CSCE 3600
Principles of Systems Programming

Python

University of North Texas

Python Basics

Using Python

- Python can be executed interactively or via a script

```
which python
python –V
python -h
```

- Put a shebang into a Python script to indicate
  - This module can be run as a script
  - Whether it can be run only on python2, python3, or if it is Python 2/3 compatible
  - On POSIX, it is necessary if you want to run the script directly without invoking python executable explicitly
- If write shebang manually in script, then always use

```
#! /usr/bin/env python
```

3

Python Library

- Python is packaged with a large library of standard modules
  - String processing
  - Operating system interfaces
  - Networking
  - Threads
  - GUI
  - Database
  - Language services
- Many third party modules
  - XML
  - Numeric processing
  - Plotting/graphics
  - Etc.

- All of these are accessed using `import`

```
import string
…
a = string.split(x)
```

4

Python Structure

- Modules: Python source files or C extensions
  - `import`, top-level via `from`, `reload`
- Statements
  - Control flow
  - Create objects
  - Indentation matters – instead of `{ }`
- Objects
  - Everything is an object
  - Automatically reclaimed when no longer needed

Can find information and tutorials on Python at:
http://python.org/

5

Interactive Shell

- Statements and expressions can be typed at prompt

```
$ python
Python 2.7.15+ (default, Oct 7 2019, 17:39:04)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> print "Hello, world"
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
...
Use quit() or Ctrl-D (i.e. EOF) to exit
```

6

Modules

- When a Python program starts, it only has access to basic functions and classes
  - `int`, `dict`, `len`, `sum`, `range`, …
- Modules contain additional functionality
  - Use `import` to tell Python to load a module
  - Example

```
import math
```

```
math.pi
math.cos(0)
math.cos(math.pi)
dir(math)
help(math)
help(math.cos)
```

  - Use `from math import *` to remove `math` prefix

7

Arithmetic Operators and Precedence

- Arithmetic operators:

`+`	addition
`-`	subtraction/negation
`*`	multiplication
`/`	division (integer division)
`%`	modulus (i.e., remainder)
`**`	exponentiation

When integers and reals are mixed, result is a real number

- Precedence:
  - `* / % **` have a higher precedence than `+ -`
  - Parentheses can be used to force a certain order of evaluation

8

Math Commands

- Has useful commands for performing calculations

Command name	Description
`abs(`***value***`)`	absolute value
`ceil(`***value***`)`	rounds up
`cos(`***value***`)`	cosine, in radians
`floor(`***value***`)`	rounds down
`log(`***value***`)`	logarithm, base *e*
`log10(`***value***`)`	logarithm, base 10
`max(`***value1***`, `***value2***`)`	larger of two values
`min(`***value1***`, `***value2***`)`	smaller of two values
`round(`***value***`)`	nearest whole number
`sin(`***value***`)`	sine, in radians
`sqrt(`***value***`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

```
from math import *
```

9

Variables

- Assignment statement
- Syntax: `name = value`
- Examples

```
x = 5
gpa = 3.14
y = 4 << 3
z = y * 4.5
w = (y+z)/2.5
x = "Hello World"
```

- Variables are dynamically typed (no explicit typing, types may change during execution)
- Variables are just names for an object (not tied to a memory location like in C)

10

Print

- `print` produces text output on the terminal

```
print "Message"
print Expression
```

Prints given text message or expression value to terminal and moves cursor down to next line

```
print Item1, Item2, …, ItemN
```

  - Prints several messages and/or expressions on same line
- Examples

```
print "Hello, world!"
age = 20
print "Only", 65 - age, "years until retirement"
```

11

Reading Input

- `input` reads a number from user input
  - You can assign/store the result of `input` into a variable
- Example

```
age = input("How old are you? ")
print "Your age is", age
print "Only", 65 - age, "years until retirement"
```

- `raw_input` reads a string of text from user input
- Example

```
name = raw_input("Hello, what is your name? ")
print "Good afternoon,", name
```

