MINDFUL LEARNING

python

A language is not worth knowing unless it teaches you to think differently

- Larry Wall, Randal Schwartz

Professional

python

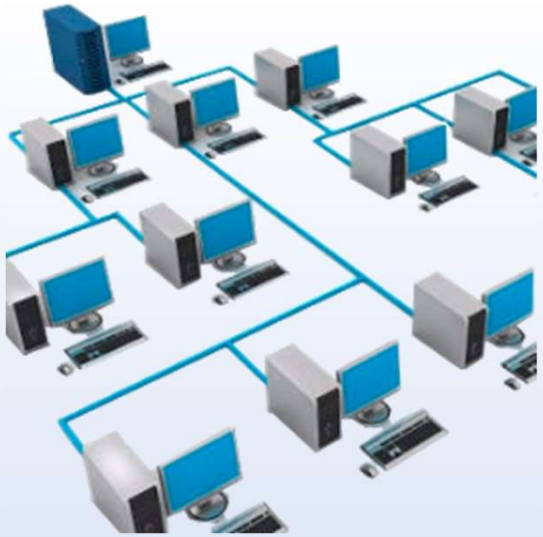
Network Programming

python

Introduction

- Internet happens to be the one of the primary application areas of Python programming language
- 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 that C/C++

The Challenge



- Communication between computers
- It’s just sending and receiving bits
- But how?

