

PROGRAMMING NETWORK SERVERS TOPIC 6, CHAPTERS 21, 22

Network Programming

Kansas State University at Salina

Decorative geometric shapes in orange, teal, and blue at the bottom of the slide.

I thought we already did servers?

- We have learned how to use sockets to listen on a port and accept connections.
- Previous programs could handle only one connection at a time.
- A server must be able to handle many clients at one time.
- Three basic approaches
 - Process creation with forks
 - Multi-threading
 - Asynchronous I/O

The skipped chapters

- Chapter 16 – SocketServer
 - General server framework
 - Connectionless protocols, like HTTP, where the server receives one request from a client and send back one reply
- Chapter 17 – SimpleXMLRPCServer
 - Module that provides a server framework for XML-RPC
 - XML-RPC client side is covered in chapter 8, which we mostly skipped
- Chapter 18 – CGI
 - Server side web programming (CMST 335)

The skipped chapters (continued)

- Chapter 19 – mod_python
 - Server side web programming support from within the Apache web server
 - More efficient than CGI
 - Same concept as PHP, except with Python
- Chapter 20 – Forking
 - Process Creation – `fork()` is the Unix system call to duplicate the current process.
 - Windows does process creation totally different. Windows based network servers do not use process creation.
 - Even in Unix, forking has limited application for network servers

Processes and Threads

- A process is a program in execution
- A thread is one line of execution within a process. A process may contain many threads.
- Thread creation has much less overhead than process creation, especially in Windows.
- Each thread has its own stack (local variables), but share global variables with the other threads.
- Global variables allow threads to share information and communicate with one another.
- Shared data introduces the need for synchronization – A can of worms

Creating threads in Python

- Three ways invoke the Thread class from the threading module
 - Create Thread instance, passing in a function
 - Create Thread instance, passing in a class
 - Subclass Thread and create subclass instance
- First method is sufficient for our needs

```
import threading
```

```
...
```

```
t = threading.Thread( target = threadcode, args = [arg1, arg2] )
```

```
t.setDaemon(1)
```

```
t.start()
```

Creating threads in Python (continued)

```
import threading
```

```
...  
t = threading.Thread( target = threadcode, args = [arg1, arg2] )  
t.setDaemon(1)  
t.start()
```

- Thread creation:
 - target identifies the code (function) for the new thread to execute
 - args is the arguments to pass to the target function
- setDaemon(1) – child thread dies when parent dies. setDaemon(0) means that the child thread can keep running after parent is finished.
- start() – begin now
- join() – parent should suspend until new thread terminates

Three parts of a multi-threaded server

- Parent thread
 - Listen and accept socket connections
 - Create and start child threads
 - Infinite loop
- Child thread
 - Receive data from client
 - Send data to client
 - Call synchronized code as needed
- Synchronized access to shared data
 - Usually a global class, else all global variables
 - Uses synchronization tools

Synchronization

- Only one thread can update global data at a time
- Multiple threads reading global data is allowed
- Critical code section – that section of the code which accesses the shared global data
- Single thread access to the critical section is easy, just acquire and release one lock
- Multiple thread access to critical section is tricky
 - Known solutions to lots of synchronization problems
 - Hardest part is framing the problem in terms of a known synchronization problem: producers and consumers, readers and writers, sleepy barber, smokers, one lane bridge, etc...
 - Take Operating Systems class or study parallel programming
 - Some Python modules implement known problem solutions

Synchronization tools (some of them)

```
import threading
L = threading.Lock()
L.acquire()
...
L.release()
```

```
S = threading.Semaphore(5)
S.acquire()
...
S.release()
```

```
import threading
C = threading.Condition()
```

```
while notReady:
    C.wait()
```

```
if oneClear:
    C.notify()
```

```
if allClear:
    C.notifyAll()
```

- Lock – Simple lock, limit access to one thread
- Semaphore(N) – General lock, limit access to N threads (Lock() and Semaphore() are almost the same) – default value of N is 1
- Condition – Allows a thread to wait and be signaled by another thread based on some condition

Asynchronous communication

- A totally different approach – non blocking sockets
- `socket.send()` and `socket.recv()` are normally blocking calls – they don't return until some potentially slow network activity completes
- Setting a socket to be non-blocking means that another system call (`poll()` or `select()`) can monitor several sockets for activity
- Since we can monitor all the client sockets at once, we do not need to create multiple processes or threads and hence we do not need the synchronization tools and all the tricky details associated with using them correctly.

Asynchronous communication (continued)

- Asynchronous servers may NOT have any slow or blocking operations – quick in and out type applications only – No databases!
- Must track the state of each client, which can make the problem harder
- According to the book and any Unix documentation you may find, `poll()` is preferred over `select()` because it is more robust.
- `poll()` is not available on Windows, only `select()`

select() and poll()

- Usage of `select()` and `poll()` is almost the same
- They both take a list of sockets to watch for activity (Unix also allows file descriptors):
 - Sockets having received data
 - Sockets ready to send data
 - Sockets in an error state
- Detailed `poll()` example in book
- Detailed `select()` example on K-State Online (`chatSelectServ.py` – I didn't write it)

When `select()` and `poll()` return

- A return with a socket in the receive list means that a `recv()` will return data immediately.
- The ready to send socket list is often omitted or ignored. A socket in this list just means that the socket is in a state where it can send data immediately.
- The `poll()` example in the book (`echoserve.py`) uses socket bitmasks and the `register()` function to force a socket into the ready to send list as soon as it's available.

Twisted – easy asynchronous communication

- Twisted implements the more challenging part of asynchronous communication – you just handle receiving and sending data.
- Based on the call-back model
- Generally need two classes, each of which inherit from Twisted classes: protocol and factory
- Pass the factory class to the Twisted reactor object. The `protocol = xxxx` definition in your factory class
- Reactor provides the main execution loop and calls methods from your protocol class.

Twisted example – from book

```
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineOnlyReceiver
from twisted.internet import reactor

class Chat(LineOnlyReceiver):
    def lineReceived(self, data):
        self.factory.sendAll("%s: %s" % (self.getId(), data))

    def getId(self):
        return str(self.transport.getPeer())

    def connectionMade(self):
        print "New connection from", self.getId()
        self.transport.write("Welcome to the chat server, %s\n" % self.getId())
        self.factory.addClient(self)

    def connectionLost(self, reason):
        self.factory.delClient(self)
```


Twisted example – from book

```
class ChatFactory(Factory):
    protocol = Chat

    def __init__(self):
        self.clients = []

    def addClient(self, newclient):
        self.clients.append(newclient)

    def delClient(self, client):
        self.clients.remove(client)

    def sendAll(self, message):
        for proto in self.clients:
            proto.transport.write(message + "\n")

reactor.listenTCP(51423, ChatFactory())
reactor.run()
```

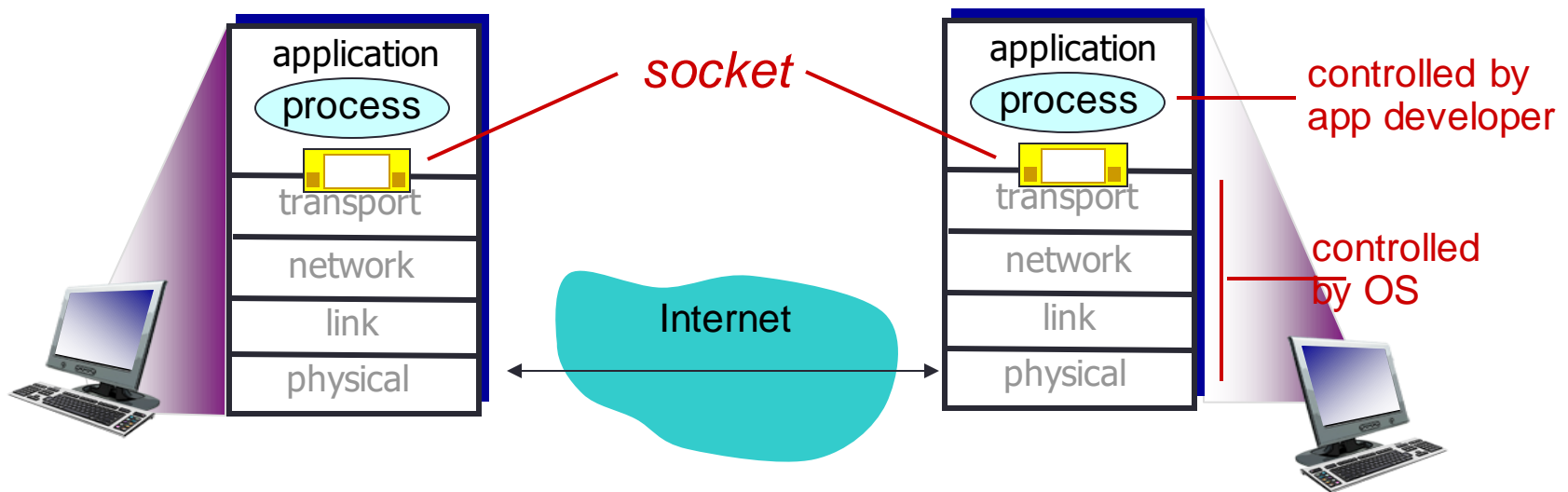
- ChatFactory gives us a class to hold our data.
- In reactor, ChatFactory is self.factory
- Reactor makes calls to functions in the protocol class, which can inherit from a base protocol class

Socket Programming

- A socket is a kind of file descriptor that is used to refer to a network connection made between the computer, often known as the client, and any remote host.
- build client/server applications that communicate using sockets
- Socket programming in python.. is somewhat similar to file manipulation
- it is the concept of files and file descriptors implemented in such a way that a connection is considered to be a file.
- In order to use the socket functions, you must import the socket module.