12

The for Loop

- The `for` loop repeats set of statements over group of values
- Syntax: **`for varName in groupOfValues:`**
  `statements`
  - Indent statements to be repeated with spaces/tabs
  - `varName` assigns name to each value, refer in `statements`
  - `groupOfValues` can be range of integers, specified with `range` function

`range(start, stop, [step])`

inclusive exclusive

- Example

```
for x in range(1, 6):
    print x, "squared is", x * x
```

13

Text Processing

- The `for` loop can be used in text processing
  - Examine each character in a string in sequence
- Example

```
# print characters one at a time
for c in "text":
    print c
```

Indentation is used to denote bodies (i.e., blocks)

14

Control Flow

- Things that are `true`
  - The Boolean value `true`
  - All non-zero numbers
  - Any string containing at least one character
  - A non-empty data structure
- Things that are `false`
  - The Boolean value `false`
  - The numbers `0` (integer), `0.0` (float) and `0j` (complex)
  - The empty string ""
  - The empty list [], empty dictionary {}, and empty set `set()`

You can terminate a Python script using the built-in functions `quit()` or `exit()`

15

The if Statement

- The `if` statement executes a group of statements only if a certain condition is true
  - Otherwise, the statements are skipped
- Syntax:

```
if condition:
        statements
```

- Example

```
gpa = 3.4
if gpa > 2.0:
    print "Your application is accepted."
```

16

The if-elif-else Statement

- Syntax:
```
if condition:
    statements
elif condition:
    statements
else:
    statements
```
optional

There is no `switch` statement

- Example
```
gpa = 1.4
if gpa > 2.0:
    print "Welcome to UNT!"
else:
    print "Your application is denied."
```

`pass` is used to denote an empty body

17

The while Loop

- The `while` loop executes a group of statements as long as a condition is true
  - Good for indefinite loops (repeat an unknown number of times)
- Syntax:
```
while condition:
    statements
```
- Example
```
num = 1
while num < 50:
    print num,
    num = num *2
```

`break` and `continue` can be used just like in C/C++

18

Logic

- Many logical expressions use relational operators

Operator	Meaning	Example	Result
==	equals	1 + 1 == 2	True
!=	does not equal	3.2 != 2.5	True
<	less than	10 < 5	False
>	greater than	10 > 5	True
<=	less than or equal to	126 <= 100	False
>=	greater than or equal to	5.0 >= 5.0	True

- Logical expressions can be combined with logical operators

Operator	Example	Result
and	9 != 6 and 2 < 3	True
or	2 == 3 or -1 < 5	True
not	not 7 > 0	False

19

Strings

- A sequence text characters in a program
  - Strings start & end with quotation mark " or apostrophe ' characters
- Examples

```
"hello"
"This is also a string"
```

- May not span across multiple lines or contain a " character

```
"This is not
a legal string."
```

- A string can represent characters by preceding them with a \
  - **\t** tab character
  - **\n** new line character
  - **\"** quotation mark character
  - **** backslash character
- Example

```
"Hello\tthere\nHow are you?"
```

20

String Indexes

- Characters in a string are numbered with indices starting at 0
- Example

```
"Mr. Hall"
```

index	0	1	2	3	4	5	6	7
character	M	r	.		H	a	l	l

- Accessing an individual character of a string

```
varName[index]
```

- Example

```
print name, "starts with", name[0]
```

21

String Properties

- `len(string)` number of characters in a string
- `str.lower(string)` lowercase version of a string
- `str.upper(string)` uppercase version of a string

- Examples

```
name = "Martin Douglas Stepp"
length = len(name)
big_name = str.upper(name)
print big_name, "has", length, "characters"
```

22

String Operations find and split

- `string.find(substring)`
  - Find start of a substring
  - Can also use `string.find(substring, pos)` to start looking at position `pos`
- `string.split(substring)`
  - Split string into parts with substring delimiter
- Examples

```
greeting = "hello there"
greeting.find("e")
greeting.find("e", 3)
greeting.split(" ")
```

