A Tutorial for Python and Network Socket Programming

Which of these languages do you know?

- C or C++
- Java
- Perl
- Scheme
- Fortran
- Python
- Matlab

A TUTORIAL FOR PYTHON PROGRAMMING LANGUAGE

Overview

- Running Python and Output
- Data Types
- Input and File I/O
- Control Flow
- Functions

Hello World

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

```
>>> 'hello world!'
'hello world!'
```

Python Overview

From *Learning Python, 2nd Edition*:

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

The print Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including functions and classes (more on this later!)
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

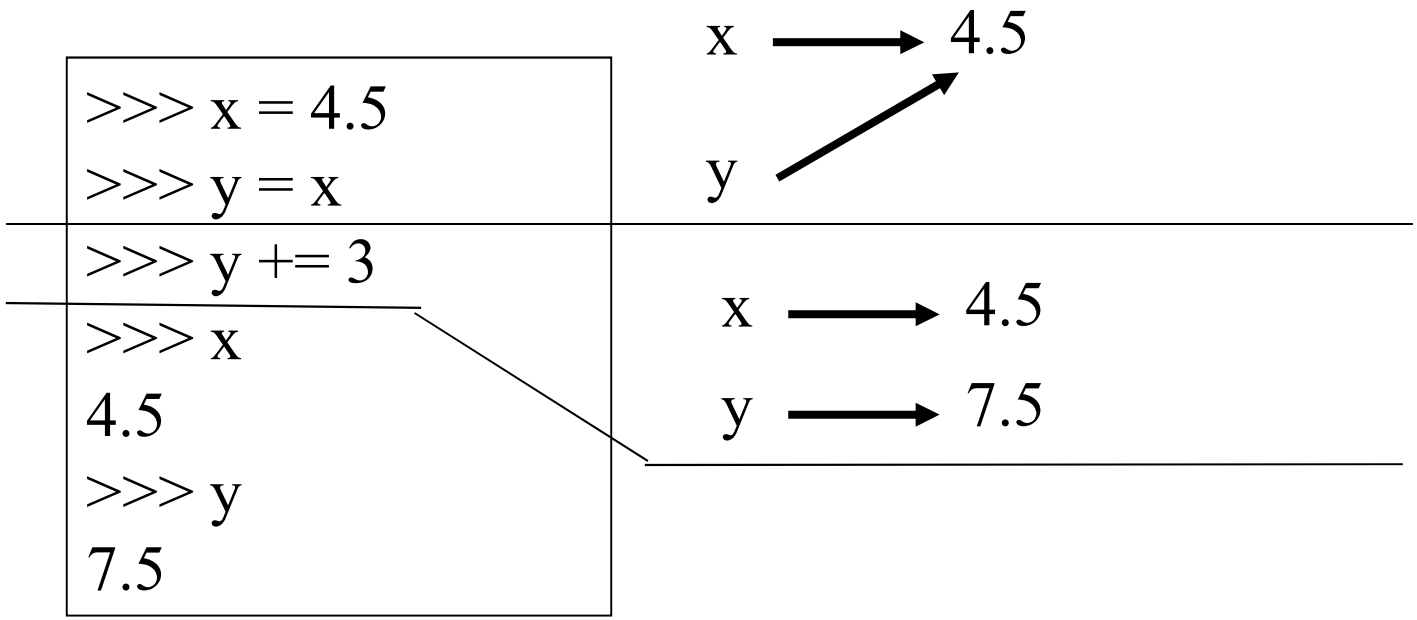
```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are immutable



String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)‘
>>> print s
And me too!
though I am much longer
than the others :)‘
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

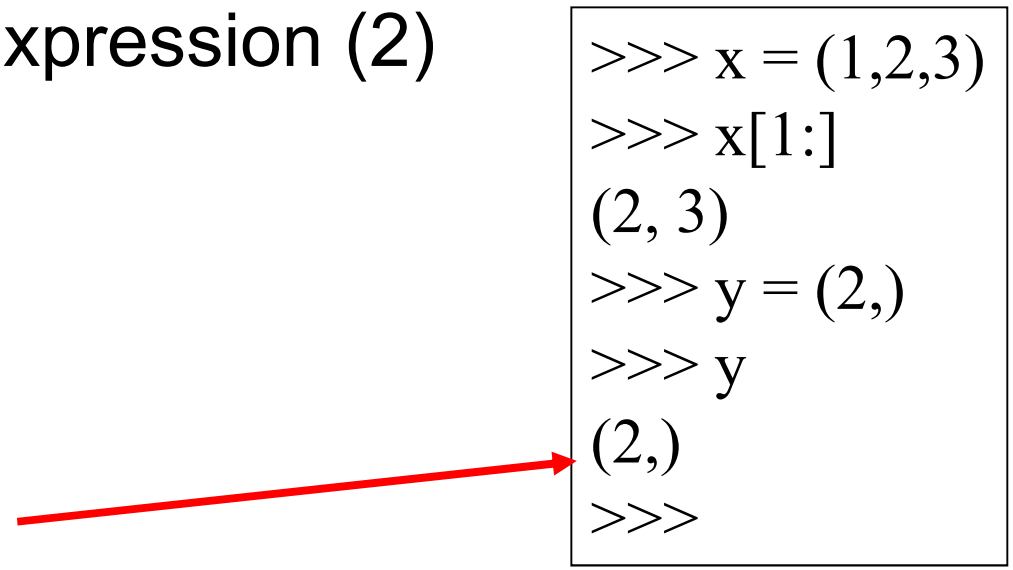
- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)



Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The **dictionary** has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print “Please input a selected year?"
youryear = int(raw_input("> "))

print "Hi %s!  the year you are selected
is %d years ago!" % (name, 2015 - youryear)
```

