### **Python Network Programming**

David M. Beazley <a href="http://www.dabeaz.com">http://www.dabeaz.com</a>

Edition: Thu Jun 17 19:49:58 2010

Copyright (C) 2010 David M Beazley All Rights Reserved

#### **Python Network Programming : Table of Contents**

1. Network Fundamentals	4
2. Client Programming	32
3. Internet Data Handling	49
4. Web Programming Basics	65
5. Advanced Networks	93

Edition: Thu Jun 17 19:49:58 2010

Slide Title Index		Threaded Server	1-50
		Forking Server (Unix)	1-51
		Asynchronous Server	1-52
0. Introduction		Utility Functions	1-53
		Omissions	1-54
		Discussion	1-55
Introduction	0-1		
Support Files	0-2	2. Client Programming	
Python Networking	0-3	_, , , , , , , , , , , , , , , , , , ,	
This Course	0 - 4		
Standard Library	0-5	Client Programming	2-1
Prerequisites	0-6	Overview	2-2
		urllib Module	2-3
1. Network Fundamentals		urllib protocols	2-5
101 (CC ) OIR I diladificitation		HTML Forms	2-6
		Web Services	2-8
Network Fundamentals	1-1	Parameter Encoding	2-9
The Problem	1-2	Sending Parameters	2-10
Two Main Issues	1-3	Response Data	2-12
Network Addressing	1-4	Response Headers	2-13
Standard Ports	1-5	Response Status	2-14
Using netstat	1-6	Exercise 2.1	2-15
Connections	1-7	urllib Limitations	2-16
Client/Server Concept	1-8	urllib2 Module	2-17
Request/Response Cycle	1-9	urllib2 Example	2-18
Using Telnet	1-10	urllib2 Requests	2-19
Data Transport	1-11	Requests with Data	2-20
Sockets	1-12	Request Headers	2-21
Socket Basics	1-13	urllib2 Error Handling	2-22
Socket Types	1-14	urllib2 Openers	2-23
Using a Socket	1-15	urllib2 build_opener()	2-24
TCP Client	1-16	Example : Login Cookies	2-25
Exercise 1.1	1-17	Discussion	2-26
Server Implementation	1-18	Exercise 2.2	2-27
TCP Server	1-19	Limitations	2-28
Exercise 1.2	1-27	ftplib	2-29
Advanced Sockets	1-28	Upload to a FTP Server	2-30
Partial Reads/Writes	1-29	httplib	2-31
Sending All Data	1-31	smtplib	2-32
End of Data	1-32	Exercise 2.3	2-33
Data Reassembly	1-33		
Timeouts	1-34	3. Internet Data Handling	
Non-blocking Sockets	1-35	or meet neer Data Transaming	
Socket Options	1-36		
Sockets as Files	1-37	Internet Data Handling	3-1
Exercise 1.3	1-39	Overview	3-2
Odds and Ends	1-40	CSV Files	3-3
UDP : Datagrams	1-41	Parsing HTML	3-4
UDP Server	1-42	Running a Parser	3-6
UDP Client	1-43	HTML Example	3-7
Unix Domain Sockets	1-44	XML Parsing with SAX	3-9
Raw Sockets	1-45	Brief XML Refresher	3-10
Sockets and Concurrency	1-46	SAX Parsing	3-11

Exercise 3.1	3-13	WSGI Example	4-37
XML and ElementTree	3-14	WSGI Applications	4-38
etree Parsing Basics	3-15	WSGI Environment	4-39
Obtaining Elements	3-17	Processing WSGI Inputs	4-41
Iterating over Elements	3-18	WSGI Responses	4-42
Element Attributes	3-19	WSGI Content	4 - 44
Search Wildcards	3-20	WSGI Content Encoding	4-45
cElementTree	3-22	WSGI Deployment	4-46
Tree Modification	3-23	WSGI and CGI	4-48
Tree Output	3-24	Exercise 4.5	4-49
Iterative Parsing	3-25	Customized HTTP	4-50
Exercise 3.2	3-28	Exercise 4.6	4-53
JSON	3-29	Web Frameworks	4-54
Sample JSON File	3-30	Commentary	4-56
Processing JSON Data	3-31		
Exercise 3.3	3-32	5. Advanced Networking	
4. Web Programming			
		Advanced Networking	5-1
		Overview	5-2
Web Programming Basics	4 - 1	Problem with Sockets	5-3
Introduction	4-2	SocketServer	5-4
Overview	4-3	SocketServer Example	5-5
Disclaimer	4 - 4	Execution Model	5-11
HTTP Explained	4-5	Exercise 5.1	5-12
HTTP Client Requests	4-6	Big Picture	5-13
HTTP Responses	4-7	Concurrent Servers	5-14
HTTP Protocol	4-8	Server Mixin Classes	5-15
Content Encoding	4-9	Server Subclassing	5-16
Payload Packaging	4-10	Exercise 5.2	5-17
Exercise 4.1	4-11	Distributed Computing	5-18
Role of Python	4-12	Discussion	5-19
Typical Python Tasks	4-13	XML-RPC	5-20
Content Generation	4-14	Simple XML-RPC	5-21
Example : Page Templates	4-15	XML-RPC Commentary	5-23
Commentary	4-17	XML-RPC and Binary	5-24
Exercise 4.2	4-18	Exercise 5.3	5-25
HTTP Servers	4-19	Serializing Python Objects	5-26
A Simple Web Server	4-20	pickle Module	5-27
Exercise 4.3	4-21	Pickling to Strings	5-28
A Web Server with CGI	4-22	Example	5-29
CGI Scripting	4-23	Miscellaneous Comments	5-31
CGI Example	4-24	Exercise 5.4	5-32
CGI Mechanics	4-27	multiprocessing	5-33
Classic CGI Interface	4-28	Connections	5-34
CGI Query Variables	4-29	Connection Use	5-35
cgi Module	4-30	Example	5-36
CGI Responses	4-31	Commentary	5-38
Note on Status Codes	4-32	What about	5-40
CGI Commentary	4-33	Network Wrap-up	5-41
Exercise 4.4	4-34	Exercise 5.5	5-42
WSGI	4-35		
WSGI Interface	4-36		

#### Section 0

### Introduction

# Support Files

Course exercises:

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

- This zip file should be downloaded and <u>extracted</u> 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++

Copyright (C) 2010, http://www.dabeaz.com

I- 3

I- 4

#### 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

Copyright (C) 2010, http://www.dabeaz.com

# 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

Copyright (C) 2010, http://www.dabeaz.com

I- 5

### Prerequisites

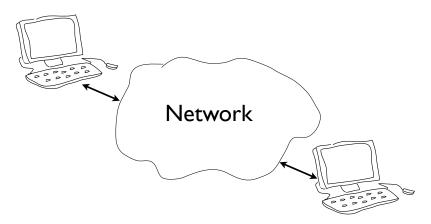
- You should already know Python basics
- However, you don't need to be an expert on all of its advanced features (in fact, none of the code to be written is highly sophisticated)
- You should have some prior knowledge of systems programming and network concepts

#### Section I

### Network Fundamentals

### The Problem

Communication between computers



• It's just sending/receiving bits

Copyright (C) 2010, http://www.dabeaz.com

### Two Main Issues

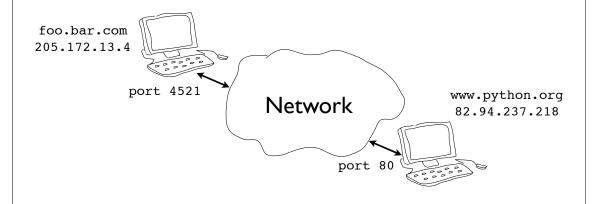
- Addressing
  - Specifying a remote computer and service
- Data transport
  - Moving bits back and forth

Copyright (C) 2010, http://www.dabeaz.com

I- 3

### Network Addressing

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



Copyright (C) 2010, http://www.dabeaz.com

#### 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

Copyright (C) 2010, http://www.dabeaz.com

I- 5

### 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
                                               * : *
               0 *:pop3s

0 localhost:mysql

0 *:pop3

0 *:imap2

0 *:8880

0 *:www

0 192.168.119.139:domain
       0
                                               *:*
tcp
tcp
          0
tcp
          0
tcp
          0
tcp
          0
tcp
tcp
          0
                  0 localhost:domain
                                                *:*
tcp
                   0 *:ssh
                                               * • *
tcp
```

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

Copyright (C) 2010, http://www.dabeaz.com

#### **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)
```

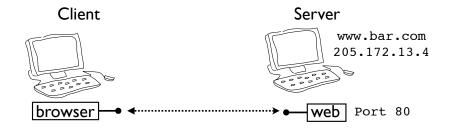
 In almost all of the network programs you'll write, you use this convention to specify a network address

Copyright (C) 2010, http://www.dabeaz.com

I- 7

### 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



Copyright (C) 2010, http://www.dabeaz.com

# 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

Copyright (C) 2010, http://www.dabeaz.com

1- 9

### Using Telnet

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

telnet hostname portnum

Example:

```
shell % telnet www.python.org 80

Trying 82.94.237.218...
Connected to www.python.org.

type this and press —> GET /index.html HTTP/1.0

return a few times 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
```

Copyright (C) 2010, http://www.dabeaz.com

1- 10

### Data Transport

- There are two basic types of communication
- <u>Streams (TCP)</u>: 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.
- <u>Datagrams (UDP)</u>: Computers send discrete packets (or messages) to each other. Each packet contains a collection of bytes, but each packet is separate and self-contained.