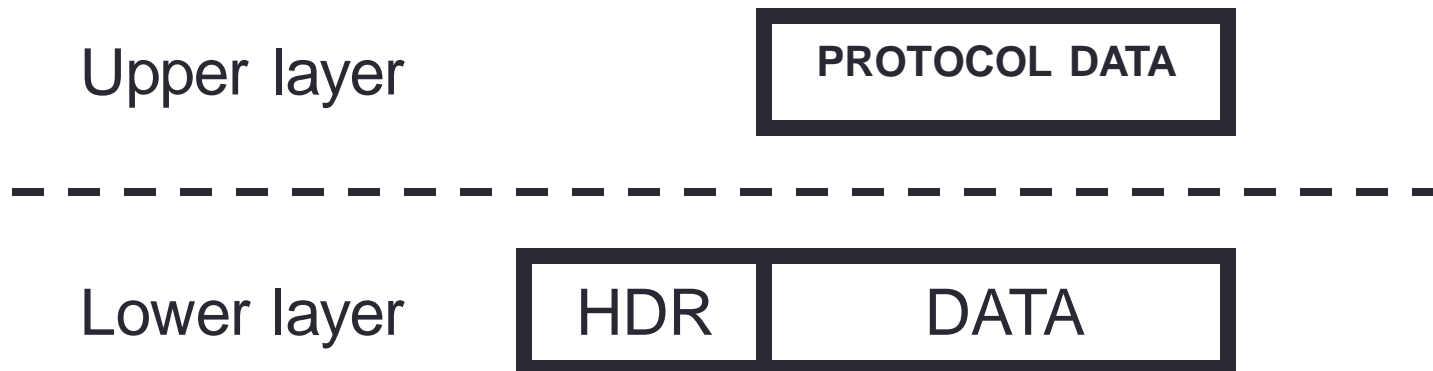
Socket programming

socket: door between application process and end-end-transport protocol



Creating a Socket

- 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 ...



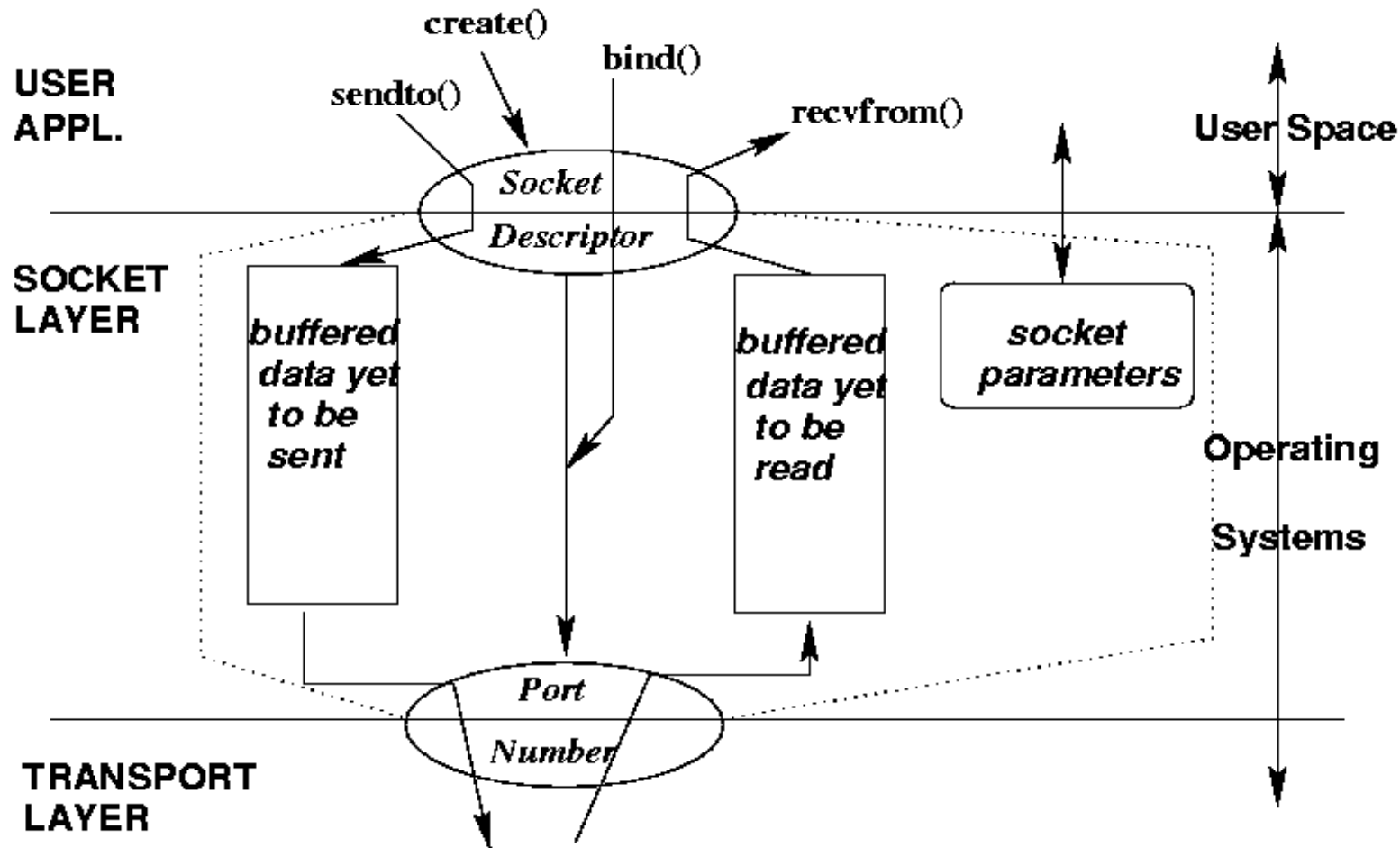
UDP Characteristics

- 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

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

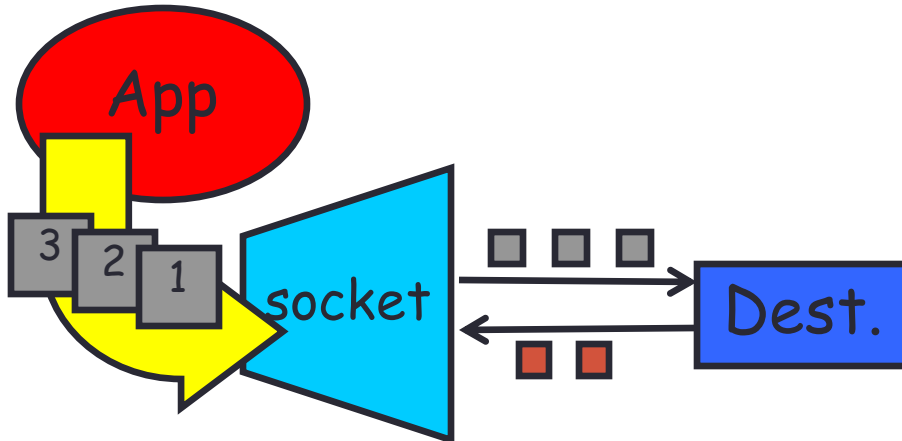
Socket: Conceptual View



Two essential types of sockets

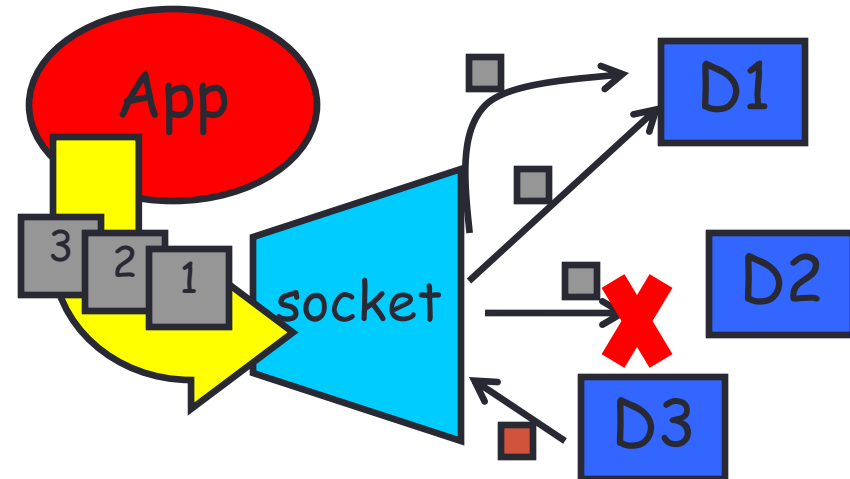
- SOCK_STREAM

- a.k.a. TCP
- reliable delivery
- in-order guaranteed
- connection-oriented
- bidirectional