23

Strings are Read Only

```
>>> str = "andrew"
>>> str[0] = "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> str = "A" + str[1:]
>>> print str
Andrew
```

24

Strings and Numbers

- `ord(string)` converts a string into a number
- Example

```
ord("a") is 97, ord("A") is 65
```

- `chr(number)` converts a number into a string
- Example

```
chr(99) is "c"
```

25

Lists

- A compound data type:

```
[0]
[2.3, 4.5]
[5, "Hello", "there", 9.8]
[]
```

- Use `len()` to get the length of a list
- Example

```
names = ["Sarah", "Claire", "Michael"]
len(names)
```

- Use `[]` to index items in the list
  - Can use negative values (i.e., relative traceback) to move backwards from the last element

26

More Lists

- Append an element

```
names.append("Ben")
```

- Remove an element by extending the list

```
del names[1]
```

- Sort by default order

```
names.sort()
```

- Reverse the elements in the list

```
names.reverse()
```

- Insert an element at some specified position

```
names.insert(1, "Jorge")
```

27

Files

- The `open()` function

```
f = open("file1", "w")   # open file for writing
g = open("file2", "r")   # open file for reading
```

- Reading and writing data

```
f.write("Hello World")
data = g.read()       # read all data
line = g.readline()   # read a single line
lines = g.readlines() # read data as list of lines
```

- Formatted I/O
  - Use the % operator for strings (works like C printf)

```
for i in range(0, 10):
    f.write("2 x %d = %d\n" % (i, 2*i))
```

28

File Processing

- Read the entire contents of a file

```
file_text = open("account.txt").read()
```

- Read a line in a file

```
f = open("names.txt")
f.readline()
```

- Output to a file

```
infile = open("names.txt")
outfile = open("out.txt", "w")
for line in infile:
    outfile.write(line)
```

w	write
a	append
wb	write in binary
r	read (default)
rb	read in binary

29

File Processing (cont'd)

- Read a file line-by-line

```
for line in open("filename").readlines():
    statements
```

- Example

```
count = 0
for line in open("account.txt").readlines():
    count = count + 1
print "The file contains", count, "lines."
```

30

Functions

- Define functions in file above point used
  - Body of function should be indented consistently
  - `def` statement creates an object and assigns a name to reference it
- Arguments are optional
  - Multiple arguments are separated by commas
- If no return statement, then "None" is returned
  - Return values can be simple types (or tuples) and may be ignored by the caller
- Example

Tuples are just values separated by commas

```
def square(n):
        return n*n
print "The square of 3 is", square(3)
```

31

Function Example

```
#! /usr/bin/env python
# print a Fibonacci series up to n

def fib(n):
    a, b = 0, 1
    while (b < n):
       print b,
       a, b = b, a+b

n = input("Enter a number: ")
fib(n)
```

```
$ python fibSeries.py
Enter a number: 100
1 1 2 3 5 8 13 21 34 55 89
```

32

Classes

```
class Classname:
    statements
```

- Example

```
#! /usr/bin/env python
# a simple example class

class MyClass:
    i = 123
    def f(self):
        return "Hello World!"

print MyClass.i
x = MyClass()
print x.f()
```

First argument of method usually called `self`

33

Classes Example

```
class Account:
    def __init__(self, initial):
        self.balance = initial
    def deposit(self, amt):
        self.balance = self.balance + amt
    def withdraw(self, amt):
        self.balance = self.balance - amt
    def getbalance(self):
        return self.balance

a = Account(1000.00)
a.deposit(550.23)
a.deposit(100)
a.withdraw(50)
print a.getbalance()
```

34

Exceptions

- The try statement

```
try:
    f = open("file1.txt")
except IOError:
    print "Could not open "file1.txt"
```

- The raise statement

```
def factorial(n):
    if n < 0:
        raise ValueError, "Expected non-negative number"
    if (n <= 1): return 1
    else: return n*factorial(n-1)
```