Files: Input

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable
- Python code files can be read into the interpreter using the **import** statement

Moving to Files

- In order to be able to find a module called myscripts.py, the interpreter scans the list sys.path of directory names.
- The module must be in one of those directories.

```
>>> import sys
>>> sys.path
['C:\\Python26\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python26.zip',
'C:\\Python26\\DLLs', 'C:\\Python26\\lib', 'C:\\Python26\\lib\\plat-win',
'C:\\Python26\\lib\\lib-tk', 'C:\\Python26', 'C:\\Python26\\lib\\site-packages']
>>> import myscripts
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import myscripts.py
ImportError: No module named myscripts.py
```

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print ‘y = ‘,
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

whileelse2.py

```
~: python whileelse2.py
1
```

For Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :
    print x
```

```
~: python forloop1.py
1
7
13
2
```

forloop2.py

```
for x in range(5) :
    print x
```

```
~: python forloop2.py
0
1
2
3
4
```

range(N) generates a list of numbers [0,1, …, n-1]

For Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print x
    break
else :
    print 'i got here'
```

elseforloop.py

```
~: python elseforloop.py
1
```

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

funcasparam.py

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

highorder.py

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
~: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Modules: Imports

import mymodule	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
from mymodule import x	Imports x from mymodule right into this namespace
from mymodule import *	Imports all elements of mymodule into this namespace

Python Socket Programming

Objectives

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

Overview

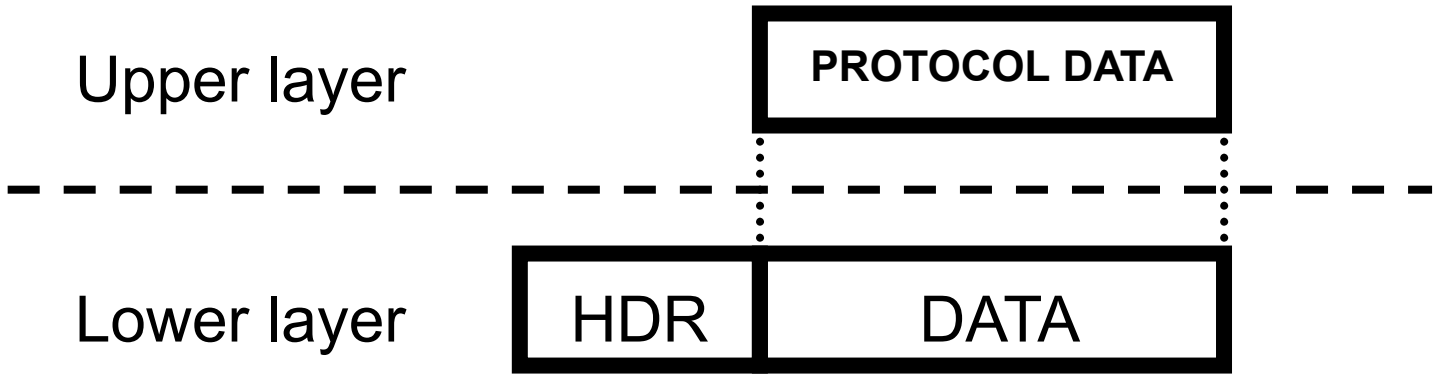
- Introduction to TCP/IP networking
  - IPv4
- Sockets: servers and clients
- Popular client libraries
- HTTP servers and clients