- SOCK_DGRAM

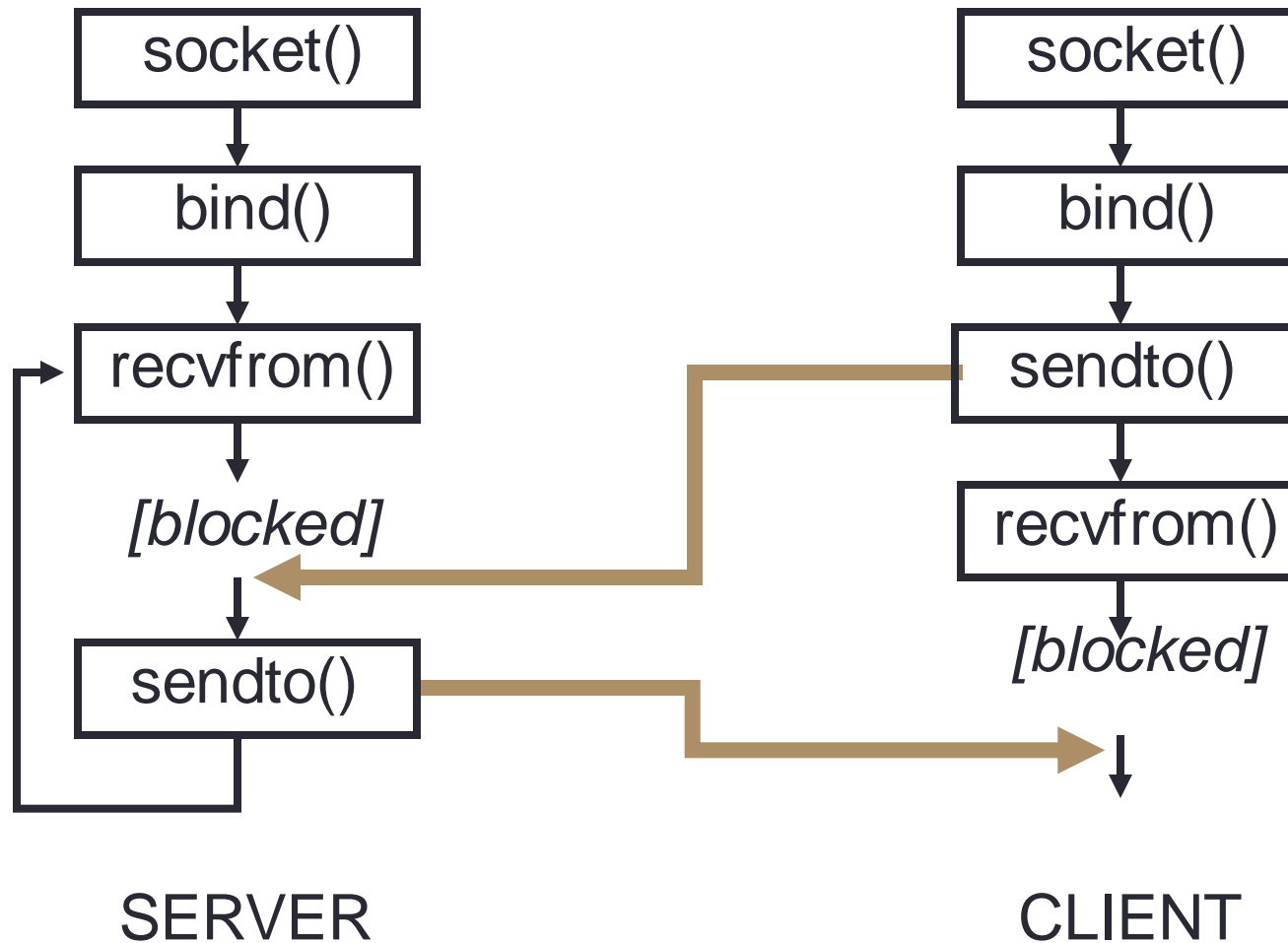
- a.k.a. UDP
- unreliable delivery
- no order guarantees
- no notion of “connection” – app indicates dest. for each packet
- can send or receive



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

Connectionless Services



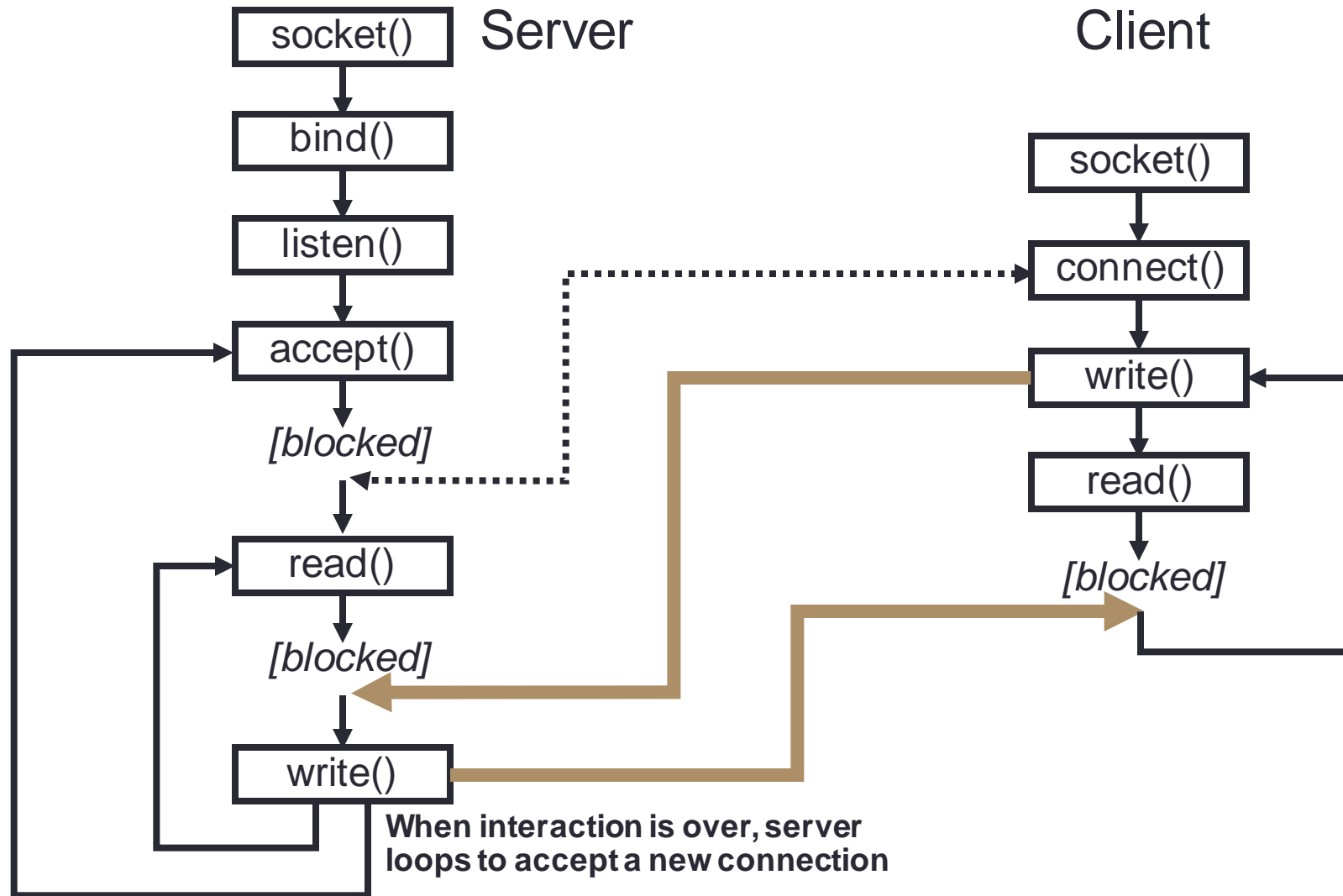
Simple Connectionless Server

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

Simple Connectionless Client

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

Connection Oriented Services



Simple 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 True:
    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

Simple 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()*

Creating a Socket

- The `socket` function from the `socket` module is used to open a new socket in much the same way that `open()` is used to open files.
- It takes two arguments: the **socket family** and the **socket type**.
- Socket Families:
 - `AF_INET`: IPv4 (you will probably use this)
 - `AF_INET6`: IPv6
 - `AF_UNIX`: unix domain
- Socket Types:
 - `SOCK_STREAM`: TCP, used for reliable connections
 - `SOCK_DGRAM`: UDP, used for games and utilities
 - `SOCK_RAW`: a raw socket

Creating a Socket

```
import socket
```

```
sock = socket.socket(socket.AF_INET,  
    socket.SOCK_STREAM)
```

- Once a socket has been created you have opened a raw file descriptor that python knows is intended to represent a specific type of network connection
- Much like files, sockets are closed with the `close()` function from the `socket` module.