- Uncaught exception

```
>>> factorial(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in factorial
ValueError: Expected non-negative number
>>>
```

35

Operating System Services

36

Operating System Services

- Python provides a wide variety of OS interfaces
  - Basic system calls
  - Operating environment
  - Processes
  - Timers
  - Signal handling
  - Error reporting
  - Users and passwords
- Implementation
  - A large portion of this functionality is contained in the `os` module
  - The interface is based on POSIX
  - Not all functions are available on all platforms

37

Process Management

- **`fork-exec-wait`**

```
os.fork()              # create a child process
os.execv(path, args)   # execute a process
os.execve(path, args, env)
os.execvp(path, args)  # execute a process, use default path
os.execvpw(path, args, env)
os.wait(pid)           # wait for child process
os.waitpid(pid, opts)  # wait for state change of child
os.system(command)     # execute system command
os._exit(n)            # exit immediately with status n
```

- Example

```
import os
pid = os.fork()      # create child
if pid == 0:
    # child process
    os.execvp("ls", ["ls", "-l"])
else:
    os.wait()
    print "done"
```

38

Pipes

- **`os.popen()`** function

```
f = popen("ls -l", "r")
data = f.read()
print data
f.close()
```

Opens a pipe to or from a command and returns a file-object

- The **`popen2`** module
  - Spawns processes, provides hooks to `stdin`, `stdout`, and `stderr`

```
popen2(cmd)   # run cmd, return (stdout, stdin)
popen3(cmd)   # run cmd, return (stdout, stdin, stderr)
```

- Example

```
…
(o, i) = popen2.popen2("wc")
i.write(data)        # write to child's input
i.close()
result = o.read()    # get child's output
print result
o.close()
```

39

Signal Handling

- The signal module

```
signal.signal(signalnum, handler) # set signal handler
signal.alarm(time)      # schedule SIGALRM signal
signal.pause()          # go to sleep until signal
signal.getsignal(signalnum)       # get signal handler
```

- Example

```
import signal
interval = 1
ticks = 0
def alarm_handler(signo, frame):
    global ticks
    print "Alarm ", ticks
    ticks = ticks + 1
    signal.alarm(interval)  # schedule new alarm

signal.signal(signal.SIGALRM, alarm_handler)
signal.alarm(interval)
while 1:
    pass
```

40

Signal Handling

- Ignoring signals

```
signal.signal(signo, signal.SIG_IGN)
```

- Default behavior

```
signal.signal(signo, signal.SIG_DFL)
```

- Comments
  - Signal handlers remain installed until explicitly reset
  - Signals are only handled between atomic instructions of interpreter
  - Certain signals cannot be handled from Python (e.g., SIGSEGV)
  - Python handles a number of signals on its own (e.g., SIGINT, SIGTERM)
  - Mixing signals and threads is extremely problematic – only main thread can deal with signals

41

Python Threads

- Thread scheduling
  - Tightly controlled by a global interpreter lock and scheduler
  - Only a single thread is allowed to be executing in the Python interpreter at once
  - Thread switching only occurs between the execution of individual byte-codes
  - Most I/O operations do not block
- Comments
  - Python threads are somewhat more restrictive than in C
  - Effectiveness may be limited on multiple CPUs (due to interpreter lock)
  - Threads can interact strangely with other Python modules (especially signal handling)
  - Not all extension modules are thread-safe

42

The Thread Module

- The `thread` module provides low-level access to threads
  - Thread creation
  - Simple mutex locks
- Creating a new thread

  `thread.start_new_thread(func.[args [.kwargs]])`
  - Executes a function in a new thread
  - Example

```
import thread
import time
def print_time(delay):
    while 1:
       time.sleep(delay)
       print time.ctime(time.time())

thread.start_new_thread(print_time, (5,))
while 1:
    pass
```

43

The Thread Module

- Thread termination
  - Thread silently exits when the function returns
  - Thread explicitly exit by calling `thread.exit()` or `sys.exit()`
  - Uncaught exception causes thread termination (prints error message)
  - Other threads continue to run even if one had an error
- Simple locks

