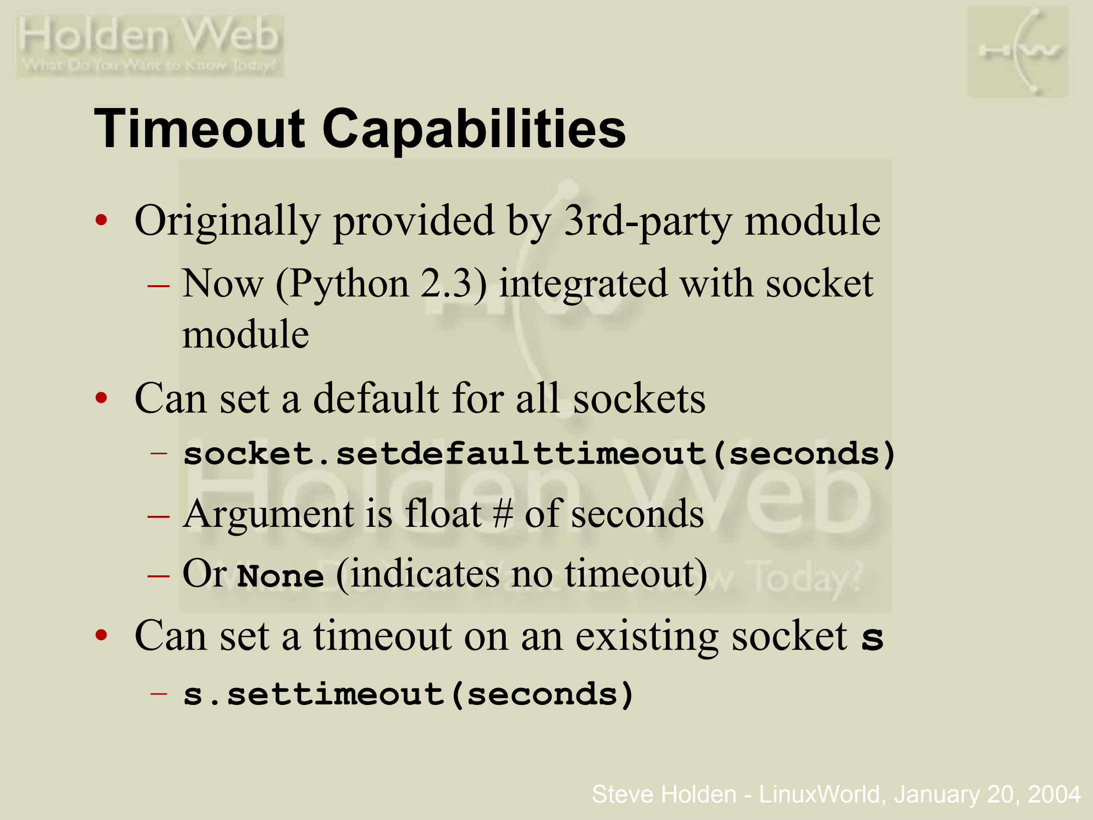
Other socket.* Constants

- The usual suspects
  - Most constants from Unix C support `SO_*`, `MSG_*`, `IP_*` and so on
- Most are rarely needed
  - C library documentation should be your guide

If you are an experienced C network programmer you will be familiar with the socket library provided to programmers in that language, which has been widely ported to various OS platforms.

Although most of the features of these libraries are available from the socket modue, it is surprising how much you can do without using more than the basics outlined in this tutorial. We are focusing on the features you need to get your network applications up and running – the rest can follow later.

There is a wealth of Python code already in the public domain that you can use as examples.



Timeout Capabilities

- Originally provided by 3rd-party module
  - Now (Python 2.3) integrated with socket module
- Can set a default for all sockets
  - `socket.setdefaulttimeout(seconds)`
  - Argument is float # of seconds
  - Or `None` (indicates no timeout)
- Can set a timeout on an existing socket `s`
  - `s.settimeout(seconds)`

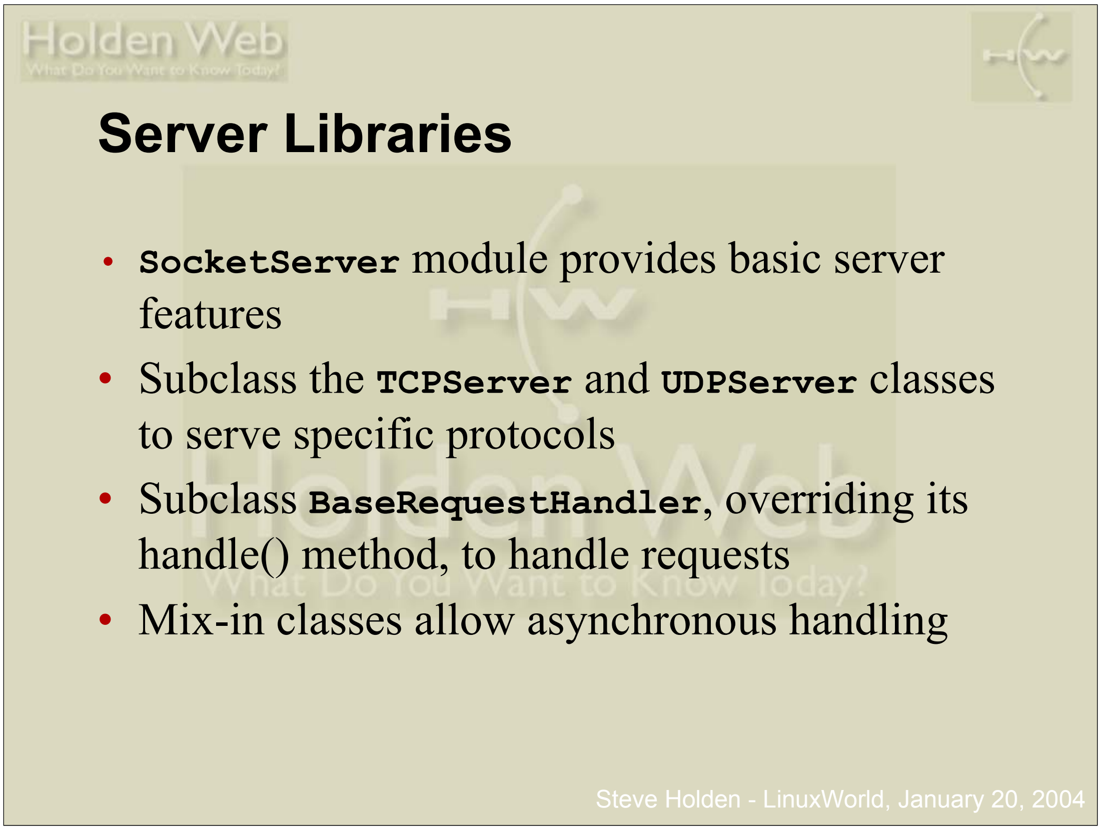
A big problem with network programming is: what do you do when the expected responses aren't received?

In the case of TCP connection attempts, if no reply is received to the initial connection attempt it isn't unusual for the adaptive timeout to keep retrying at exponentially larger intervals, with the result that no exception occurs for over an hour. UDP sockets will hang indefinitely if they are awaiting a message that somehow gets lost, or is never transmitted.

Timeouts are therefore a valuable way to ensure that failures are detected in a timely manner, and I encourage you to use this feature, which has been added to the standard library in the most recent release. If you are using an older release then you should download the **`timeoutsocket`** module from

`http://www.timo-tasi.org/python/timeoutsocket.py`

Having made you aware of the problems we are going to ignore them for the rest of the class – timeouts, although invaluable in practice, can complicate the code to the extent that useful programs take more than one slide ☺



Server Libraries

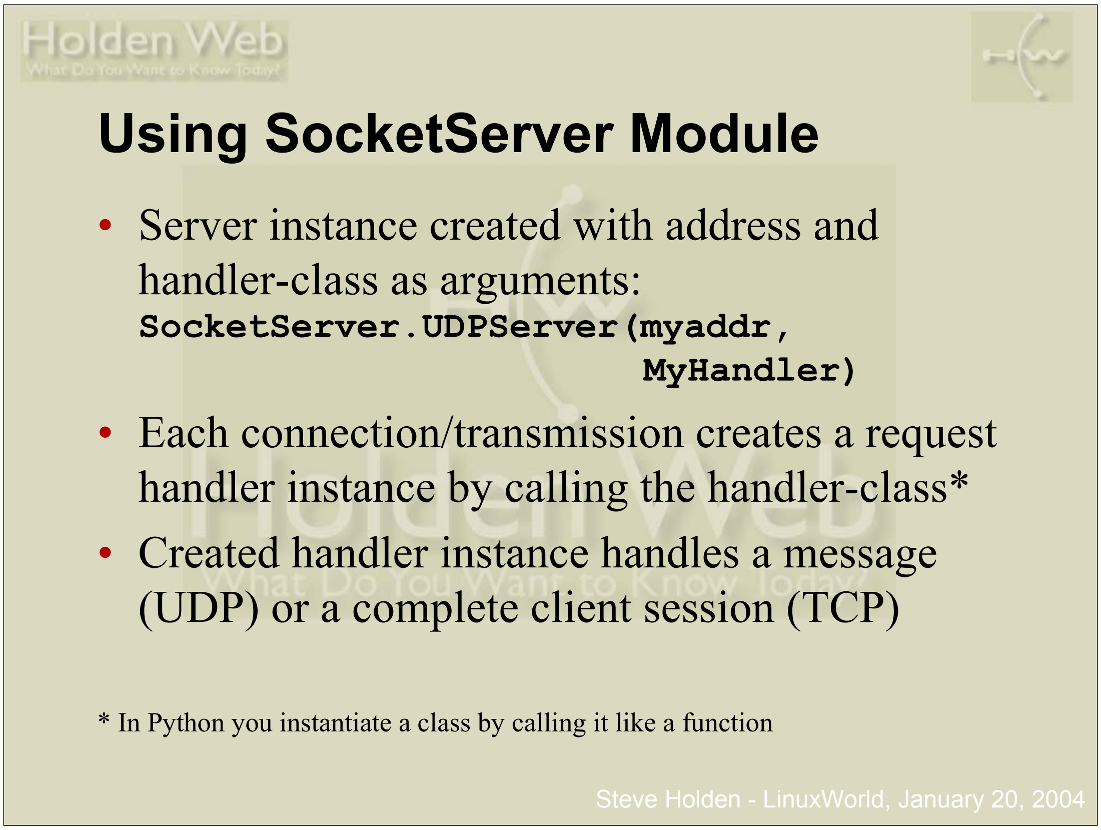
- **`SocketServer`** module provides basic server features
- Subclass the **`TCPServer`** and **`UDPServer`** classes to serve specific protocols
- Subclass **`BaseRequestHandler`**, overriding its handle() method, to handle requests
- Mix-in classes allow asynchronous handling

Steve Holden - LinuxWorld, January 20, 2004

The **`SocketServer`** module is the basis for (among other things) a family of HTTP servers which are also a part of the standard Python library. It offers a framework that allows you to write some quite sophisticated servers, and is very useful for rapid network experimentation. You should be aware, though, that robustness and throughput are both somewhat below professional servers, so you should not make this the basis of production servers unless you are fairly certain the load will be light.

Having said that, the ability to build an experimental server with a (very) few lines of code is invaluable, so **`SocketServer`** should definitely be a part of your programming vocabulary if you do a lot of protocol development or similar work. Python's conciseness and readability are a real asset here.

Python's ability to handle multiple inheritance is also useful -- the module lets you add the ability to handle each request in a separate thread or a separate process by adding so-called *mixin* classes to your server's base classes. This is particularly useful for connection-oriented servers, which would otherwise stall additional clients until an existing session was completely terminated, as we saw in the earlier demonstration.



Holden Web

What Do You Want to Know Today?

Using SocketServer Module

- Server instance created with address and handler-class as arguments:
  `SocketServer.UDPServer(myaddr,`
  `MyHandler)`
- Each connection/transmission creates a request handler instance by calling the handler-class*
- Created handler instance handles a message (UDP) or a complete client session (TCP)

* In Python you instantiate a class by calling it like a function

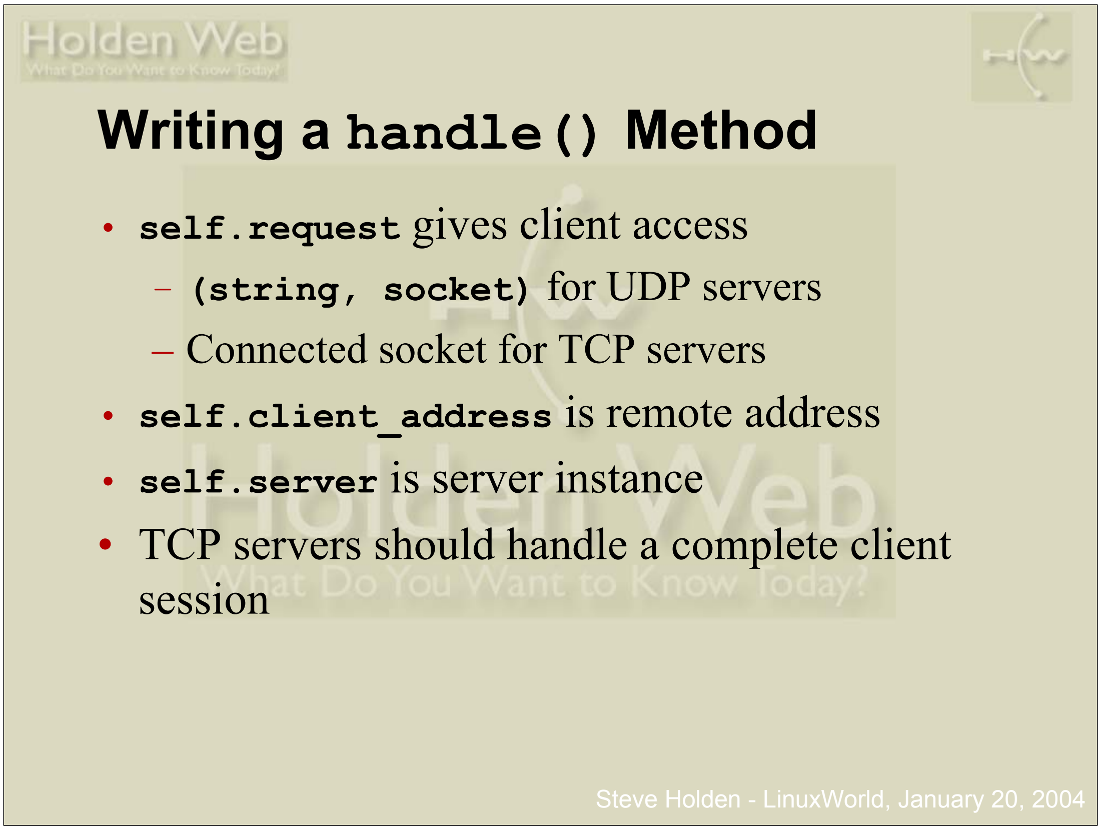
Steve Holden - LinuxWorld, January 20, 2004

The **`SocketServer`** framework is quite easy to use and yet, as the HTTP server modules we examine later will show, it can be the basis of some very powerful functionality.

The UDP server code treats each incoming packet as a standalone request, which is pretty much what you expect with UDP – there's no such thing as a connection, and so all the request's data has to arrive at once.

The address you provide is normally a two-element tuple: the first element is a string containing the address you want to bind to (the empty string means "bind to all addresses") and the port number to listen on. This approach is common to almost all Python socket code.

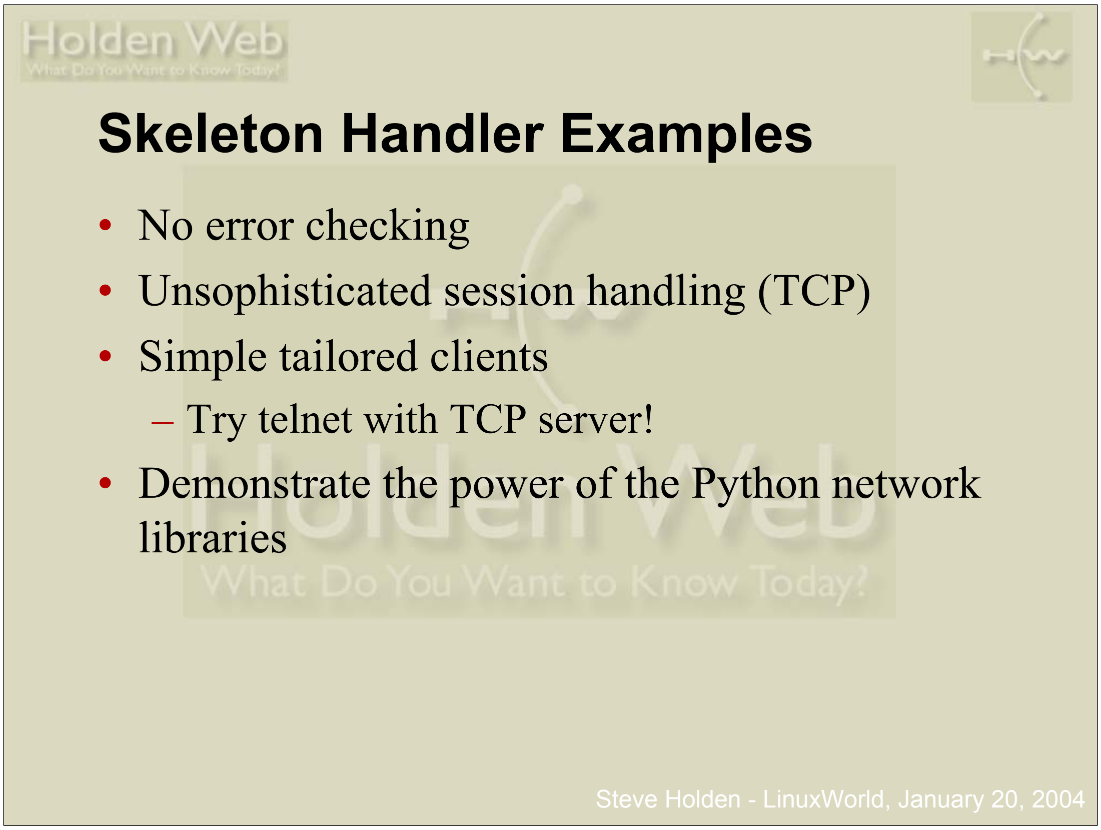
For the handler you provide a class, and the server framework creates a brand new instance of that class each time a new request comes in. [Technically you provide a *callable*, but when you call it the result must be an object that possesses all required methods and attributes: in practice it's easiest to provide a subclass of the **`SocketServer.BaseRequestHandler`** class provided with the library]. Usually all you have to define in your subclass is your own **`handle()`** method.



