**Advanced Python Programming**

**David M. Beazley**

**Department of Computer Science**

**University of Chicago**

**beazley@cs.uchicago.edu**

**O’Reilly Open Source Conference**

**July 17, 2000**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 1

July 17, 2000, beazley@cs.uchicago.edu

**Overview**

**Advanced Programming Topics in Python**

A brief introduction to Python

Working with the filesystem.

Operating system interfaces

Programming with Threads

Network programming

Database interfaces

Restricted execution

Extensions in C.

**This is primarily a tour of the Python library**

Everything covered is part of the standard Python distribution.

Goal is to highlight many of Python’s capabilities.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 2

July 17, 2000, beazley@cs.uchicago.edu

**Preliminaries**

**Audience**

Experienced programmers who are familiar with advanced programming topics in other languages.

Python programmers who want to know more.

Programmers who aren’t afraid of gory details.

**Disclaimer**

This tutorial is aimed at an advanced audience

I assume prior knowledge of topics in Operating Systems and Networks.

Prior experience with Python won’t hurt as well.

**My Background**

I was drawn to Python as a C programmer.

Primary interest is using Python as an interpreted interface to C programs.

Wrote the "Python Essential Reference" in 1999 (New Riders Publishing).

All of the material presented here can be found in that source.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 3

July 17, 2000, beazley@cs.uchicago.edu

**A Very Brief Tour of Python**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 4

July 17, 2000, beazley@cs.uchicago.edu

**Starting and Stopping Python**

**Unix**

unix % **python**

Python 1.5.2 (#1, Sep 19 1999, 16:29:25) [GCC 2.7.2.3] on linux2

Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

>>>

**On Windows and Macintosh**

Python is launched as an application.

An interpreter window will appear and you will see the prompt.

**Program Termination**

Programs run until EOF is reached.

Type Control-D or Control-Z at the interactive prompt.

Or type

raise SystemExit

O’Reilly OSCON 2000, Advanced Python Programming, Slide 5

July 17, 2000, beazley@cs.uchicago.edu

**Your First Program**

**Hello World**

>

>> **print "Hello World"**

Hello World

>>>

**Putting it in a file**

#

hello.py

print "Hello World"

**Running a file**

unix % python hello.py

**Or you can use the familiar #! trick**

#

!/usr/local/bin/python

print "Hello World"

O’Reilly OSCON 2000, Advanced Python Programming, Slide 6

July 17, 2000, beazley@cs.uchicago.edu

**Variables and Expressions**

**Expressions**

Standard mathematical operators work like other languages:

3

+ 5

+ (5\*4)

3

3

Hello’ + ’World’

\*\* 2

’

**Variable assignment**

a = 4 << 3

b = a \* 4.5

c = (a+b)/2.5

a = "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.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 7

July 17, 2000, beazley@cs.uchicago.edu

**Conditionals**

**if-else**

#

Compute maximum (z) of a and b

if a < b:

z = b

else:

z = a

**The pass statement**

if a < b:

pass

# Do nothing

else:

z = a

**Notes:**

Indentation used to denote bodies.

pass used to denote an empty body.

There is no ’?:’ operator.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 8

July 17, 2000, beazley@cs.uchicago.edu

**Conditionals**

**elif statement**

if a == ’+’:

op = PLUS

elif a == ’-’:

op = MINUS

elif a == ’\*’:

op = MULTIPLY

else:

op = UNKNOWN

Note: There is no switchstatement.

**Boolean expressions: and, or, not**

if b >= a and b <= c:

print "b is between a and c"

if not (b < a or b > c):

print "b is still between a and c"

O’Reilly OSCON 2000, Advanced Python Programming, Slide 9

July 17, 2000, beazley@cs.uchicago.edu

**Basic Types (Numbers and Strings)**

**Numbers**

a = 3

b = 4.5

# Integer

# Floating point

c = 517288833333L # Long integer (arbitrary precision)

d = 4 + 3j

# Complex (imaginary) number

**Strings**

a = ’Hello’

b = "World"

# Single quotes

# Double quotes

c = "Bob said ’hey there.’" # A mix of both

d = ’’’A triple quoted string

can span multiple lines

like this’’’

e = """Also works for double quotes"""