  `allocate_lock()` creates a lock object, initially unlocked

```
import thread
lk = thread.allocate_lock()
def some_example():
    lk.acquire()      # acquire lock
    critical section
    lk.release()      # release lock
```

- Only one thread can acquire lock at once
- Threads block indefinitely until lock becomes available

44

The Thread Module

- The main thread
  - When Python starts, it runs as a single thread of execution
  - This is called the "main thread"
- Termination of the main thread
  - If the main thread exits and other threads are active, the behavior is system dependent
  - Usually, this immediately terminates the execution of all other threads without cleanup
  - Cleanup actions of the main thread may be limited as well
- Signal handling
  - Signals can only be caught and handled by the main thread of execution
  - Otherwise, you will get an error (in the signal module)
  - The keyboard-interrupt can be caught by any thread (non-deterministically)

45

Network Programming

46

Network Overview

- Python provides a wide assortment of network support
  - Low-level programming with sockets
  - Support for existing network protocols (HTTP, FTP, SMTP, etc.)
  - Web programming (CGI scripting and HTTP servers)
  - Data encoding

47

Socket Example

- The socket module
  - Provides access to low-level network programming functions
  - Example

Server returns the current time

```
from socket import *
import time

s = socket(AF_INET, SOCK_STREAM)
s.bind(("", 8888))   # bind to port 8888
s.listen(5)       # start listening

while 1:
    client,addr = s.accept() # wait for connection
    print "Connection received from ", addr
    client.send(time.ctime(time.time()))
    client.close()
```

  - Socket first opened by server is not same one used to exchange data
  - Instead, `accept()` function returns a new socket for this client

48

Socket Example (cont'd)

- Client program
  - Connect to time server and get current time

```
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.connect(("cse05.cse.unt.edu", 8888))
tm = s.recv(1024)    # receive up to 1024 bytes
s.close()
print "The time is ", tm
```

  - Once connection is established, server/client communicate using `send()` and `recv()`

49

The Socket Module

- Socket methods

```
s.accept()          # accept new connection
s.bimd(addr)        # bind to address and port
s.close()           # close socket
s.connect(addr)     # connect to remote socket
s.fileno()          # return file descriptor
s.getpeername()     # get name of remote machine
s.getsockname()     # get socket address
s.getsockopt(…)     # get socket options
s.listen(backlog)   # start listening for connections
s.makefile(mode)    # turn socket into file-object
s.recv(bufsize)     # receive data
s.recvfrom(bufsize) # receive data (UDP)
s.send(string)      # send data
s.sendto(str, addr) # send packet (UDP
s.setblocking(flag) # set blocking/nonblocking mode
s.setsockopt(...)   # set socket options
s.shutdown(how)     # shutdown one or both connections
```

50

Socket Basics

- To create a socket

```
import socket
s = socket.socket(addr_family, type)
```

- Address families

`socket.AF_UNIX`	Unix domain
`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)
```

51

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.
```

Server message

52

TCP Server

- 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 the socket to a specific address

- 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, can bind to a specific address

53

TCP Server

- Start listening for 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
  - Not related to max number of clients

54

TCP Server

- 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 a new client connection

- `s.accept()` blocks until connection received
  - accept returns a pair `(client_socket, addr)`, where `addr` is the network/port address of the client that connected
- Server sleeps if nothing is happening

55

TCP Server

- 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 for transmitting data as the server socket is only used for accepting new connections

56

TCP Server

- 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()
```

Close client connection

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

57

TCP Server

- Waiting for the 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

58

TCP Client

- How to make an outgoing connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.connect(("",9000))        # Connect
data = s.recv(10000)        # Get response
print "Server message:", data
s.close()
```

- `s.connect(addr)` makes a connection
- Once connected, use `send()`, `recv()` to transmit and receive data
- `close()` shuts down the connection