Mindful Learning

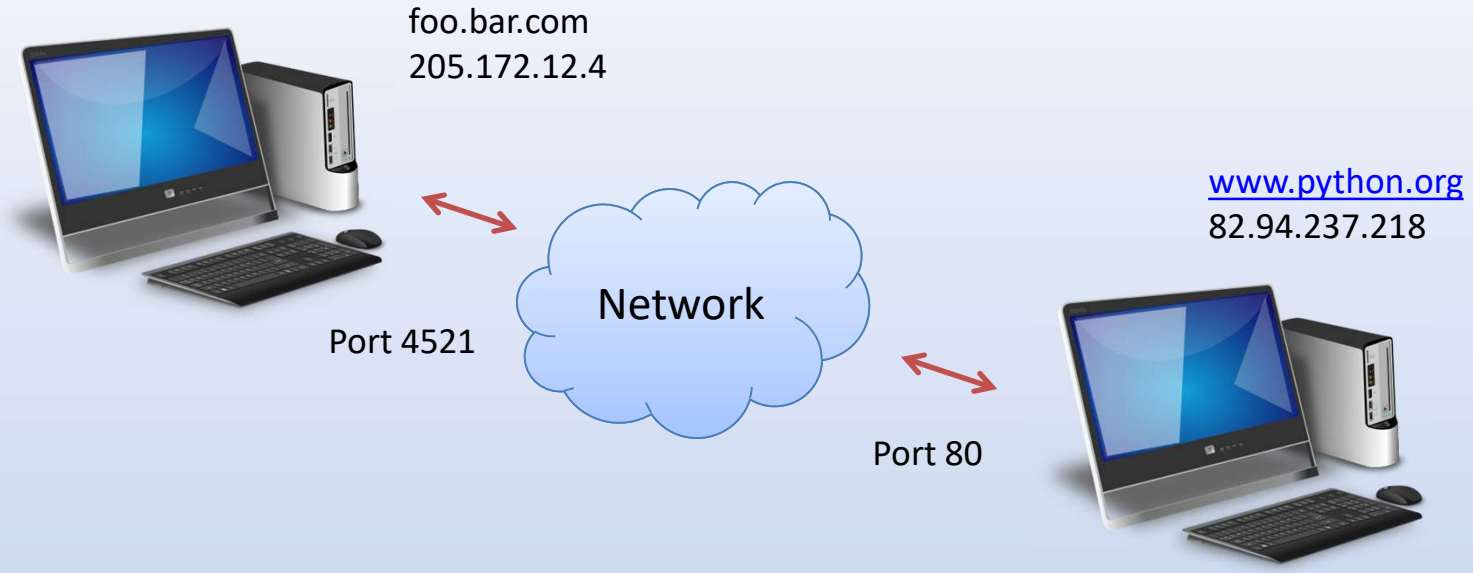
python

Two Main Issues

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

Network Addressing

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



Standard Ports

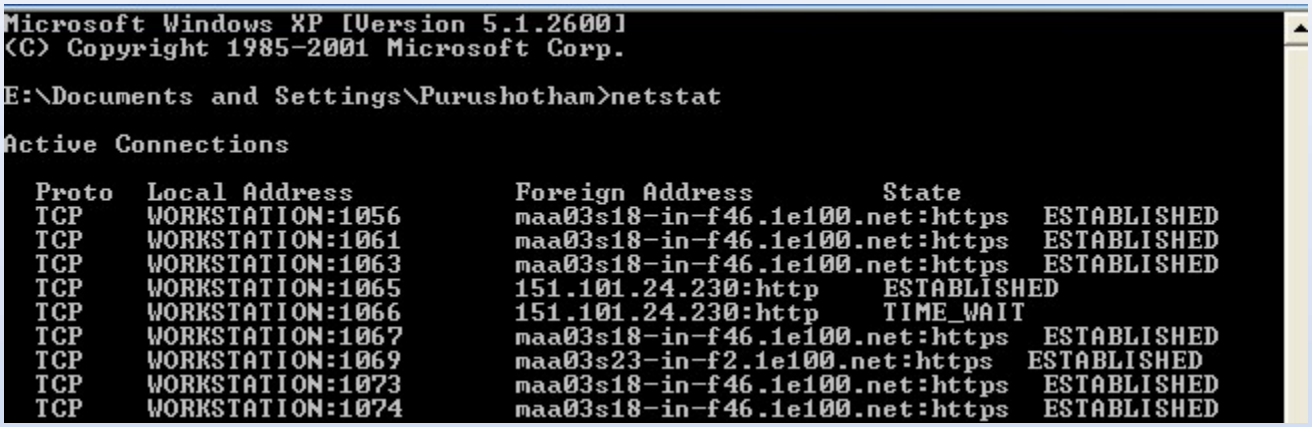
- Ports for common services are pre-assigned

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

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

Using netstat

- Use ‘netstat’ to view active network connections
- Can be used in command line mode in both Windows and Linux machines



Mindful Learning

python

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 generally use this convention to specify a network address