Copyright (C) 2010, http://www.dabeaz.com

1- 11

#### 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

Copyright (C) 2010, http://www.dabeaz.com

1- 12

#### **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)
```

Copyright (C) 2010, http://www.dabeaz.com

I- I3

# 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)
```

1- 14

# 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

Copyright (C) 2010, http://www.dabeaz.com

1- 15

1- 16

#### 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()
```

s.connect(addr) makes a connection

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

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

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 1.1

Time: 10 Minutes

Copyright (C) 2010, http://www.dabeaz.com

1- 17

# 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

Copyright (C) 2010, http://www.dabeaz.com

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 nost.
%
Server message
```

Copyright (C) 2010, http://www.dabeaz.com

I- I9

#### 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()
```

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
```

Copyright (C) 2010, http://www.dabeaz.com

Start listening for connections

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

Copyright (C) 2010, http://www.dabeaz.com

1-21

#### TCP Server

Accepting a new connection

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

Copyright (C) 2010, http://www.dabeaz.com

Client socket and address

```
from socket import *
s = socket(AF INET, SOCK STREAM)
s.bind(("",9000))
s.listen(5)
                   Accept returns a pair (client socket,addr)
while True
     c,a = s.accept()
     print "Received connection from", a
      .send("Hello %s\n" % a[0])
       .close()
<socket. socketobject</pre>
                              ("104.23.11.4", 27743)
 object at 0x3be30>
                                This is the network/port
 This is a new socket
                                address of the client that
  that's used for data
                                       connected
```

Copyright (C) 2010, http://www.dabeaz.com

I- 23

### **TCP Server**

Sending data

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

Copyright (C) 2010, http://www.dabeaz.com

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

Copyright (C) 2010, http://www.dabeaz.com

I- 25

#### TCP Server

Waiting for the next connection

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

Copyright (C) 2010, http://www.dabeaz.com

### Exercise 1.2

Time: 20 Minutes

Copyright (C) 2010, http://www.dabeaz.com

I- 27

### **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

Copyright (C) 2010, http://www.dabeaz.com

I- 29

#### 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)

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.

Copyright (C) 2010, http://www.dabeaz.com

### 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.)

Copyright (C) 2010, http://www.dabeaz.com

1- 31

### 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.

Copyright (C) 2010, http://www.dabeaz.com

I- 33

#### **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)
```

Copyright (C) 2010, http://www.dabeaz.com

# Non-blocking Sockets

Instead of timeouts, can set non-blocking

```
>>> s.setblocking(False)
```

 Future send(),recv() operations will raise an exception if the operation would have blocked

```
>>> s.setblocking(False)
>>> s.recv(1000) 
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
socket.error: (35, 'Resource temporarily unavailable')
>>> s.recv(1000) 
'Hello World\n'
>>>
Data arrived
```

Sometimes used for polling

Copyright (C) 2010, http://www.dabeaz.com

I- 35

I- 36

### Socket Options

- Sockets have a large number of parameters
- Can be set using s.setsockopt()
- Example: Reusing the port number

```
>>> s.bind(("",9000))
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   File "<string>", line 1, in bind
socket.error: (48, 'Address already in use')
>>> s.setsockopt(socket.SOL_SOCKET,
... socket.SO_REUSEADDR, 1)
>>> s.bind(("",9000))
>>>
```

Consult reference for more options

Copyright (C) 2010, http://www.dabeaz.com

#### Sockets as Files

 Sometimes it is easier to work with sockets represented as a "file" object

```
f = s.makefile()
```

This will wrap a socket with a file-like API

```
f.read()
f.readline()
f.write()
f.writelines()
for line in f:
    ...
f.close()
```

Copyright (C) 2010, http://www.dabeaz.com

I- 37

#### Sockets as Files

- Commentary: From personal experience, putting a file-like layer over a socket rarely works as well in practice as it sounds in theory.
- Tricky resource management (must manage both the socket and file independently)
- It's easy to write programs that mysteriously "freeze up" or don't operate quite like you would expect.

#### Exercise 1.3

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

I- 39

### Odds and Ends

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

Copyright (C) 2010, http://www.dabeaz.com

# **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.)

Copyright (C) 2010, http://www.dabeaz.com

1- 41

#### **UDP** Server

A simple datagram server

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

Copyright (C) 2010, http://www.dabeaz.com

#### **UDP** Client

Sending a datagram to a server

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

Copyright (C) 2010, http://www.dabeaz.com

I- 43

#### **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

Copyright (C) 2010, http://www.dabeaz.com

#### Raw Sockets

- If you have root/admin access, can gain direct access to raw network packets
- Depends on the system
- Example: Linux packet sniffing

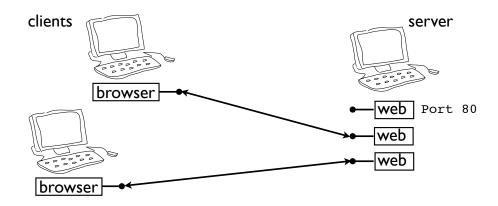
```
s = socket(AF_PACKET, SOCK_DGRAM)
s.bind(("eth0",0x0800))  # Sniff IP packets
while True:
    msg,addr = s.recvfrom(4096)  # get a packet
...
```

Copyright (C) 2010, http://www.dabeaz.com

I- 45

# Sockets and Concurrency

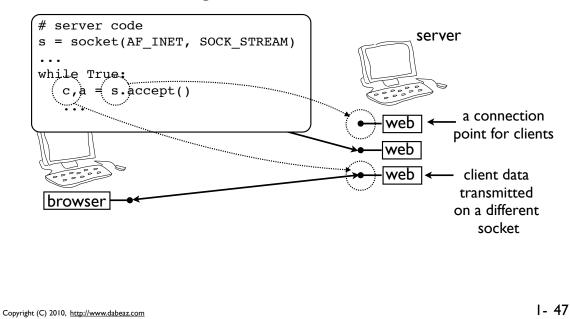
Servers usually handle multiple clients



Copyright (C) 2010, http://www.dabeaz.com

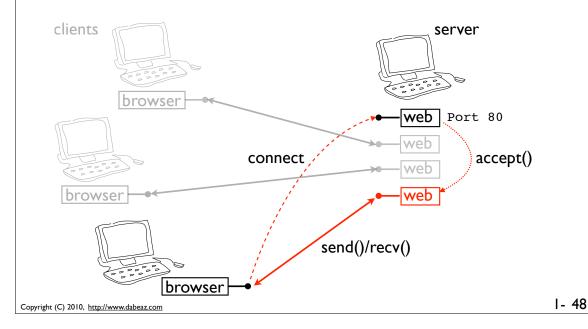
# Sockets and Concurrency

• Each client gets its own socket on server





• New connections make a new socket



# Sockets and Concurrency

- To manage multiple clients,
  - Server must always be ready to accept new connections
  - Must allow each client to operate independently (each may be performing different tasks on the server)
- Will briefly outline the common solutions

Copyright (C) 2010, http://www.dabeaz.com

1-49

#### Threaded Server

Each client is handled by a separate thread

Copyright (C) 2010, http://www.dabeaz.com

# Forking Server (Unix)

Each client is handled by a subprocess

```
import os
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    if os.fork() == 0:
        # Child process. Manage client
        ...
        c.close()
        os._exit(0)
else:
        # Parent process. Clean up and go
        # back to wait for more connections
        c.close()
```

Note: Omitting some critical details

Copyright (C) 2010, http://www.dabeaz.com

1-51

### Asynchronous Server

• Server handles all clients in an event loop

Frameworks such as Twisted build upon this

Copyright (C) 2010, http://www.dabeaz.com

# **Utility Functions**

• Get the hostname of the local machine

```
>>> socket.gethostname()
'foo.bar.com'
>>>
```

Get the IP address of a remote machine

```
>>> socket.gethostbyname("www.python.org")
'82.94.237.218'
>>>
```

• Get name information on a remote IP

```
>>> socket.gethostbyaddr("82.94.237.218")
('dinsdale.python.org', [], ['82.94.237.218'])
>>>
```

Copyright (C) 2010, http://www.dabeaz.com

I- 53

#### **Omissions**

- socket module has hundreds of obscure socket control options, flags, etc.
- Many more utility functions
- IPv6 (Supported, but new and hairy)
- Other socket types (SOCK\_RAW, etc.)
- More on concurrent programming (covered in advanced course)

#### Discussion

- It is often unnecessary to directly use sockets
- Other library modules simplify use
- However, those modules assume some knowledge of the basic concepts (addresses, ports, TCP, UDP, etc.)
- Will see more in the next few sections...

Copyright (C) 2010, http://www.dabeaz.com

I- 55

#### Section 2

# Client Programming

#### Overview

- Python has library modules for interacting with a variety of standard internet services
- HTTP, FTP, SMTP, NNTP, XML-RPC, etc.
- In this section we're going to look at how some of these library modules work
- Main focus is on the web (HTTP)

#### urllib Module

- A high level module that allows clients to connect a variety of internet services
  - HTTP
  - HTTPS
  - FTP
  - Local files
- Works with typical URLs on the web...

Copyright (C) 2010, http://www.dabeaz.com

2- 3

#### urllib Module

• Open a web page: urlopen()