Writing a `handle()` Method

- `self.request` gives client access
  - `(string, socket)` for UDP servers
  - Connected socket for TCP servers
- `self.client_address` is remote address
- `self.server` is server instance
- TCP servers should handle a complete client session

The key to **`SocketServer`** programs is to get the **`handle()`** method right. It isn't difficult, as the framework is fairly intelligent and does most of what needs to be done.



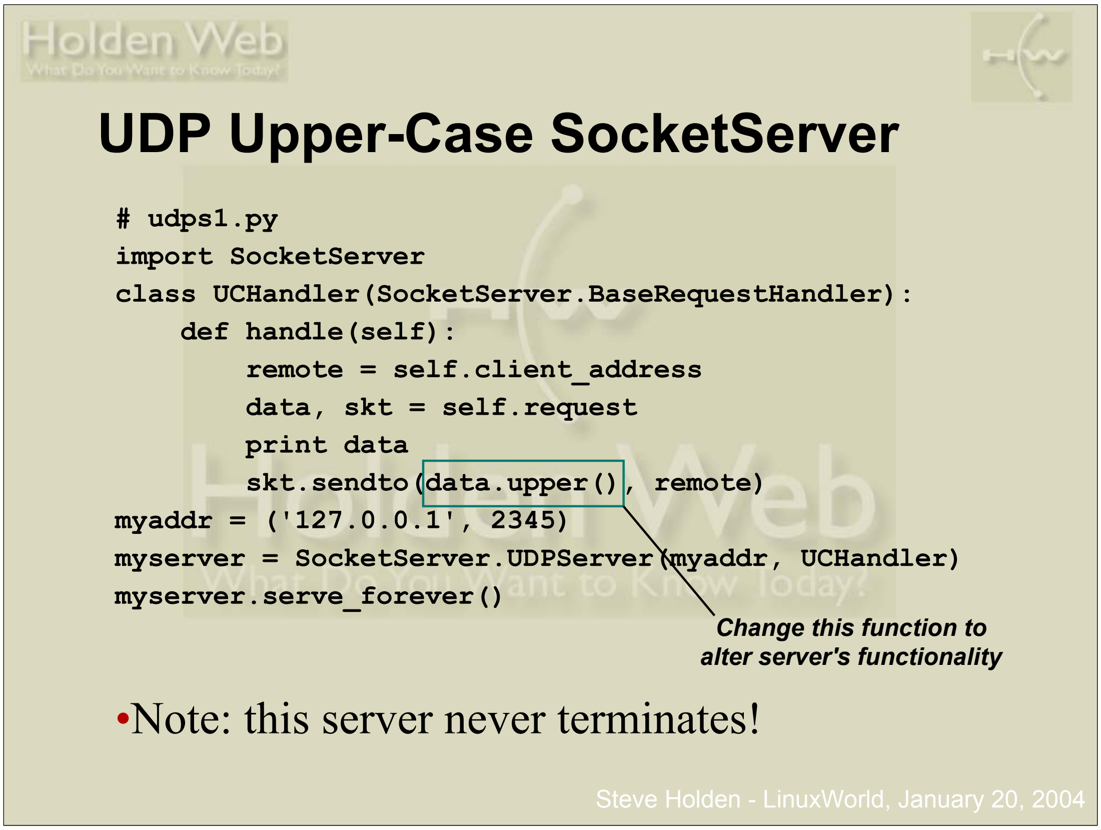
Skeleton Handler Examples

- No error checking
- Unsophisticated session handling (TCP)
- Simple tailored clients
  - Try telnet with TCP server!
- Demonstrate the power of the Python network libraries

Steve Holden - LinuxWorld, January 20, 2004

To give an immediate flavor of the utility of this library, we present an example of both a connectionless and a connection-oriented server. The service is simply that of translating the case of text.

As the slides point out, though, the code is easily adapted to providing other functionality – the server framework gives you pretty much everything you need.



UDP Upper-Case SocketServer

```
# udps1.py
import SocketServer
class UCHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        remote = self.client_address
        data, skt = self.request
        print data
        skt.sendto(data.upper(), remote)
myaddr = ('127.0.0.1', 2345)
myserver = SocketServer.UDPServer(myaddr, UCHandler)
myserver.serve_forever()
```

Change this function to alter server's functionality

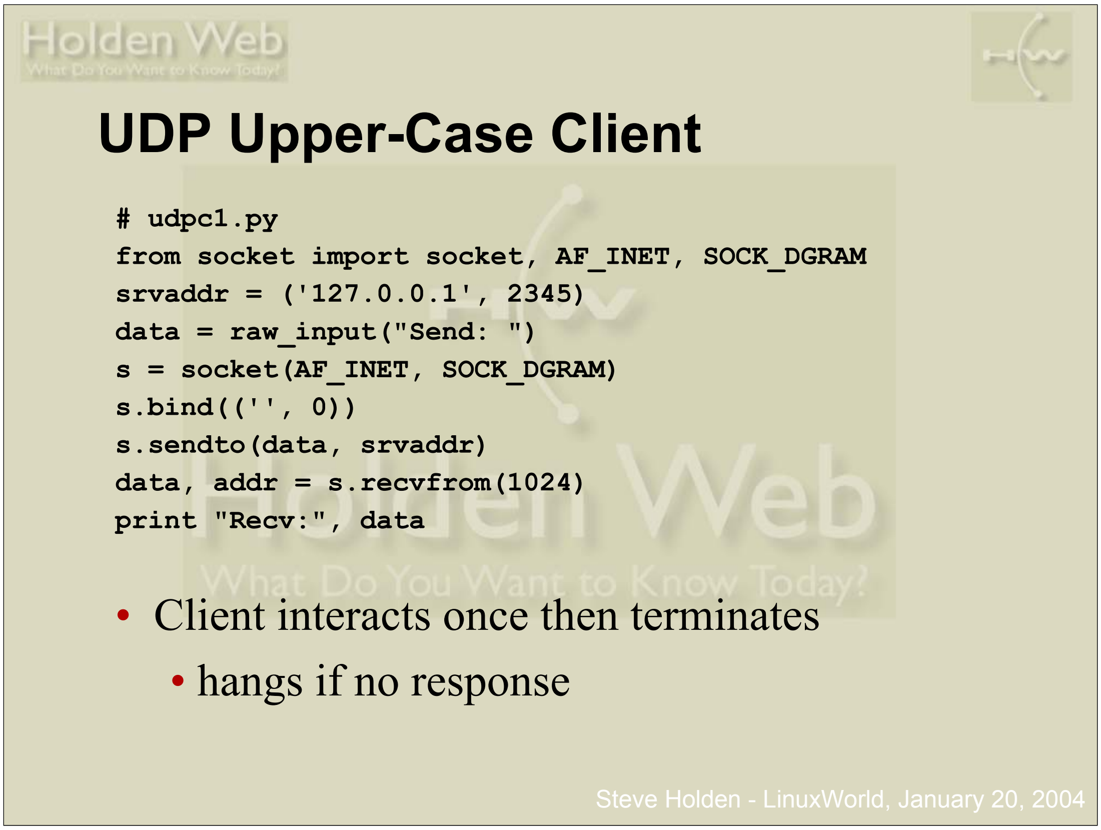
- Note: this server never terminates!

Steve Holden - LinuxWorld, January 20, 2004

This server example shows the basic way to reply to a UDP client.

UCHandler is an upper-case handler class. Its **handle()** method gets the data and the return address from the instance's **request** attribute. It replies to the client using the socket's **sendto()** method.

Python strings are objects with their own methods. The **upper()** method of a string simply returns the upper-case conversion of the string. It would be easy to replace the **data.upper()** expression with a more complicate function of **data** to implement different handler functionality.



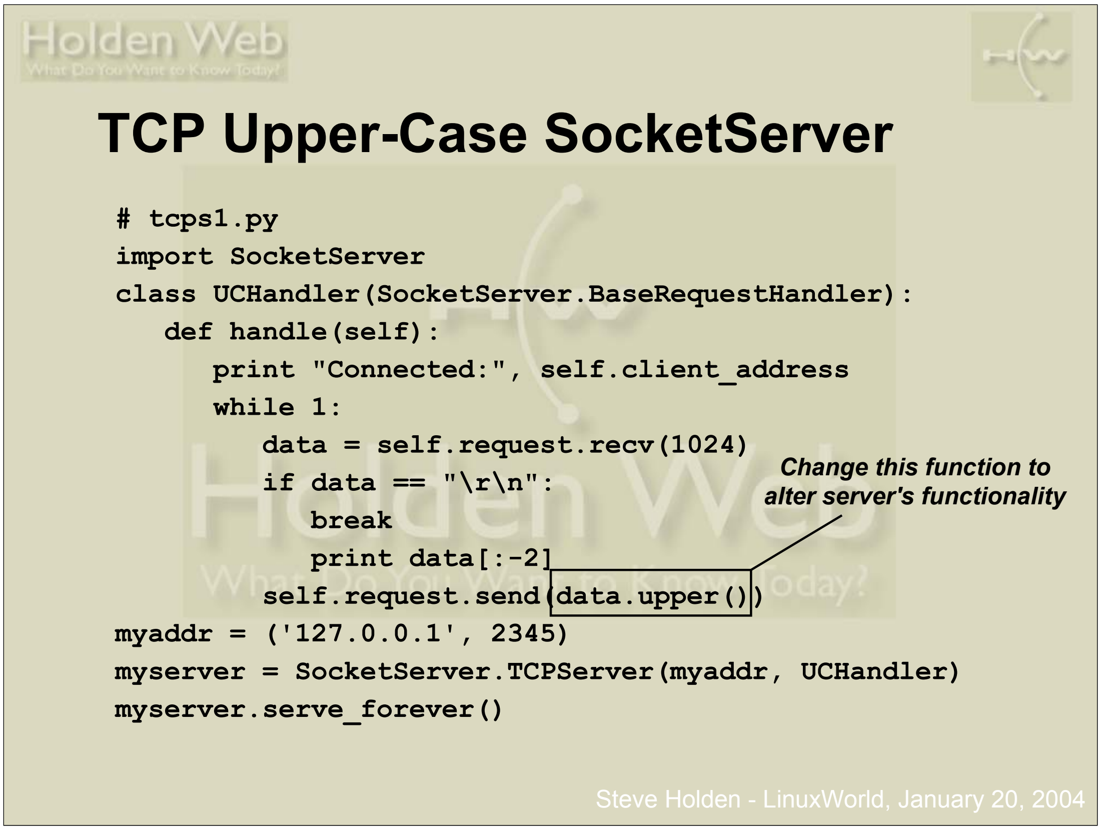
UDP Upper-Case Client

```
# udpc1.py
from socket import socket, AF_INET, SOCK_DGRAM
srvaddr = ('127.0.0.1', 2345)
data = raw_input("Send: ")
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('', 0))
s.sendto(data, srvaddr)
data, addr = s.recvfrom(1024)
print "Recv:", data
```

- Client interacts once then terminates
  - hangs if no response

Since we can't make connections using UDP, each input generates a datagram to the server, and the client then waits for the server's return datagram.

In practice, of course, it would be rather better to set a timeout on the receive so that the program doesn't hang forever if something goes wrong.



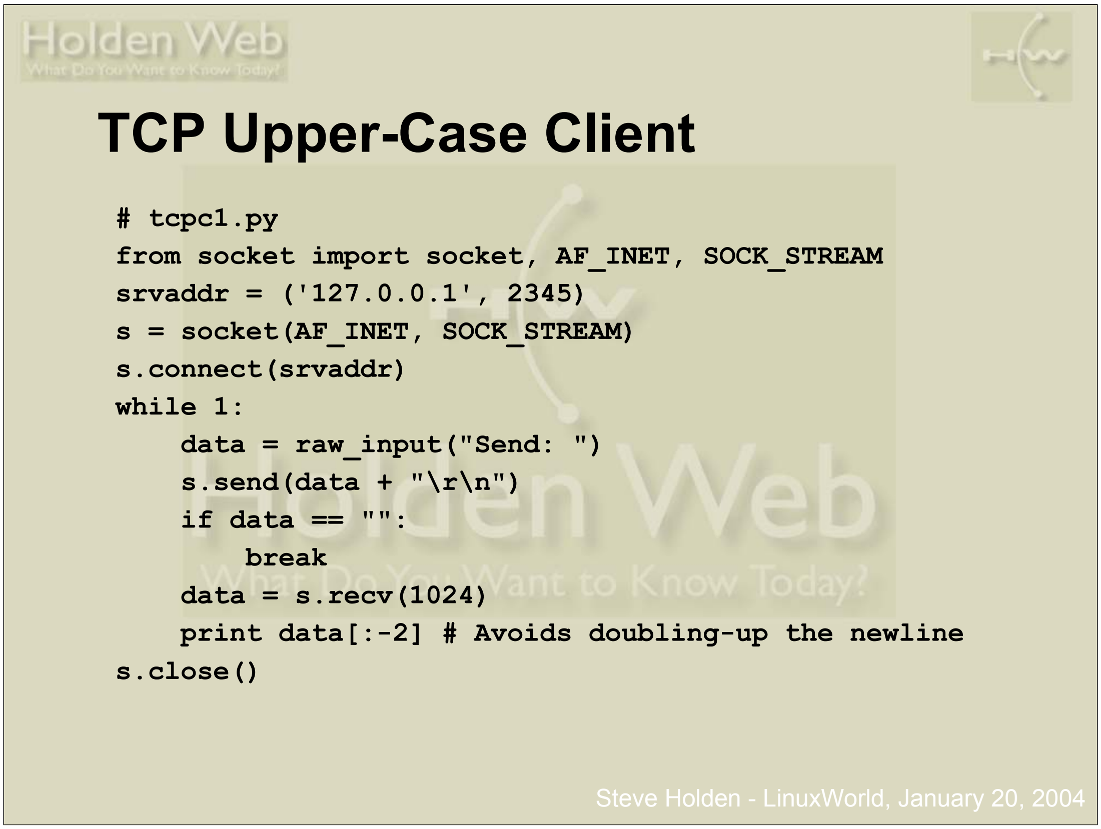
TCP Upper-Case SocketServer

```
# tcps1.py
import SocketServer
class UCHandler(SocketServer.BaseRequestHandler):
   def handle(self):
      print "Connected:", self.client_address
      while 1:
         data = self.request.recv(1024)
         if data == "\r\n":
            break
         print data[:-2]
         self.request.send(data.upper())
myaddr = ('127.0.0.1', 2345)
myserver = SocketServer.TCPServer(myaddr, UCHandler)
myserver.serve_forever()
```

Change this function to alter server's functionality

The TCP server structure is somewhat different from the UDP server because the **handle()** method has to handle a complete connection from beginning to end, potentially involving a number of interations between the client and the server. So it implements a loop, which requires a terminating condition.

This server's **handle()** method terminates when it receives a blank line from the client. It would be more usual to terminate when a zero-length input was received, as this is the conventional indication that the client has closed the socket from its end. However that would need a client that closes the socket, so the empty line is rather easier to deal with on the client side.

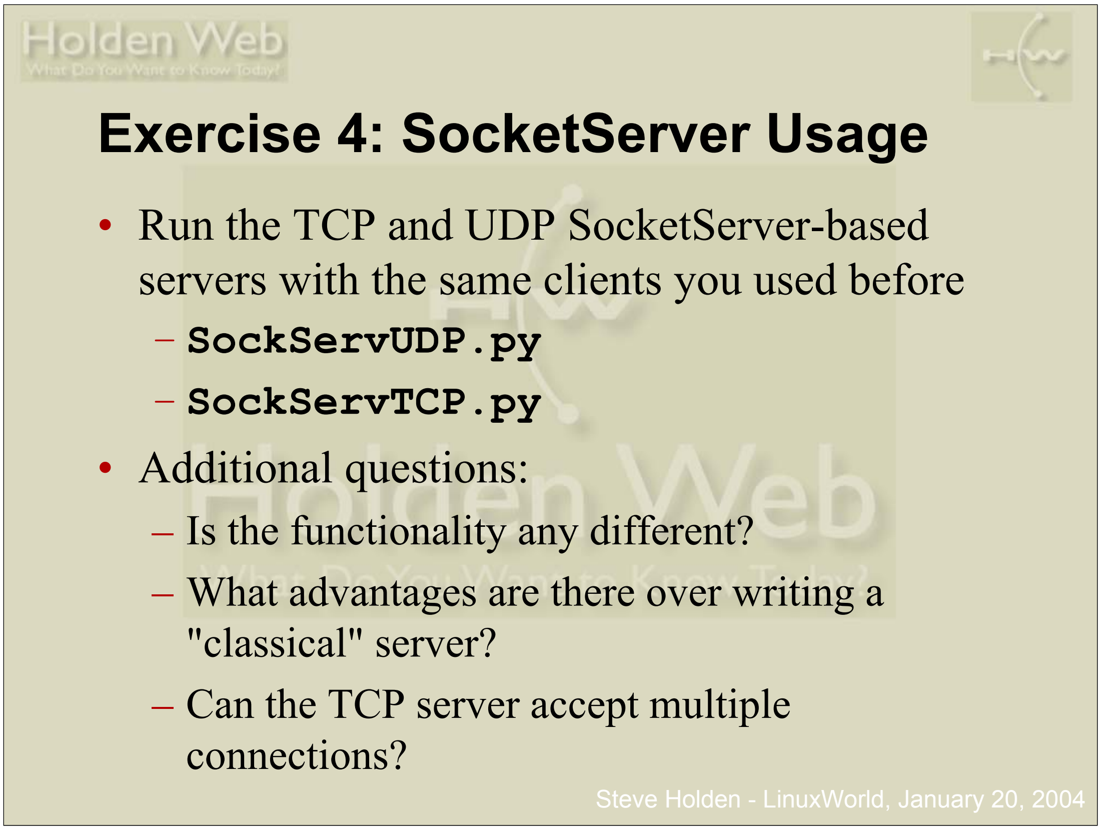


TCP Upper-Case Client

```
# tcpc1.py
from socket import socket, AF_INET, SOCK_STREAM
srvaddr = ('127.0.0.1', 2345)
s = socket(AF_INET, SOCK_STREAM)
s.connect(srvaddr)
while 1:
    data = raw_input("Send: ")
    s.send(data + "\r\n")
    if data == "":
        break
    data = s.recv(1024)
    print data[:-2] # Avoids doubling-up the newline
s.close()
```