Socket programming *transport services:*

Two socket types for two transport services:

- **UDP:** unreliable datagram
- **TCP:** reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming *with UDP*

UDP: no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server

Socket programming *with UDP...*

- 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
- The client sends data to the server, which awakes from its blocked state and starts to compute its response.
- **sendto()** is a function that sends packets to the socket. It takes two tuples (serverName, Port)
- **recvfrom(buflen)** : Receive data from the socket, up to buflen bytes, returning also the remote host and port from which the data came

Client/server socket interaction: UDP

server (running on serverIP)

create socket, port= x:
`serverSocket =
socket(AF_INET,SOCK_DGRAM)`

↓
read datagram from
`serverSocket`

↓
write reply to
`serverSocket`
specifying
client address,
port number

client

create socket:
`clientSocket =
socket(AF_INET,SOCK_DGRAM)`

↓
Create datagram with server IP and
port=x; send datagram via
`clientSocket`

↓
read datagram from
`clientSocket`

↓
close
`clientSocket`

Example app: UDP client

Python UDPClient

include Python's socket
library

→ `from socket import *`

`serverName = '127.0.0.1'`

`serverPort = 12000`

create UDP socket for
server

→ `clientSocket = socket(AF_INET, SOCK_DGRAM)`

`message = raw_input('Input lowercase sentence:')`

get user keyboard
input

→ `clientSocket.sendto(message, (serverName, serverPort))`

Attach server name, port to
message; send into socket

→ `modifiedMessage, serverAddress =`

read reply characters from
socket into string

`clientSocket.recvfrom(2048)`

`print modifiedMessage`

print out received string
and close socket

→ `clientSocket.close()`

Example app: UDP server

Python UDPServer

```
from socket import *  
serverPort = 12000  
  
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)  
bind socket to local port  
number 12000 → serverSocket.bind(('', serverPort))  
  
print "The server is ready to receive"  
  
loop forever → while 1:  
    Read from UDP socket into  
    message, getting client's  
    address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)  
    modifiedMessage = message.upper()  
    send upper case string  
    back to this client → serverSocket.sendto(modifiedMessage, clientAddress)
```

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

Connecting a Socket

- After socket is created, it is necessary to connect the socket to a remote host. This is done with the **connect()** function on the client side.
- The **connect()** function takes two arguments
 - the **hostname** in string form, and a **port number** to connect to in integer form.

```
import socket
```

```
sock = socket.socket(socket.AF_INET,  
    socket.SOCK_STREAM)
```

```
sock.connect(("amu.edu.et", 123))
```

- If you are the client of your connection, you need not worry about Binding and Accepting, and can move on to Sending and Receiving.

Binding and Accepting

- This is only relevant if your socket is intended to be a server.

Binding

- If you are planning to accept incoming connections, you must bind your socket.
- Whereas connecting forms a connection with a remote socket,
- **binding** tells your socket that it should use a specific port when looking for incoming connections.

Binding and Accepting...

- You bind a socket using the **bind()** function, which takes 2 arguments:
- the **hostname** that you wish to bind to, and the **port** you wish to bind to.
- In general, the hostname will be your hostname, so you can use the `gethostbyname()` function from the `socket` module as the hostname argument.
- Once it is bound to your hostname and to a specific port, the socket knows that when told to listen, it should listen at that port.

Binding and Accepting...

```
import socket  
server = socket.socket(socket.AF_INET,  
                        socket.SOCK_STREAM)  
server.bind(socket.gethostbyname(), 123) or  
server.bind("", 31337) or
```

Listening

- Once the socket is bound to a port, you must tell it to listen at that port.
- **Listening**; refers to monitoring a port so that it can handle anything that tries to connect to that port.

Binding and Accepting...

- The **listen()** function does this, listening to connection attempts.
- It takes one argument, an integer value representing the maximum number of queued connection attempts to hold before dropping older ones.

```
import socket
```

```
server = socket.socket(socket.AF_INET,  
                        socket.SOCK_STREAM)
```

```
server.bind("", 123)
```

```
server.listen(1) #listen at port 123 and queue only 1  
connection at a time
```

- After a connection is made, it must be accepted.

Binding and Accepting...

Accepting

- `listen()` finds connection attempts, but you must use then `accept` them to form a connection between your host and the remote client.
- You use the aptly-named `accept()` function in order to do this.

```
import socket
```

```
server = socket.socket(socket.AF_INET,  
    socket.SOCK_STREAM)
```

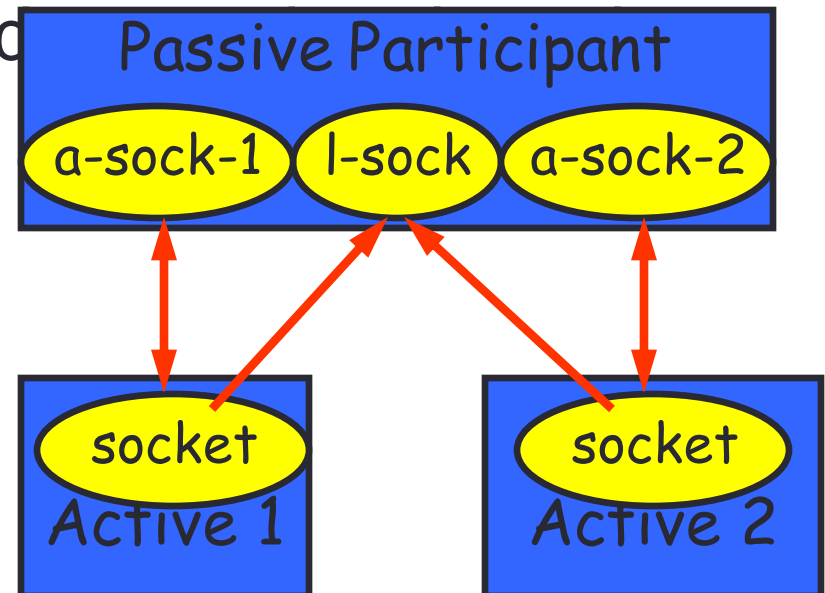
```
server.bind(“”,123)
```

```
server.listen(1)
```

```
connection, sock_addr = sock.accept()
```

Binding and Accepting...

- As you can see, the `accept()` function creates an entirely new socket
- your original socket (in this case 'sock') can continue listening for new connections
- while the newly created socket (in this case 'connection') can be used to read data from the client.



Sending and Receiving

- Whether you have connected to a remote host or accepted a connection from a client, sending and receiving are what allow data to be transferred.
- The **send()** and **recv()** functions from the socket module are basically identical - the only difference is whether data is being sent or received.
- They both take one argument
- **send()** takes the data to be sent as an argument,
- **recv()** takes the number of bytes to be received as an argument. **recv()** returns the data received.

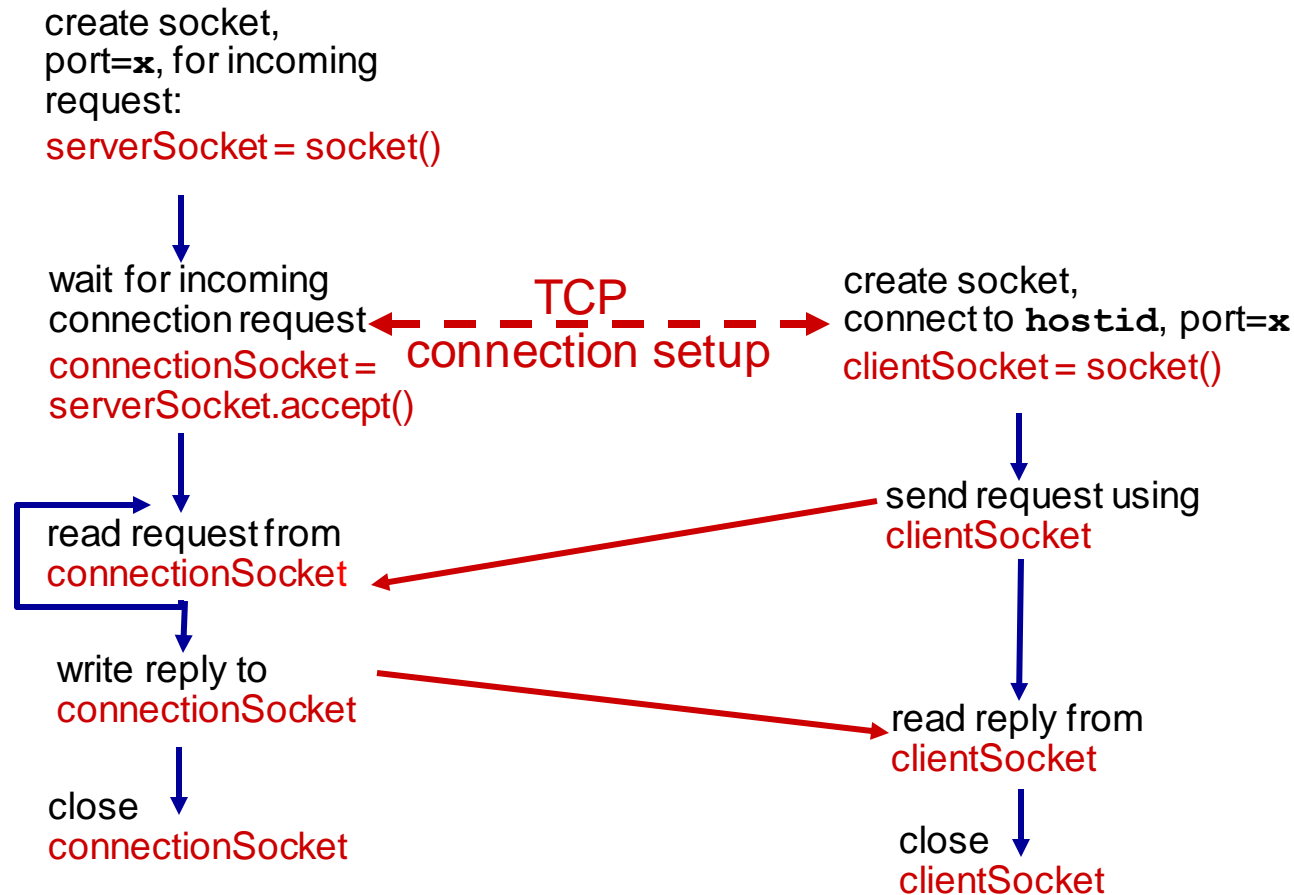
Sending and Receiving...

```
sock = socket.socket(socket.AF_INET,  
    socket.SOCK_STREAM  
sock.connect(("iamu.edu.et", 123  
nickname = "NICK mamush"  
encoded_nick = bytes(nickname, 'utf-8')  
sock.send(encoded_nick)  
sock.send(encoded_user)
```