Network Layering

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

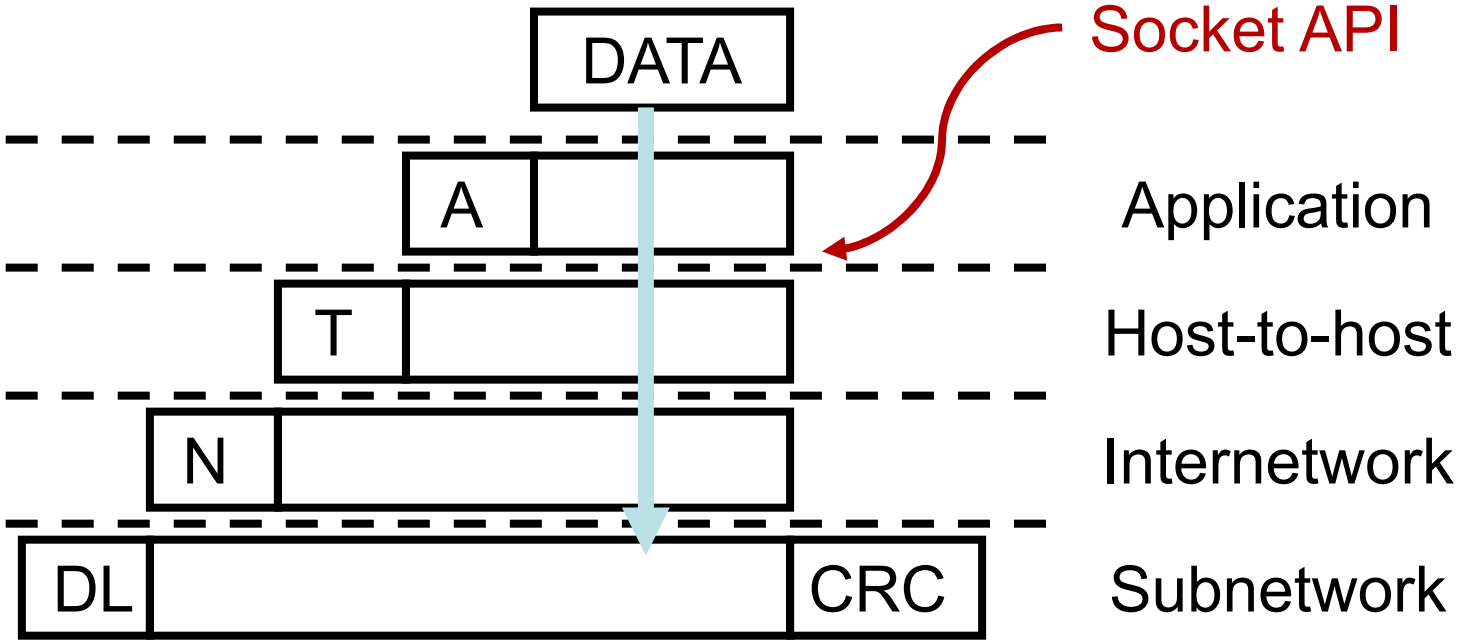
Inter-Layer Relationships

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



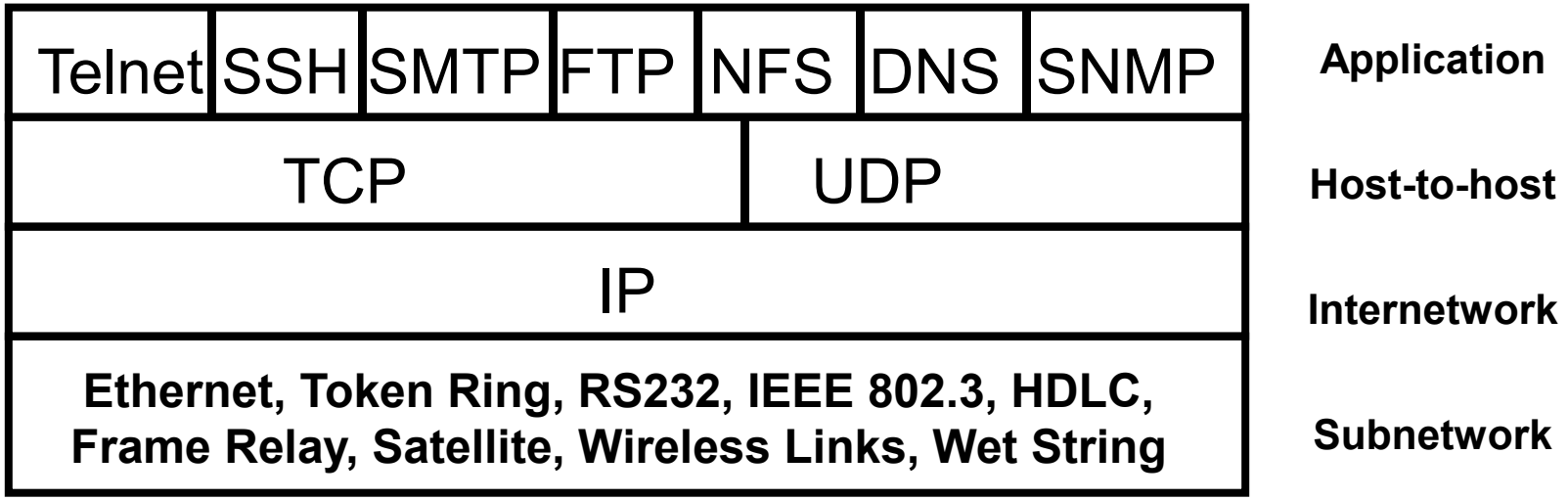
The TCP/IP Layering Model

- Simpler than OSI model, with four layers