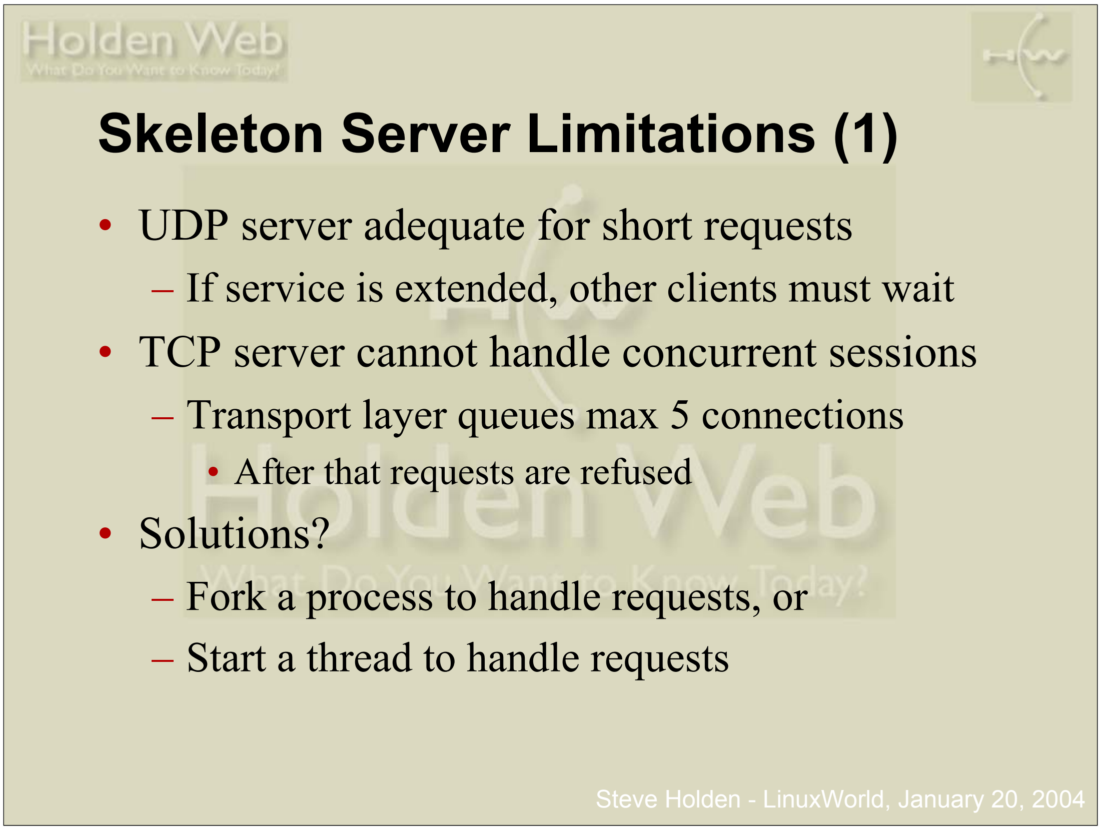
As you can see, the client connects to the server and then repeatedly reads data from the user. In line with the server's requirements, the client sends the blank line that terminates the input before closing the socket.

The Python socket is supposed to be closed automatically when the program ends, but it is akways safe to explicitly close the socket, and it's better practice than relying on something that may or may not happen.



Exercise 4: SocketServer Usage

- Run the TCP and UDP SocketServer-based servers with the same clients you used before
  - `SockServUDP.py`
  - `SockServTCP.py`
- Additional questions:
  - Is the functionality any different?
  - What advantages are there over writing a "classical" server?
  - Can the TCP server accept multiple connections?

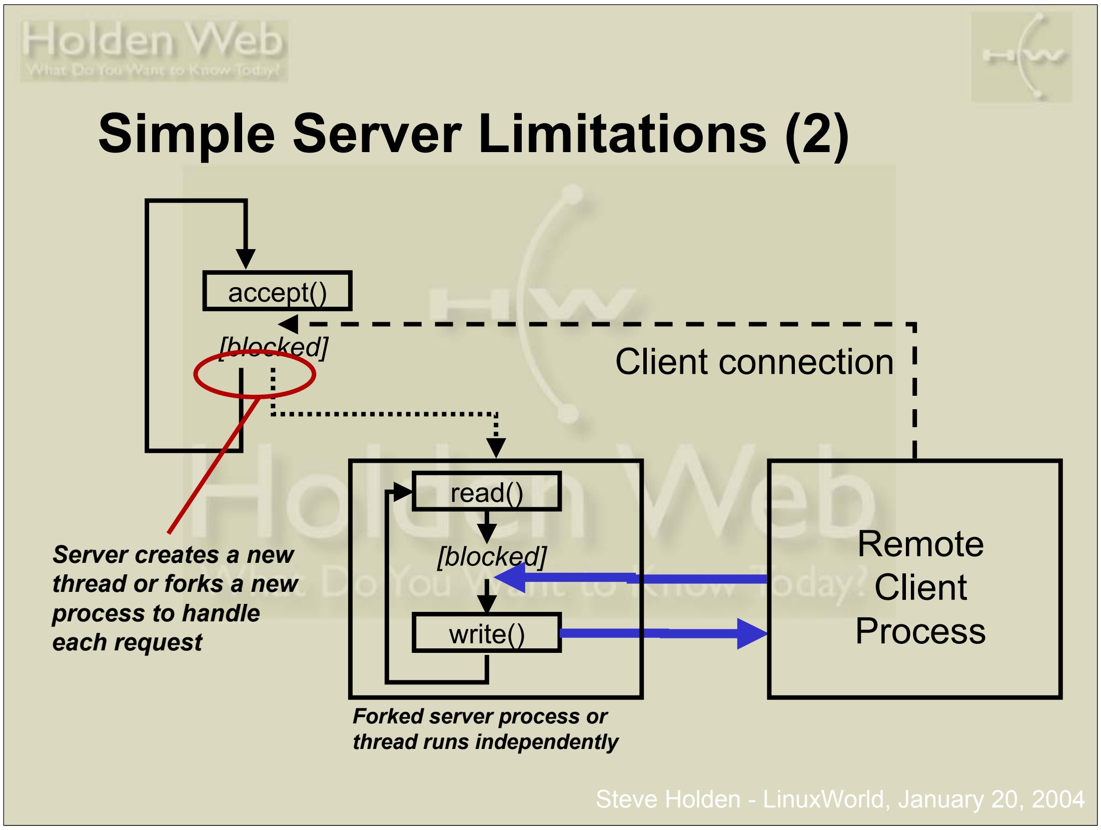


Skeleton Server Limitations (1)

- UDP server adequate for short requests
  - If service is extended, other clients must wait
- TCP server cannot handle concurrent sessions
  - Transport layer queues max 5 connections
    - After that requests are refused
- Solutions?
  - Fork a process to handle requests, or
  - Start a thread to handle requests

Clearly a server will be able to handle more work if processing one request doesn't hold others up. The basic **`SocketServer`** server, however, only handles a single request at a time.

Fortunately the solution is ready to hand – the **`SocketServer`** module's author has already thought about this problem, and provided mixin classes to give forking and threading as alternative server behaviors.



Holden Web
What Do You Want to Know Today?

Simple Server Limitations (2)

accept()

[blocked]

Client connection

read()

[blocked]

write()

Remote Client Process

Server creates a new thread or forks a new process to handle each request

Forked server process or thread runs independently

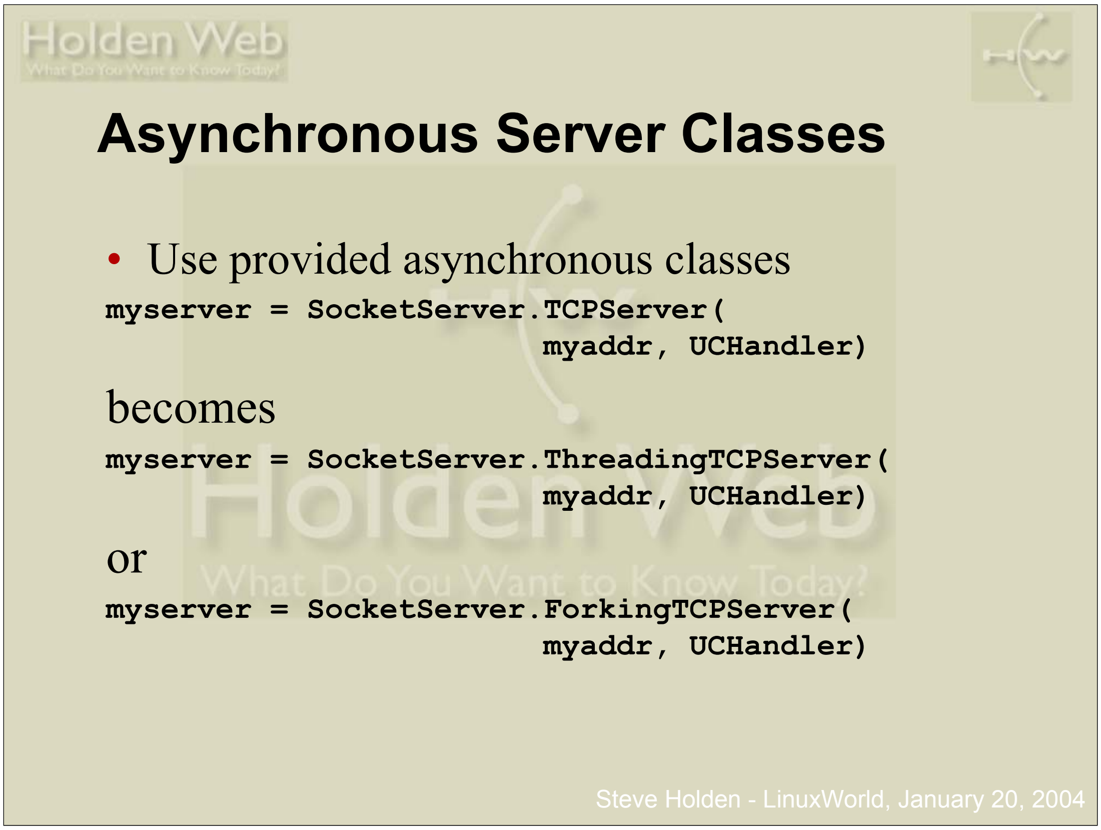
Steve Holden - LinuxWorld, January 20, 2004

This diagram shows what needs to happen to allow parallel handling of concurrent requests.

In the case of a forking server process, the forked child handles the request and the parent loops around to **`accept()`** another connection.

In the case of a threading server the two threads coexist in the same process, and the process shares its CPU allocation between the new thread that processes the request and the original thread, which loops around to **`accept()`** another connection.

The essential point is that the server no longer has to handle one connection completely before it can handle the next request. This isn't so important for UDP servers, where the request and response tend to be short and sweet. In the TCP world, however, a "request" is actually a connection that can be exceedingly long-lived – think of a Telnet or ssh session, for example.



Asynchronous Server Classes

- Use provided asynchronous classes

```
myserver = SocketServer.TCPServer(
                        myaddr, UCHandler)
```

becomes

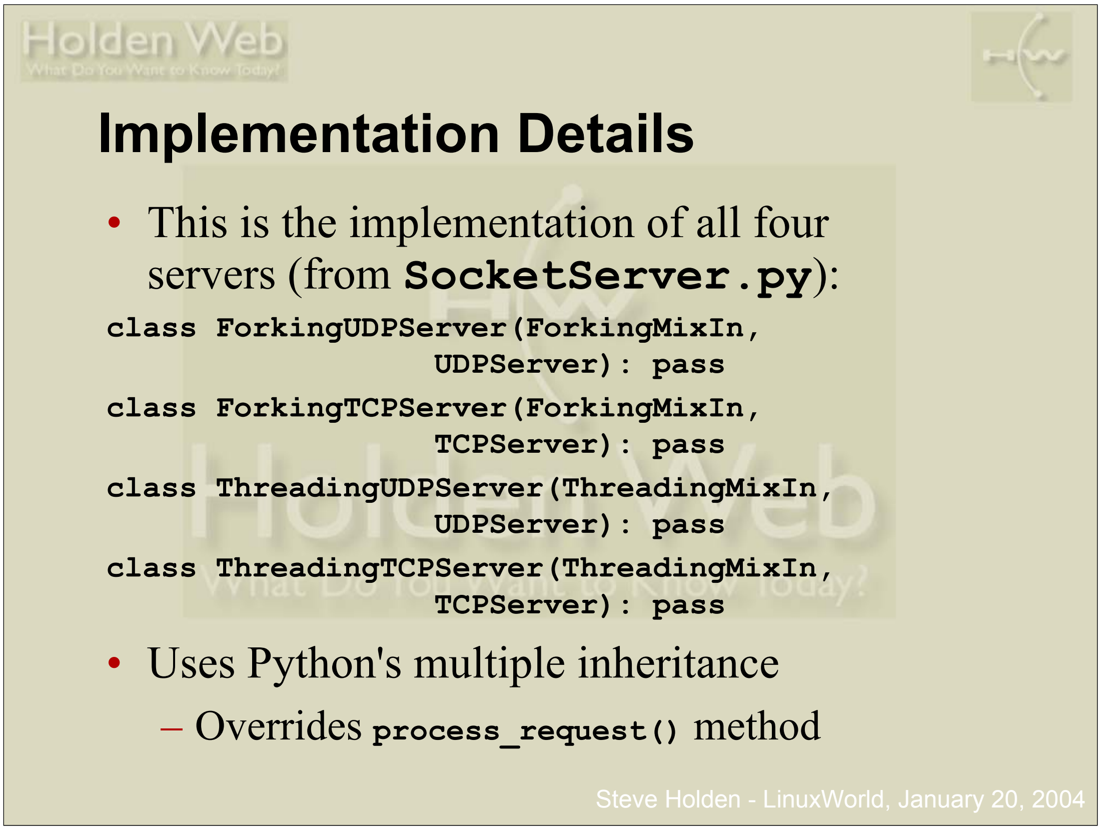
```
myserver = SocketServer.ThreadingTCPServer(
                        myaddr, UCHandler)
```

or

```
myserver = SocketServer.ForkingTCPServer(
                        myaddr, UCHandler)
```

This is a very powerful way to make your servers asynchronous. In the forking and threading variants, service for one request becomes independent of the main server loop because the handler gets its own process (in the forking model) or thread (in the threading model). Once the server has started the request handler it is free to loop round again and accept another request almost immediately.

A measure of Python's power is how easily it is possible to adapt the standard servers – the ThreadingTCPServer and ThreadingTCPServer classes are actually quite simple, as you will see on the next slide.



Implementation Details

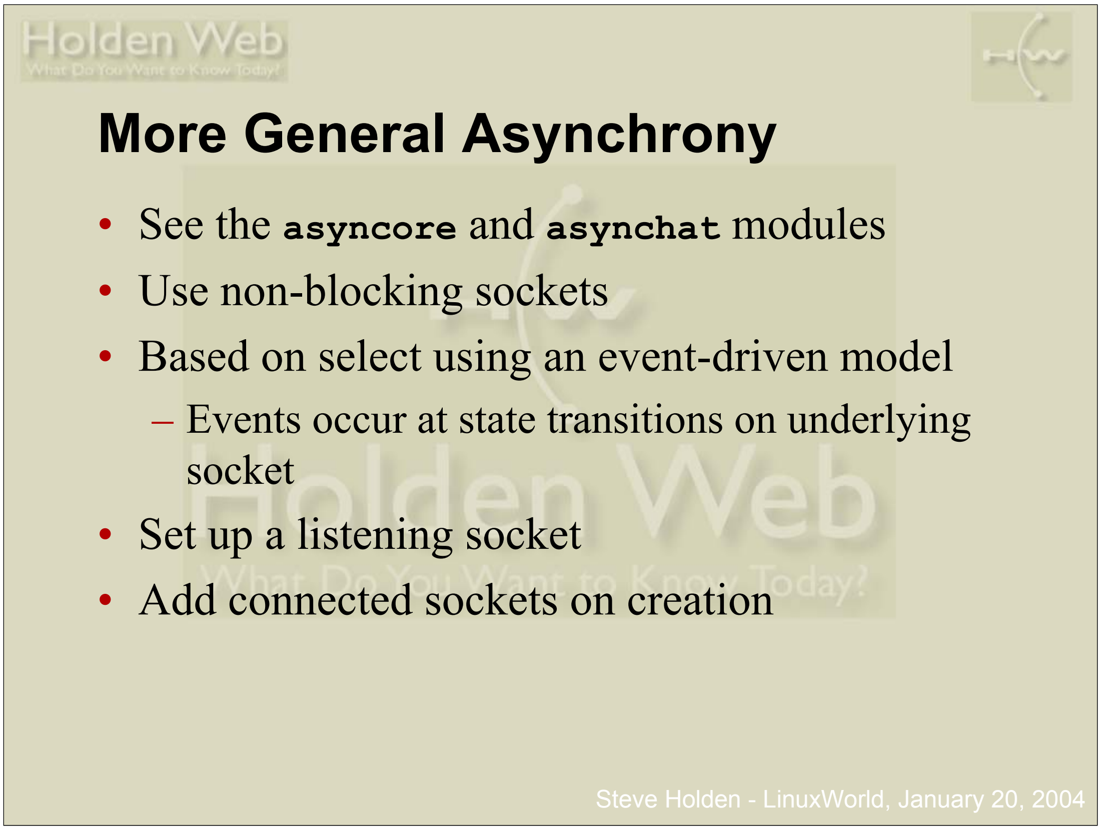
- This is the implementation of all four servers (from **SocketServer.py**):

```
class ForkingUDPServer(ForkingMixIn,
                       UDPServer): pass
class ForkingTCPServer(ForkingMixIn,
                       TCPServer): pass
class ThreadingUDPServer(ThreadingMixIn,
                       UDPServer): pass
class ThreadingTCPServer(ThreadingMixIn,
                       TCPServer): pass
```

- Uses Python's multiple inheritance
  - Overrides **process_request()** method

You might think the mix-in classes would be complex, but this is not the case. Forking adds maybe thirty lines of code, and threading around twenty lines. The mix-in classes override the **process_request()** method of their client classes, and so their definition is used in preference to the definition in the server class (**TCPServer** or **UDPServer**). This inheritance model makes it relatively easy to add orthogonal functionality to classes.

The forking variants do not work on Windows platforms because the **os.fork()** primitive is unavailable. The threading variants work on all common platforms.