O’Reilly OSCON 2000, Advanced Python Programming, Slide 10

July 17, 2000, beazley@cs.uchicago.edu

**Basic Types (Lists)**

**Lists of Arbitrary Objects**

a = [2, 3, 4]

b = [2, 7, 3.5, "Hello"]

c = []

d = [2, [a,b]]

e = a + b

# A list of integers

# A mixed list

# An empty list

# A list containing a list

# Join two lists

**List Manipulation**

x = a[1]

# Get 2nd element (0 is first)

# Return a sublist

y = b[1:3]

z = d[1][0][2]

b[0] = 42

# Nested lists

# Change an element

O’Reilly OSCON 2000, Advanced Python Programming, Slide 11

July 17, 2000, beazley@cs.uchicago.edu

**Basic Types (Tuples)**

**Tuples**

f = (2,3,4,5)

# A tuple of integers

g = (,)

# An empty tuple

h = (2, [3,4], (10,11,12))

# A tuple containing mixed objects

**Tuple Manipulation**

x = f[1]

y = f[1:3]

z = h[1][1]

# Element access. x = 3

# Slices. y = (3,4)

# Nesting. z = 4

**Comments**

Tuples are like lists, but size is fixed at time of creation.

Can’t replace members (said to be "immutable")

O’Reilly OSCON 2000, Advanced Python Programming, Slide 12

July 17, 2000, beazley@cs.uchicago.edu

**Basic Types (Dictionaries)**

**Dictionaries (Associative Arrays)**

a = { }

# An empty dictionary

b = { ’x’: 3, ’y’: 4 }

c = { ’uid’: 105,

’

login’: ’beazley’,

’

name’ : ’David Beazley’

}

**Dictionary Access**

u = c[’uid’]

# Get an element

c[’shell’] = "/bin/sh"

if c.has\_key("directory"):

d = c[’directory’]

else:

# Set an element

# Check for presence of an member

d = None

d = c.get("directory",None)

# Same thing, more compact

O’Reilly OSCON 2000, Advanced Python Programming, Slide 13

July 17, 2000, beazley@cs.uchicago.edu

**Loops**

**The while statement**

while a < b:

#

Do something

a = a + 1

**The for statement (loops over members of a sequence)**

for i in [3, 4, 10, 25]:

print i

#

for c in "Hello World":

print c

Print characters one at a time

#

for i in range(0,100):

print i

Loop over a range of numbers

O’Reilly OSCON 2000, Advanced Python Programming, Slide 14

July 17, 2000, beazley@cs.uchicago.edu

**Functions**

**The def statement**

#

Return the remainder of a/b

def remainder(a,b):

q = a/b

r = a - q\*b

return r

#

Now use it

a = remainder(42,5)

# a = 2

**Returning multiple values**

def divide(a,b):

q = a/b

r = a - q\*b

return q,r

x,y = divide(42,5)

# x = 8, y = 2

O’Reilly OSCON 2000, Advanced Python Programming, Slide 15

July 17, 2000, beazley@cs.uchicago.edu

**Classes**

**The class statement**

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

**Using a class**

a = Account(1000.00)

a.deposit(550.23)

a.deposit(100)

a.withdraw(50)

print a.getbalance()

O’Reilly OSCON 2000, Advanced Python Programming, Slide 16

July 17, 2000, beazley@cs.uchicago.edu

**Exceptions**

**The try statement**

try:

f = open("foo")

except IOError:

print "Couldn’t open ’foo’. Sorry."

**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 (innermost last):

File "<stdin>", line 1, in ?

File "<stdin>", line 3, in factorial

ValueError: Expected non-negative number

>>>

O’Reilly OSCON 2000, Advanced Python Programming, Slide 17

July 17, 2000, beazley@cs.uchicago.edu

**Files**

**The open() function**

f = open("foo","w")

g = open("bar","r")

# Open a file for writing

# Open a file for reading

**Reading and writing data**

f.write("Hello World")

data = g.read()

line = g.readline()

lines = g.readlines()

# Read all data

# Read a single line

# Read data as a list of lines

**Formatted I/O**

Use the % operator for strings (works like C printf)

for i in range(0,10):

f.write("2 times %d = %d\n" % (i, 2\*i))

O’Reilly OSCON 2000, Advanced Python Programming, Slide 18

July 17, 2000, beazley@cs.uchicago.edu

**Modules**

**Large programs can be broken into modules**

#

numbers.py

def divide(a,b):

q = a/b

r = a - q\*b

return q,r

def gcd(x,y):

g = y

while x > 0:

g = x

x = y % x

y = g

return g

**The import statement**

import numbers

x,y = numbers.divide(42,5)

n = numbers.gcd(7291823, 5683)

import creates a namespace and executes a file.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 19

July 17, 2000, beazley@cs.uchicago.edu

**Python Library**

**Python is packaged with a large library of standard modules**

String processing

Operating system interfaces

Networking

Threads

GUI

Database

Language services

Security.