Client/Server Concept

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

python

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

MINDFUL LEARNING

python

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.
Escape character is '^]'.
GET /index.html HTTP/1.0

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

Type this and press ENTER a few times

Mindful Learning

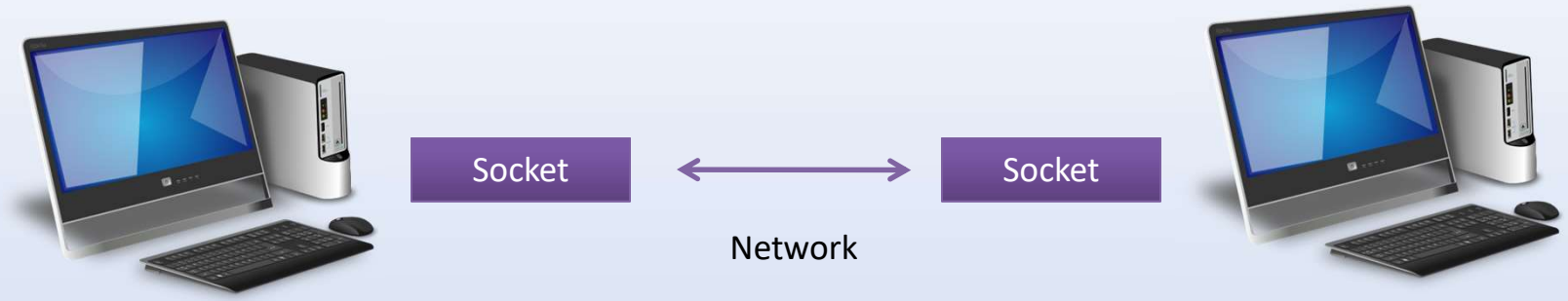
python

Data Transport

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

Sockets

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



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

Socket Basics

- To create a socket

```
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)
```