More General Asynchrony

- See the **`asyncore`** and **`asynchat`** modules
- Use non-blocking sockets
- Based on select using an event-driven model
  - Events occur at state transitions on underlying socket
- Set up a listening socket
- Add connected sockets on creation

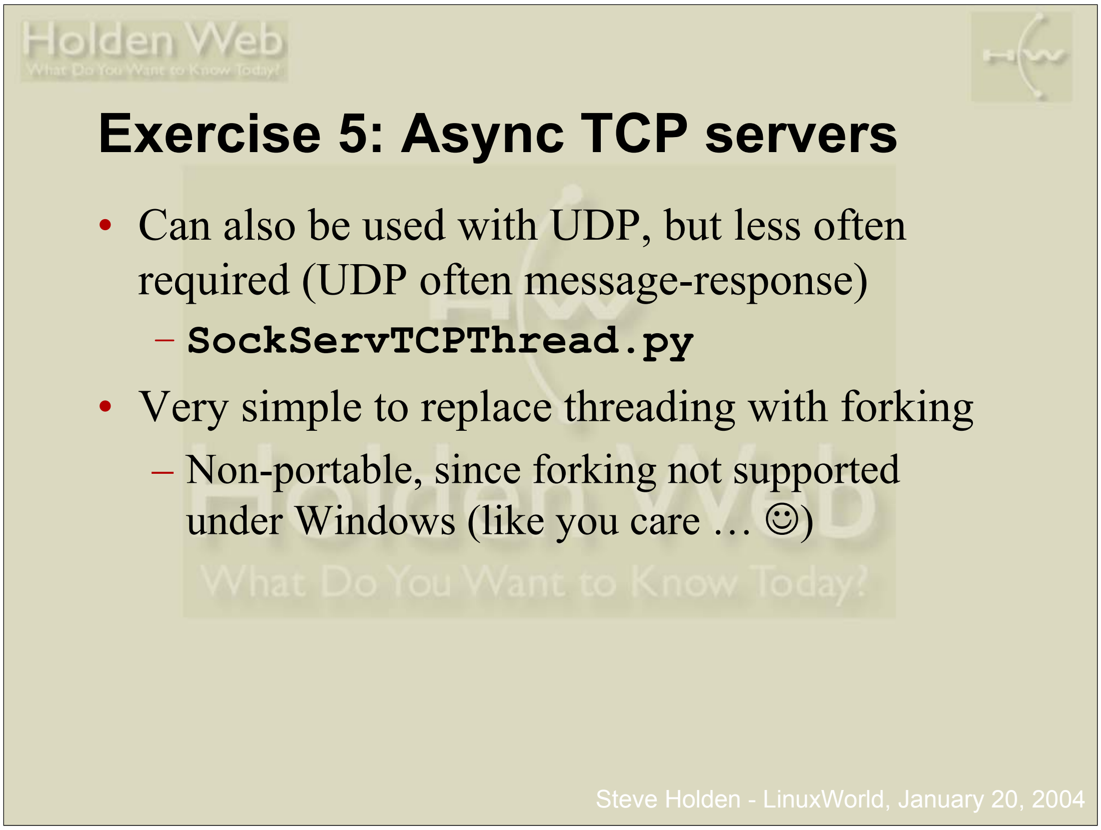
Steve Holden - LinuxWorld, January 20, 2004

Another, different way to specify asynchronous processing uses these modules. They are based on the use of the **`select()`** system call to define "events" that can occur asynchronously, and each active or lstening socket is a "channel".

A basic **`asyncore`** server program will create a single channel which is its listening socket, and then call the **`asyncore.loop()`** function. As connections come in the server will add further channels to handle events related to these connections.

Events cause the loop to call channel methods such as **`handle_read()`** and **`handle_write()`**, which cause data transfer between the server and the remote client.

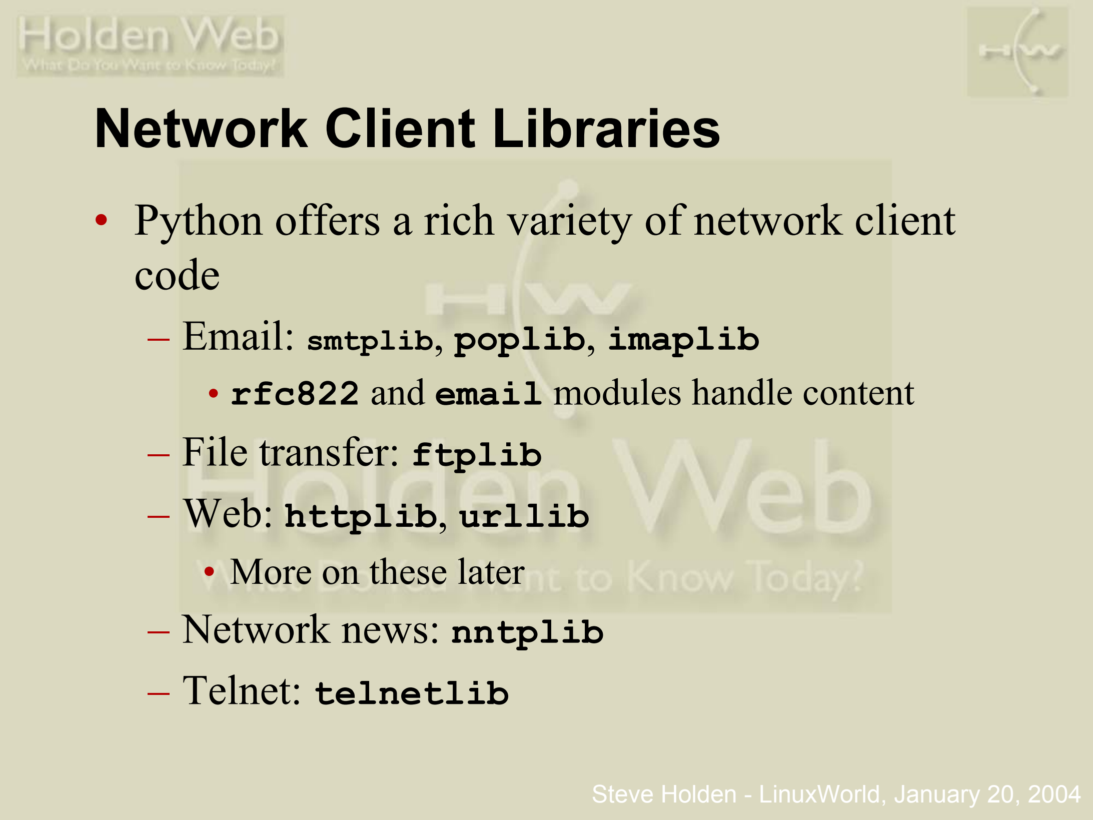
The **`asynchat`** module adds the ability to generate events when particular strings appear in the incoming data stream, which is useful for handling protocols that use delimiters rather than fixed-length strings.



Exercise 5: Async TCP servers

- Can also be used with UDP, but less often required (UDP often message-response)
  - `SockServTCPThread.py`
- Very simple to replace threading with forking
  - Non-portable, since forking not supported under Windows (like you care … ☺)

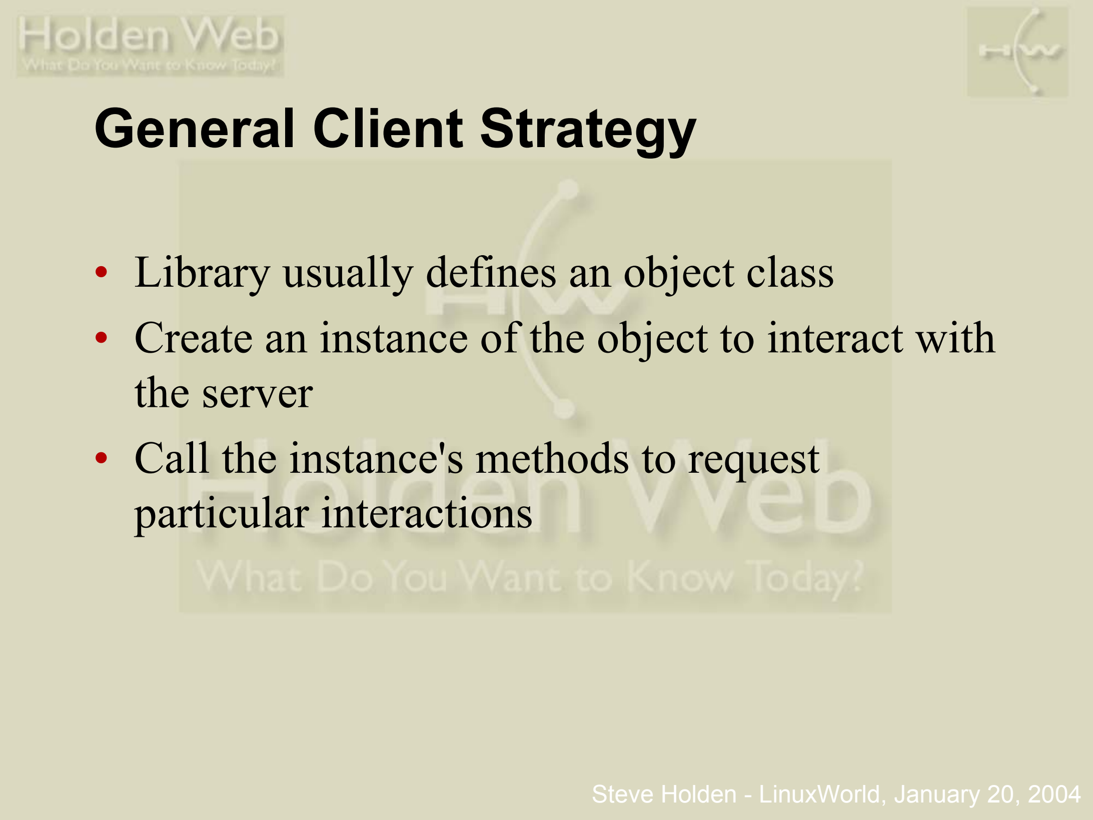
This shows that it isn't too difficult to adapt the original servers so they can handle multiple concurrent connections. SocketServer limits Windows to multi-threading solutions, but on Unix you have the choice of multithreading or process forking.



Network Client Libraries

- Python offers a rich variety of network client code
  - Email: **`smtplib`**, **`poplib`**, **`imaplib`**
    - **`rfc822`** and **`email`** modules handle content
  - File transfer: **`ftplib`**
  - Web: **`httplib`**, **`urllib`**
    - More on these later
  - Network news: **`nntplib`**
  - Telnet: **`telnetlib`**

The news is realtively good if you want to use the more common protocols. Lots of batteries included, although it would always be nice to see more client modules incorporated into the core.



General Client Strategy

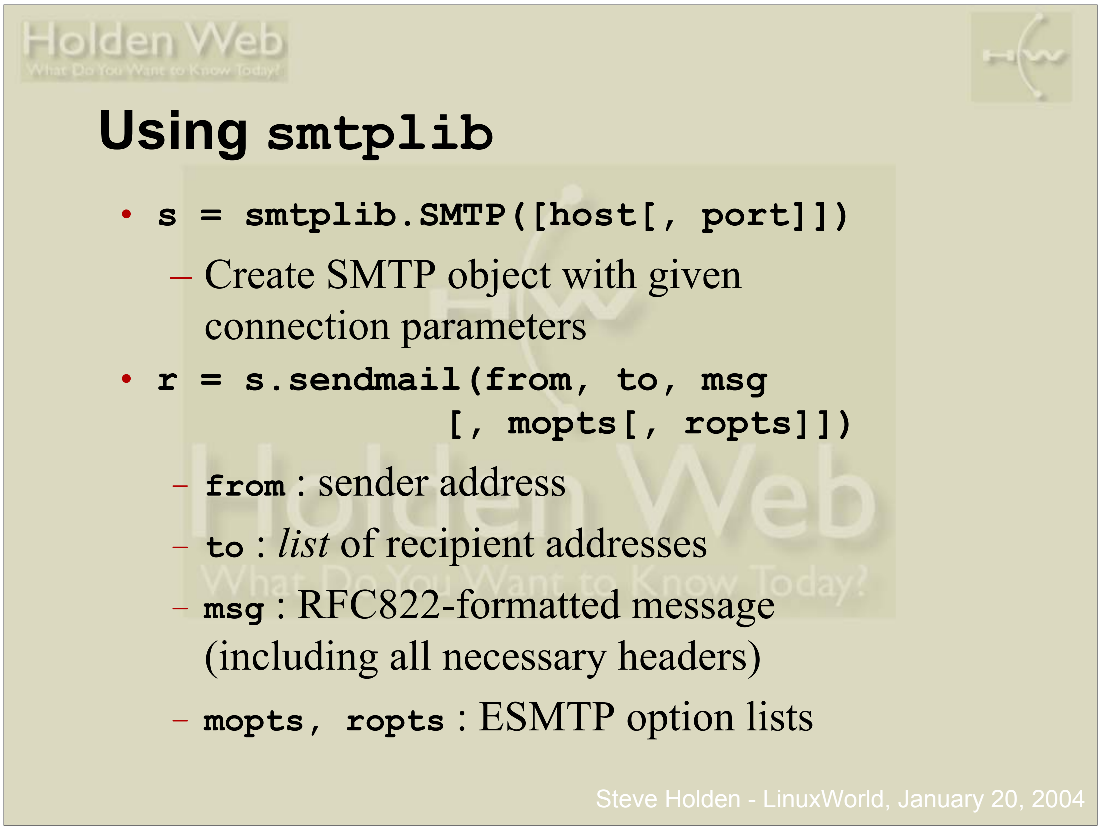
- Library usually defines an object class
- Create an instance of the object to interact with the server
- Call the instance's methods to request particular interactions

Steve Holden - LinuxWorld, January 20, 2004

Since Python is inherently object-oriented, it makes sense to encapsulate the connection with the server as an instance of some class. Then you have no problems about where to store connection state, and it's easy to generate multiple connections in the same program – if you do that, of course, you can get problems unless each client has its own thread.

As we've seen, adding asynchronous behavior is relatively easy on the server side, because mixin classes are provided for explicit support. It's less straightforward on the client side, but it's less frequently required too, so it still tends not to be a problem.

The third-party *twisted* framework is worth investigating (http://www.twistedmatrix.com/) if you have a need for an event-driven netwrok programming framework; the twisted code has been used to implement a wide variety of client- and server-side network functionality.



Using `smtplib`

- `s = smtplib.SMTP([host[, port]])`
  - Create SMTP object with given connection parameters

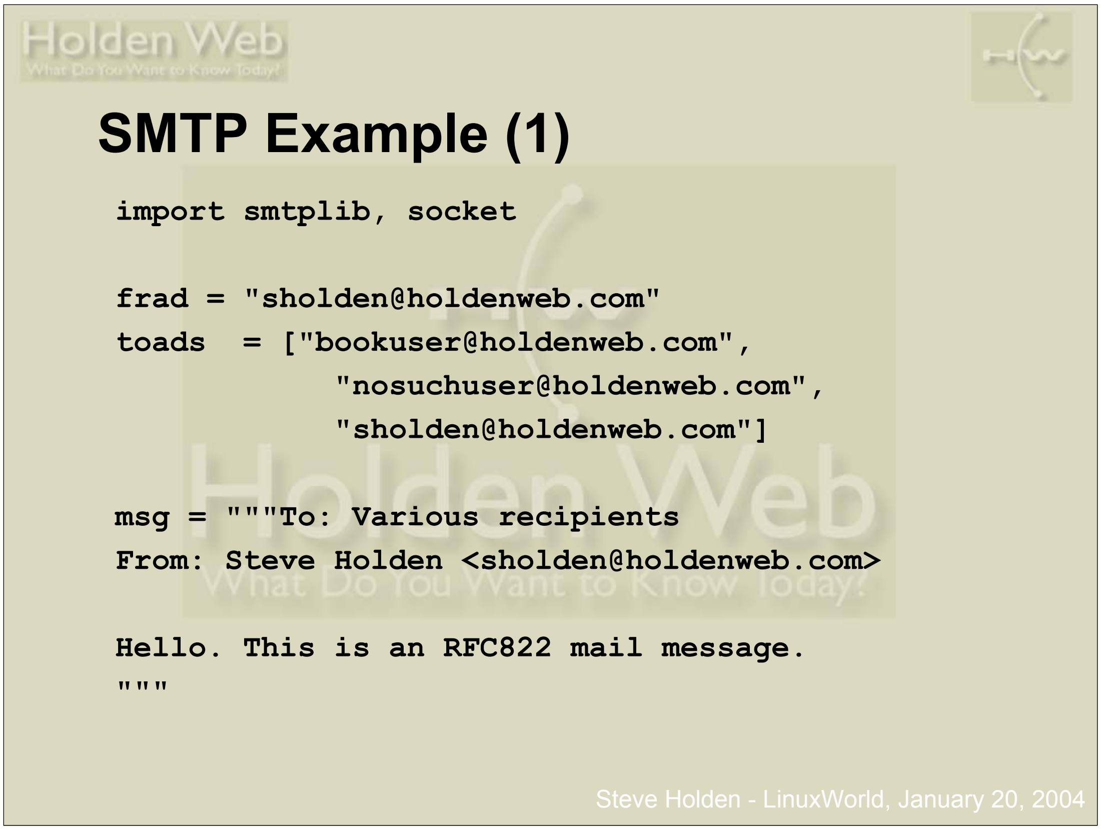
-
  ```
  r = s.sendmail(from, to, msg
                 [, mopts[, ropts]])
  ```
  - `from` : sender address
  - `to` : *list* of recipient addresses
  - `msg` : RFC822-formatted message (including all necessary headers)
  - `mopts, ropts` : ESMTP option lists

Steve Holden - LinuxWorld, January 20, 2004

The **`sendmail()`** method either raises an exception (if none of the recipients is accepted) or returns a dictionary.

Each dictionary entry has an unacceptable recipient address as a key, and the value associated with thast key is a tuple consisting of the numeric error code and the error message returned by the SMTP server.

For a completely successful send the returned dictionary will be empty. This makes it relatively easy to decode the results of an email transmission.