**And there are many third party modules**

XML

Numeric Processing

Plotting/Graphics

etc.

**All of these are accessed using ’import’**

import string

...

a = string.split(x)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 20

July 17, 2000, beazley@cs.uchicago.edu

**Quick Summary**

**This is not an introductory tutorial**

Consult online docs or Learning Python for a gentle introduction.

Experiment with the interpreter.

Generally speaking, most programmers don’t have trouble picking up Python.

**Rest of this tutorial**

A fearless tour of various library modules.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 21

July 17, 2000, beazley@cs.uchicago.edu

**String Processing**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 22

July 17, 2000, beazley@cs.uchicago.edu

**The string module**

**Various string processing functions**

string.atof(s)

# Convert to float

# Convert to integer

# Convert to long

string.atoi(s)

string.atol(s)

string.count(s,pattern)

string.find(s,pattern)

string.split(s, sep)

# Count occurrences of pattern in s

# Find pattern in s

# String a string

string.join(strlist, sep) # Join a list of string

string.replace(s,old,new) # Replace occurrences of old with new

**Examples**

s = "Hello World"

a = string.split(s)

# a = [’Hello’,’World’]

b = string.replace(s,"Hello","Goodbye")

c = string.join(["foo","bar"],":") # c = "foo:bar"

O’Reilly OSCON 2000, Advanced Python Programming, Slide 23

July 17, 2000, beazley@cs.uchicago.edu

**Regular Expressions**

**Background**

Regular expressions are patterns that specify a matching rule.

Generally contain a mix of text and special characters

foo.\*

# Matches any string starting with foo

# Match any number decimal digits

# Match a sequence of one or more letters

\

d\*

[a-zA-Z]+

**The re module**

Provides regular expression pattern matching and replacement.

Details follow.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 24

July 17, 2000, beazley@cs.uchicago.edu

**Regular Expressions**

**Regular expression pattern rules**

text

Match literal text

Match any character except newline

Match the start of a string

Match the end of a string

Match 0 or more repetitions

Match 1 or more repetitions

Match 0 or 1 repetitions

Match 0 or more, few as possible

Match 1 or more, few as possible

Match m to n repetitions

Match m to n repetitions, few as possible

Match a set of characters

.

^

$

\*

+

?

\*

+

{

{

[

[

?

?

m,n}

m,n}?

...]

^...]

Match characters not in set

Match A or B

Match regex in parenthesis as a group

A | B

(...)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 25

July 17, 2000, beazley@cs.uchicago.edu

**Regular Expressions**

**Special characters**

\

\

\

\

\

\

\

\

\

\

\

\

number

Matches text matched by previous group

Matches start of string

A

b

B

d

D

s

S

w

W

Z

\

Matches empty string at beginning or end of word

Matches empty string not at begin or end of word

Matches any decimal digit

Matches any non-digit

Matches any whitespace

Matches any non-whitespace

Matches any alphanumeric character

Matches characters not in \w

Match at end of string.

Literal backslash

**Raw strings**

Because of backslashes and special characters, raw strings are used.

Raw strings don’t interpret backslash as an escape code

expr = r’(\d+)\.(\d\*)’

# Matches numbers like 3.4772

O’Reilly OSCON 2000, Advanced Python Programming, Slide 26

July 17, 2000, beazley@cs.uchicago.edu

**The re Module**

**General idea**

Regular expressions are specified using syntax described.

Compiled into a regular expression "object".

This is used to perform matching and replacement operations.

**Example**

import re

pat = r’(\d+)\.(\d\*)’

r = re.compile(pat)

m = r.match(s)

if m:

# My pattern

# Compile it

# See if string s matches

#

Yep, it matched

..

.

else:

#

Nope.

...