MINDFUL LEARNING

python

Socket Types

- Almost all code will use one of the following:

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

- Most common case: TCP Connection

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

Using a Socket

- Creating a socket is only the first step

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

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

TCP Client

MINDFUL LEARNING

python

- 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

Mindful Learning

python

Server Implementation

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

MINDFUL LEARNING

python

TCP Server

- A simple server

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

- Send a message back to a client

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

TCP Server

MINDFUL LEARNING

python

- Address binding

```
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()
```

Binds socket to specific address

TCP Server

MINDFUL LEARNING

python

- Addressing

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

Binds to localhost

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

TCP Server

MINDFUL LEARNING

python

- Start listening to connections

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

Tells OS to start listening for connections on the socket

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

TCP Server

MINDFUL LEARNING

python

- Accepting a new connection

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

Accept new client connections

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

TCP Server

- Client socket and address

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

Accept returns a pair:
(client_socket, addr)

c,a = s.accept()

("104.23.11.4",27743)
This is a network /port address of the client that is connected

<socket._socketobject object at 0x3be30>
This is a new socket that is used for data

TCP Server

MINDFUL LEARNING

python

- Sending data

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

Send data to client

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

TCP Server

MINDFUL LEARNING

python

- 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()
```

Closing the connection

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

TCP Server

MINDFUL LEARNING

python

- Waiting for next connection

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

Wait for next connection

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

Mindful Learning

python

Advanced Sockets

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

Partial Reads/Writes

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

```
>>> len(data)
1000000
>>> s.send(data)
37722
>>>
>>> data = s.recv(10000)
>>> len(data)
6420
>>>
```

Sent partial data

Received less than max

Partial Reads/Writes

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

```
# Client
...
s.send(data)
s.send(moredata)
...
# Server
...
data = s.recv(maxsize)
...
```

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

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

Sending All Data

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

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

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

MINDFUL LEARNING

python

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.

MINDFUL LEARNING

python

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)
```

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'
>>>
```

No data available

Data arrived

- Sometimes used for polling

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

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()
```

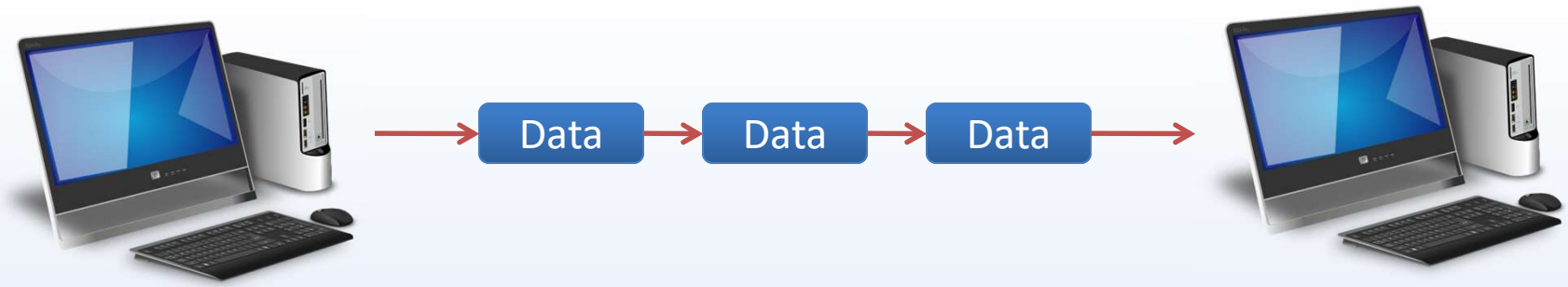
Sockets as Files

- 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.