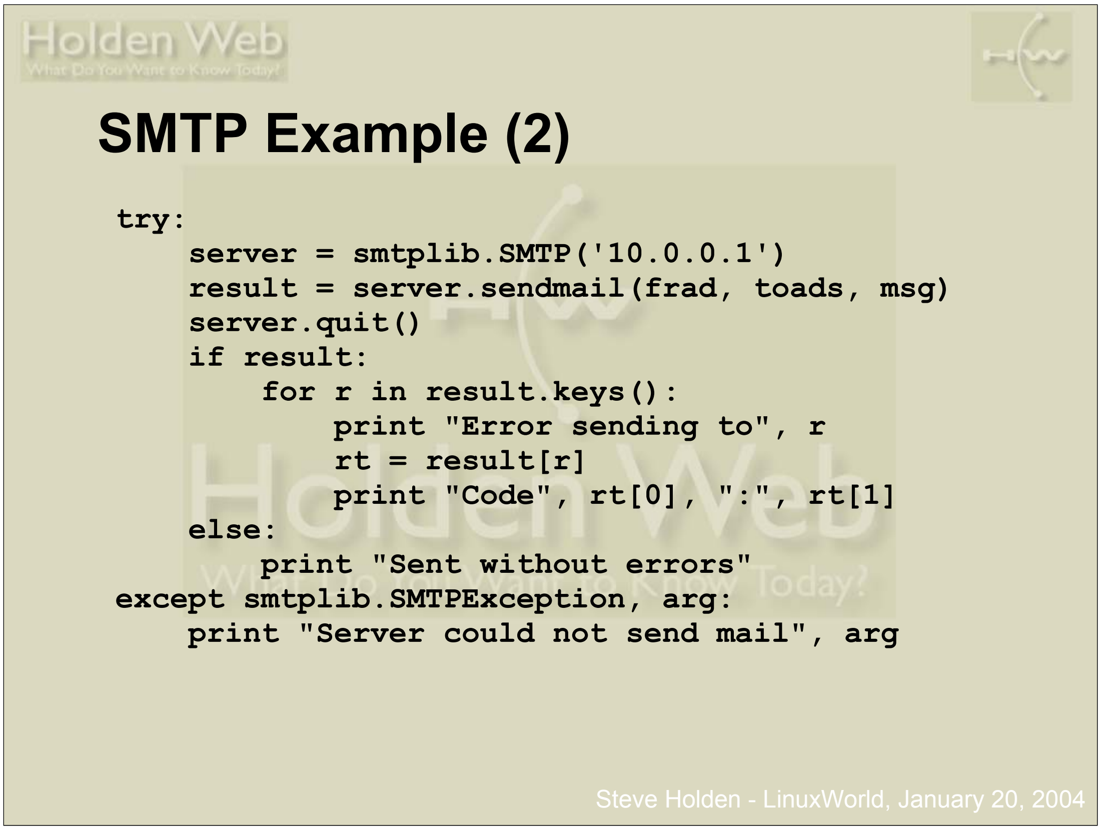


SMTP Example (1)

```
import smtplib, socket

frad = "sholden@holdenweb.com"
toads  = ["bookuser@holdenweb.com",
            "nosuchuser@holdenweb.com",
            "sholden@holdenweb.com"]

msg = """To: Various recipients
From: Steve Holden <sholden@holdenweb.com>

Hello. This is an RFC822 mail message.
"""
```

Note that the addresses of the sender and recipients aren't taken from the message headers, but from the arguments to the **`sendmail()`** method. While the two sets of addresses are usually the same, this isn't invariably the case.

This is one reason why spam is so common now – the SMTP protocol was never intended to cope with people who *lied* about who they are!



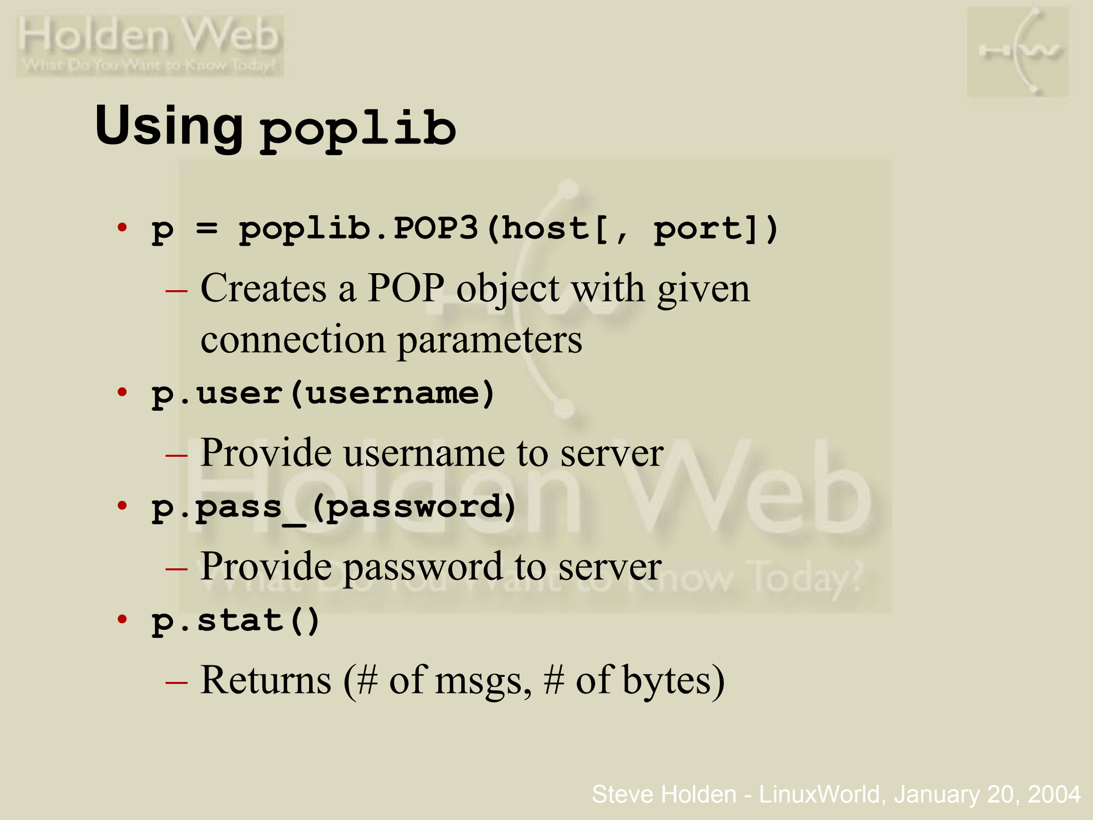
SMTP Example (2)

```
try:
    server = smtplib.SMTP('10.0.0.1')
    result = server.sendmail(frad, toads, msg)
    server.quit()
    if result:
        for r in result.keys():
            print "Error sending to", r
            rt = result[r]
            print "Code", rt[0], ":", rt[1]
    else:
        print "Sent without errors"
except smtplib.SMTPException, arg:
    print "Server could not send mail", arg
```

Since it's entirely possible that the transmission will generate exceptions, it's always safest to allow for them.

The sample code in this presentation isn't the best example of defensive programming, but the slides would have been rather too busy if all possible exceptions had been handled.

At least it does report both partial and total failure.



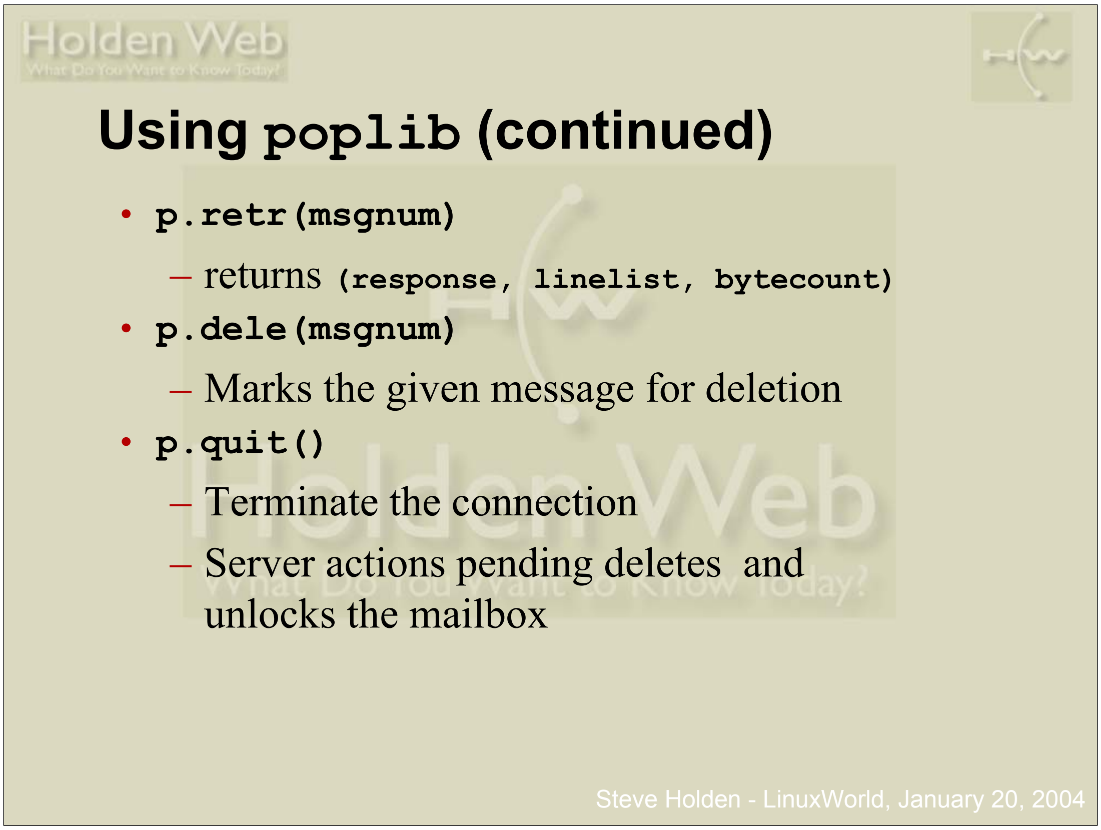
Using `poplib`

- `p = poplib.POP3(host[, port])`
  - Creates a POP object with given connection parameters
- `p.user(username)`
  - Provide username to server
- `p.pass_(password)`
  - Provide password to server
- `p.stat()`
  - Returns (# of msgs, # of bytes)

Steve Holden - LinuxWorld, January 20, 2004

POP is probably the most popular mail user agent protocol. IMAP is coming up fast, but it's a *much* more complex protocol, and well outside the scope of anything shorter than a book.

Once you have connected to a server and authenticated yourself, you can find out how much mail is waiting in your mailbox.

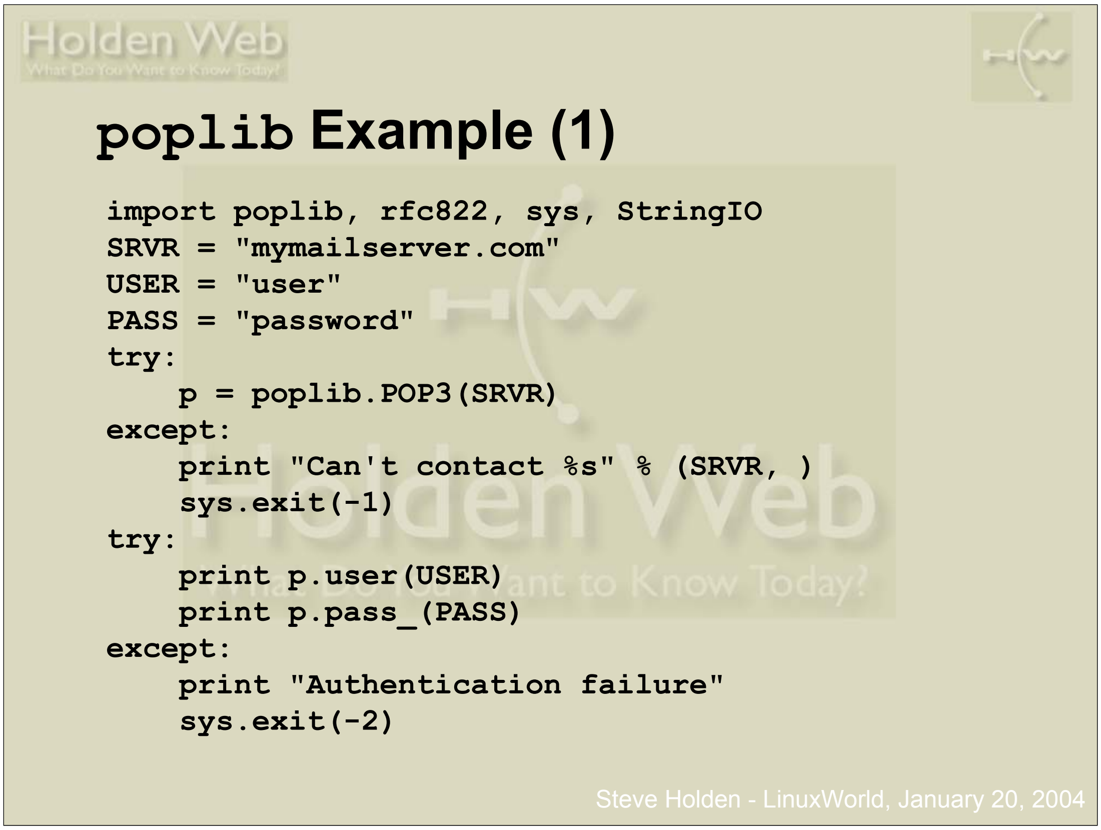


Using `poplib` (continued)

- `p.retr(msgnum)`
  - returns `(response, linelist, bytecount)`
- `p.dele(msgnum)`
  - Marks the given message for deletion
- `p.quit()`
  - Terminate the connection
  - Server actions pending deletes and unlocks the mailbox

You then retrieve the messages one at a time, and can mark them for deletion or not, as the case may be.

The POP protocol specifies that the mailbox doesn't get updated until the client terminates the session. That's why you sometimes get duplicate messages when a network connection goes down during a POP3 session.



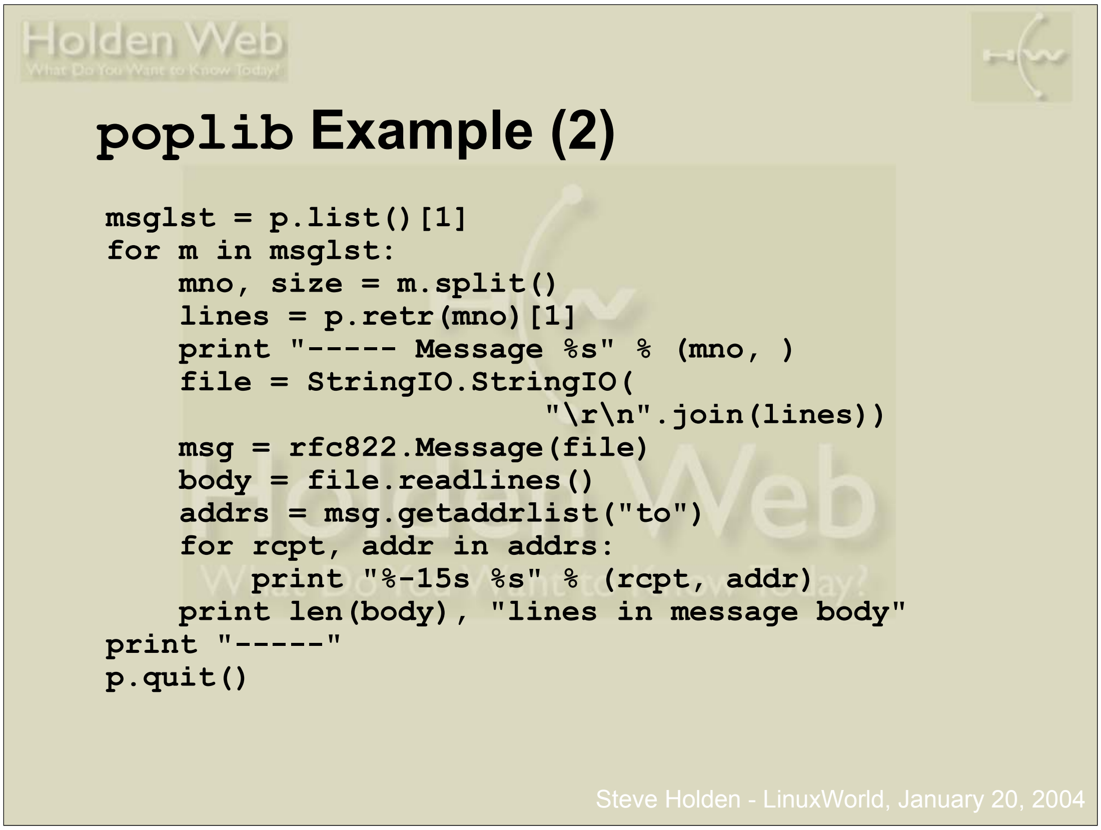
poplib Example (1)

```
import poplib, rfc822, sys, StringIO
SRVR = "mymailserver.com"
USER = "user"
PASS = "password"
try:
    p = poplib.POP3(SRVR)
except:
    print "Can't contact %s" % (SRVR, )
    sys.exit(-1)
try:
    print p.user(USER)
    print p.pass_(PASS)
except:
    print "Authentication failure"
    sys.exit(-2)
```

Steve Holden - LinuxWorld, January 20, 2004

Again the error checking is a little vestigial, but it's important that you realise how easy it is for Python to deal with the kind of problem that network code can generate.

At the end of this first page of the code, **p** is an authenticated POP3 connection waiting and ready to go.