O’Reilly OSCON 2000, Advanced Python Programming, Slide 27

July 17, 2000, beazley@cs.uchicago.edu

**The re Module (cont)**

**Regular Expression Objects**

Objects created by re.compile() have these methods

r.search(s [,pos [,endpos]])

r.match(s [,pos [,endpos]])

r.split(s)

# Search for a match

# Check string for match

# Split on a regex match

# Find all matches

r.findall(s)

r.sub(repl,s)

# Replace all matches with repl

When a match is found a ’MatchObject’ object is returned.

This contains information about where the match occurred.

Also contains group information.

**Notes**

The search method looks for a match anywhere in a string.

The match method looks for a match starting with the first character.

The pos and endpos parameters specify starting and ending positions for the search/match.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 28

July 17, 2000, beazley@cs.uchicago.edu

**The re Module (cont)**

**Match Objects**

Contain information about the match itself

But it is based on a notion of "groups"

**Grouping Rules**

(\d+)\.(\d\*)

This regular expression has 3 groups

group 0 : The entire regular expression

group 1 : The (\d+) part

group 2 : The (\d\*) part

Group numbers are assigned left to right in the pattern

**Obtaining match information**

m.group(n)

m.start(n)

m.end(n)

# Return text matched for group n

# Return starting index for group n

# Return ending index for group n

O’Reilly OSCON 2000, Advanced Python Programming, Slide 29

July 17, 2000, beazley@cs.uchicago.edu

**The re Module (cont)**

**Matching Example**

import re

r = re.compile(r’(\d+)\.(\d\*)’)

m = r.match("42.37")

a = m.group(0)

b = m.group(1)

c = m.group(2)

print m.start(2)

# Returns ’42.37’

# Returns ’42’

# Returns ’37’

# Prints 3

**A more complex example**

#

Replace URL such as http://www.python.org with a hyperlink

pat = r’(http://[\w-]+(\.[\w-]+)\*((/[\w-~]\*)?))’

r = re.compile(pat)

r.sub(’<a href="\\1">\\1</a>’,s)

# Replace in string

**Where to go from here?**

*Mastering Regular Expressions*, by Jeffrey Friedl

Online docs

Experiment

O’Reilly OSCON 2000, Advanced Python Programming, Slide 30

July 17, 2000, beazley@cs.uchicago.edu

**Working with Files**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 31

July 17, 2000, beazley@cs.uchicago.edu

**File Objects**

**open(filename [,mode])**

Opens a file and returns a file object

By default, opens a file for reading.

File open modes

"

"

"

"

"

r"

Open for reading

Open for writing (truncating to zero length)

Open for append

w"

a"

r+"

Open for read/write (updates)

w+"

Open for read/write (with truncation to zero length)

**Notes**

A ’b’ may be appended to the mode to indicate binary data.

This is required for portability to Windows.

"+" modes allow random-access updates to the file.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 32

July 17, 2000, beazley@cs.uchicago.edu

**File Objects**

**File Methods**

The following methods can be applied to an open file *f*

f.read([n])

f.readline([n])

f.readlines()

f.write(s)

f.writelines(ls)

f.close()

f.tell()

# Read at most n bytes

# Read a line of input with max length of n

# Read all input and return a list of lines

# Write string s

# Write a list of strings

# Close a file

# Return current file pointer

f.seek(offset [,where]) # Seek to a new position

#

#

#

# Return 1 if interactive terminal

# Flush output

where = 0: Relative to start

where = 1: Relative to current

where = 2: Relative to end

f.isatty()

f.flush()

f.truncate([size])

f.fileno()

# Truncate file to at most size bytes

# Return integer file descriptor

O’Reilly OSCON 2000, Advanced Python Programming, Slide 33

July 17, 2000, beazley@cs.uchicago.edu

**File Objects**

**File Attributes**

The following attributes provide additional file information

f.closed

f.mode

# Set to 1 if file object has been closed

# I/O mode of the file

f.name

# Name of file if created using open().

#

Otherwise, a string indicating the source

f.softspace

# Boolean indicating if extra space needs to be

#

printed before another value when using print.

**Other notes**

File operations on lines are aware of local conventions (\n\r vs. \n).