TCP/IP Components

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



IP Characteristics

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

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

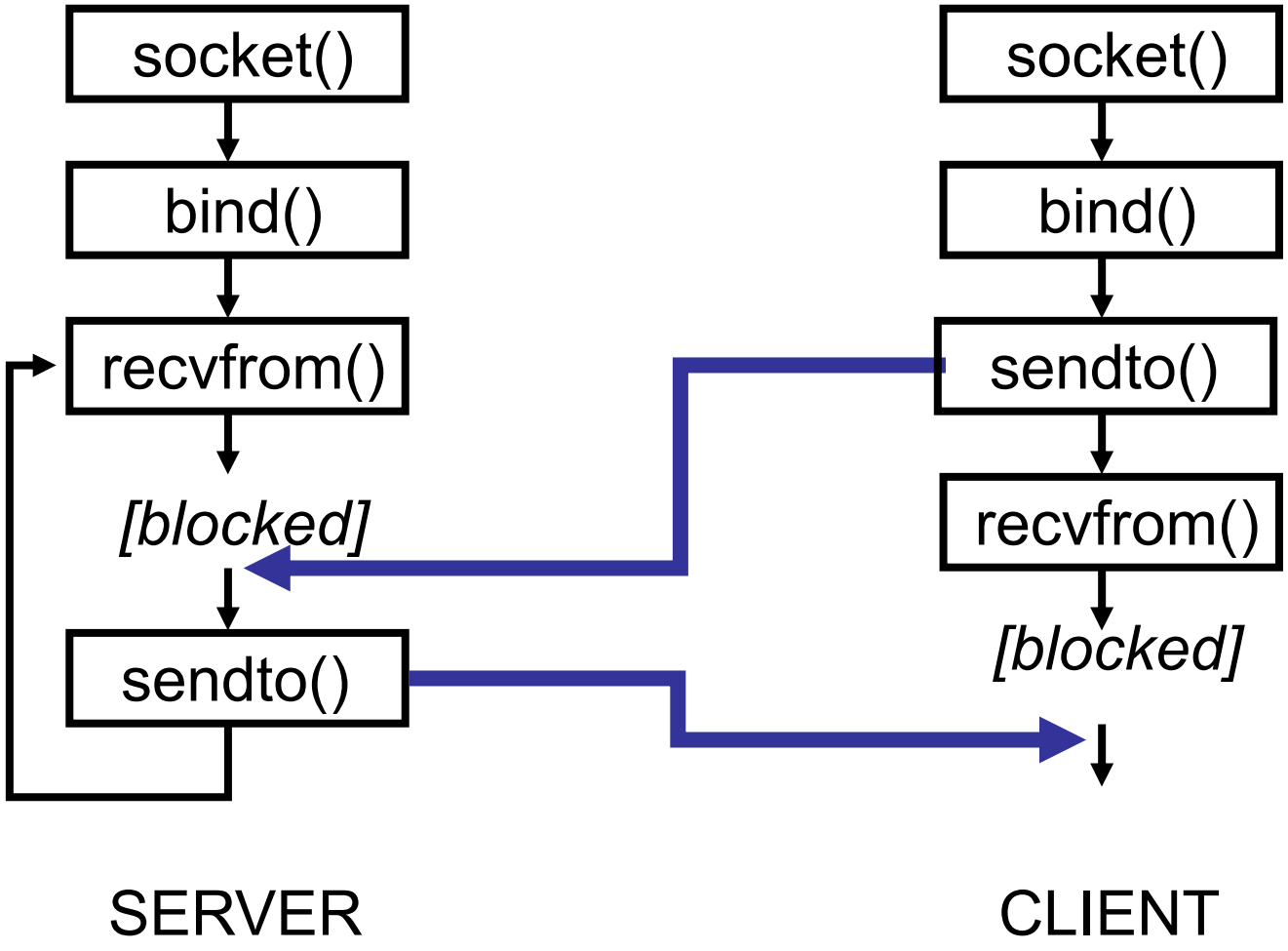
TCP Characteristics

- 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