```
>>> import urllib
>>> u = urllib.urlopen("http://www.python/org/index.html")
>>> data = u.read()
>>> print data
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML ...
...
>>>
```

- urlopen() returns a file-like object
- Read from it to get downloaded data

### urllib protocols

Supported protocols

```
u = urllib.urlopen("http://www.foo.com")
u = urllib.urlopen("https://www.foo.com/private")
u = urllib.urlopen("ftp://ftp.foo.com/README")
u = urllib.urlopen("file:///Users/beazley/blah.txt")
```

 Note: HTTPS only supported if Python configured with support for OpenSSL

Copyright (C) 2010, http://www.dabeaz.com

2- 5

#### **HTML Forms**

• One use of urllib is to automate forms

Your name:	
Your email:	
Subscribe	

• Example HTML source for the form

```
<FORM ACTION="/subscribe" METHOD="POST">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe">
```

Copyright (C) 2010, http://www.dabeaz.com

#### **HTML Forms**

 Within the form, you will find an action and named parameters for the form fields

```
<FORM ACTION="/subscribe" METHOD="POST">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe">
```

Action (a URL)

http://somedomain.com/subscribe

Parameters:

name email

Copyright (C) 2010, http://www.dabeaz.com

2- 7

#### Web Services

- Another use of urllib is to access web services
  - Downloading maps
  - Stock quotes
  - Email messages
- Most of these are controlled and accessed in the same manner as a form
- There is a particular request and expected set of parameters for different operations

### Parameter Encoding

- urlencode()
- Takes a dictionary of fields and creates a URL-encoded string of parameters

```
fields = {
    'name' : 'Dave',
    'email' : 'dave@dabeaz.com'
}
parms = urllib.urlencode(fields)
```

Sample result

```
>>> parms
'name=Dave&email=dave%40dabeaz.com'
>>>
```

Copyright (C) 2010, http://www.dabeaz.com

2- 9

### Sending Parameters

Case I : GET Requests

```
<FORM ACTION="/subscribe" METHOD="GET">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe";</pre>
```

• Example code:

```
fields = { ... }
parms = urllib.urlencode(fields)
u = urllib.urlopen("http://somedomain.com/subscribe?"+parms)
```

You create a long URL by concatenating the request with the parameters

http://somedomain.com/subscribe?name=Dave&email=dave%40dabeaz.com

Copyright (C) 2010, http://www.dabeaz.com

### Sending Parameters

• Case 2 : POST Requests

```
<FORM ACTION="/subscribe" METHOD="POST">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe">
```

Example code:

Copyright (C) 2010, http://www.dabeaz.com

### Response Data

 To read response data, treat the result of urlopen() as a file object

```
>>> u = urllib.urlopen("http://www.python.org")
>>> data = u.read()
>>>
```

- Be aware that the response data consists of the raw bytes transmitted
- If there is any kind of extra encoding (e.g., Unicode), you will need to decode the data with extra processing steps.

Copyright (C) 2010, http://www.dabeaz.com 2- 12

### Response Headers

HTTP headers are retrieved using .info()

```
>>> u = urllib.urlopen("http://www.python.org")
>>> headers = u.info()
>>> headers
<httplib.HTTPMessage instance at 0x1118828>
>>> headers.keys()
['content-length', 'accept-ranges', 'server',
'last-modified', 'connection', 'etag', 'date',
'content-type']
>>> headers['content-length']
'13597'
>>> headers['content-type']
'text/html'
>>>
```

A dictionary-like object

Copyright (C) 2010, http://www.dabeaz.com

2- 13

### Response Status

- urlopen() ignores HTTP status codes (i.e., errors are silently ignored)
- Can manually check the response code

```
u = urllib.urlopen("http://www.python.org/java")
if u.code == 200:
    # success
    ...
elif u.code == 404:
    # Not found!
    ...
elif u.code == 403:
    # Forbidden
```

Unfortunately a little clumsy (fixed shortly)

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 2.1

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

2- 15

#### urllib Limitations

- urllib only works with simple cases
- Does not support cookies
- Does not support authentication
- Does not report HTTP errors gracefully
- Only supports GET/POST requests

#### urllib2 Module

- urllib2 The sequel to urllib
- Builds upon and expands urllib
- Can interact with servers that require cookies, passwords, and other details
- Better error handling (uses exceptions)
- Is the preferred library for modern code

Copyright (C) 2010, http://www.dabeaz.com

2- 17

### urllib2 Example

• urllib2 provides urlopen() as before

```
>>> import urllib2
>>> u = urllib2.urlopen("http://www.python.org/index.html")
>>> data = u.read()
>>>
```

- However, the module expands functionality in two primary areas
  - Requests
  - Openers

### urllib2 Requests

Requests are now objects

```
>>> r = urllib2.Request("http://www.python.org")
>>> u = urllib2.urlopen(r)
>>> data = u.read()
```

- Requests can have additional attributes added
- User data (for POST requests)
- Customized HTTP headers

Copyright (C) 2010, http://www.dabeaz.com

2- 19

### Requests with Data

Create a POST request with user data

 Note: You still use urllib.urlencode() from the older urllib library

Copyright (C) 2010, http://www.dabeaz.com

### Request Headers

Adding/Modifying client HTTP headers

 This can be used if you need to emulate a specific client (e.g., Internet Explorer, etc.)

Copyright (C) 2010, http://www.dabeaz.com

2-21

### urllib2 Error Handling

HTTP Errors are reported as exceptions

```
>>> u = urllib2.urlopen("http://www.python.org/perl")
Traceback...
urllib2.HTTPError: HTTP Error 404: Not Found
>>>
```

• Catching an error

```
try:
    u = urllib2.urlopen(url)
except urllib2.HTTPError,e:
    code = e.code  # HTTP error code
```

 Note: urllib2 automatically tries to handle redirection and certain HTTP responses

Copyright (C) 2010, http://www.dabeaz.com 2- 22

### urllib2 Openers

- The function urlopen() is an "opener"
- It knows how to open a connection, interact with the server, and return a response.
- It only has a few basic features---it does not know how to deal with cookies and passwords
- However, you can make your own opener objects with these features enabled

Copyright (C) 2010, http://www.dabeaz.com

2- 23

### urllib2 build\_opener()

• build opener() makes an custom opener

Can add a set of new features from this list

CacheFTPHandler
HTTPBasicAuthHandler
HTTPCookieProcessor
HTTPDigestAuthHandler
ProxyHandler
ProxyBasicAuthHandler
ProxyDigestAuthHandler

Copyright (C) 2010, http://www.dabeaz.com

### Example: Login Cookies

```
fields = {
    'txtUsername' : 'dave',
    'txtPassword': '12345',
    'submit login' : 'Log In'
opener = urllib2.build_opener(
            urllib2.HTTPCookieProcessor()
         )
request = urllib2.Request(
      "http://somedomain.com/login.asp",
      urllib.urlencode(fields))
# Login
u = opener.open(request)
resp = u.read()
# Get a page, but use cookies returned by initial login
u = opener.open("http://somedomain.com/private.asp")
resp = u.read()
```

Copyright (C) 2010, http://www.dabeaz.com

2- 25

#### Discussion

- urllib2 module has a huge number of options
- Different configurations
- File formats, policies, authentication, etc.
- Will have to consult reference for everything

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 2.2

Time: 15 Minutes

Password: guido456

Copyright (C) 2010, http://www.dabeaz.com

2- 27

#### Limitations

- urllib and urllib2 are useful for fetching files
- However, neither module provides support for more advanced operations
- Examples:
  - Uploading to an FTP server
  - File-upload via HTTP Post
  - Other HTTP methods (e.g., HEAD, PUT)

### ftplib

- A module for interacting with FTP servers
- Example : Capture a directory listing

Copyright (C) 2010, http://www.dabeaz.com

2- 29

### Upload to a FTP Server

```
host = "ftp.foo.com"
username = "dave"
password = "1235"
filename = "somefile.dat"

import ftplib
ftp_serv = ftplib.FTP(host,username,password)

# Open the file you want to send
f = open(filename, "rb")

# Send it to the FTP server
resp = ftp_serv.storbinary("STOR "+filename, f)

# Close the connection
ftp_serv.close()
```

Copyright (C) 2010, http://www.dabeaz.com

### httplib

 A module for implementing the client side of an HTTP connection

```
import httplib
c = httplib.HTTPConnection("www.python.org",80)
c.putrequest("HEAD","/tut/tut.html")
c.putheader("Someheader","Somevalue")
c.endheaders()

r = c.getresponse()
data = r.read()
c.close()
```

 Low-level control over HTTP headers, methods, data transmission, etc.

Copyright (C) 2010, http://www.dabeaz.com

2-31

#### smtplib

A module for sending email messages

```
import smtplib
serv = smtplib.SMTP()
serv.connect()

msg = """\
From: dave@dabeaz.com
To: bob@yahoo.com
Subject: Get off my lawn!