String data read and written to files may contain embedded nulls and other binary content.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 34

July 17, 2000, beazley@cs.uchicago.edu

**Standard Input, Output, and Error**

**Standard Files**

sys.stdin - Standard input

sys.stdout - Standard output

sys.stderr - Standard error

**These are used by several built-in functions**

print outputs to sys.stdout

input() and raw\_input() read from sys.stdin

s = raw\_input("type a command : ")

print "You typed ", s

Error messages and the interactive prompts go to sys.stderr

**You can replace these with other files if you want**

import sys

sys.stdout = open("output","w")

O’Reilly OSCON 2000, Advanced Python Programming, Slide 35

July 17, 2000, beazley@cs.uchicago.edu

**File and Path Manipulation**

**os.path - Functions for portable path manipulation**

abspath(path)

basename(path)

dirname(path)

normcase(path)

normpath(path)

split(path)

splitdrive(path)

splitext(path)

expanduser(path)

expandvars(path)

join(p1,p2,...)

# Returns the absolute pathname of a path

# Returns filename component of path

# Returns directory component of path

# Normalize capitalization of a name

# Normalize a pathname

# Split path into (directory, file)

# Split path into (drive, pathname)

# Split path into (filename, suffix)

# Expands ~user components

# Expands environment vars ’$name’ or ’${name}’

# Join pathname components

**Examples**

abspath("../foo")

basename("/usr/bin/python")

dirname("/usr/bin/python")

# Returns "/home/beazley/blah/foo"

# Returns "python"

# Returns "/usr/bin"

normpath("/usr/./bin/python") # Returns "/usr/bin/python"

split("/usr/bin/python")

splitext("index.html")

# Returns ("/usr/bin","python")

# Returns ("index",".html")

O’Reilly OSCON 2000, Advanced Python Programming, Slide 36

July 17, 2000, beazley@cs.uchicago.edu

**File Tests**

**os.path - Functions for portable filename inquires**

exists(path)

isabs(path)

isfile(path)

isdir(path)

islink(path)

ismount(path)

getatime(path)

getmtime(path)

getsize(path)

# Test for existence

# Return 1 if path is an absolute pathname

# Return 1 if path is a regular file

# Return 1 if path is a directory

# Return 1 if path is a symlink

# Return 1 if path is a mountpoint

# Get access time

# Get modification time

# Get file size in bytes

samefile(path1,path2) # Return 1 if path1 and path2 are the same file

sameopenfile(f1,f2)

# Return 1 if file objects f1 and f2 are same file.

**Notes:**

samefile() and sameopenfile() useful if file referenced by symbolic links or aliases.

The stat module provides lower-level functions for file inquiry.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 37

July 17, 2000, beazley@cs.uchicago.edu

**Globbing**

**glob module**

Returns filenames in a directory that match a pattern

import glob

a = glob.glob("\*.html")

b = glob.glob("image[0-5]\*.gif")

Pattern matching is performed using rules of Unix shell.

Tilde (~) and variable expansion is not performed.

**fnmatch module**

Matches filenames according to rules of Unix shell

import fnmatch

if fnmatch(filename,"\*.html"):

...

Case-sensitivity depends on the operating system.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 38

July 17, 2000, beazley@cs.uchicago.edu

**Low-Level File I/O**

**os.open(file [,flags [,mode]])**

Opens a file and returns an integer file descriptor

flags is the bitwise-or of the following

O\_RDONLY

O\_WRONLY

O\_RDWR

Open file for reading

Open file for writing

Open file for read/write

Append to the end of the file

Create file if it doesn’t exit

Don’t block on open,read, or write.

Truncate to zero length

Text mode (Windows)

O\_APPEND

O\_CREAT

O\_NONBLOCK

O\_TRUNC

O\_TEXT

O\_BINARY

Binary mode (Windows)

mode is file access mode according to standard Unix conventions

**Example**

import os

f = os.open("foo", O\_WRONLY | O\_CREAT, 0644)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 39

July 17, 2000, beazley@cs.uchicago.edu

**Low-Level I/O operations**

**The os module contains a variety of low-level I/O functions**

os.close(fd)

# Close a file

# Duplicate file descriptor fd

# Duplicate oldfd to newfd

os.dup(fd)

os.dup2(oldfd,newfd)

os.fdopen(fd [,mode [,bufsize]]) # Create a file object from an fd

os.fstat(fd)

os.fstatvfs(fd)

os.ftruncate(fd,size)

os.lseek(fd,pos,how)