Client/server socket interaction: TCP

server (running on `hostid`)

client



Example app:TCP client

Python TCPClient

```
from socket import *
```

```
serverName = 'servername'
```

```
serverPort = 1200
```

create TCP socket for
server, remote port 12000

→ `clientSocket = socket(AF_INET, SOCK_STREAM)`

```
clientSocket.connect((serverName,serverPort))
```

```
sentence = raw_input('Input lowercase sentence:')
```

No need to attach server
name, port

→ `clientSocket.send(sentence)`

```
modifiedSentence = clientSocket.recv(1024)
```

```
print 'From Server:', modifiedSentence
```

```
clientSocket.close()
```

Example app: TCP server

Python TCPServer

create TCP welcoming
socket



server begins listening for
incoming TCP requests



loop forever



server waits on accept()
for incoming requests, new
socket created on return



read bytes from socket (but
not address as in UDP)



close connection to this
client (but *not* welcoming
socket)



```
from socket import *
serverPort = 1200
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

Python Internet Modules

- A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtpplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

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

SMTP Example

```
import smtplib
smtpObj = smtplib.SMTP("localhost")
sender = 'nebfikru@gmail.com'
receivers = ['nebiat.fikru@amu.edu.et']
message = """From:<nebfikru@gmail.com>
To: <nebiat.fikru@amu.edu.et>
Subject: SMTP e-mail test

This is a test e-mail message.
"""
smtpObj.sendmail(sender, receivers, message)
print("Successfully sent email")
```

Chapter 2: summary

our study of network apps now complete!

- application architectures
 - client-server
 - P2P
- application service requirements:
 - reliability, bandwidth, delay
- Internet transport service model
 - connection-oriented, reliable: TCP
 - unreliable, datagrams: UDP
- ❖ specific protocols:
 - HTTP
 - FTP
 - SMTP, POP, IMAP
 - DNS
 - P2P: BitTorrent, DHT
- ❖ socket programming: TCP, UDP sockets

Chapter 2: summary

most importantly: learned about protocols!

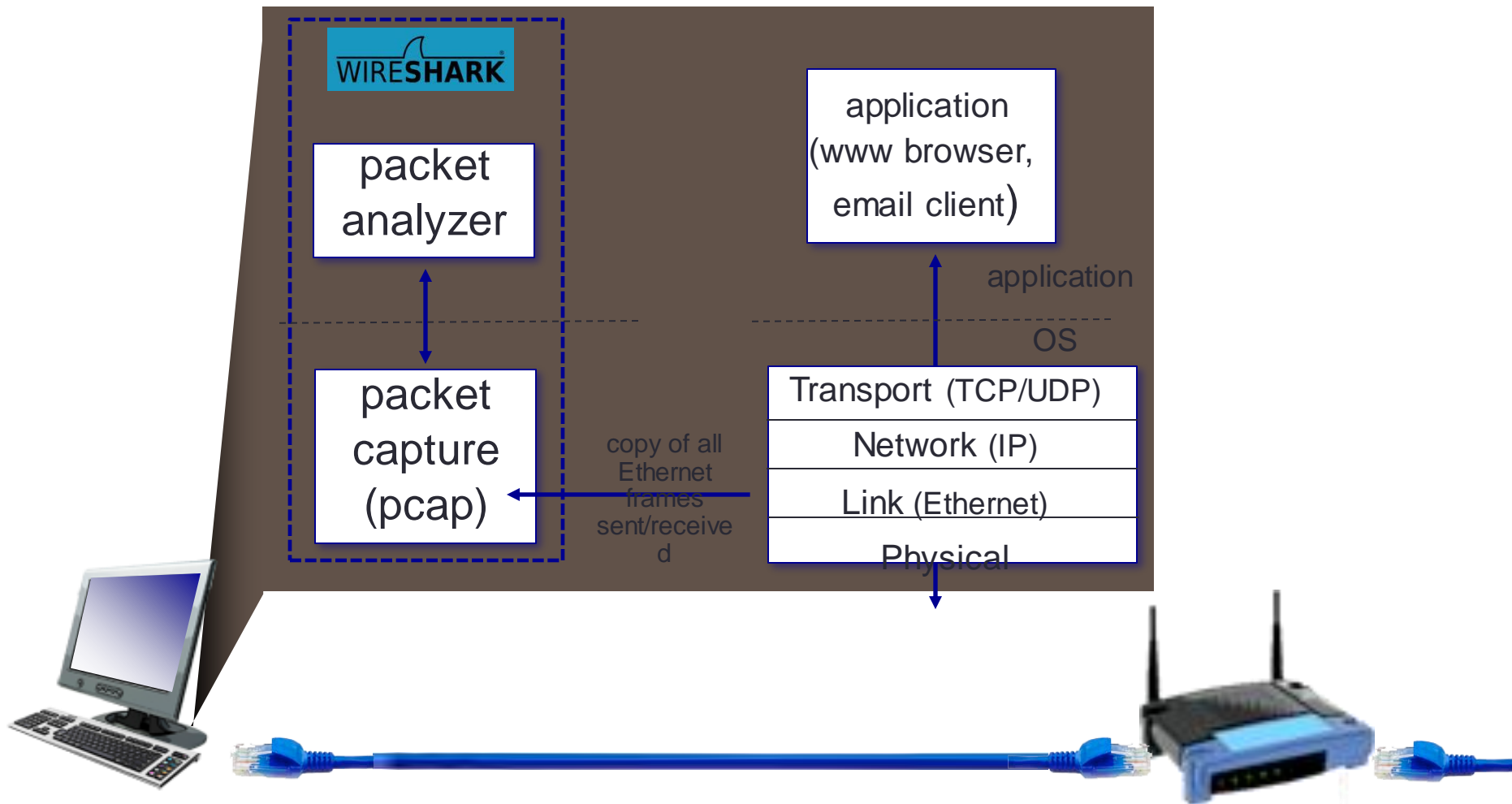
- typical request/reply message exchange:
 - client requests info or service
 - server responds with data, status code
- message formats:
 - headers: fields giving info about data
 - data: info being communicated

important themes:

- ❖ control vs. data msgs
 - in-band, out-of-band
- ❖ centralized vs. decentralized
- ❖ stateless vs. stateful
- ❖ reliable vs. unreliable msg transfer
- ❖ “complexity at network edge”

Chapter I

Additional Slides



DATABASE PROGRAMMING

Why is a database useful for a big web site?

- For CNN.com:
 - Can have multiple authors and editors creating multiple stories distributed all over a network.
 - Can pull the content automatically via a program and merge all the stories into one big website
- Works similarly for other kinds of large websites
 - Amazon.com
 - Where do you think their catalog and review is stored?
 - EBay.com
 - Where do you think all those pictures and descriptions and bid information is stored?

Why databases?

- Rather: Why not just use files?
 - Why do we care about using some extra software for storing our bytes?
- Databases provide efficient access to data in a standardized mechanism.
 - Databases are fast.
 - Databases can be accessed from more than one place in more than one way.
 - Databases store relations between data

Databases are fast because of indices

- Filenames are *indexed* just by name.
- Usually, you care about information that is found by something other than a filename.
 - For example, you may care about someone's information identified by last name or by SSN or even birthdate or city/state of residence.

Databases are standardized

- There are many different standard databases.
 - In the UNIX and open source markets: bsddb, gdbm, MySQL
 - In the commercial markets: Microsoft Access, Informix, Oracle, Sybase
- Information stored in a standard database can be accessed and manipulated via many different tools and languages.

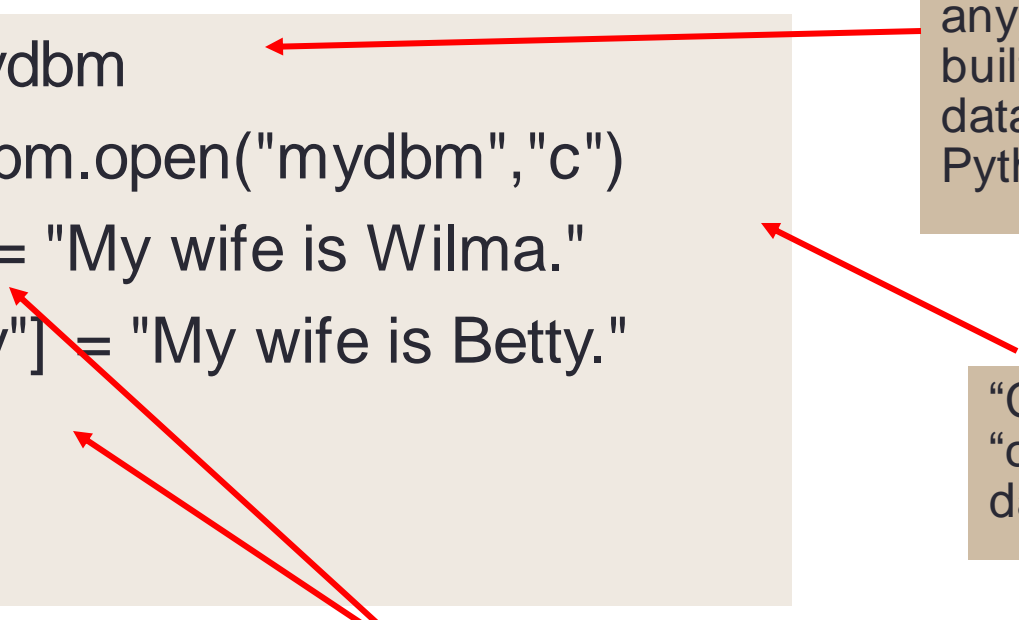
Databases store *relations*

- Recall our list representation of pixels.
 - It was just a list of five numbers.
 - Who knew that the first two numbers were x and y positions, and the last three were RGB values?
 - Only us—it wasn't recorded anywhere.
- Databases can store *names* for the *fields* of data
- They can store which fields are important (and thus *indexed* for rapid access), and how fields are related (e.g., that each pixel has three color components, that each student has one transcript)

Simplest databases in Python