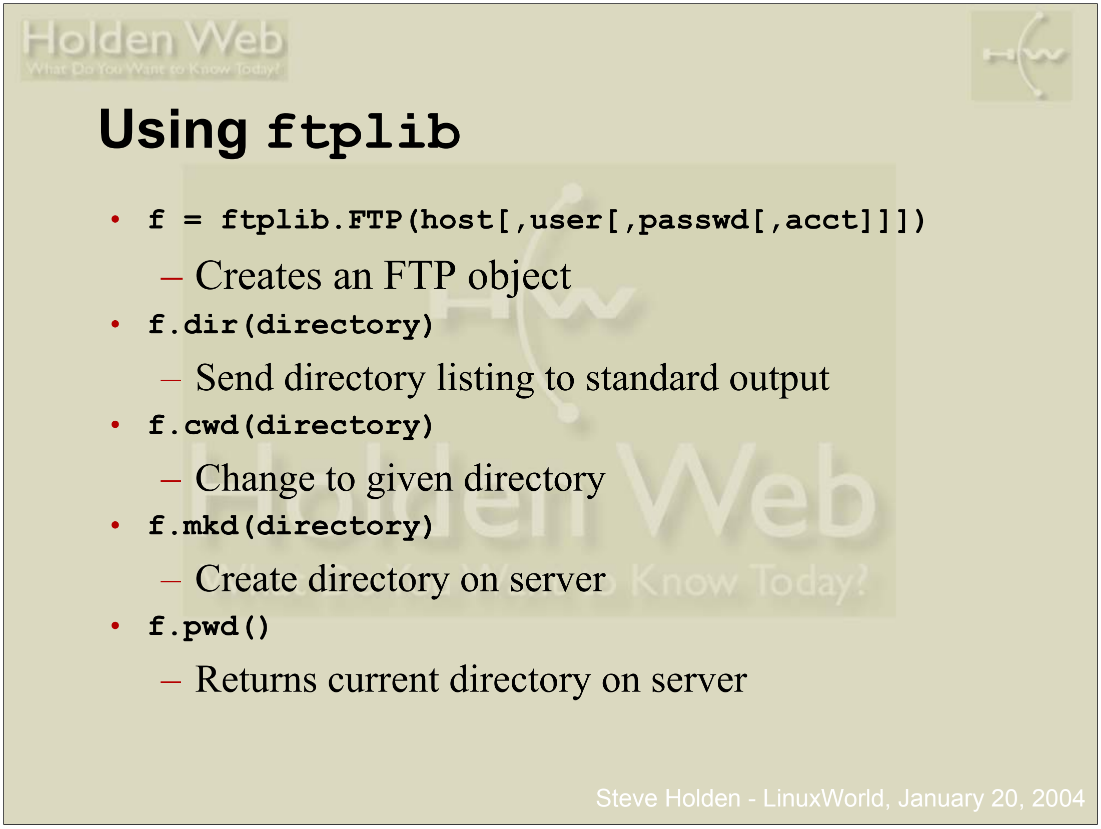
poplib Example (2)

```
msglst = p.list()[1]
for m in msglst:
    mno, size = m.split()
    lines = p.retr(mno)[1]
    print "----- Message %s" % (mno, )
    file = StringIO.StringIO(
                        "\r\n".join(lines))
    msg = rfc822.Message(file)
    body = file.readlines()
    addrs = msg.getaddrlist("to")
    for rcpt, addr in addrs:
        print "%-15s %s" % (rcpt, addr)
    print len(body), "lines in message body"
print "-----"
p.quit()
```

The server returns a list of message descriptions as the second element of the result of the **`list()`** method. The **`for`** loop iterates over this list of message descriptions to process the whole content of the mailbox.

Each message description is itself made up of a message number and a size. These are saved in an unpacking assignment, and the message number is used to retrieve the message. The message is reconstituted as a file-like **`StringIO`** object to allow it to be parsed using the standard **`rfc822`** library (more modern code would use the **`email`** library).

Some of the message's details are then printed out. This shows you how easy it is to handle email in Python – a complete mailbox processor in less than thirty lines of code!

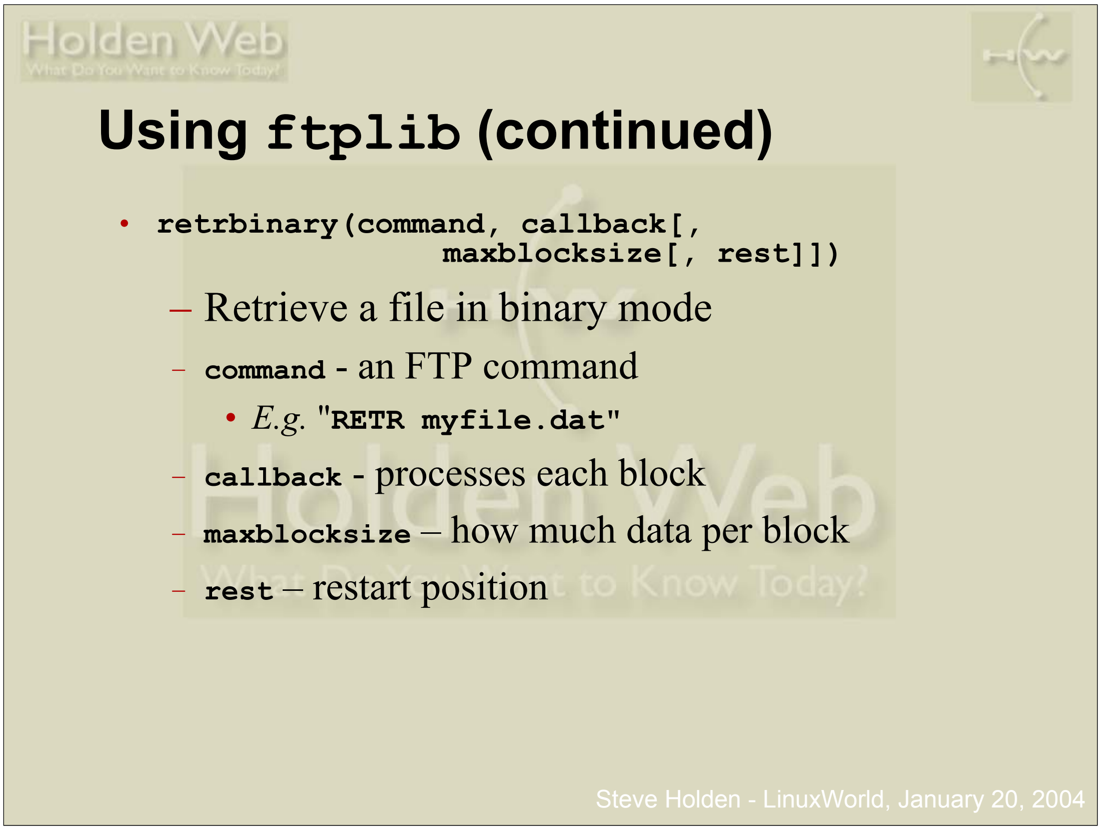


Using `ftplib`

- `f = ftplib.FTP(host[,user[,passwd[,acct]]])`
  - Creates an FTP object
- `f.dir(directory)`
  - Send directory listing to standard output
- `f.cwd(directory)`
  - Change to given directory
- `f.mkd(directory)`
  - Create directory on server
- `f.pwd()`
  - Returns current directory on server

FTP is a rather more complex protocol, so the coverage here is limited to the bare essentials.

I focus on the things that I found most difficult to comprehend when I was a Python beginner.



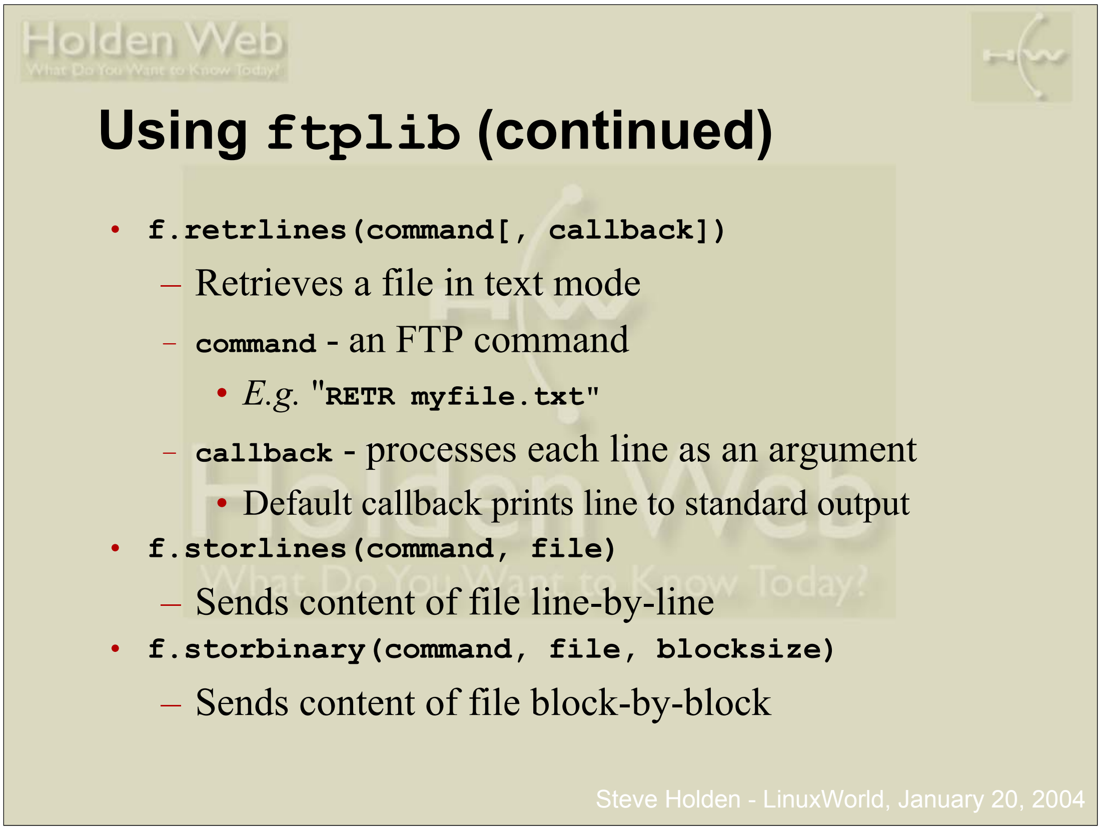
Using ftplib (continued)

-
  ```
  retrbinary(command, callback[,
                       maxblocksize[, rest]])
  ```
  - Retrieve a file in binary mode
  - command - an FTP command
    - *E.g.* "RETR myfile.dat"
  - callback - processes each block
  - maxblocksize – how much data per block
  - rest – restart position

Steve Holden - LinuxWorld, January 20, 2004

The retrieval methods use a callback paradigm – you call the retrieval method, providing a callback function which the method's code will call whenever it has data to dispose of. The callback can be a plain function, or any other Python callable.

When binary files are transferred then typically the callback will be a function (or the bound method of an object) that writes each chunk out to a local file as it is received.



Using `ftplib` (continued)

- `f.retrlines(command[, callback])`
  - Retrieves a file in text mode
  - `command` - an FTP command
    - *E.g.* "`RETR myfile.txt`"
  - `callback` - processes each line as an argument
    - Default callback prints line to standard output
- `f.storlines(command, file)`
  - Sends content of file line-by-line
- `f.storbinary(command, file, blocksize)`
  - Sends content of file block-by-block

Steve Holden - LinuxWorld, January 20, 2004

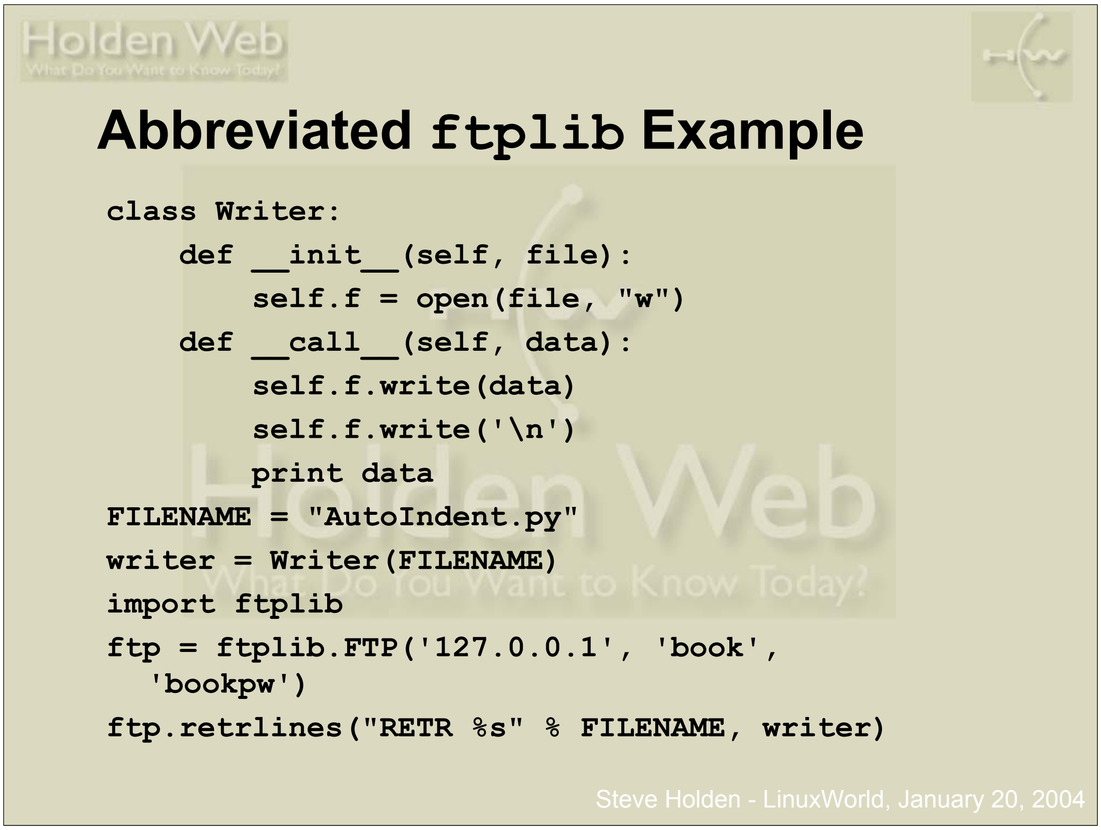
Text retrieval is similar to binary retrieval, with the exception that the callback is actiavted for each received line.

The storage methods use a file that you have already opened to provide the transfer content, and simply read the file as appropriate and necessary.

One of the weaknesses of the FTP module is that you have to know enough about the FTP protocol to be able to formulate the commands to the server, which isn't strictly necessary. There's a more complicated FTP example at

    http://www.holdenweb.com/Python/PDCode/ftpStream.py

This shows transfers of different sizes, and explains how it's relatively easy to layer the behavior you want over a Python library whose interface isn't convenient for you.

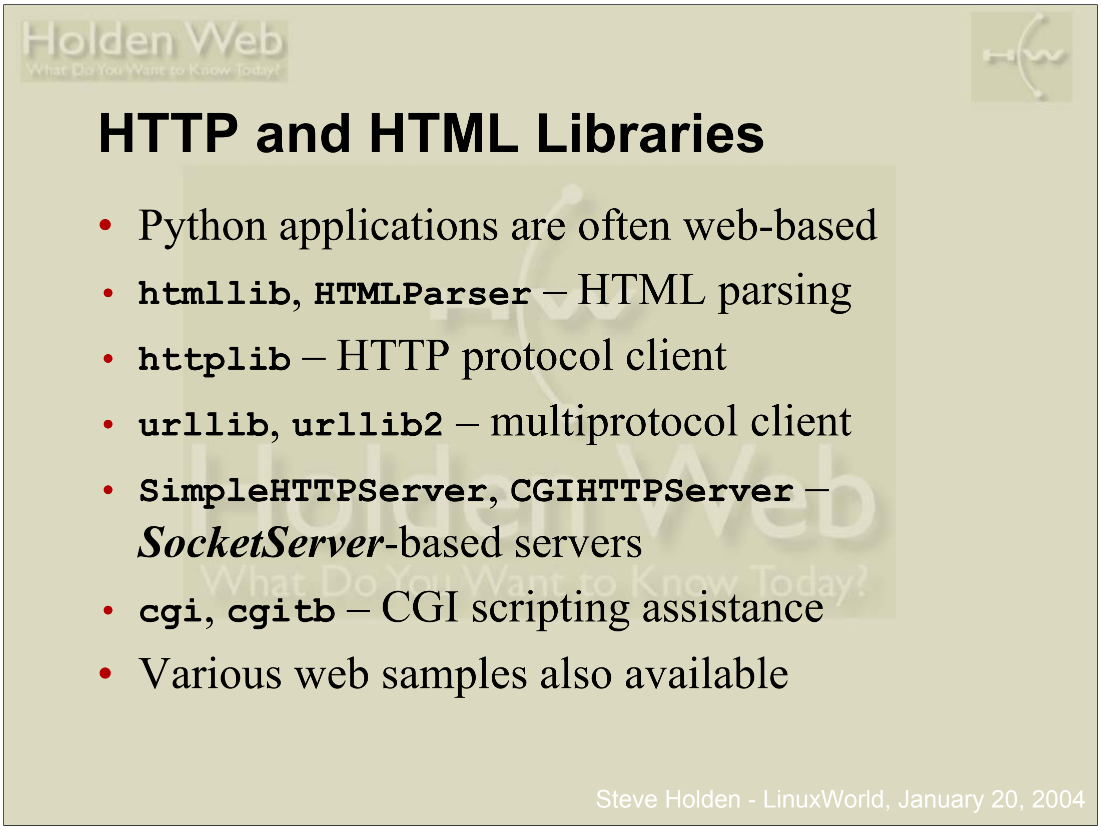


Abbreviated `ftplib` Example

```
class Writer:
    def __init__(self, file):
        self.f = open(file, "w")
    def __call__(self, data):
        self.f.write(data)
        self.f.write('\n')
        print data
FILENAME = "AutoIndent.py"
writer = Writer(FILENAME)
import ftplib
ftp = ftplib.FTP('127.0.0.1', 'book',
  'bookpw')
ftp.retrlines("RETR %s" % FILENAME, writer)
```

Steve Holden - LinuxWorld, January 20, 2004

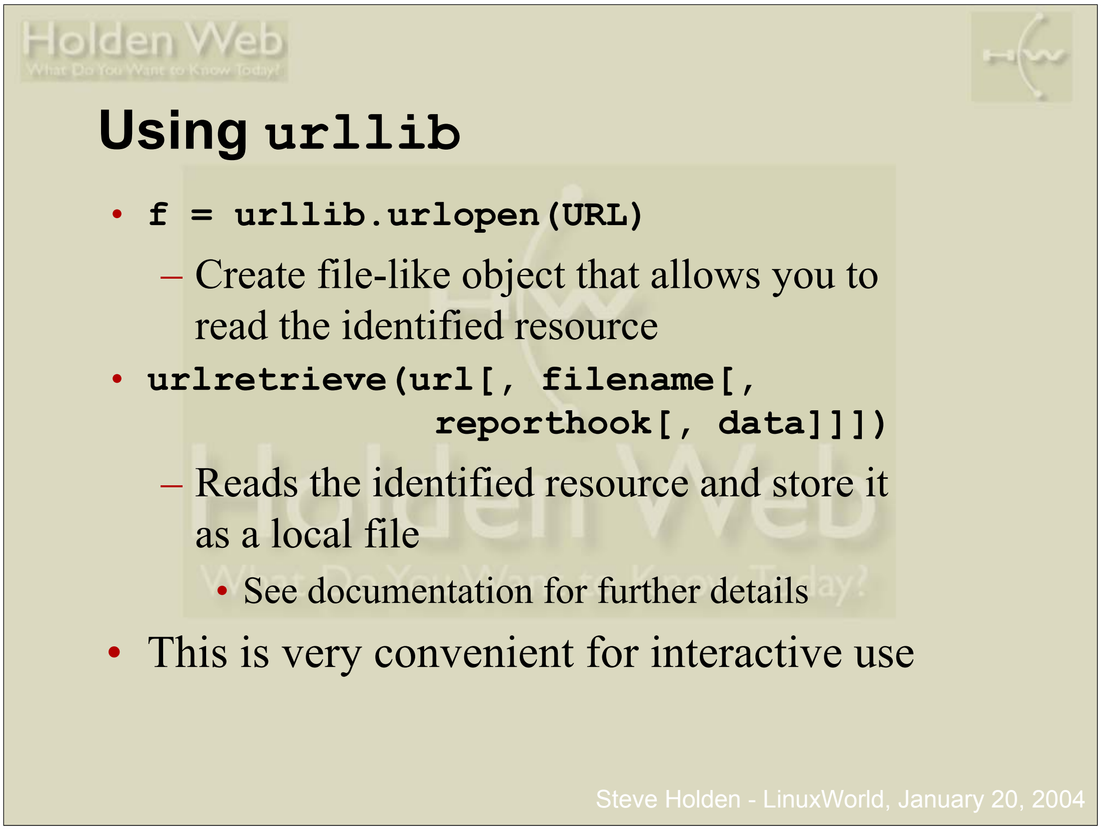
A rather simple example, but at least it shoes you how you can use the library to perform real work. Note that a **`Writer`** instance is callable – what actually gets called is its **`__call__()`** special method.