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 (udpSimpleServer1.py)

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

- How much easier does it need to be?

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

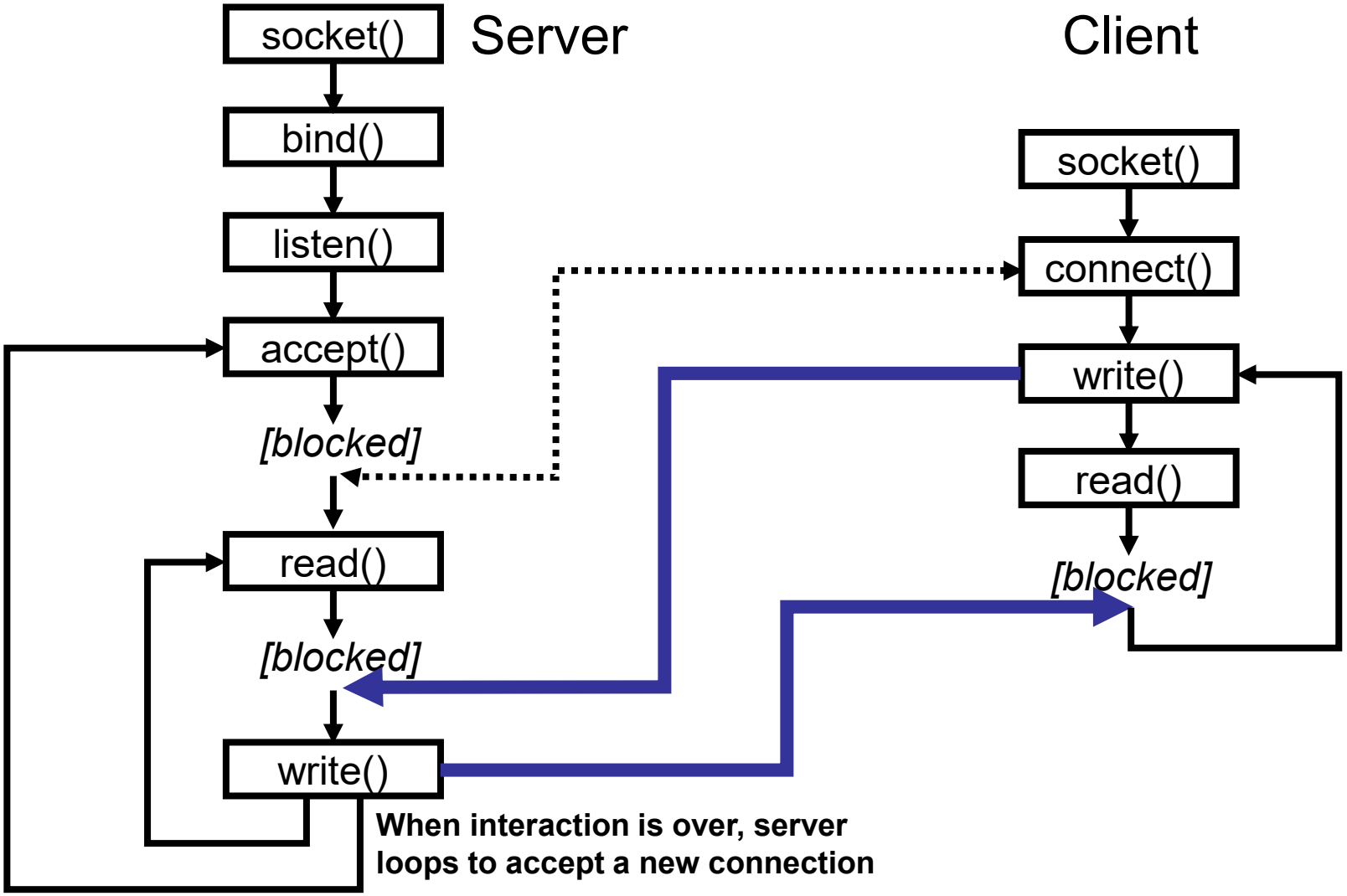
Simple Connectionless Client (udpSimpleClient1.py)