Blah blah blah"""

serv.sendmail("dave@dabeaz.com",['bob@yahoo.com'],msg)
```

• Useful if you want to have a program send you a notification, send email to customers, etc.

#### Exercise 2.3

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

# Section 3 Internet Data Handling

#### Overview

- If you write network clients, you will have to worry about a variety of common file formats
- CSV, HTML, XML, JSON, etc.
- In this section, we briefly look at library support for working with such data

#### **CSV Files**

• Comma Separated Values

```
Elwood, Blues, "1060 W Addison, Chicago 60637", 110 McGurn, Jack, "4902 N Broadway, Chicago 60640", 200
```

Parsing with the CSV module

```
import csv
f = open("schmods.csv","r")
for row in csv.reader(f):
    # Do something with items in row
```

• Understands quoting, various subtle details

Copyright (C) 2010, http://www.dabeaz.com

3- 3

### Parsing HTML

- Suppose you want to parse HTML (maybe obtained via urlopen)
- Use the HTMLParser module
- A library that processes HTML using an "event-driven" programming style

### Parsing HTML

 Define a class that inherits from HTMLParser and define a set of methods that respond to different document features

```
from HTMLParser import HTMLParser
class MyParser(HTMLParser):
    def handle_starttag(self,tag,attrs):
        ...
    def handle_data(self,data):
        ...
    def handle_endtag(self,tag):
        ...

starttag
    data endttag

<tag attr="value" attr="value">data</tag>
```

Copyright (C) 2010, http://www.dabeaz.com

3- 5

### Running a Parser

 To run the parser, you create a parser object and feed it some data

```
# Fetch a web page
import urllib
u = urllib.urlopen("http://www.example.com")
data = u.read()

# Run it through the parser
p = MyParser()
p.feed(data)
```

 The parser will scan through the data and trigger the various handler methods

Copyright (C) 2010, http://www.dabeaz.com

### HTML Example

An example: Gather all links

Copyright (C) 2010, http://www.dabeaz.com

3- 7

## HTML Example

Running the parser

Copyright (C) 2010, http://www.dabeaz.com

### XML Parsing with SAX

- The event-driven style used by HTMLParser is sometimes used to parse XML
- Basis of the SAX parsing interface
- An approach sometimes seen when dealing with large XML documents since it allows for incremental processing

Copyright (C) 2010, http://www.dabeaz.com

3- 9

#### Brief XML Refresher

• XML documents use structured markup

```
<contact>
  <name>Elwood Blues</name>
  <address>1060 W Addison</address>
  <city>Chicago</city>
  <zip>60616</zip>
</contact>
```

Documents made up of elements

<name>Elwood Blues</name>

- Elements have starting/ending tags
- May contain text and other elements

### SAX Parsing

Define a special handler class

```
import xml.sax

class MyHandler(xml.sax.ContentHandler):
    def startDocument(self):
        print "Document start"

def startElement(self,name,attrs):
        print "Start:", name

def characters(self,text):
        print "Characters:", text

def endElement(self,name):
        print "End:", name
```

 In the class, you define methods that capture elements and other parts of the document

Copyright (C) 2010, http://www.dabeaz.com

3- 11

### SAX Parsing

 To parse a document, you create an instance of the handler and give it to the parser

```
# Create the handler object
hand = MyHandler()

# Parse a document using the handler
xml.sax.parse("data.xml",hand)
```

 This reads the file and calls handler methods as different document elements are encountered (start tags, text, end tags, etc.)

Copyright (C) 2010, http://www.dabeaz.com 3- 12

#### Exercise 3.1

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

3- 13

#### XML and ElementTree

- xml.etree.ElementTree module is one of the easiest ways to parse XML
- Lets look at the highlights

### etree Parsing Basics

Parsing a document

```
from xml.etree.ElementTree import parse
doc = parse("recipe.xml")
```

- This builds a complete parse tree of the entire document
- To extract data, you will perform various kinds of queries on the document object

Copyright (C) 2010, http://www.dabeaz.com

3- 15

#### etree Parsing Basics

- A mini-reference for extracting data
- Finding one or more elements

```
elem = doc.find("title")
for elem in doc.findall("ingredients/item"):
    statements
```

Element attributes and properties

Copyright (C) 2010, http://www.dabeaz.com 3- 16

### **Obtaining Elements**

```
<?xml version="1.0" encoding="iso-8859-1"?>
       <recipe>
         <title>Famous Guacamole</title>
          <description>
            A southwest favorite!
          </description>
          <ingredients>
              <item num="2">1
                                doc = parse("recipe.xml")
              <item num="1">To
              <item num="1/2"
                                desc_elem = doc.find("description")
              <item num="1" ur
                                desc text = desc elem.text
              <item num="1">J
              <item num="1" un
              <item num="3" ur
                                doc = parse("recipe.xml")
              <item num="6" ur
                                desc text = doc.findtext("description")
          </ingredients>
          <directions>
            Combine all ingredients and nand whisk to desired consistency.
            Serve and enjoy with ice-cold beers.
          </directions>
       </recipe>
                                                                             3- 17
Copyright (C) 2010, http://www.dabeaz.com
```

## Iterating over Elements

```
<?xml version="1.
                   doc = parse("recipe.xml")
<recipe>
                   for item in doc.findall("ingredients/item"):
 <title>Famous G
  <description>
                          statements
    A southwest
  </description>
  <ingredients>
      <item num="2">Large avocados, chopped</item>
      <item num="1">Tomato, chopped</item>
      <item num="1/2" units="C">White onion, chopped</item>
      <item num="1" units="tbl">Fresh squeezed lemon juice</item>
      <item num="1">Jalapeno pepper, diced</item>
      <item num="1" units="tbl">Fresh cilantro, minced</item>
      <item num="3" units="tsp">Sea Salt</item>
      <item num="6" units="bottles">Ice-cold beer</item>
  </ingredients>
    Combine all ingredients and hand whisk to desired consistency.
    Serve and enjoy with ice-cold beers.
  </directions>
</recipe>
                                                                    3- 18
```

Copyright (C) 2010, http://www.dabeaz.com

#### **Element Attributes**

```
<?xml version="1.0" encoding="iso-8859-1"?>
       <recipe>
         <title>Famous Guacamole</title>
          <description>
            A southwest favorite!
          </description>
          <ingredients>
              for item in doc.findall("ingredients/item"):
                         = item.get("num")
                                                                     </item>
                  units = item.get("units")
              <item num="/1" units="tbl">Fresh cilantro, minced</item>
              <item num='3" units="tsp">Sea Salt</item>
              <item num="6" units="bottles">Ice-cold beer</item>
          </ingredients>
          <directions>
            Combine all ingredients and hand whisk to desired consistency.
            Serve and enjoy with ice-cold beers.
          </directions>
       </recipe>
                                                                              3-19
Copyright (C) 2010, http://www.dabeaz.com
```

#### Search Wildcards

Specifying a wildcard for an element name

```
items = doc.findall("*/item")
items = doc.findall("ingredients/*")
```

- The \* wildcard only matches a single element
- Use multiple wildcards for nesting

Copyright (C) 2010, http://www.dabeaz.com

#### Search Wildcards

Wildcard for multiple nesting levels (//)

```
items = doc.findall("//item")
```

More examples

Copyright (C) 2010, http://www.dabeaz.com

3- 21

#### cElementTree

 There is a C implementation of the library that is significantly faster

```
import xml.etree.cElementTree
doc = xml.etree.cElementTree.parse("data.xml")
```

- For all practical purposes, you should use this version of the library given a choice
- Note: The C version lacks a few advanced customization features, but you probably won't need them

Copyright (C) 2010, http://www.dabeaz.com 3- 22

#### Tree Modification

- ElementTree allows modifications to be made to the document structure
- To add a new child to a parent node

```
node.append(child)
```

To insert a new child at a selected position

```
node.insert(index,child)
```

• To remove a child from a parent node

```
node.remove(child)
```

Copyright (C) 2010, http://www.dabeaz.com

3- 23

### Tree Output

- If you modify a document, it can be rewritten
- There is a method to write XML

```
doc = xml.etree.ElementTree.parse("input.xml")
# Make modifications to doc
...
# Write modified document back to a file
f = open("output.xml","w")
doc.write(f)
```

Individual elements can be turned into strings

```
s = xml.etree.ElementTree.tostring(node)
```

### Iterative Parsing

An alternative parsing interface

```
from xml.etree.ElementTree import iterparse
parse = iterparse("file.xml", ('start','end'))

for event, elem in parse:
    if event == 'start':
        # Encountered an start <tag ...>
        ...
    elif event == 'end':
        # Encountered an end </tag>
        ...
```

- This sweeps over an entire XML document
- Result is a sequence of start/end events and element objects being processed

Copyright (C) 2010, http://www.dabeaz.com

3- 25

### Iterative Parsing

- If you combine iterative parsing and tree modification together, you can process large XML documents with almost no memory overhead
- Programming interface is significantly easier to use than a similar approach using SAX
- General idea: Simply throw away the elements no longer needed during parsing

Copyright (C) 2010, http://www.dabeaz.com 3- 26

### Iterative Parsing

Programming pattern

The last step is the critical part

Copyright (C) 2010, http://www.dabeaz.com

3- 27

#### Exercise 3.2

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

### JSON

- Javascript Object Notation
- A data encoding commonly used on the web when interacting with Javascript
- Sometime preferred over XML because it's less verbose and faster to parse
- Syntax is almost identical to a Python dict

Copyright (C) 2010, http://www.dabeaz.com

3- 29

### Sample JSON File

Copyright (C) 2010, http://www.dabeaz.com

### Processing JSON Data

• Parsing a JSON document

```
import json
doc = json.load(open("recipe.json"))
```

Result is a collection of nested dict/lists

```
ingredients = doc['recipe']['ingredients']
for item in ingredients:
    # Process item
```

Dumping a dictionary as JSON