```
>>> import anydbm
>>> db = anydbm.open("mydbm","c")
>>> db["fred"] = "My wife is Wilma."
>>> db["barney"] = "My wife is Betty."
>>> db.close()
```

anydbm is a built-in database to Python.



“C” for “create” the database

Keys on which the database is indexed.

Accessing our simple database

```
>>> db = anydbm.open("mydbm","r")
>>> print db.keys()
['barney', 'fred']
>>> print db['barney']
My wife is Betty.
>>> for k in db.keys():
...     print db[k]
...
My wife is Betty.
My wife is Wilma.
>>> db.close()
```



Now, open for Reading

Disadvantages of the simple database

- Keys and values can only be simple strings.
- Can only have a single index.
 - Can't index, say, on last name and SSN.
- Doesn't store field names.
- There's no real search or manipulation capability built in other than simply using Python.

Shelves store anything

```
>>> import shelve
>>> db=shelve.open("mysshelf","c")
>>> db["one"]=["This is",["a","list"]]
>>> db["two"]=12
>>> db.close()
>>> db=shelve.open("mysshelf","r")
>>> print db.keys()
['two', 'one']
>>> print db['one']
['This is', ['a', 'list']]
>>> print db['two']
12
```

Can use shelves to store in standardized database formats, but not really useful for Python-specific data.

Well, not quite anything


- Can we use the shelve module to store and retrieve our media?
- It's not made for data like that.
 - Lists of pictures didn't come back from the database the way they were stored.
 - Lists got mangled: Sub-lists in sub-lists, etc.
 - Media have many, many more elements than simple databases can handle.

Powerful, relational databases

- Modern databases are mostly *relational*
- Relational databases store information in *tables* where *columns* of information are named and *rows* of data are assumed to be related.
- You work with multiple tables to store complex relationships.

A simple table

Fields



Name	Age
Mark	40
Matthew	11
Brian	38

The implied relation
of this row is Mark is
40 years old.

More complex tables

Picture	PictureID
Class1.jpg	P1
Class2.jpg	P2

StudentName	StudentID
Katie	S1
Brittany	S2
Carrie	S3

PictureID	StudentID
P1	S1
P1	S2
P2	S3

How to use complex tables

- What picture is Brittany in?

- Look up her ID in the student table
- Look up the corresponding PictureID in the PictureID-StudentID table
- Look up the picture in the Picture table
 - Answer: Class1.jpg

StudentName	StudentID
Katie	S1
Brittany	S2
Carrie	S3

Picture	PictureID
Class1.jpg	P1
Class2.jpg	P2

PictureID	StudentID
P1	S1
P1	S2
P2	S3

Another Use

- Who is in “Class1.jpg”?
 - Look up the picture in the Picture table to get the ID
 - Look up the corresponding PictureID in the PictureID-StudentID table
 - Look up the StudentNames in the Student picture
 - Answer: Katie and Brittany

StudentName	StudentID
Katie	S1
Brittany	S2
Carrie	S3

Picture	PictureID
Class1.jpg	P1
Class2.jpg	P2

PictureID	StudentID
P1	S1
P1	S2
P2	S3

A Database *Join*

- We call this kind of access across multiple tables a *join*
- By joining tables, we can represent more complex relationships than with just a single table.
- Most database systems provide the ability to join tables.
- Joining works better if the tables are well-formed:
 - Simple
 - Containing only a single relation per row

Creating Relational Databases using Simple Python Databases

- We can create structures like relational databases using our existing Python tools.
- We start by introducing *hash tables* (also called *associative arrays*)
 - Think of these as arrays whose indices are *strings*, not *numbers*

Hash tables in Python

```
>>> row={'StudentName':'Katie','StudentID':'S1'}
>>> print row
{'StudentID': 'S1', 'StudentName': 'Katie'}
>>> print row['StudentID']
S1
>>> print row['StudenName']
Attempt to access a key that is not in a dictionary.
>>> print row['StudentName']
Katie
```

Building a Hash Table more Slowly

```
>>> picturerow = {}  
>>> picturerow['Picture']='Class1.jpg'  
>>> picturerow['PictureID']='P1'  
>>> print picturerow  
{'Picture': 'Class1.jpg', 'PictureID': 'P1'}  
>>> print picturerow['Picture']  
Class1.jpg
```

Building relational database out of shelves of hash tables

- For each row of the table, we can use a hash table.
- We can store collections of rows in the same database.
 - We search for something by using a for loop on the keys() of the database

Creating a database

```
import shelve
def createDatabases():
    #Create Student Database
    students=shelve.open("students.db","c")
    row = {'StudentName':'Katie','StudentID':'S1'}
    students['S1']=row
    row =
        {'StudentName':'Brittany','StudentID':'S2'}
    students['S2']=row
    row = {'StudentName':'Carrie','StudentID':'S3'}
    students['S3']=row
    students.close()
```

The keys in the database really don't matter in this example.

```
#Create Picture Database
pictures=shelve.open("pictures.db","c")
row = {'Picture':'Class1.jpg','PictureID':'P1'}
pictures['P1']=row
row = {'Picture':'Class2.jpg','PictureID':'P2'}
pictures['P2']=row
pictures.close()
#Create Picture-Student Database
pictures=shelve.open("pict-students.db","c")
row = {'PictureID':'P1','StudentID':'S1'}
pictures['P1S1']=row
row = {'PictureID':'P1','StudentID':'S2'}
pictures['P1S2']=row
row = {'PictureID':'P2','StudentID':'S3'}
pictures['P2S3']=row
pictures.close()
```

Doing a join: Who is in Class1.jpg?

```
def whoInClass1():  
    # Get the pictureID  
  
    pictures=shelve.open("pictures.db",  
        "r")  
    for key in pictures.keys():  
        row = pictures[key]  
        if row['Picture'] == 'Class1.jpg':  
            id = row['PictureID']  
    pictures.close()
```

This can be made MUCH easier
with some sub-functions! Like:
findStudentWithID()

```
# Get the students' IDs  
studentslist=[]  
pictures=shelve.open("pict-students.db","c")  
for key in pictures.keys():  
    row = pictures[key]  
    if row['PictureID']==id:  
        studentslist.append(row['StudentID'])  
pictures.close()  
print "We're looking for:",studentslist  
# Get the students' names  
students = shelve.open("students.db","r")  
for key in students.keys():  
    row = students[key]  
    if row['StudentID'] in studentslist:  
        print row['StudentName'], "is in the picture"  
students.close()
```

Running the Join