# Return file status for fd

# Return file system info for fd

# Truncate file to given size

# Seek to new position

#

#

#

how = 0: beginning of file

how = 1: current position

how = 2: end of file

os.read(fd,n)

os.write(fd,str)

# Read at most n bytes

# Write data in str

**Notes**

The os.fdopen() and f.fileno() methods convert between file objects and file descriptors.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 40

July 17, 2000, beazley@cs.uchicago.edu

**Low-level File and Directory Manipulation**

**The os module also contains functions manipulating files and directories**

os.access(path,accessmode)

os.chmod(path,mode)

os.chown(path,uid,gid)

os.link(src,dst)

# Checks access permissions on a file

# Change file permissions

# Change owner and group permissions

# Create a hard link

os.listdir(path)

# Return a list of names in a directory

# Create a directory

os.mkdir(path [,mode])

os.remove(path)

os.rename(src,dst)

os.rmdir(path)

os.stat(path)

os.statvfs(path)

os.symlink(src,dst)

os.unlink(path)

os.utime(path,(atime,mtime)) # Change access and modification times

# Remove a file

# Rename a file

# Remove a directory

# Return file information

# Return filesystem information

# Create a symbolic link

# Remove a file (same as remove)

**Notes**

If you care about portability, better to use the os.path module for some of these operations.

Note all operations have been listed. Consult a reference.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 41

July 17, 2000, beazley@cs.uchicago.edu

**Other File-Related Modules**

**fcntl**

Provides access to the fcntl() system call and file-locking operations

import fcntl, FCNTL

Lock a file

fcntl.flock(f.fileno(),FCNTL.LOCK\_EX)

#

**tempfile**

Creates temporary files

**gzip**

Creates file objects with compression/decompression

Compatible with the GNU gzip program.

import gzip

f = gzip.open("foo","wb")

f.write(data)

f.close()

O’Reilly OSCON 2000, Advanced Python Programming, Slide 42

July 17, 2000, beazley@cs.uchicago.edu

**Strings and Files**

**The StringIO and cStringIO modules**

Provide a file-like object that reads/writes from a string buffer

Example:

import StringIO

f = StringIO.StringIO()

f.write("Hello World\n")

.

..

s = f.getvalue()

# Get saved string value

**Notes**

StringIO objects support most of the normal file operations

cStringIO is implemented in C and is significantly faster.

StringIO is implemented in Python and can be subclassed.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 43

July 17, 2000, beazley@cs.uchicago.edu

**Object Serialization and Persistence**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 44

July 17, 2000, beazley@cs.uchicago.edu

**Object Serialization**

**Motivation**

Sometimes you need to save an object to disk and restore it later.

Or maybe you need to ship it across the network.

**Problem**

Manual implementation requires a lot of work.

Must come up with some kind of encoding scheme.

Must write code to marshal objects to and from the encoding.

**Fortunately...**

Python provides several modules to do all of this for you

O’Reilly OSCON 2000, Advanced Python Programming, Slide 45

July 17, 2000, beazley@cs.uchicago.edu

**The pickle and cPickle Module**

**The pickle and cPickle modules serialize objects to and from files**

To serialize, you ’pickle’ an object

import pickle

p = pickle.Pickler(file)

p.dump(obj)

# file is an open file object

# Dump object

To unserialize, you ’unpickle’ an object

p = pickle.Unpickler(file)

obj = p.load()

# file is an open file

# Load object

**Notes**

Most built-in types can be pickled except for files, sockets, execution frames, etc...

The data-encoding is Python-specific.

Any file-like object that provides write(),read(), and readline() methods can be used as a file.

Recursive objects are correctly handled.

cPickle is like pickle, but written in C and is substantially faster.

pickle can be subclassed, cPickle can not.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 46

July 17, 2000, beazley@cs.uchicago.edu

**The marshal Module**

**The marshal module can also be used for serialization**

To serialize

import marshal

marshal.dump(obj,file)

# Write obj to file

To unserialize

obj = marshal.load(file)

**Notes**

marshal is similiar to pickle, but is intended only for simple objects

Can’t handle recursion or class instances.

On the plus side, it’s pretty fast if you just want to save simple objects to a file.