HTTP and HTML Libraries

- Python applications are often web-based
- `htmllib`, `HTMLParser` – HTML parsing
- `httplib` – HTTP protocol client
- `urllib`, `urllib2` – multiprotocol client
- `SimpleHTTPServer`, `CGIHTTPServer` – ***SocketServer***-based servers
- `cgi`, `cgitb` – CGI scripting assistance
- Various web samples also available

There's a large variety of web-related libraries. Many of the techniques you need are the same as those used in the earlier examples.

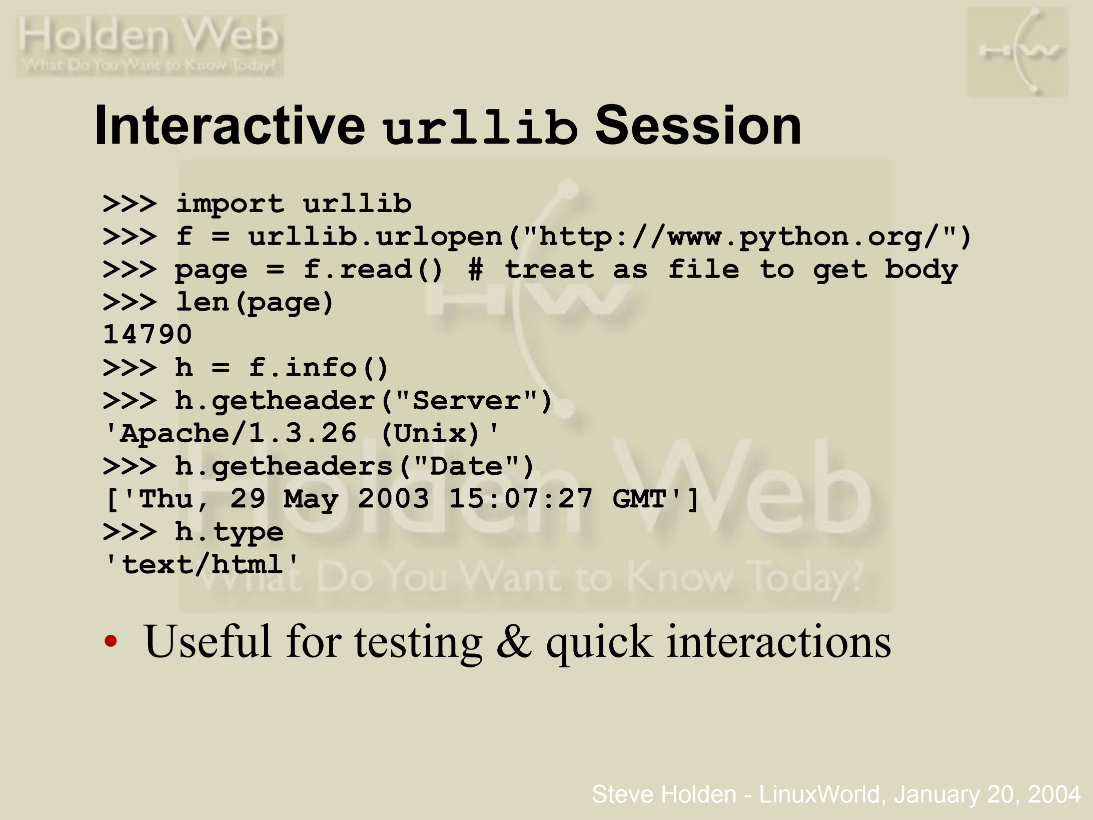


Using `urllib`

- **`f = urllib.urlopen(URL)`**
  - Create file-like object that allows you to read the identified resource
- **`urlretrieve(url[, filename[, reporthook[, data]]])`**
  - Reads the identified resource and store it as a local file
    - See documentation for further details
- This is very convenient for interactive use

The **`urllib`** code can handle just about anything that a modern browser can handle, though it isn't going to support secure communications any time soon. You need **`urllib2`** for that.

Ongoing efforts in the Python community continue to try to make web services more accessible. There are also SOAP libraries, and *lots* of XML handling code of various types.



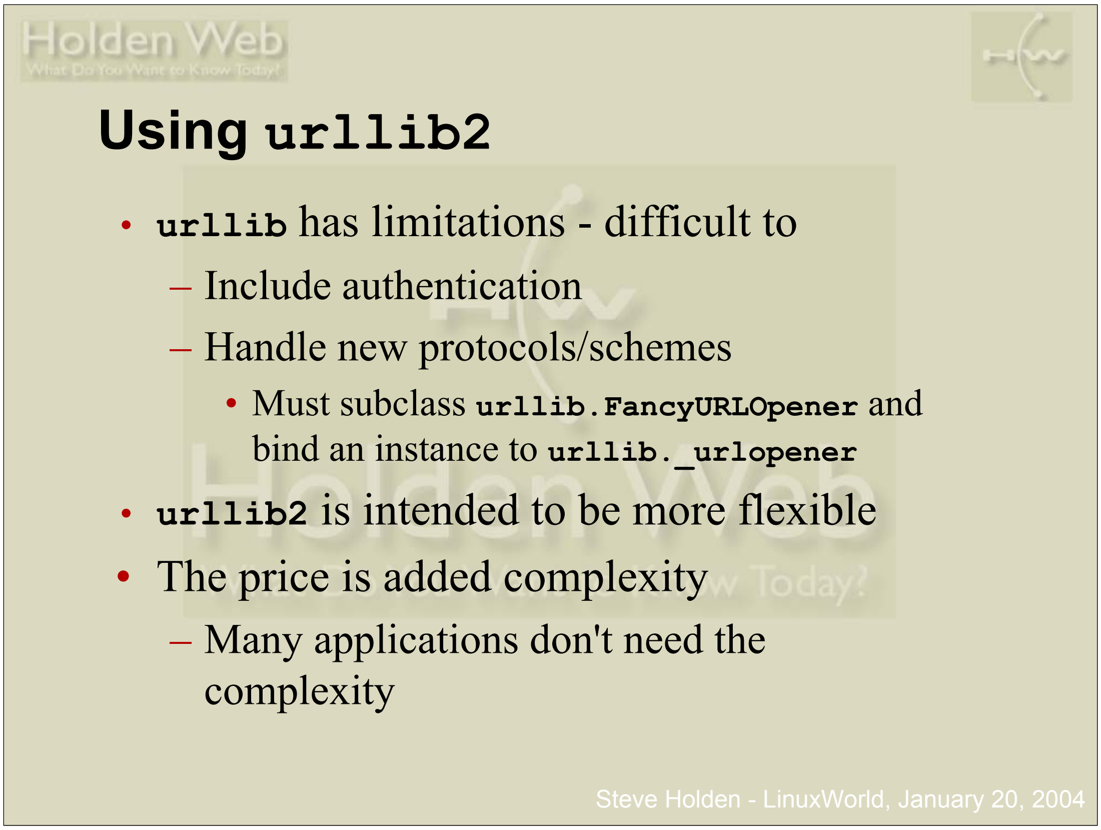
Interactive `urllib` Session

```
>>> import urllib
>>> f = urllib.urlopen("http://www.python.org/")
>>> page = f.read() # treat as file to get body
>>> len(page)
14790
>>> h = f.info()
>>> h.getheader("Server")
'Apache/1.3.26 (Unix)'
>>> h.getheaders("Date")
['Thu, 29 May 2003 15:07:27 GMT']
>>> h.type
'text/html'
```

- Useful for testing & quick interactions

The interactive interpreter is a very easy way to find out how the various modules work., and you are encouraged to try things out to assist your understanding from the documentation.

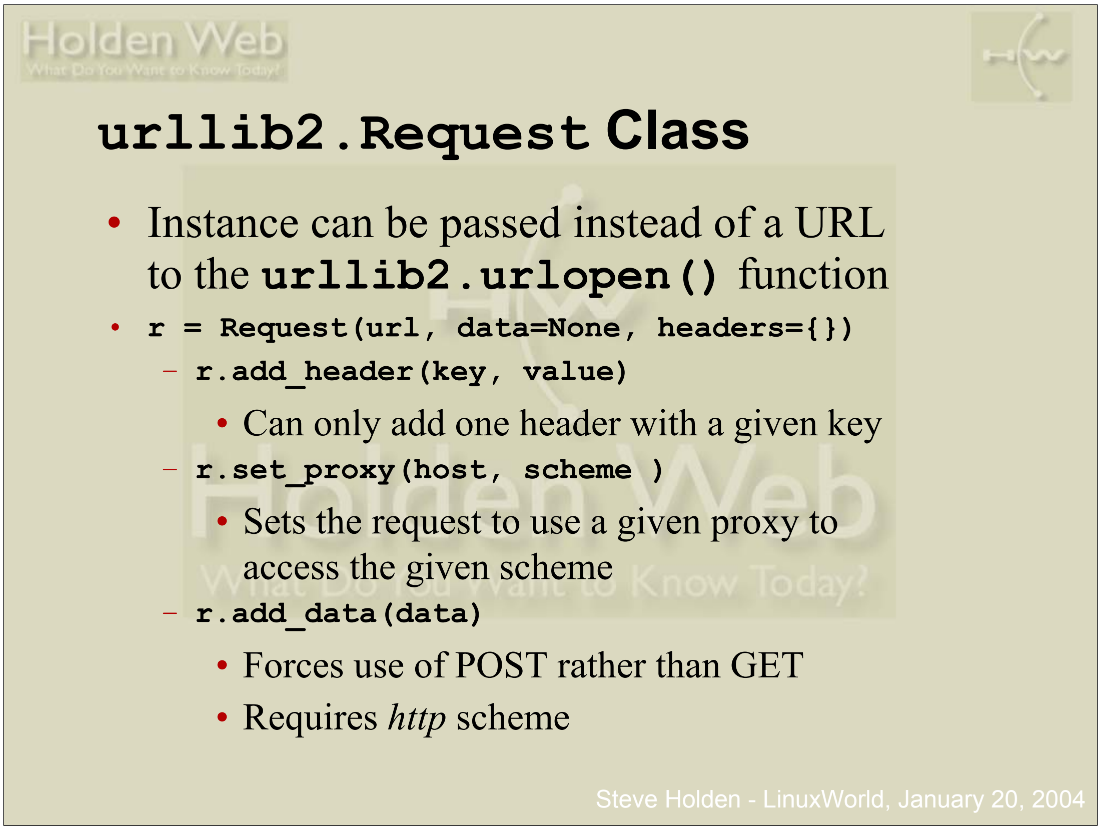
Here we see a simple exposition of **urllib**.



Using `urllib2`

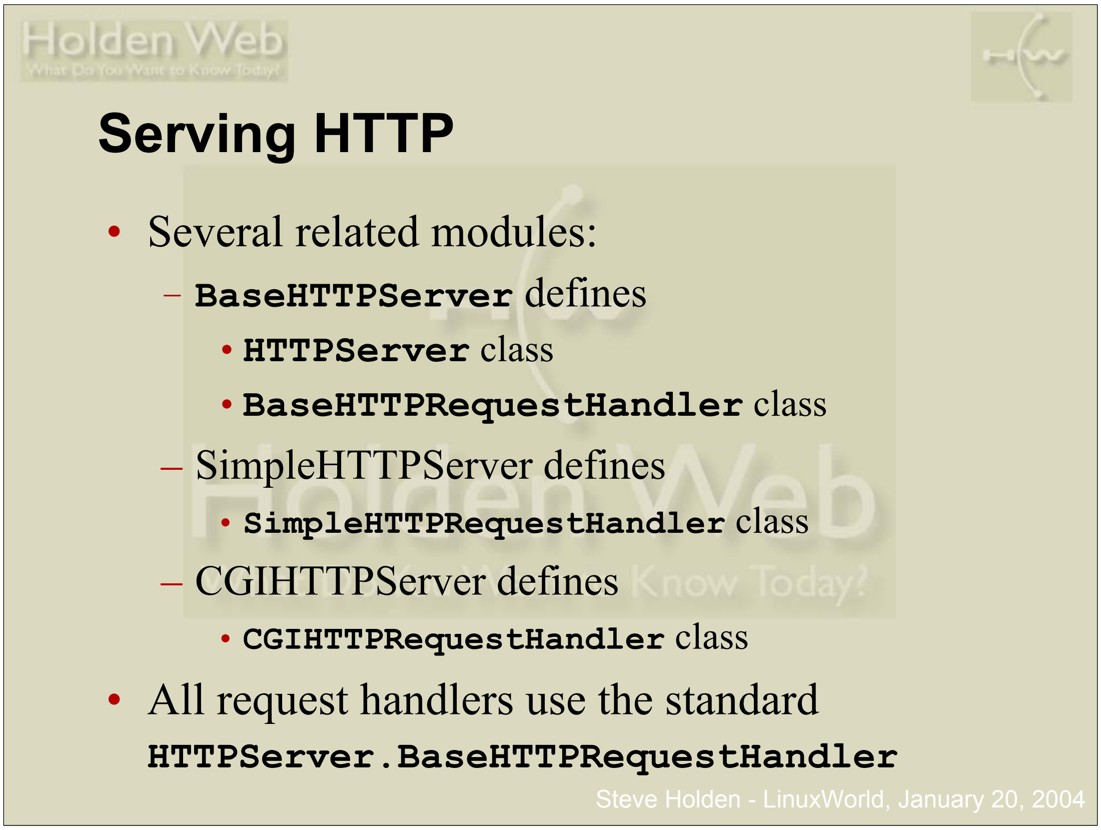
- `urllib` has limitations - difficult to
  - Include authentication
  - Handle new protocols/schemes
    - Must subclass `urllib.FancyURLOpener` and bind an instance to `urllib._urlopener`
- `urllib2` is intended to be more flexible
- The price is added complexity
  - Many applications don't need the complexity

The newer library is better organized, which means it is more flexible (for example, people have managed to handle https communications with it), but this makes it correspondingly more difficult to use.If **urllib** can handle your needs it's the best option for a quick solution. For a better-engineered solution, however, you will find **urllib2** more satisfactory.



`urllib2.Request` Class

- Instance can be passed instead of a URL to the `urllib2.urlopen()` function
- `r = Request(url, data=None, headers={})`
  - `r.add_header(key, value)`
    - Can only add one header with a given key
  - `r.set_proxy(host, scheme )`
    - Sets the request to use a given proxy to access the given scheme
  - `r.add_data(data)`
    - Forces use of POST rather than GET
    - Requires *http* scheme



Serving HTTP

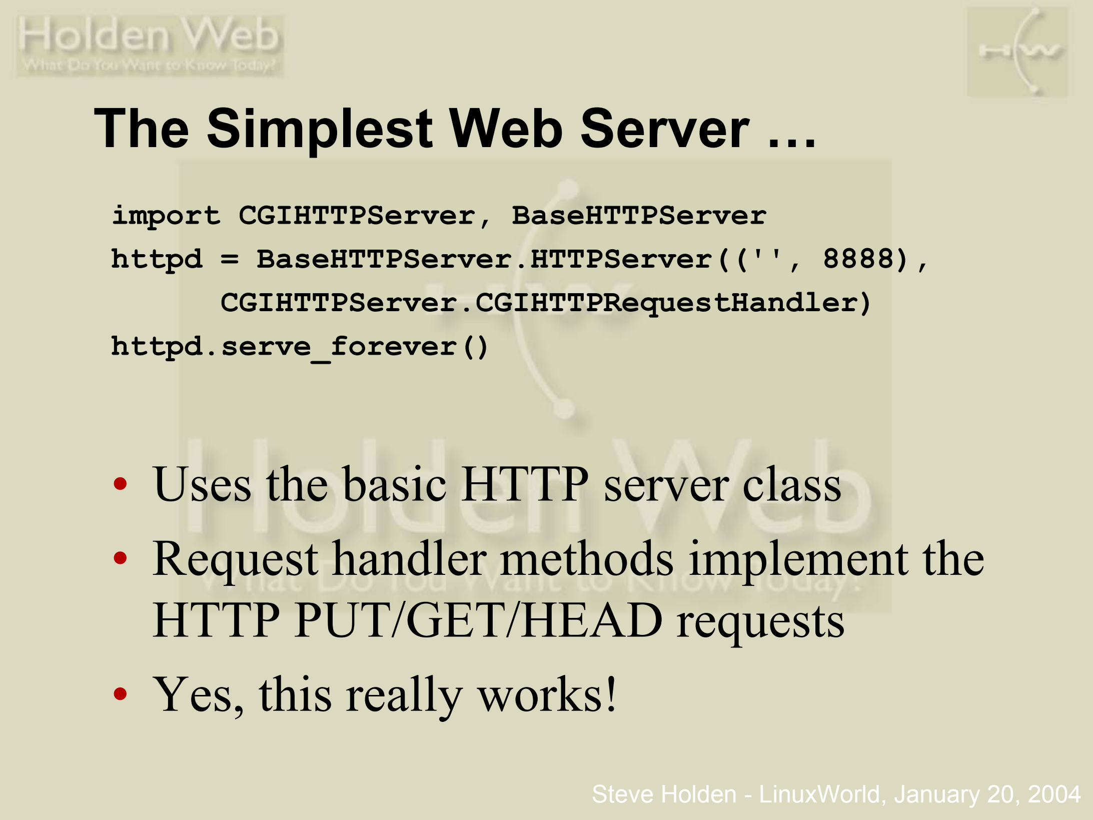
- Several related modules:
  - **`BaseHTTPServer`** defines
    - **`HTTPServer`** class
    - **`BaseHTTPRequestHandler`** class
  - SimpleHTTPServer defines
    - **`SimpleHTTPRequestHandler`** class
  - CGIHTTPServer defines
    - **`CGIHTTPRequestHandler`** class
- All request handlers use the standard **`HTTPServer.BaseHTTPRequestHandler`**

Steve Holden - LinuxWorld, January 20, 2004

All the core Python web servers are based on the same server code.