```
f = open("file.json","w")
json.dump(doc,f)
```

Copyright (C) 2010, http://www.dabeaz.com

3- 31

#### Exercise 3.3

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

#### Section 4

## Web Programming Basics

#### Introduction

- The web is (obviously) so pervasive, knowing how to write simple web-based applications is basic knowledge that all programmers should know about
- In this section, we cover the absolute basics of how to make a Python program accessible through the web

#### Overview

- Some basics of Python web programming
- HTTP Protocol
- CGI scripting
- WSGI (Web Services Gateway Interface)
- Custom HTTP servers

Copyright (C) 2010, http://www.dabeaz.com

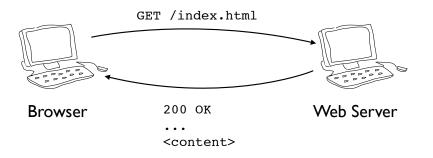
4- 3

#### Disclaimer

- Web programming is a huge topic that could span an entire multi-day class
- It might mean different things
  - Building an entire website
  - Implementing a web service
- Our focus is on some basic mechanisms found in the Python standard library that all Python programmers should know about

## HTTP Explained

- HTTP is the underlying protocol of the web
- Consists of requests and responses



Copyright (C) 2010, http://www.dabeaz.com

4- 5

## HTTP Client Requests

• Client (Browser) sends a request

```
GET /index.html HTTP/1.1

Host: www.python.org

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-U

Accept: text/xml,application/xml,application/xhtml+xml,text

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

<base>blank line>
```

 Request line followed by headers that provide additional information about the client

Copyright (C) 2010, http://www.dabeaz.com

## **HTTP** Responses

Server sends back a response

# HTTP/1.1 200 OK Date: Thu, 26 Apr 2007 19:54:01 GMT Server: Apache/2.0.54 (Debian GNU/Linux) DAV/2 SVN/1.1.4 mc Last-Modified: Thu, 26 Apr 2007 18:40:24 GMT Accept-Ranges: bytes Content-Length: 14315 Connection: close Content-Type: text/html

 Response line followed by headers that further describe the response contents

Copyright (C) 2010, http://www.dabeaz.com

4- 7

#### **HTTP Protocol**

There are a small number of request types

GET POST HEAD

There are standardized response codes

200 OK 403 Forbidden 404 Not Found 501 Not implemented

But, this isn't an exhaustive tutorial

,

## Content Encoding

Content is described by these header fields:

```
Content-type:
Content-length:
```

Example:

```
Content-type: image/jpeg
Content-length: 12422
```

- Of these, Content-type is the most critical
- Length is optional, but it's polite to include it if it can be determined in advance

Copyright (C) 2010, http://www.dabeaz.com

4- 9

## Payload Packaging

Responses must follow this formatting

```
Headers
...
Content-type: image/jpeg
Content-length: 12422
...
\r\n (Blank Line)

Content
(12422 bytes)
```

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 4.1

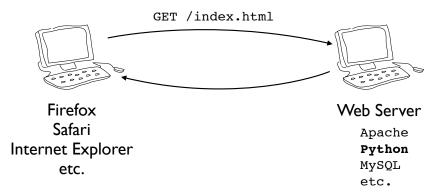
Time: 10 Minutes

Copyright (C) 2010, http://www.dabeaz.com

4- 11

## Role of Python

 Most web-related Python programming pertains to the operation of the server



 Python scripts used on the server to create, manage, or deliver content back to clients

Copyright (C) 2010, http://www.dabeaz.com

## Typical Python Tasks

- <u>Static content generation</u>. One-time generation of static web pages to be served by a standard web server such as Apache.
- <u>Dynamic content generation</u>. Python scripts that produce output in response to requests (e.g., form processing, CGI scripting).

Copyright (C) 2010, http://www.dabeaz.com

4- 13

#### Content Generation

- It is often overlooked, but Python is a useful tool for simply creating static web pages
- Example : Taking various pages of content, adding elements, and applying a common format across all of them.
- Web server simply delivers all of the generated content as normal files

## Example: Page Templates

Create a page "template" file

```
<html>
        <body>
          template.html
            Your Logo: Navigation Links
                                             Your Logo: Navigation Links
            <hr>
                                             Scontent.
            Note the
            Copyright (C) 2008
  special -
           → $content
 $variable
            <em>Copyright (C) 2008</em>
            </body>
      </html>
                                                                   4- 15
Copyright (C) 2010, http://www.dabeaz.com
```

## Example: Page Templates

• Use template strings to render pages

```
from string import Template

# Read the template string
pagetemplate = Template(open("template.html").read())

# Go make content
page = make_content()

# Render the template to a file
f = open(outfile,"w")
f.write(pagetemplate.substitute(content=page))
```

 Key idea: If you want to change the appearance, you just change the template

Copyright (C) 2010, http://www.dabeaz.com 4- 16

## Commentary

- Using page templates to generate static content is extremely common
- For simple things, just use the standard library modules (e.g., string.Template)
- For more advanced applications, there are numerous third-party template packages

Copyright (C) 2010, http://www.dabeaz.com

4- 17

#### Exercise 4.2

Time: 10 Minutes

Copyright (C) 2010, http://www.dabeaz.com

#### **HTTP Servers**

- Python comes with libraries that implement simple self-contained web servers
- Very useful for testing or special situations where you want web service, but don't want to install something larger (e.g., Apache)
- Not high performance, sometimes "good enough" is just that

Copyright (C) 2010, http://www.dabeaz.com

4- 19

## A Simple Web Server

• Serve files from a directory

```
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
import os
os.chdir("/home/docs/html")
serv = HTTPServer(("",8080),SimpleHTTPRequestHandler)
serv.serve_forever()
```

- This creates a minimal web server
- Connect with a browser and try it out

#### Exercise 4.3

Time: 10 Minutes

Copyright (C) 2010, http://www.dabeaz.com

4- 21

#### A Web Server with CGI

Serve files and allow CGI scripts

```
from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler
import os
os.chdir("/home/docs/html")
serv = HTTPServer(("",8080),CGIHTTPRequestHandler)
serv.serve_forever()
```

 Executes scripts in "/cgi-bin" and "/htbin" directories in order to create dynamic content

Copyright (C) 2010, http://www.dabeaz.com

## CGI Scripting

- <u>Common Gateway Interface</u>
- A common protocol used by existing web servers to run server-side scripts, plugins
- Example: Running Python, Perl, Ruby scripts under Apache, etc.
- Classically associated with form processing, but that's far from the only application

Copyright (C) 2010, http://www.dabeaz.com

4- 23

## CGI Example

A web-page might have a form on it

Your name:	
Your email:	
Subscribe	

• Here is the underlying HTML code

```
<FORM ACTION="/cgi-bin/subscribe.py" METHOD="POST">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe">
```

Specifies a CGI program on the server

Copyright (C) 2010, http://www.dabeaz.com 4- 24

## CGI Example

Forms have submitted fields or parameters

```
<FORM ACTION="/cgi-bin/subscribe.py" METHOD="POST">
Your name: <INPUT type="text" name="name" size="30"><br>
Your email: <INPUT type="text" name="email" size="30"><br>
<INPUT type="submit" name="submit-button" value="Subscribe">
```

 A request will include both the URL (cgi-bin/ subscribe.py) along with the field values

Copyright (C) 2010, http://www.dabeaz.com

4- 25

## CGI Example

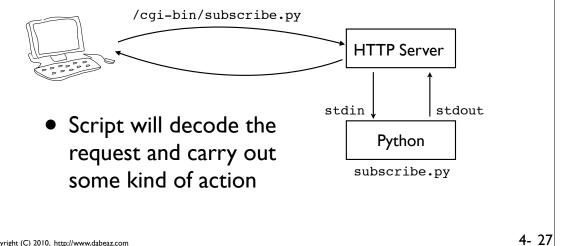
Request encoding looks like this:

```
Request 
POST /cgi-bin/subscribe.py HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS Accept: text/xml,application/xml,application/xhtml Accept-Language: en-us,en;q=0.5
...
```

- Request tells the server what to run
- Query string contains encoded form fields

#### **CGI** Mechanics

 CGI was originally implemented as a scheme for launching processing scripts as a subprocess to a web server



#### Classic CGI Interface

• Server populates environment variables with information about the request

```
import os
os.environ['SCRIPT NAME']
os.environ['REMOTE_ADDR']
os.environ['QUERY_STRING']
os.environ['REQUEST METHOD']
os.environ['CONTENT TYPE']
os.environ['CONTENT LENGTH']
os.environ['HTTP_COOKIE']
```

stdin/stdout provide I/O link to server

```
sys.stdin
                  # Read to get data sent by client
                 # Write to create the response
sys.stdout
```

Copyright (C) 2010, http://www.dabeaz.com

Copyright (C) 2010, http://www.dabeaz.com

## CGI Query Variables

For GET requests, an env. variable is used

```
query = os.environ['QUERY STRING']
```

For POST requests, you read from stdin

```
if os.environ['REQUEST_METHOD'] == 'POST':
    size = int(os.environ['CONTENT_LENGTH'])
    query = sys.stdin.read(size)
```

This yields the raw query string

```
name=David+Beazley&email=dave
%40dabeaz.com&submit-button=Subscribe
```

Copyright (C) 2010, http://www.dabeaz.com

4- 29

## cgi Module

- A utility library for decoding requests
- Major feature: Getting the passed parameters

- All CGI scripts start like this
- FieldStorage parses the incoming request into a dictionary-like object for extracting inputs

Copyright (C) 2010, http://www.dabeaz.com

## **CGI** Responses

 CGI scripts respond by simply printing response headers and the raw content

```
name = form.getvalue('name')
email = form.getvalue('email')
... do some kind of processing ...