Data is stored in a binary architecture independent format

O’Reilly OSCON 2000, Advanced Python Programming, Slide 47

July 17, 2000, beazley@cs.uchicago.edu

**The shelve Module**

**The shelve module provides a persistent dictionary**

Idea: works like a dictionary, but data is stored on disk

import shelve

d = shelve.open("data")

d[’foo’] = 42

# Open a ’shelf’

# Save data

# Retrieve data

x = d[’bar’]

Shelf operations

d[key] = obj

obj = d[key]

del d[key]

d.has\_key(key)

d.keys()

# Store an object

# Retrieve an object

# Delete an object

# Test for existence of key

# Return a list of all keys

# Close the shelf

d.close()

**Comments**

Keys must be strings.

Data can be any object serializable with pickle.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 48

July 17, 2000, beazley@cs.uchicago.edu

**DBM-Style Databases**

**Python provides a number of DBM-style database interfaces**

Key-based databases that store arbitrary strings.

Similar to shelve, but can’t store arbitrary objects (strings only)

Examples: dbm, gdbm, bsddb

**Example:**

import dbm

d = dbm.open("database","r")

d["foo"] = "bar"

s = d["spam"]

del d["name"]

d.close()

# Store a value

# Retrieve a value

# Delete a value

# Close the database

**Comments**

The availability of DBM modules depends on optional libraries and may vary.

Don’t use these if you should really be using a relational database (e.g., MySQL).

O’Reilly OSCON 2000, Advanced Python Programming, Slide 49

July 17, 2000, beazley@cs.uchicago.edu

**Operating System Services**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 50

July 17, 2000, beazley@cs.uchicago.edu

**Operating System Services**

**Python provides a wide variety of operating system 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 (especially Windows/Mac).

**Let’s take a tour...**

I’m not going to cover everything.

This is mostly a survey of what Python provides.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 51

July 17, 2000, beazley@cs.uchicago.edu

**Process Environment**

**Environment Variables**

os.environ - A dictionary containing current environment variables

user = os.environ[’USER’]

os.environ[’PATH’] = "/bin:/usr/bin"

**Current directory and umask**

os.chdir(path)

os.getcwd()

os.umask(mask)

# Change current working directory

# Get current working directory

# Change umask setting. Returns previous umask

**User and group identification**

os.getegid()

os.geteuid()

os.getgid()

# Get effective group id

# Get effective user id

# Get group id

# Get user id

os.getuid()

os.setgid(gid)

os.setuid(uid)

# Set group id

# Set user id

O’Reilly OSCON 2000, Advanced Python Programming, Slide 52

July 17, 2000, beazley@cs.uchicago.edu

**Process Creation and Destruction**

**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)

os.execvpe(path,args, env)

os.wait([pid)]

# Execute process, use default path

# Wait for child process

os.waitpid(pid,options)

os.system(command)

os.\_exit(n)

# Wait for change in state of child

# Execute a system command

# Exit immediately with status n.

**Canonical Example**

import os

pid = os.fork()

if pid == 0:

# Create child

#

Child process

os.execvp("ls", ["ls","-l"])

else:

os.wait()

# Wait for child

O’Reilly OSCON 2000, Advanced Python Programming, Slide 53

July 17, 2000, beazley@cs.uchicago.edu

**Pipes**

**os.popen() function**

f = popen("ls -l", "r")

data = f.read()

f.close()

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

**The popen2 module**

Spawns processes and provides hooks to stdin, stdout, and stderr

popen2(cmd) # Run cmd and return (stdout, stdin)

popen3(cmd) # Run cmd and return (stdout, stdin, stderr)

Example

(

i.write(data)

i.close()

result = o.read()

o.close()

o,i) = popen2.popen2("wc")

# Write to child’s input

# Get child’s output

O’Reilly OSCON 2000, Advanced Python Programming, Slide 54

July 17, 2000, beazley@cs.uchicago.edu

**The commands Module**

**The easy way to capture the output of a subprocess**

import commands

data = commands.getoutput("ls -l")

Also includes a quoting function

arg = mkarg(str) # Turns str into a argument suitable

#

for use in the shell (to prevent mischief)

**Comments**

Really this is just a wrapper over the popen2 module.

Only available on Unix (sorry).

O’Reilly OSCON 2000, Advanced Python Programming, Slide 55

July 17, 2000, beazley@cs.uchicago.edu

**Error Handling**

**System-related errors are typically translated into the following**

OSError - General operating system error

IOError - I/O related system error

**Cause of the error is contained in errno attribute of exception**

Can use the errno module for symbolic error names

**Example:**

import os, errno

.