```
>>> whoInClass1()
```

```
We're looking for: ['S2', 'S1']
```

```
Brittany is in the picture
```

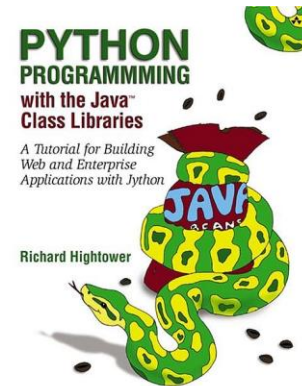
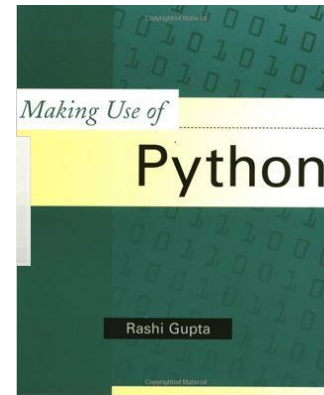
```
Katie is in the picture
```

An Example using MySQL

- We're going to use an example using MySQL
 - MySQL is a popular open source database that runs on many platforms.
 - It's powerful and can handle large, complex table manipulations.
- The goal is not for you to learn to use MySQL.
 - Very similar things can be done with Microsoft Access, SimpleDB/InstantDB, Oracle, Informix.
- Just using MySQL as an example.

For More Information on Databases and SQL in Python (and Jython)

- *Making Use of Python* by Rashi Gupta (Wiley: 2002)
- *Python Programming with the Java Class Libraries* by Richard Hightower (Addison-Wesley: 2003)

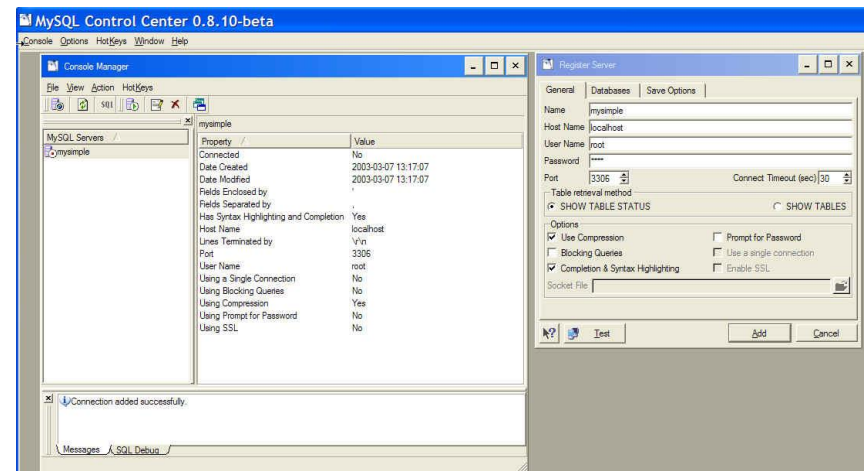


WARNING: We're Going to Get Detailed and Technical Here!

- If we ask you to do any database work on assignment, it will only be with anydbm and shelve.
- However, if you do any database work in your professional life, you *will* be using relational databases and SQL.
 - We won't be asking you to do that for homework in this class.
- The next few slides give you the pointers on how to set up MySQL on your own computer.
 - But it's not for the faint of heart!
 - If you'd like to avoid technical details, ignore the next FOUR slides

Installing MySQL

- Go to <http://www.mysql.com/downloads/index.html>
 - Download and install MySQL
 - Suggestion: Download and install MySQLcc (Command Center)
- Run the Command Center to create a connection
 - Automatically also creates a database connection named “Test”
- Run “mysqld” to get MySQL running (in the background)



Getting Python to talk to MySQL

- You have to modify your JES to work with MySQL
 - anydbm and shelve are built into JES, but not the MySQL connection
- Download the MySQL connection for Java from the MySQL web site.
 - Place the .jar file that you download in your JES\jython “Lib” folder

Setting up the database connection

- The following is how you do it in Jython to talk to MySQL.
 - Talking to Python is different only for this slide. The rest is the same.

```
from com.ziclix.python.sql import zxJDBC
db =zxJDBC.connect("jdbc:mysql://localhost/test", "root", None,
"com.mysql.jdbc.Driver")
#This is the name of your database connection, the database
"username" you used, the password you used, and the Driver you
need.
con = db.cursor()
```

Put it in a function

- All these details are hard to remember, so hide it all in a function and just say **con = getConnection()**

```
from com.ziclix.python.sql import zxJDBC
def getConnection():
    db =zxJDBC.connect("jdbc:mysql://localhost/test", "root", None,
"com.mysql.jdbc.Driver")
    con = db.cursor()
    return con
```

Executing SQL Commands

(Back to the generally relevant lecture)

- Once you have a database connection (called a cursor in SQL), you can start executing commands in your database using the **execute** method, e.g.

```
con.execute("create table Person (name VARCHAR(50), age  
INT)")
```

SQL: Structured Query Language

- SQL is usually pronounced “S.Q.L.” or “Sequel”
- It’s a language for database creation and manipulation.
 - Yes, a whole new language, like Python or Java
 - It actually has several parts, such as DDL (Data Definition Language) and DML (Data Manipulation Language), but we’re not going to cover each part.
- We’re not going to cover all of SQL
 - There’s a lot there
 - And what’s there depends, on part, on the database you’re using.

Creating tables in SQL

- Create table *tablename* (*columnname datatype, ...*)
 - Tablename is the name you want to use for the table
 - Columnname is what you want to call that field of information.
 - Datatype is what kind of data you're going to store there.
 - Examples: NUMERIC, INT, FLOAT, DATE, TIME, YEAR, VARCHAR(number-of-bytes), TEXT
- We can define some columns as **index** fields, and then create an index based on those fields, which speeds access.

Inserting data via SQL

- Insert into *tablename* values (*columnvalue1*, *columnvalue2...*)
 - For our Person table:
`con.execute('insert into Person values ("Mark",40)')`
- Here's where those two kinds of quotes comes in handy!

Selecting data in a database

- Select *column1,column2* from *tablename*
- Select *column1,column2* from *tablename* where *condition*

Select * from Person

Select name,age from Person

Select * from Person where age>40

Select name,age from Person where age>40

Doing this from Python

- When you use a *select* from Python,
 - Your cursor has a variable ***rowcount*** that tells you how many rows were selected.
 - This is called an *instance variable*
 - It's a variable known just to that object, similar to how a method is a function known just to that object.
 - Method ***fetchone()*** gives you the next selected row.
- ***Fetchone()*** returns a list

Selecting from the command area

```
>>> con.execute("select name,age from Person")
>>> print con.rowcount
3
>>> print con.fetchone()
('Mark', 40)
>>> print con.fetchone()
('Barb', 41)
>>> print con.fetchone()
('Brian', 36)
```

Selecting and printing from a function

```
def showPersons(con):  
    con.execute("select name, age from Person")  
    for i in range(0,con.rowcount):  
        results=con.fetchone()  
        print results[0]+" is "+str(results[1])+" years old"
```

Running our selection function

```
>>> showPersons(con)
```

```
Mark is 40 years old
```

```
Barb is 41 years old
```

```
Brian is 36 years old
```

Selecting and printing with a condition

```
def showSomePersons(con, condition):  
    con.execute("select name, age from Person "+condition)  
    for i in range(0,con.rowcount):  
        results=con.fetchone()  
        print results[0]+" is "+str(results[1])+" years old"
```

Running the conditional show

```
>>> showSomePersons(con,"where age >= 40")
```

Mark is 40 years old

Barb is 41 years old

The Point of the Conditional Show

- Why are we doing the conditional show?
 - First, to show that we can have tests on our queries which makes processing easier.
 - Second, because this is how we're going to generate HTML: By assembling pieces as strings.

We can do joins, too (But more complicated)

- Answering: What picture is Brittany in?

Select

p.picture,
s.studentName

From

Students as s,
IDs as i,
Pictures as p

Where

(s.studentName="Brittany") and
(s.studentID=i.studentID) and
(i.pictureID=p.pictureID)

StudentName	StudentID
Katie	S1
Brittany	S2
Carrie	S3

Picture	PictureID
Class1.jpg	P1
Class2.jpg	P2

PictureID	StudentID
P1	S1
P1	S2
P2	S3

Week 8

The Natural Language Toolkit (NLTK)

Except where otherwise noted, this work is licensed under:
<http://creativecommons.org/licenses/by-nc-sa/3.0>

List methods

- Getting information about a list
 - `list.index(item)`
 - `list.count(item)`
- These **modify the list in-place**, unlike str operations
 - `list.append(item)`
 - `list.insert(index, item)`
 - `list.remove(item)`
 - `list.extend(list2)`
 - same as `list += list2`
 - `list.sort()`
 - `list.reverse()`

List exercise

- Write a script to print the most frequent token in a text file.

And now for something completely different

Programming tasks?

- So far, we've studied programming syntax and techniques
- What about tasks for programming?
 - Homework
 - Mathematics, statistics (Sage)
 - Biology (Biopython)
 - Animation (Blender)
 - Website development (Django)
 - Game development (PyGame)
 - **Natural language processing (NLTK)**

Natural Language Processing (NLP)

- How can we make a computer understand language?
 - Can a human write/talk to the computer?
 - Or can the computer guess/predict the input?
 - Can the computer talk back?
 - Based on language rules, patterns, or statistics
 - For now, statistics are more accurate and popular

Some areas of NLP

- shallow processing – the surface level
 - **tokenization**
 - **part-of-speech tagging**
 - forms of words
- deep processing – the underlying structures of language
 - **word order (syntax)**
 - meaning
 - translation
- **natural language generation**

The NLTK

- A collection of:
 - Python functions and objects for accomplishing NLP tasks
 - sample texts (corpora)
- Available at: <http://nltk.sourceforge.net>
 - Requires Python 2.4 or higher
 - Click 'Download' and follow instructions for your OS

Tokenization

- Say we want to know the words in Marty's vocabulary
 - *"You know what I hate? Anybody who drives an S.U.V. I'd really like to find Mr. It-Costs-Me-100-Dollars-To-Gas-Up and kick him square in the teeth. Booyah. Be like, I'm Marty Stepp, the best ever. Booyah!"*
- *How do we split his speech into tokens?*

Tokenization (cont.)

- How do we split his speech into tokens?