# Output a response
print "Status: 200 OK"
print "Content-type: text/html"
print
print "<html><head><title>Success!</title></head><body>"
print "Hello %s, your email is %s" % (name,email)
print "</body>"
```

 Normally you print HTML, but any kind of data can be returned (for web services, you might return XML, JSON, etc.)

Copyright (C) 2010, http://www.dabeaz.com

4-31

#### Note on Status Codes

 In CGI, the server status code is set by including a special "Status:" header field

```
import cgi
form = cgi.FieldStorage()
name = form.getvalue('name')
email = form.getvalue('email')

...

print "Status: 200 OK"
print "Content-type: text/html"
print
print "<html><head><title>Success!</title></head><body>"
print "Hello %s, your email is %s" % (name,email)
print "</body>"
```

 This is a special server directive that sets the response status

Copyright (C) 2010, http://www.dabeaz.com 4- 32

# **CGI** Commentary

- There are many more minor details (consult a reference on CGI programming)
- The basic idea is simple
  - Server runs a script
  - Script receives inputs from environment variables and stdin
  - Script produces output on stdout
- It's old-school, but sometimes it's all you get

Copyright (C) 2010, http://www.dabeaz.com

4- 33

#### Exercise 4.4

Time: 25 Minutes

#### **WSGI**

- Web Services Gateway Interface (WSGI)
- This is a standardized interface for creating Python web services
- Allows one to create code that can run under a wide variety of web servers and frameworks as long as they also support WSGI (and most do)
- So, what is WSGI?

Copyright (C) 2010, http://www.dabeaz.com

4- 35

#### WSGI Interface

- WSGI is an application programming interface loosely based on CGI programming
- In CGI, there are just two basic features
  - Getting values of inputs (env variables)
  - Producing output by printing
- WSGI takes this concept and repackages it into a more modular form

## WSGI Example

- With WSGI, you write an "application"
- An application is just a function (or callable)

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    response.append("You requested :"+environ['PATH_INFO]')
    return response
```

 This function encapsulates the handling of some request that will be received

Copyright (C) 2010, http://www.dabeaz.com

4- 37

## WSGI Applications

Applications always receive just two inputs

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    response.append("You requested :"+environ['PATH_INFO]')
    return response
```

- environ A <u>dictionary</u> of input parameters
- start\_response A callable (e.g., function)

#### **WSGI** Environment

The environment contains CGI variables

```
def hello_app(environ, start_response):
    status = "200 OK"
    response headers = [ ('Content-type','text/plain')]

environ['REQUEST_METHOD']
    environ['SCRIPT_NAME']
    environ['PATH_INFO']
    environ['QUERY_STRING']
    environ['CONTENT_TYPE']
    environ['CONTENT_LENGTH']
    environ['SERVER_NAME']
    ...
```

 The meaning and values are exactly the same as in traditional CGI programs

Copyright (C) 2010, http://www.dabeaz.com

4-39

#### **WSGI** Environment

Environment also contains some WSGI variables

```
def hello_app(environ, start_response):
    status = "200 OK"
    response headers = [ ('Content-type','text/plain')]

environ['wsgi.input']
    environ['wsgi.errors']
    environ['wsgi.url_scheme']
    environ['wsgi.multithread']
    environ['wsgi.multithread']
    environ['wsgi.multiprocess']
...
```

- wsgi.input A file-like object for reading data
- wsgi.errors File-like object for error output

Copyright (C) 2010, http://www.dabeaz.com

## **Processing WSGI Inputs**

Parsing of query strings is similar to CGI

 You use FieldStorage() as before, but give it extra parameters telling it where to get data

Copyright (C) 2010, http://www.dabeaz.com

4-41

## WSGI Responses

 The second argument is a function that is called to initiate a response

- You pass it two parameters
  - A status string (e.g., "200 OK")
  - A list of (header, value) HTTP header pairs

Copyright (C) 2010, http://www.dabeaz.com

## **WSGI** Responses

- start\_response() is a hook back to the server
- Gives the server information for formulating the response (status, headers, etc.)
- Prepares the server for receiving content data

Copyright (C) 2010, http://www.dabeaz.com

4- 43

#### **WSGI** Content

• Content is returned as a sequence of byte strings

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    response.append("You requested :"+environ['PATH_INFO]')
    return response
```

 Note: This differs from CGI programming where you produce output using print.

## WSGI Content Encoding

- WSGI applications must always produce bytes
- If working with Unicode, it must be encoded

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/html')]
    start_response(status,response_headers)
    return [u"That's a spicy Jalape\u00f1o".encode('utf-8')]
```

 This is a little tricky--if you're not anticipating Unicode, everything can break if a Unicode string is returned (be aware that certain modules such as database modules may do this)

Copyright (C) 2010, http://www.dabeaz.com

4- 45

## WSGI Deployment

- The main point of WSGI is to simplify deployment of web applications
- You will notice that the interface depends on no third party libraries, no objects, or even any standard library modules
- That is intentional. WSGI apps are supposed to be small self-contained units that plug into other environments

## WSGI Deployment

Running a simple stand-alone WSGI server

```
from wsgiref import simple_server
httpd = simple_server.make_server("",8080,hello_app)
httpd.serve_forever()
```

- This runs an HTTP server for testing
- You probably wouldn't deploy anything using this, but if you're developing code on your own machine, it can be useful

Copyright (C) 2010, http://www.dabeaz.com

4-47

#### WSGI and CGI

 WSGI applications can run on top of standard CGI scripting (which is useful if you're interfacing with traditional web servers).

```
#!/usr/bin/env python
# hello.py

def hello_app(environ, start_response):
    ...

import wsgiref.handlers
wsgiref.handlers.CGIHandler().run(hello_app)
```

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 4.5

Time: 20 Minutes

Copyright (C) 2010, http://www.dabeaz.com

4- 49

#### **Customized HTTP**

- Can implement customized HTTP servers
- Use BaseHTTPServer module
- Define a customized HTTP handler object
- Requires some knowledge of the underlying HTTP protocol

#### **Customized HTTP**

• Example: A Hello World Server

Defined a method for "GET" requests

Copyright (C) 2010, http://www.dabeaz.com

4- 51

#### **Customized HTTP**

A more complex server

Can customize everything (requires work)

Copyright (C) 2010, http://www.dabeaz.com 4- 52

#### Exercise 4.6

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

4- 53

#### Web Frameworks

- Python has a huge number of web frameworks
  - Zope
  - Django
  - Turbogears
  - Pylons
  - CherryPy
  - Google App Engine
- Frankly, there are too many to list here..

Copyright (C) 2010, http://www.dabeaz.com

#### Web Frameworks

- Web frameworks build upon previous concepts
- Provide additional support for
  - Form processing
  - Cookies/sessions
  - Database integration
  - Content management
- Usually require their own training course

Copyright (C) 2010, http://www.dabeaz.com

4- 55

## Commentary

- If you're building small self-contained components or middleware for use on the web, you're probably better off with WSGI
- The programming interface is minimal
- The components you create will be selfcontained if you're careful with your design
- Since WSGI is an official part of Python, virtually all web frameworks will support it

#### Section 5

## Advanced Networking

#### Overview

- An assortment of advanced networking topics
- The Python network programming stack
- Concurrent servers
- Distributed computing
- Multiprocessing

#### Problem with Sockets

- In part I, we looked at low-level programming with sockets
- Although it is possible to write applications based on that interface, most of Python's network libraries use a higher level interface
- For servers, there's the SocketServer module

Copyright (C) 2010, http://www.dabeaz.com

5- 3

#### SocketServer

- A module for writing custom servers
- Supports TCP and UDP networking
- The module aims to simplify some of the low-level details of working with sockets and put to all of that functionality in one place

Copyright (C) 2010, http://www.dabeaz.com

## SocketServer Example

- To use SocketServer, you define handler objects using classes
- Example: A time server

```
import SocketServer
import time

class TimeHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.sendall(time.ctime()+"\n")

serv = SocketServer.TCPServer(("",8000),TimeHandler)
serv.serve_forever()
```

Copyright (C) 2010, http://www.dabeaz.com

5- 5

## SocketServer Example

Handler Class

Copyright (C) 2010, http://www.dabeaz.com

## SocketServer Example

#### Handler Class

```
import SocketServer
import time

class TimeHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.sendall(time.ctime())

serv = SocketServer.TCPServer(("",8000),TimeHandler)
serv.serve forever()
```

Must inherit from

Copyright (C) 2010, http://www.dabeaz.com

5- 7

## SocketServer Example

#### • handle() method

```
import SocketServer
import time

class TimeHandler(SocketServer action

def handle(self):
    self.request.sendall(time.ctime())

serv = SocketServer.TCPServer(("",8000),TimeHandler)
serv.serve_forever()
```

Copyright (C) 2010, http://www.dabeaz.com

## SocketServer Example

Client socket connection