Odds and Ends

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

UDP: Datagrams



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

UDP Server

MINDFUL LEARNING

python

- A simple datagram server

```
from socket import *
s = socket(AF_INET,SOCK_DGRAM)
s.bind(("",10000))
while True:
  data, addr = s.recvfrom(maxsize)
  resp = "Get off my lawn!"
  s.sendto(resp,addr)
```

Create datagram socket

Bind to a port

Wait for a message

Send response

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

UDP Client

MINDFUL LEARNING

python

- Sending a datagram to a server

```
from socket import *
s = socket(AF_INET,SOCK_DGRAM)
msg = "Hello World"
s.sendto(msg,("server.com",10000))
data, addr = s.recvfrom(maxsize)
```

Create datagram socket

Send message

Wait for a message

Returned data

Remote address

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

python

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)
```

- Rest of the programming interface is the same
- Address is just a "filename"

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

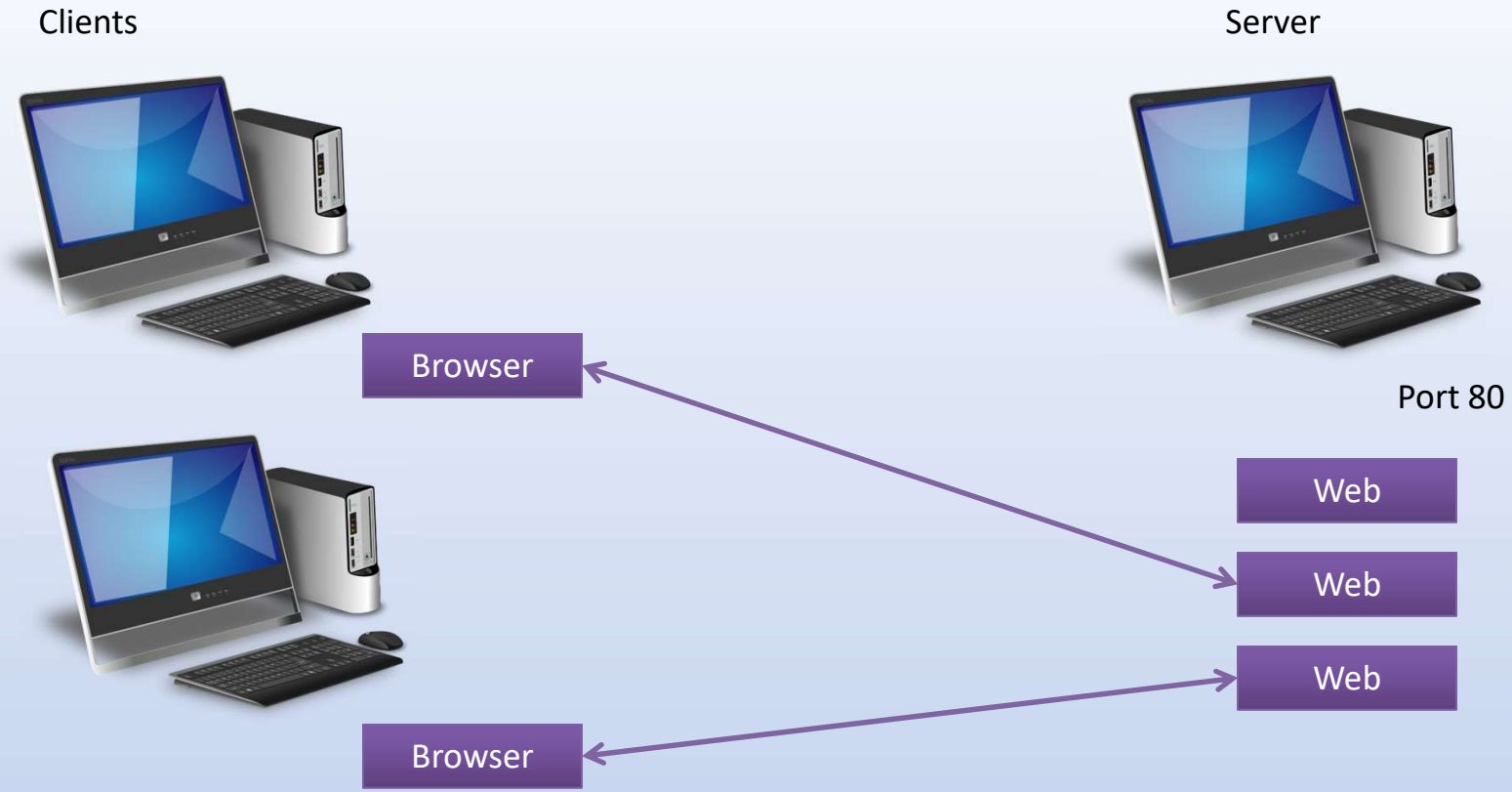
Raw Sockets

- 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
  ...
```