59

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
>>>
```

Send partial data

Received less than max

60

Partial Reads/Writes

- Be aware that for TCP, the data stream is continuous (that is, 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.

61

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.)
- 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)

62

Data Reassembly

- Receivers often need to reassemble messages from a series of small chunks
- Here is a sample programming template to accomplish this

```
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)
```

- You can also use string concat (+=), but it is slower

63

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)
```

64

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 blocks

```
>>> 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'
>>>
```

- Sometimes used for polling

65

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))
>>>
```

66

UDP Server

- A simple datagram server

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

Create datagram socket

Bind to a specific port

Wait for a message

Send response (optional)

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

67

UDP Client

- Sending a datagram to a server

```
import socket
serverName = raw_input("What CSE machine connecting to? ")
serverPort = 12000
clientSocket = socket.socket(socket.AF_INET,
                             socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence: ')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

Returned data

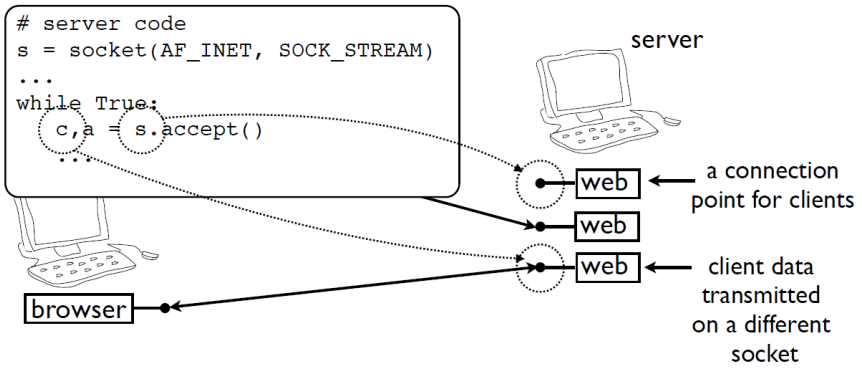
Remote address

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

68

Sockets and Concurrency

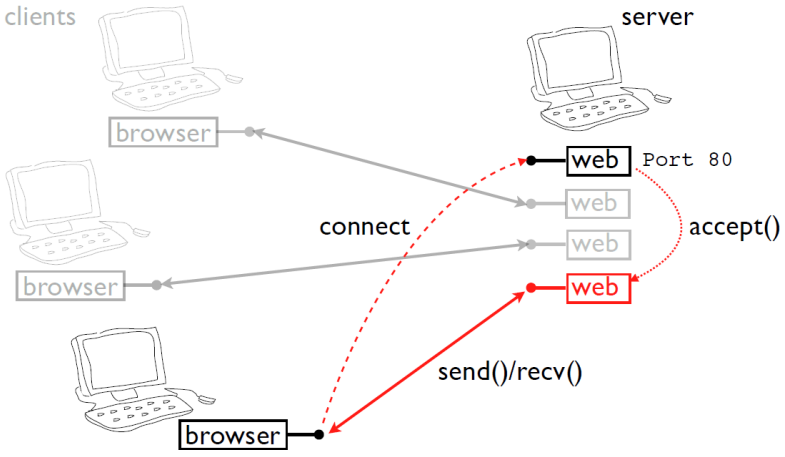
- Each client gets its own socket on the server



69

Sockets and Concurrency

- New connections make a new socket



70

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,))
```

71

Forking Server

- Each client is handled by a subprocess

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

- Note that some critical details have been omitted

72

Asynchronous Server

- Server handles all clients in an event loop

```
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:
    ...
```

73

Utility Functions

- Get the hostname of the local machine

```
>>> import socket
>>> socket.gethostname()
'cse04'
>>>
```

- Get the IP address of a remote machine

```
>>> import socket
>>> socket.gethostbyname("www.unt.edu")
'129.120.231.230'
>>>
```

- Get name information on a remote IP

```
>>> import socket
>>> socket.gethostbyaddr("129.120.151.98")
('cse05.cse.unt.edu', ['cse05'], ['129.120.151.98'])
>>>
```

74