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

Exercise 1: UDP Client/Server

- Run the sample UDP client and server I have provided
  - **`udpSimpleServer1.py`**
  - **`udpSimpleClient1.py`**
- Additional questions:
  - How easy is it to change the port number and address used by the service?
  - What happens if you run the client when the server isn't listening?

Connection-Oriented Services



Connection-Oriented Server (tcpSimpleServer1.py)

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

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

Connection-Oriented Client (tcpSimpleClient1.py)

```
from socket import \
socket, AF_INET, SOCK_STREAM
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()***

Some socket Utility Functions

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

Handling Names & Addresses

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

Treating Sockets as Files

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

TCP Client/Server

- Run the sample UDP client and server I have provided
  - **`tcpSimpleServer1.py`**
  - **`tcpSimpleClient1.py`**
- Additional questions:
  - What happens if the client aborts (try entering CTRL/D as input, for example)?
  - Can you run two clients against the same server?

Summary of Address Families

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

Summary of Socket Types

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

Other socket.* Constants

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

Timeout Capabilities

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

Server Libraries

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

Using SocketServer Module

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

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

Writing a `handle()` Method

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

Skeleton Handler Examples

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

UDP Upper-Case SocketServer (`udpSimpleServer2.py`)

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

Change this function to alter server's functionality

- Note: this server never terminates!

UDP Upper-Case Client (udpSimpleClient2.py)