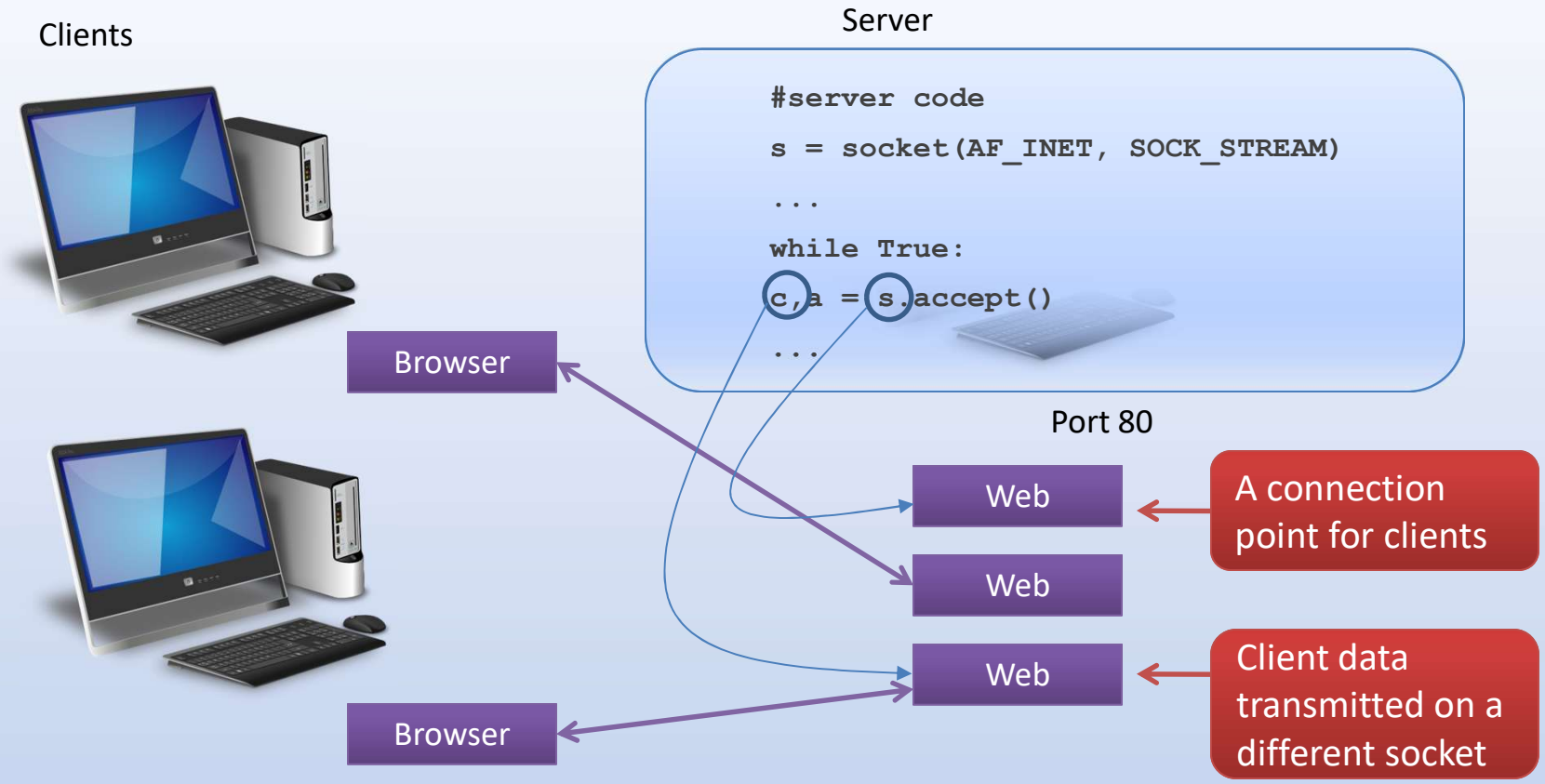
Sockets and Concurrency

- Servers usually handle multiple clients



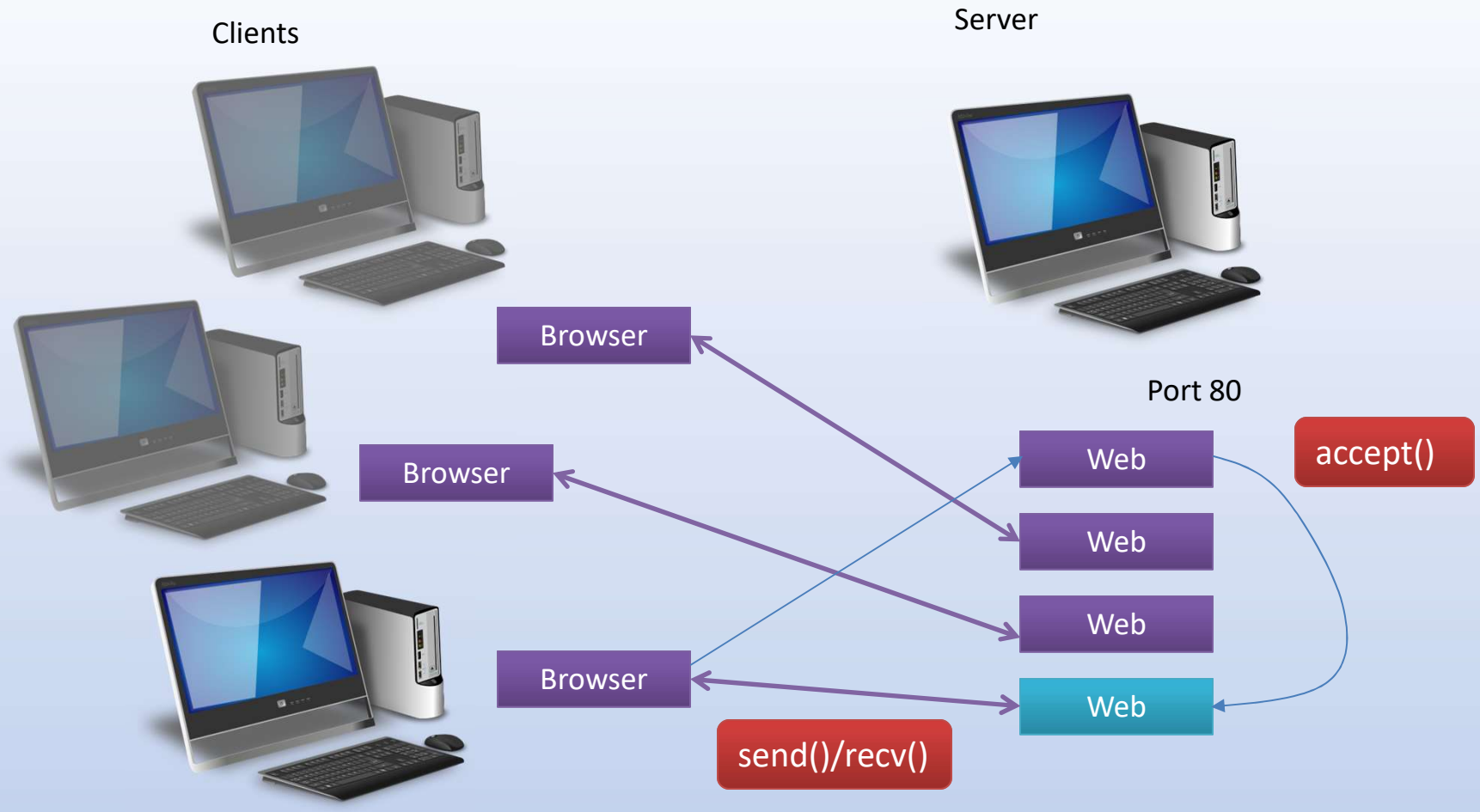
Sockets and Concurrency

- Each Client gets its own socket on server



Sockets and Concurrency

- 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

python

Threaded Server

- Each client is handled by a separate thread

```
import threading
from socket import *
def handle_client(c):
   ... whatever ...
   c.close()
   return
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
   c,a = s.accept()
   t = threading.Thread(target=handle_client,args=(c,))
```

Forking Server

- Each client is handled by a sub-process

```
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()
```

python

Asynchronous Server

- Server handles all clients in an event loop
- Frameworks such as Twisted is built upon this concept

```
import select
from socket import *
s = socket(AF_INET,SOCK_STREAM)
...
clients = [] # List of all active client sockets
while True:
   # Look for activity on any of my sockets
   input,output,err = select.select(s+clients,clients, clients)

   # Process all sockets with input
   for i in input:
   ...
   # Process all sockets ready for output
   for o in output:
   ...
```

Utility Functions

- Get the hostname of the local machine

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

- Get name information on a remote IP

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

- Get the IP address of a remote machine

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

Mindful Learning

python

What's Not Covered Here?

- socket module has hundreds of obscure
- socket control options, flags, etc.
- Many more utility functions
- IPv6 (Supported but it new and experimental)
- 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 sessions...

MINDFUL LEARNING

python

Advanced Networking Concepts

Overview

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

Problem with Sockets

- In part 1, 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

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

MINDFUL LEARNING

python

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()
```

MINDFUL LEARNING

python

SocketServer Example

- Handler Class

Server implemented by a handler class

```
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()
```

SocketServer Example

- Handler Class

Must inherit from BaseRequestHandler

```
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()
```

MINDFUL LEARNING

python

SocketServer Example

- handle() method

```
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()
```

Define handle() to implement the server action

MINDFUL LEARNING

python

SocketServer Example

- Client socket connection

```
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()
```

Socket object for client connection

This is a bare socket object

MINDFUL LEARNING

python

SocketServer Example

- Creating and running the 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()
```

Socket object for client connection

Runs server forever

Mindful Learning

python

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

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.

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()
```

MINDFUL LEARNING

python

Server Mixin Classes

- SocketServer defines these mixin classes

```
ForkingMixIn
ThreadingMixIn
```

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