The facilities of the server are enhanced by adding methods to the request handlers.

For example, to handle POST requests the handler has to provide a **do_post** method.



The Simplest Web Server …

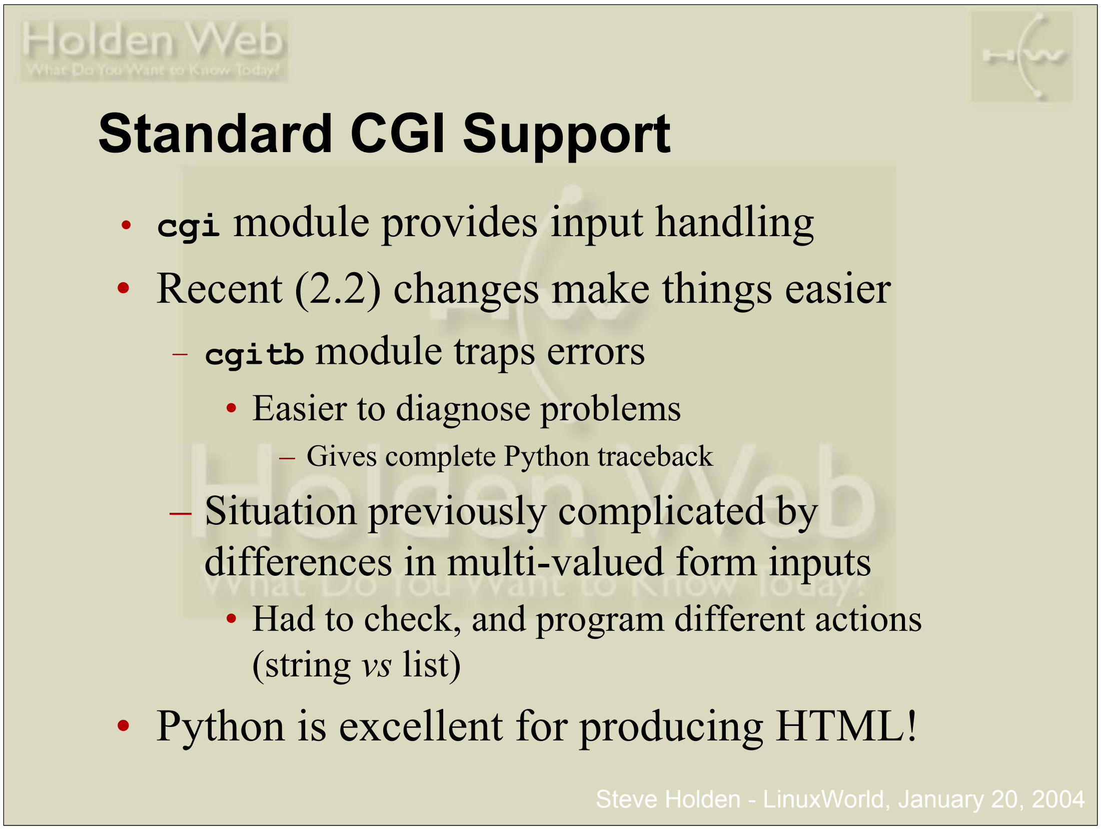
```
import CGIHTTPServer, BaseHTTPServer
httpd = BaseHTTPServer.HTTPServer(('', 8888),
      CGIHTTPServer.CGIHTTPRequestHandler)
httpd.serve_forever()
```

- Uses the basic HTTP server class
- Request handler methods implement the HTTP PUT/GET/HEAD requests
- Yes, this really works!

Steve Holden - LinuxWorld, January 20, 2004

When I first published this code, sombody mailed me to compain that writing a web server couldn't possibly be that simple! It does seem quite amazing that three (logical) lines of code will get you on the web.

The Python tutor group spent some time playing with this code. It was fascinating for language beginners to be able to test their own web content with such a small piece of code. Although the server has some limitations it's ideal for local use. Your browser gets to do real HTTP (1.0) interactions!



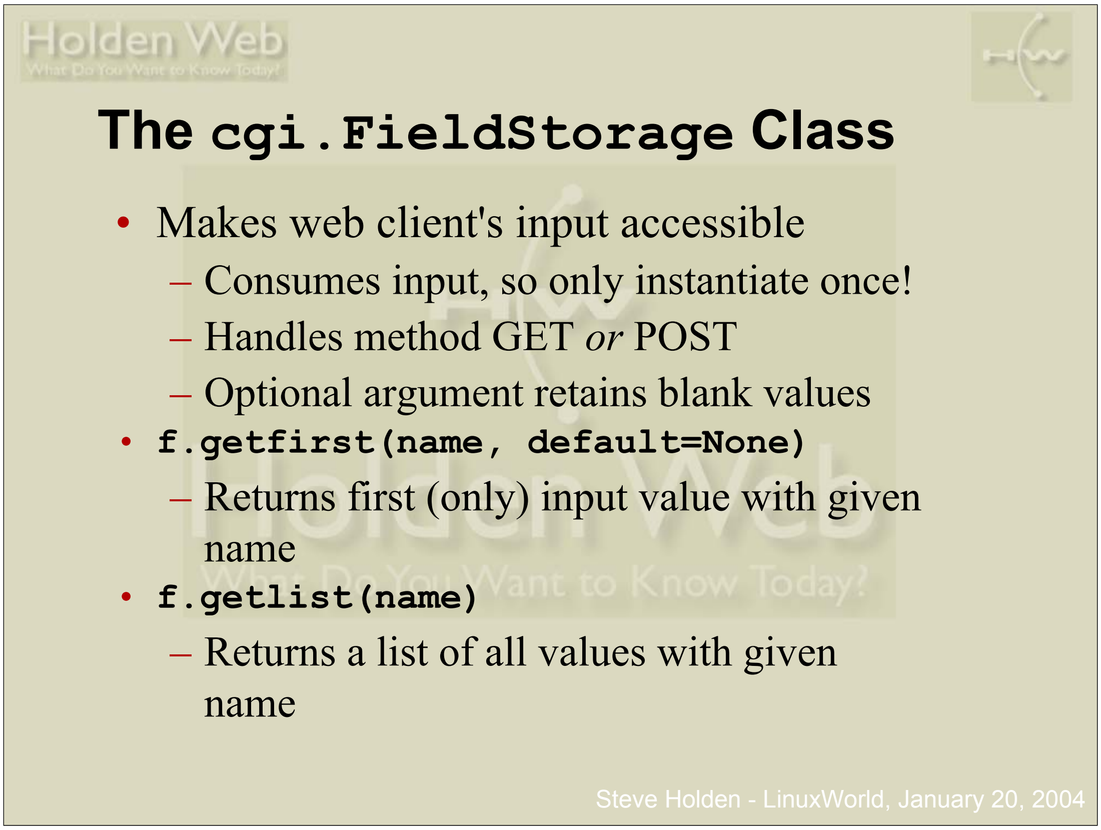
Standard CGI Support

- `cgi` module provides input handling
- Recent (2.2) changes make things easier
  - `cgitb` module traps errors
    - Easier to diagnose problems
      - Gives complete Python traceback
  - Situation previously complicated by differences in multi-valued form inputs
    - Had to check, and program different actions (string *vs* list)
- Python is excellent for producing HTML!

Steve Holden - LinuxWorld, January 20, 2004

The code of the CGI module is rather complicated, mostly because of the need to maintain backward compatibility with older versions and because it needs to handle the rare complexity of forms with client-side files embedded.

Since this complexity isn't needed by most applications, I focus on the bits you will need most of the time. Again, you will find that you can write CGI code quite easily in Python under a variety of web servers – the beauty of a standard environment!



The `cgi.FieldStorage` Class

- Makes web client's input accessible
  - Consumes input, so only instantiate once!
  - Handles method GET *or* POST
  - Optional argument retains blank values
- **`f.getfirst(name, default=None)`**
  - Returns first (only) input value with given name
- **`f.getlist(name)`**
  - Returns a list of all values with given name

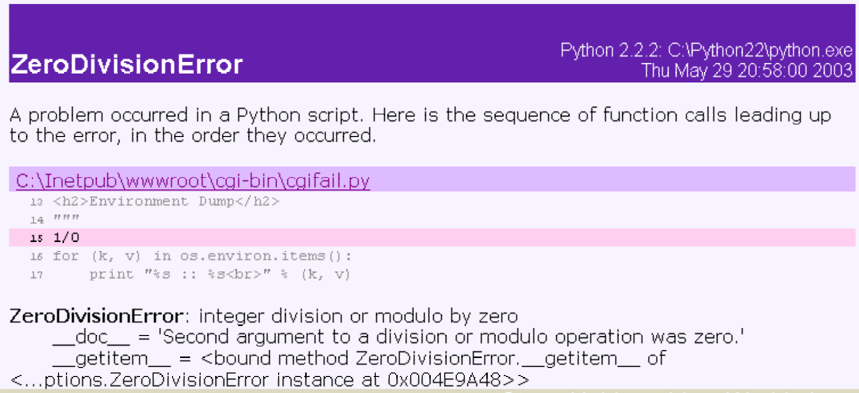
Strictly speakling, each value stored in the **`FieldStorage`** can be another instance of **`FieldStorage`**. Often it will instead be an instance of the (signature-compatible but considerably less complex) **`MiniFieldStorage`** class, internal to the **`cgi`** module.

Error Handling

- Should use for *all* CGI scripts!

  `import cgitb; cgitb.enable()`

- Traps any errors, producing legible trace



ZeroDivisionError

Python 2.2.2: C:\Python22\python.exe
Thu May 29 20:58:00 2003

A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.

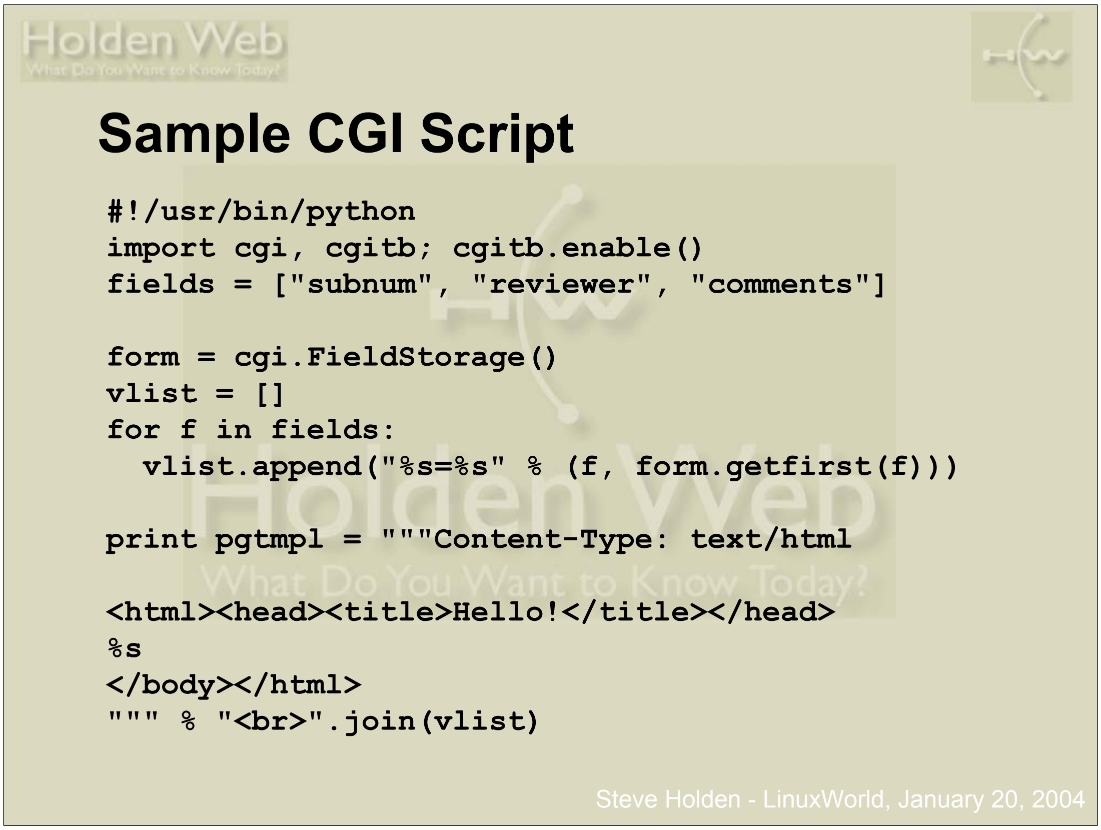
C:\Inetpub\wwwroot\cgi-bin\cgifail.py

```
<h2>Environment Dump</h2>
"""
1/0
for (k, v) in os.environ.items():
    print "%s :: %s<br>" % (k, v)
```

```
ZeroDivisionError: integer division or modulo by zero
      __doc__ = 'Second argument to a division or modulo operation was zero.'
      __getitem__ = <bound method ZeroDivisionError.__getitem__ of
<...ptions.ZeroDivisionError instance at 0x004E9A48>>
```

There's nothing worse than a mysterious web application error that just produces a blank page. Unfortunately this can often be what the user sees if you just let a Python failure generate a traceback on standard error – the traceback gets recorded in the server logs, but this isn't ideal, especially when the server isn't directly under your control.

The cgitb module saves you from such embarrassments. Here you see the traceback from a deliberately-provoked divide-by-zero error, somewhat truncated for space reasons.



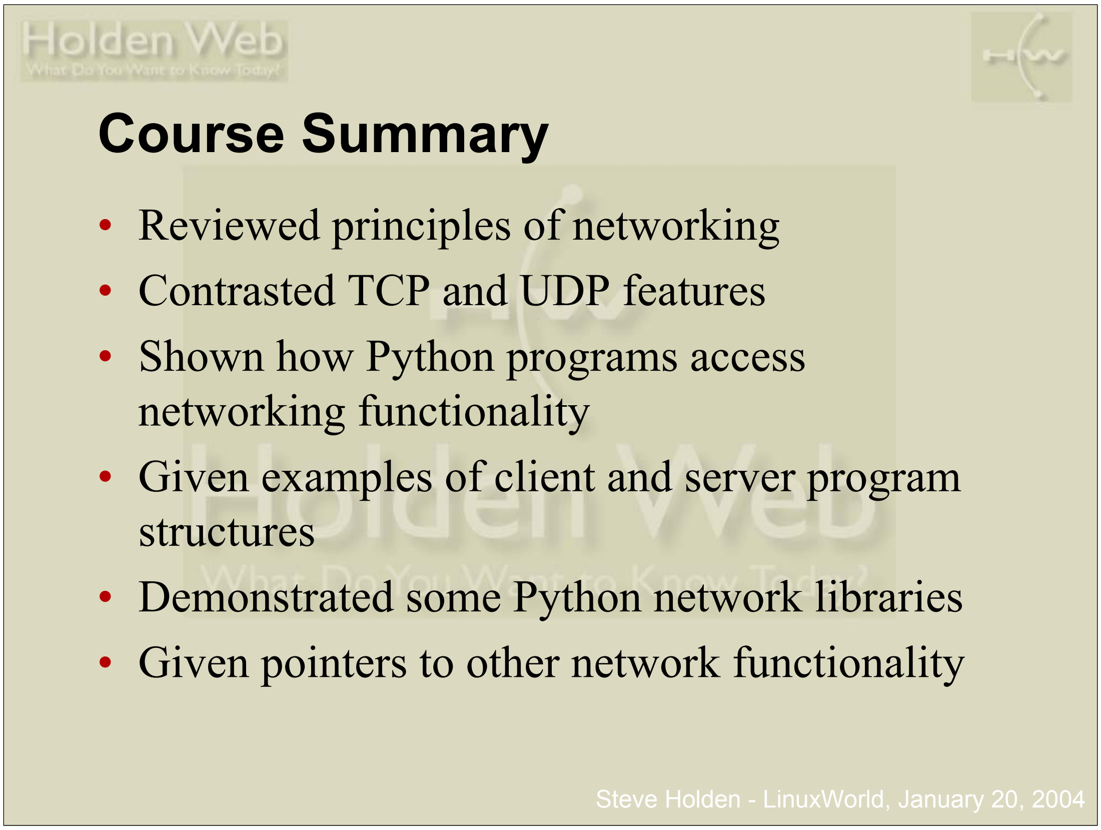
Sample CGI Script

```
#!/usr/bin/python
import cgi, cgitb; cgitb.enable()
fields = ["subnum", "reviewer", "comments"]

form = cgi.FieldStorage()
vlist = []
for f in fields:
  vlist.append("%s=%s" % (f, form.getfirst(f)))

print pgtmpl = """Content-Type: text/html

<html><head><title>Hello!</title></head>
%s
</body></html>
""" % "<br>".join(vlist)
```

This is about as short and simple as a CGI can be. It dumps the value of three named form fields to the page output.

Note that like all CGIs it has to produce appropriate HTTP headers. The Python triple-quote notation is very useful for generating string constants with multi-line values.

You also see here a couple of examples of string formatting, modelled after the UNIX C library's **`sprintf`** family of functions. The left-hand operand of the **`%`** operator is the format string, the right-hand operand is a tuple of values to be substituted. If you only have one value for substitution you don't even need to use a tuple, as the second example at the end of the code shows.



Course Summary

- Reviewed principles of networking
- Contrasted TCP and UDP features
- Shown how Python programs access networking functionality
- Given examples of client and server program structures
- Demonstrated some Python network libraries
- Given pointers to other network functionality

Need more training?
Want to offer feedback?

Need help with a Python project?
Got budget?

Get in touch:

sholden@holdenweb.com

Steve Holden - LinuxWorld, January 20, 2004

If you have been, thanks for listening!

Other sources of information and assistance to Python users:

`www.python.org`	Main Python web site and a pointer to many other useful resources
`comp.lang.python`, or **`python-list@python.org`**	Relatively friendly newsgroup/mailing list with a very inventive and helpful readership
`tutor@python.org`	Mailing list especially for learners
`http://www.amk.ca/python/howto/sockets/`	More background on socket programming in Python

I spend a lot of time explaining technologies, many of them open source, and helping clients to implement them to solve their problems.

If I can help you, or if you have further questions, please feel free to mail me at

`sholden@holdenweb.com`