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

- Client interacts once then terminates
  - hangs if no response

TCP Upper-Case SocketServer (`tcpSimpleServer2.py`)

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

Change this function to alter server's functionality

TCP Upper-Case Client (tcpSimpleClient2.py)

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

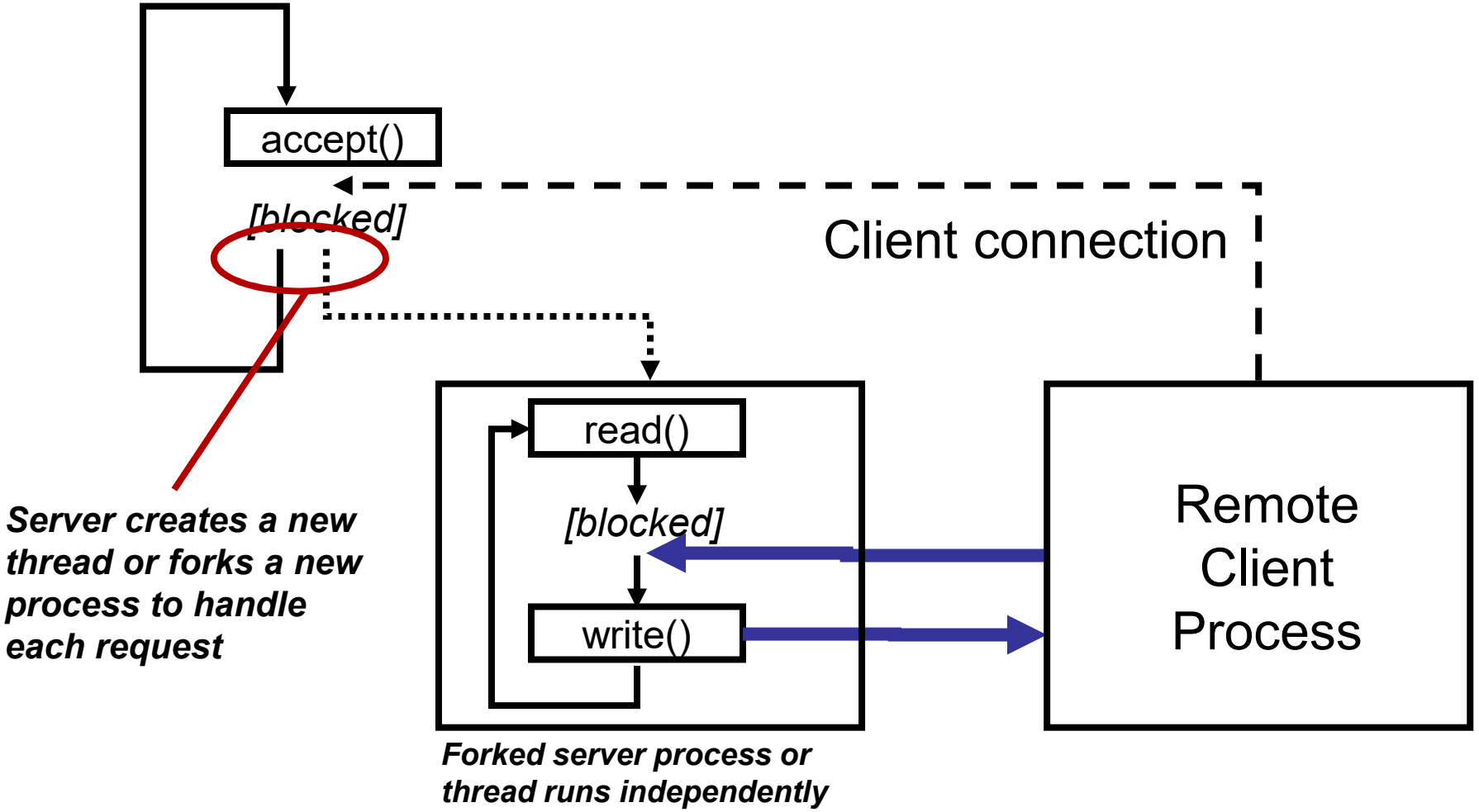
Exercise 3: SocketServer Usage

- Run the TCP and UDP SocketServer-based servers with the same clients you used before
  - `tcpSimpleServer2.py`
  - `tcpSimpleClient2`
- Additional questions:
  - Is the functionality any different?
  - Can the TCP server accept multiple connections?

Skeleton Server Limitations (1)

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

Simple Server Limitations (2)



Asynchronous Server Classes

- Use provided asynchronous classes

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

becomes

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

or

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

Implementation Details

- This is the implementation of all four servers:

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

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

More General Asynchrony

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

Network Client Libraries

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

General Client Strategy

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

HTTP and HTML Libraries

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

Using `urllib`

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

Interactive `urllib` Session

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

- Useful for testing & quick interactions

Using `urllib2`

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

`urllib2.Request` Class

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

Serving HTTP

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

The Simplest Web Server (simpleWebServer.py)

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

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

Summary

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