```
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
from SocketServer import ThreadingMixIn
class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
  pass
serv = ThreadedHTTPServer(("",8080),
                          SimpleHTTPRequestHandler)
```

MINDFUL LEARNING

python

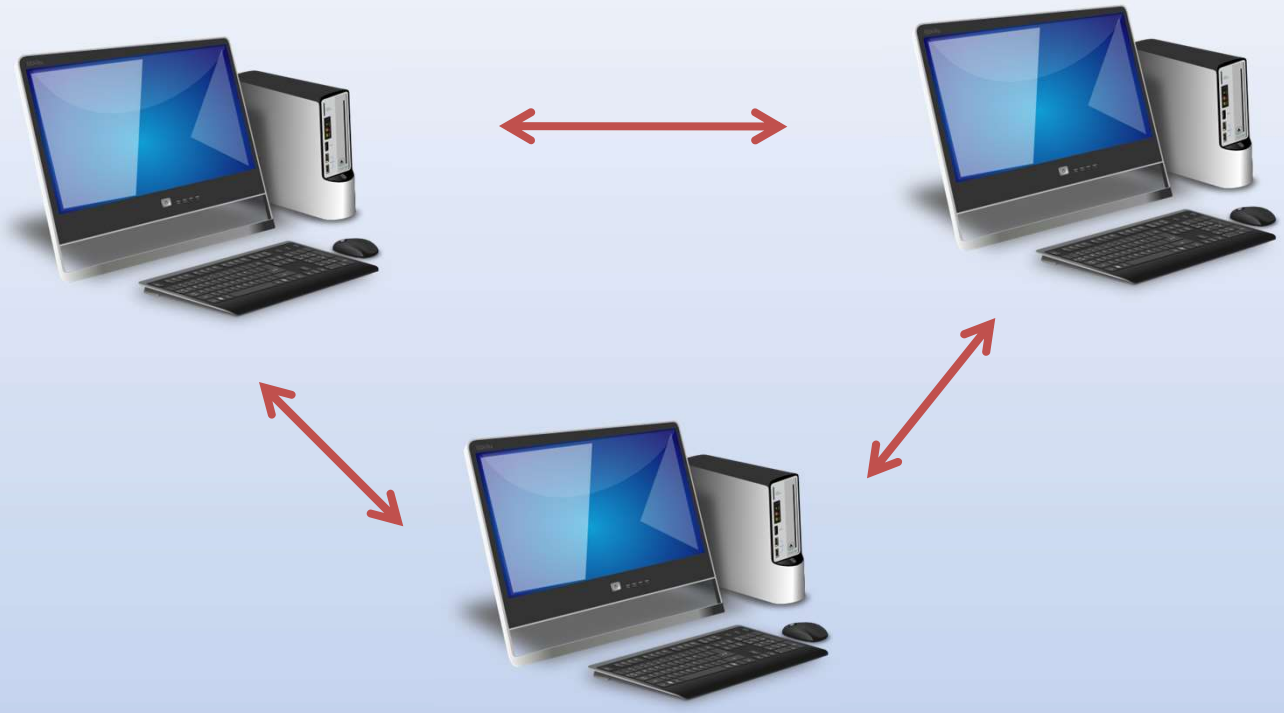
Server Subclassing

- 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()
```

Distributed Computing

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



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

XML-RPC

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

Simple XML-RPC

MINDFUL LEARNING

python

- 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"
>>>
```

MINDFUL LEARNING

python

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()
```

Mindful Learning

python

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

Mindful Learning

python

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

python

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

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.

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

MINDFUL LEARNING

python

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

MINDFUL LEARNING

python

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

Comments

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

Mindful Learning

python

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 inter-process communication, parallel computing, worker pools, etc.
- Will only show a few network features here

Mindful Learning

python

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()
```

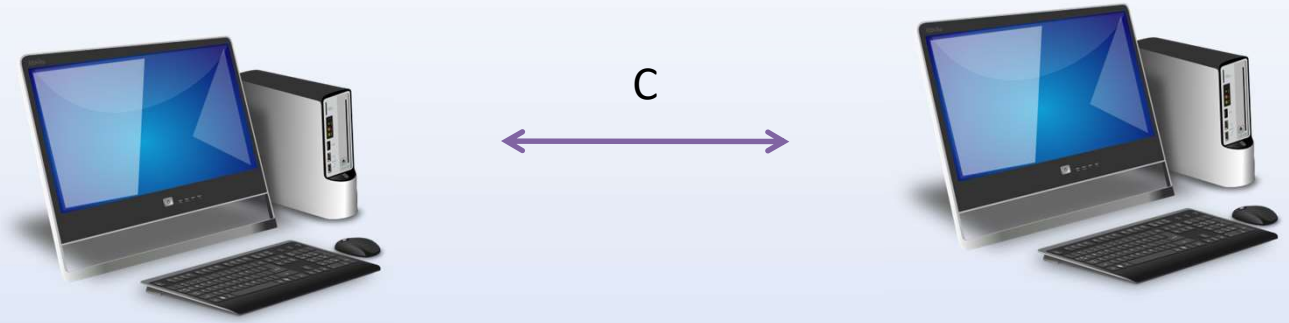
- Client process

```
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

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

MINDFUL LEARNING

python

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

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