```
>>> martySpeech.split()  
['You', 'know', 'what', 'I', 'hate?', 'Anybody',  
'who', 'drives', 'an', 'S.U.V.', "I'd", 'really',  
'like', 'to', 'find', 'Mr.', 'It-Costs-Me-100-  
Dollars-To-Gas-Up', 'and', 'kick', 'him',  
'square', 'in', 'the', 'teeth.', 'Booyah.', 'Be',  
'like,', "I'm", 'Marty', 'Stepp,', 'the', 'best',  
'ever.', 'Booyah!']
```

- Now, how often does he use the word "booyah"?

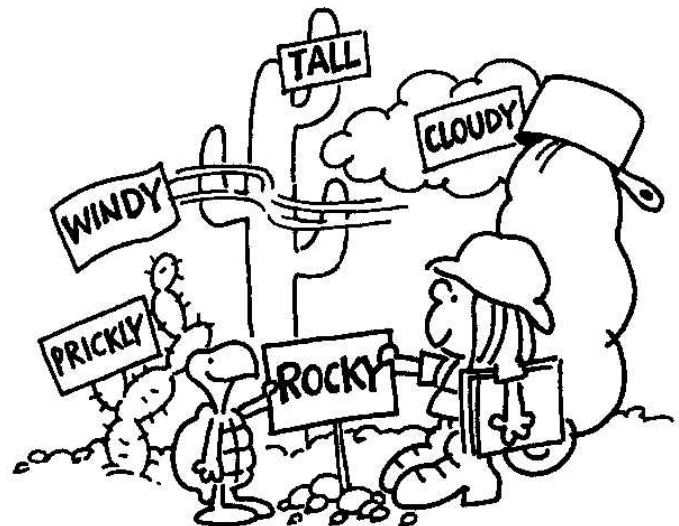
```
>>> martySpeech.split().count("booyah")  
0  
>>> # What the!
```

Tokenization (cont.)

- We could lowercase the speech
- We could write our own method to split on "." split on ",", split on "-", etc.
- The NLTK already has several tokenizer options
- Try:
 - `nltk.tokenize.WordPunctTokenizer`
 - tokenizes on all punctuation
 - `nltk.tokenize.PunktWordTokenizer`
 - **trained** algorithm to statistically split on words

Part-of-speech (POS) tagging

- If you know a token's POS you know:
 - is it the subject?
 - is it the verb?
 - is it introducing a grammatical structure?
 - is it a proper name?



Part-of-speech (POS) tagging

- Exercise: most frequent proper noun in the Penn Treebank?
 - Try:
 - `nltk.corpus.treebank`
 - Python's **`dir()`** to list attributes of an object
 - Example:

```
>>> dir("hello world!")  
[..., 'capitalize', 'center', 'count',  
'decode', 'encode', 'endswith', 'expandtabs',  
'find', 'index', 'isalnum', 'isalpha',  
'isdigit', 'islower', 'isspace', 'istitle',  
'isupper', 'join', 'ljust', 'lower', ...]
```

Tuples

- `tagged_words()` gives us a list of **tuples**
 - **tuple**: the same thing as a list, but you can't change it
 - in this case, the tuples are a (word, tag) pairs

```
>>> # Get the (word, tag) pair at list index 0
...
>>> pair = nltk.corpus.treebank.tagged_words()[0]
>>> pair
('Pierre', 'NNP')
>>> word = pair[0]
>>> tag = pair[1]
>>> print word, tag
Pierre NNP
>>> word, tag = pair                                # or unpack in 1 line!
>>> print word, tag
Pierre NNP
```

POS tagging (cont.)

- How do we tag plain sentences?
 - A NLTK tagger needs a list of tagged sentences to train on
 - We'll use `nltk.corpus.treebank.tagged_sents()`
 - Then it is ready to tag any input! (but how well?)
 - Try these tagger objects:
 - `nltk.UnigramTagger(tagged_sentences)`
 - `nltk.TrigramTagger(tagged_sentences)`
 - Call the tagger's `tag(tokens)` method

```
>>> tagger = nltk.UnigramTagger(tagged_sentences)
>>> result = tagger.tag(tokens)
>>> result
[('You', 'PRP'), ('know', 'VB'), ('what', 'WP'),
 ('I', 'PRP'), ('hate', None), ('?', '.'), ...]
```

POS tagging (cont.)

- Exercise: Mad Libs
 - I have a passage I want filled with the right parts of speech
 - Let's use random picks from our own data!
 - This code will print it out:

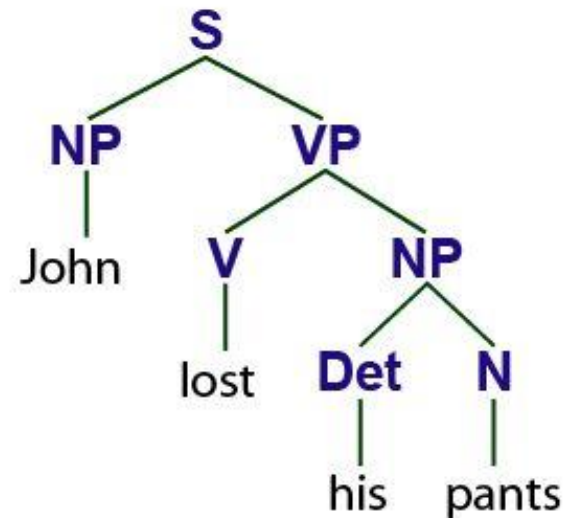
```
print properNoun1, "has always been a", adjective1, \  
    singularNoun, "unlike the", adjective2, \  
    properNoun2, "who I", pastVerb, "as he was", \  
    ingVerb, "yesterday."
```

Eliza (NLG)

- Eliza simulates a Rogerian psychotherapist
- With while loops and tokenization, you can make a chat bot!
 - Try:
 - `nltk.chat.eliza.eliza_chat()`

Parsing

- Syntax is as important for a compiler as it is for natural language
- Realizing the hidden structure of a sentence is useful for:
 - translation
 - meaning analysis
 - relationship analysis
 - a cool demo!
 - Try:
 - `nltk.draw.rdparsers.demo()`



Conclusion

- NLTK: NLP made easy with Python
 - Functions and objects for:
 - tokenization, tagging, generation, parsing, ...
 - and much more!
 - Even armed with these tools, NLP has a lot of difficult problems!
- Also saw:
 - List methods
 - `dir()`
 - Tuples

<http://www.nltk.org/book>

NLTK is on berry patch machines!

```
>>>from nltk.book import *
```

```
>>> text1
```

```
<Text: Moby Dick by Herman Melville 1851>
```

```
>>> text1.name
```

```
'Moby Dick by Herman Melville 1851'
```

```
>>> text1.concordance("monstrous")
```

```
>>> dir(text1)
```

```
>>> text1.tokens
```

```
>>> text1.index("my")
```

```
4647
```

```
>>> sent2
```

```
['The', 'family', 'of', 'Dashwood', 'had', 'long', 'been', 'settled', 'in',  
 'Sussex', '.']
```

Classify Text

```
>>> def gender_features(word):  
...     return {'last_letter': word[-1]}  
  
>>> gender_features('Shrek')  
{'last_letter': 'k'}  
  
>>> from nltk.corpus import names  
  
>>> import random  
  
>>> names = ([ (name, 'male') for name in names.words('male.txt')] +  
...          [ (name, 'female') for name in names.words('female.txt')])  
  
>>> random.shuffle(names)
```

Featurize, train, test, predict

```
>>> featuresets = [(gender_features(n), g) for (n,g) in names]
```

```
>>> train_set, test_set = featuresets[500:], featuresets[:500]
```

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

```
>>> print nltk.classify.accuracy(classifier, test_set)
```

```
0.726
```

```
>>> classifier.classify(gender_features('Neo'))
```

```
'male'
```

from `nltk.corpus` import `reuters`

- Reuters Corpus: 10,788 news
1.3 million words.
- Been classified into 90 topics
- Grouped into 2 sets, "training" and "test"
- Categories overlap with each other

<http://nltk.googlecode.com/svn/trunk/doc/book/ch02.html>

Reuters

```
>>> from nltk.corpus import reuters
```

```
>>> reuters.fileids()
```

```
['test/14826', 'test/14828', 'test/14829', 'test/14832', ...]
```

```
>>> reuters.categories()
```

```
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa', 'coconut',  
 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn', 'cotton', 'cotton-oil',  
 'cpi', 'cpu', 'crude', 'dfl', 'dlr', ...]
```