```
import SocketServer
import time

class TimeHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.sendall(time.ctime())

serv = SocketS
    Socket object
    for client connection
8000), TimeHandler)
```

This is a bare socket object

Copyright (C) 2010, http://www.dabeaz.com

5- 9

## SocketServer Example

Creating and running the server

Copyright (C) 2010, http://www.dabeaz.com

#### **Execution Model**

- Server runs in a loop waiting for requests
- On each connection, the server creates a new instantiation of the handler class
- The handle() method is invoked to handle the logic of communicating with the client
- When handle() returns, the connection is closed and the handler instance is destroyed

Copyright (C) 2010, http://www.dabeaz.com

5- 11

#### Exercise 5.1

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

## Big Picture

- A major goal of SocketServer is to simplify the task of plugging different server handler objects into different kinds of server implementations
- For example, servers with different implementations of concurrency, extra security features, etc.

Copyright (C) 2010, http://www.dabeaz.com

5- 13

#### **Concurrent Servers**

 SocketServer supports different kinds of concurrency implementations

```
TCPServer - Synchronous TCP server (one client)
ForkingTCPServer - Forking server (multiple clients)
ThreadingTCPServer - Threaded server (multiple clients)
```

 Just pick the server that you want and plug the handler object into it

```
serv = SocketServer.ForkingTCPServer(("",8000),TimeHandler)
serv.serve_forever()

serv = SocketServer.ThreadingTCPServer(("",8000),TimeHandler)
serv.serve_forever()
```

Copyright (C) 2010, http://www.dabeaz.com

#### Server Mixin Classes

SocketServer defines these mixin classes

```
ForkingMixIn
ThreadingMixIn
```

 These can be used to add concurrency to other server objects (via multiple inheritance)

Copyright (C) 2010, http://www.dabeaz.com

5-15

## Server Subclassing

- SocketServer objects are also subclassed to provide additional customization
- Example: Security/Firewalls

```
class RestrictedTCPServer(TCPServer):
    # Restrict connections to loopback interface
    def verify_request(self,request,addr):
        host, port = addr
        if host != '127.0.0.1':
            return False
        else:
            return True

serv = RestrictedTCPServer(("",8080),TimeHandler)
serv.serve_forever()
```

Copyright (C) 2010, http://www.dabeaz.com

#### Exercise 5.2

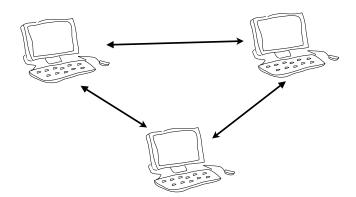
Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

5- 17

# Distributed Computing

 It is relatively simple to build Python applications that span multiple machines or operate on clusters



Copyright (C) 2010, http://www.dabeaz.com

#### Discussion

- Keep in mind: Python is a "slow" interpreted programming language
- So, we're not necessarily talking about high performance computing in Python (e.g., number crunching, etc.)
- However, Python can serve as a very useful distributed scripting environment for controlling things on different systems

Copyright (C) 2010, http://www.dabeaz.com

5-19

#### XML-RPC

- Remote Procedure Call
- Uses HTTP as a transport protocol
- Parameters/Results encoded in XML
- Supported by languages other than Python

Copyright (C) 2010, http://www.dabeaz.com

# Simple XML-RPC

How to create a stand-alone server

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x,y):
    return x+y

s = SimpleXMLRPCServer(("",8080))
s.register_function(add)
s.serve_forever()
```

How to test it (xmlrpclib)

```
>>> import xmlrpclib
>>> s = xmlrpclib.ServerProxy("http://localhost:8080")
>>> s.add(3,5)
8
>>> s.add("Hello","World")
"HelloWorld"
>>>
```

Copyright (C) 2010, http://www.dabeaz.com

5-21

# Simple XML-RPC

Adding multiple functions

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

s = SimpleXMLRPCServer(("",8080))
s.register_function(add)
s.register_function(foo)
s.register_function(bar)
s.serve_forever()
```

Registering an instance (exposes all methods)

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

s = SimpleXMLRPCServer(("",8080))
obj = SomeObject()
s.register_instance(obj)
s.serve_forever()
```

Copyright (C) 2010, http://www.dabeaz.com

# XML-RPC Commentary

- XML-RPC is extremely easy to use
- Almost too easy--you might get the perception that it's extremely limited or fragile
- I have encountered a lot of major projects that are using XML-RPC for distributed control
- Users seem to love it (I concur)

Copyright (C) 2010, http://www.dabeaz.com

5-23

# XML-RPC and Binary

- One wart of caution...
- XML-RPC assumes all strings are UTF-8 encoded Unicode
- Consequence: You can't shove a string of raw binary data through an XML-RPC call
- For binary: must base64 encode/decode
- base64 module can be used for this

### Exercise 5.3

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

5-25

# Serializing Python Objects

- In distributed applications, you may want to pass various kinds of Python objects around (e.g., lists, dicts, sets, instances, etc.)
- Libraries such as XML-RPC support simple data types, but not anything more complex
- However, serializing arbitrary Python objects into byte-strings is quite simple

Copyright (C) 2010, http://www.dabeaz.com

## pickle Module

- A module for serializing objects
- Serializing an object onto a "file"

```
import pickle
...
pickle.dump(someobj,f)
```

Unserializing an object from a file

```
someobj = pickle.load(f)
```

• Here, a file might be a file, a pipe, a wrapper around a socket, etc.

Copyright (C) 2010, http://www.dabeaz.com

5-27

5-28

# Pickling to Strings

Pickle can also turn objects into byte strings

```
import pickle
# Convert to a string
s = pickle.dumps(someobj, protocol)
...
# Load from a string
someobj = pickle.loads(s)
```

 This can be used if you need to embed a Python object into some other messaging protocol or data encoding

# Example

Using pickle with XML-RPC

```
# addserv.py
import pickle

def add(px,py):
    x = pickle.loads(px)
    y = pickle.loads(py)
    return pickle.dumps(x+y)

from SimpleXMLRPCServer import SimpleXMLRPCServer
serv = SimpleXMLRPCServer(("",15000))
serv.register_function(add)
serv.serve forever()
```

 Notice: All input arguments and return values are encoded/decoded with pickle

Copyright (C) 2010, http://www.dabeaz.com

5-29

## Example

Passing Python objects from the client

```
>>> import pickle
>>> import xmlrpclib
>>> serv = xmlrpclib.ServerProxy("http://localhost:15000")
>>> a = [1,2,3]
>>> b = [4,5]
>>> r = serv.add(pickle.dumps(a),pickle.dumps(b))
>>> c = pickle.loads(r)
>>> c
[1, 2, 3, 4, 5]
>>>
```

 Again, all input and return values are processed through pickle

### Miscellaneous Comments

- Pickle is really only useful if used in a Pythononly environment
- Would not use if you need to communicate to other programming languages
- There are also security concerns
- Never use pickle with untrusted clients (malformed pickles can be used to execute arbitrary system commands)

Copyright (C) 2010, http://www.dabeaz.com

5-31

### Exercise 5.4

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

## multiprocessing

- Python 2.6/3.0 include a new library module (multiprocessing) that can be used for different forms of distributed computation
- It is a substantial module that also addresses interprocess communication, parallel computing, worker pools, etc.
- Will only show a few network features here

Copyright (C) 2010, http://www.dabeaz.com

5- 33

### **Connections**

- Creating a dedicated connection between two Python interpreter processes
- Listener (server) process

```
from multiprocessing.connection import Listener
serv = Listener(("",16000),authkey="12345")
c = serv.accept()
```

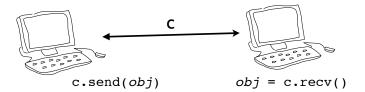
Client process

```
from multiprocessing.connection import Client
c = Client(("servername",16000),authkey="12345")
```

• On surface, looks similar to a TCP connection

## Connection Use

 Connections allow bidirectional message passing of arbitrary Python objects



- Underneath the covers, everything routes through the pickle module
- Similar to a network connection except that you just pass objects through it

Copyright (C) 2010, http://www.dabeaz.com

5- 35

# Example

• Example server using multiprocessing

```
# addserv.py

def add(x,y):
    return x+y

from multiprocessing.connection import Listener
serv = Listener(("",16000),authkey="12345")
c = serv.accept()
while True:
    x,y = c.recv()  # Receive a pair
    c.send(add(x,y))  # Send result of add(x,y)
```

 Note: Omitting a variety of error checking/ exception handling

# Example

Client connection with multiprocessing

```
>>> from multiprocessing.connection import Client
>>> client = Client(("",16000),authkey="12345")
>>> a = [1,2,3]
>>> b = [4,5]
>>> client.send((a,b))
>>> c = client.recv()
>>> c
[1, 2, 3, 4, 5]
>>>
```

 Even though pickle is being used underneath the covers, you don't see it here

Copyright (C) 2010, http://www.dabeaz.com

5-37

## Commentary

- Multiprocessing module already does the work related to pickling, error handling, etc.
- Can use it as the foundation for something more advanced
- There are many more features of multiprocessing not shown here (e.g., features related to distributed objects, parallel processing, etc.)

# Commentary

- Multiprocessing is a good choice if you're working strictly in a Python environment
- It will be faster than XML-RPC
- It has some security features (authkey)
- More flexible support for passing Python objects around

Copyright (C) 2010, http://www.dabeaz.com

5-39

## What about...

- CORBA? SOAP? Others?
- There are third party libraries for this
- Honestly, most Python programmers aren't into big heavyweight distributed object systems like this (too much trauma)
- However, if you're into distributed objects, you should probably look at the Pyro project (http://pyro.sourceforge.net)

Copyright (C) 2010, http://www.dabeaz.com

## Network Wrap-up

- Have covered the basics of network support that's bundled with Python (standard lib)
- Possible directions from here...
  - Concurrent programming techniques (often needed for server implementation)
  - Parallel computing (scientific computing)
  - Web frameworks

Copyright (C) 2010, http://www.dabeaz.com

5-41

### Exercise 5.5

Time: 15 Minutes

Copyright (C) 2010, http://www.dabeaz.com

### **Python Network Programming Index**

#### A

accept() method, of sockets, 1-19, 1-22 Address binding, TCP server, 1-20 Addressing, network, 1-4 Asynchronous network server, 1-52

#### B

BaseRequestHandler, SocketServer module, 5-5 bind() method, of sockets, 1-19, 1-20, 1-42 Browser, emulating in HTTP requests, 2-21 build\_opener() function, urllib2 module, 2-24

### $\mathbf{C}$

cElementTree module, 3-22 cgi module, 4-30 CGI scripting, 4-23, 4-24, 4-25, 4-26, 4-27 CGI scripting, and WSGI, 4-48 CGI scripting, creating a response, 4-31, 4-32 CGI scripting, environment variables, 4-28 CGI scripting, I/O model, 4-28 CGI scripting, parsing query variables, 4-30 CGI scripting, query string, 4-26 CGI scripting, query variables, 4-29 CherryPy, 4-54 Client objects, multiprocessing module, 5-34 Client/Server programming, 1-8 close() method, of sockets, 1-16, 1-25 Concurrency, and socket programming, 1-46 connect() method, of sockets, 1-16 Connections, network, 1-7 Content encoding, HTTP responses, 4-9 Cookie handling and HTTP requests, 2-25 Cookies, and urllib2 module, 2-17 CORBA, 5-40 Creating custom openers for HTTP requests, 2-24 csv module, 3-3

#### D

Datagram, 1-43 Distributed computing, 5-18, 5-19 Django, 4-54 dump() function, pickle module, 5-27 dumps() function, pickle module, 5-28

#### $\mathbf{E}$

ElementTree module, modifying document structure, 3-23
ElementTree module, performance, 3-22
ElementTree module, xml.etree package, 3-14
ElementTree, attributes, 3-19
ElementTree, incremental XML parsing, 3-25
ElementTree, wildcards, 3-20
ElementTree, writing XML, 3-24
End of file, of sockets, 1-32
environ variable, os module, 4-28
Error handling, HTTP requests, 2-22

#### F

FieldStorage object, cgi module, 4-30
File upload, via urllib, 2-28
Files, creating from a socket, 1-37
Forking server, 1-51
ForkingMixIn class, SocketServer module, 5-15
ForkingTCPServer, SocketServer module, 5-14
ForkingUDPServer, SocketServer module, 5-14
Form data, posting in an HTTP request, 2-10, 2-11, 2-20
FTP server, interacting with, 2-29
FTP, uploading files to a server, 2-30
ftplib module, 2-29

### G

gethostbyaddr() function, socket module, 1-53 gethostbyname() function, socket module, 1-53 gethostname() function, socket module, 1-53 Google AppEngine, 4-54

#### H

Hostname, 1-4 Hostname, obtaining, 1-53 HTML, parsing of, 3-4, 3-7 HTMLParser module, 3-5, 3-7 HTTP cookies, 2-25
HTTP protocol, 4-5
HTTP request, with cookie handling, 2-25
HTTP status code, obtaining with urllib, 2-14
HTTP, client-side protocol, 2-31
HTTP, methods, 4-8
HTTP, request structure, 4-6
HTTP, response codes, 4-8
HTTP, response content encoding, 4-9
HTTP, response structure, 4-7, 4-10, 4-12
httplib module, 2-31

#### Ι

Interprocess communication, 1-44 IP address, 1-4 IPC, 1-44 IPv4 socket, 1-13 IPv6 socket, 1-13

### J

JSON, 3-29 json module, 3-31

#### L

Limitations, of urllib module, 2-28 listen() method, of sockets, 1-19, 1-21 Listener objects, multiprocessing module, 5-34 load() function, pickle module, 5-27 loads() function, pickle module, 5-28

#### M

makefile() method, of sockets, 1-37 multiprocessing module, 5-33

#### N

netstat, 1-6 Network addresses, 1-4, 1-7 Network programming, client-server concept, 1-8 Network programming, standard port assignments, 1-5

### 0

Objects, serialization of, 5-26 Opener objects, urllib2 module, 2-23 OpenSSL, 2-5

#### P

Parsing HTML, 3-7
Parsing, JSON, 3-29
Parsing, of HTML, 3-5
pickle module, 5-27
POST method, of HTTP requests, 2-6, 2-7
Posting form data, HTTP requests, 2-10, 2-11, 2-20
Pylons, 4-54

### Q

Query string, and CGI scripting, 4-26

### R

Raw Sockets, 1-45 recv() method, of sockets, 1-16 recvfrom() method, of sockets, 1-42, 1-43 Request objects, urllib2 module, 2-19 Request-response cycle, network programming, 1-9 RFC-2822 headers, 4-6

### S

sax module, xml package, 3-11 select module, 1-52 select() function, select module, 1-52 send() method, of sockets, 1-16, 1-24 sendall() method, of sockets, 1-31 Sending email, 2-32 sendto() method, of sockets, 1-42, 1-43 Serialization, of Python objects, 5-26 serve\_forever() method, SocketServer, 5-5 setsockopt() method, of sockets, 1-36 settimeout() method, of sockets, 1-34 SimpleXMLRPCServer module, 5-21

simple\_server module, wsgiref package, 4-46, UDPServer, SocketServer module, 5-14 4-47 Unix domain sockets, 1-44 Uploading files, to an FTP server, 2-30 smtplib module, 2-32 URL, parameter encoding, 2-6, 2-7 SOAP, 5-40 urlencode() function, urllib module, 2-9 socket module, 1-13 urllib module, 2-3 socket() function, socket module, 1-13 Socket, using for server or client, 1-15 urllib module, limitations, 2-28 Socket, wrapping with a file object, 1-37 urllib2 module, 2-17 Sockets, 1-12, 1-13 urllib2 module, error handling, 2-22 Sockets, and concurrency, 1-46 urllib2 module, Request objects, 2-19 Sockets, asynchronous server, 1-52 urlopen() function, obtaining response headers, Sockets, end of file indication, 1-32 2 - 13Sockets, forking server example, 1-51 urlopen() function, obtaining status code, 2-14 Sockets, partial reads and writes, 1-29 urlopen() function, reading responses, 2-12 Sockets, setting a timeout, 1-34 urlopen() function, urllib module, 2-4 urlopen() function, urllib2 module, 2-18 Sockets, setting options, 1-36 Sockets, threaded server, 1-50 urlopen(), posting form data, 2-10, 2-11, 2-20 urlopen(), supported protocols, 2-5 SocketServer module, 5-4 SocketServer, subclassing, 5-16 User-agent, setting in HTTP requests, 2-21 Standard port assignments, 1-5  $\mathbf{V}$  $\mathbf{T}$ viewing open network connections, 1-6 TCP, 1-13, 1-14 TCP, accepting new connections, 1-22  $\mathbf{W}$ TCP, address binding, 1-20 TCP, client example, 1-16 Web frameworks, 4-54, 4-55 TCP, communication with client, 1-23 Web programming, and WSGI, 4-35, 4-36 TCP, example with SocketServer module, 5-5 Web programming, CGI scripting, 4-23, 4-24, TCP, listening for connections, 1-21 4-25, 4-26, 4-27 TCP, server example, 1-19 Web services, 2-8 TCPServer, SocketServer module, 5-10 Webday, 2-28 Telnet, using with network applications, 1-10 WSGI, 4-36 Threaded network server, 1-50 WSGI (Web Services Gateway Interface), 4-35 ThreadingMixIn class, SocketServer module, WSGI, and CGI environment variables, 4-39 5-15 WSGI, and wsgi.\* variables, 4-40 ThreadingTCPServer, SocketServer module, 5-14 WSGI, application inputs, 4-38 Threading UDPServer, SocketServer module, 5-14 WSGI, applications, 4-37 Threads, and network servers, 1-50 WSGI, parsing query string, 4-41 Timeout, on sockets, 1-34 WSGI, producing content, 4-44 Turbogears, 4-54 WSGI, response encoding, 4-45 Twisted framework, 1-52 WSGI, responses, 4-42 WSGI, running a stand-alone server, 4-46, 4-47

WSGI, running applications within a CGI script,

WWW, see HTTP, 4-5

#### U

UDP, 1-13, 1-41 UDP, client example, 1-43 UDP, server example, 1-42

### $\mathbf{X}$

XML, element attributes, 3-19

XML, element wildcards, 3-20

XML, ElementTree interface, 3-15, 3-16

XML, ElementTree module, 3-14

XML, finding all matching elements, 3-18

XML, finding matching elements, 3-17

XML, incremental parsing of, 3-25

XML, modifying documentation structu with

ElementTree, 3-23

XML, parsing with SAX, 3-9

XML, writing to files, 3-24

XML-RPC, 5-20

### $\mathbf{Z}$

Zope, 4-54