..

try:

os.execlp("foo")

except OSError,e:

if e.errno == errno.ENOENT:

print "Program not found. Sorry"

elif e.errno == errno.ENOEXEC:

print "Program not executable."

else:

#

Some other kind of error

O’Reilly OSCON 2000, Advanced Python Programming, Slide 56

July 17, 2000, beazley@cs.uchicago.edu

**Signal Handling**

**Signals**

Usually correspond to external events and arrive asynchronously.

Example: Expiration of a timer, arrival of input, program fault.

**The signal module**

Provides functions for writing Unix-style signal handlers in Python.

signal.signal(signalnum, handler) # Set a signal handler

signal.alarm(time)

signal.pause()

signal.getsignal(signalnum)

# Schedules a SIGALRM signal

# Go to sleep until signal

# Get signal handler

**Supported signals (platform specific)**

SIGABRT

SIGALRM

SIGBUS

SIGCHLD

SIGCLD

SIGCONT

SIGXCPU

SIGFPE

SIGHUP

SIGILL

SIGINT

SIGIO

SIGIOT

SIGXFSZ

SIGKILL

SIGPIPE

SIGPOLL

SIGPROF

SIGPWR

SIGSEGV

SIGSTOP

SIGTERM

SIGTRAP

SIGTSTP

SIGTTIN

SIGTTOU

SIGURG

SIGUSR1

SIGUSR2

SIGVTALRM

SIGWINCH

SIGQUIT

O’Reilly OSCON 2000, Advanced Python Programming, Slide 57

July 17, 2000, beazley@cs.uchicago.edu

**Signal Handling (Cont)**

**Example: A Periodic Timer**

import signal

interval = 1.0

ticks = 0

def alarm\_handler(signo,frame):

global ticks

print "Alarm ", ticks

ticks = ticks + 1

signal.alarm(interval)

# Schedule a new alarm

signal.signal(signal.SIGALRM, alarm\_handler)

signal.alarm(interval)

#

Spin forever--should see handler being called every second

while 1:

pass

O’Reilly OSCON 2000, Advanced Python Programming, Slide 58

July 17, 2000, beazley@cs.uchicago.edu

**Signal Handling (Cont)**

**Ignoring signals**

signal.signal(signo, signal.SIG\_IGN)

**Default behavior**

signal.signal(signo, signal.SIG\_DFL)

**Comments**

Signal handlers remain installed until explicitly reset.

It is not possible to temporarily disable signals.

Signals are only handled between atomic instructions of the interpreter.

If a signal occurs during an I/O operation, it may fail with an exception (errno == EINTR).

Certain signals can’t be handled from Python (SIGSEGV for instance).

Python handles a number of signals on its own (SIGINT, SIGTERM).

Mixing signals and threads is extremely problematic. Only main thread can deal with signals.

Signal handling on Windows and Macintosh is of limited functionality.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 59

July 17, 2000, beazley@cs.uchicago.edu

**Time**

**The time module**

A variety of time related functions

time.clock()

# Current CPU time in seconds

time.time()

# Current time (GMT) in seconds since epoch

# Convert time to local time (returns a tuple).

# Convert time to GMT (returns a tuple)

# Creates a string representing the time

# Create a string representing local time

# Convert time tuple to seconds

time.localtime(secs)

time.gmtime(secs)

time.asctime(tuple)

time.ctime(secs)

time.mktime(tuple)

time.sleep(secs)

# Go to sleep for awhile

**Example**

import time

t = time.time()

#

Returns (year,month,day,hour,minute,second,weekday,day,dst)

tp = time.localtime(t)

Produces a string like ’Mon Jul 12 14:45:23 1999’

print time.localtime(tp)

#

O’Reilly OSCON 2000, Advanced Python Programming, Slide 60

July 17, 2000, beazley@cs.uchicago.edu

**Getting User and Group Information**

**The pwd module**

Provides access to the Unix password database

pwd.getpwuid(uid)

pwd.getpwname(login)

pwd.getpwall()

# Returns passwd entry for uid

# Returns passwd entry for login

# Get all entries

x = pwd.getpwnam(’beazley’)

#

#

x = (’beazley’,’x’,100,1,’David M. Beazley’, ’/home/beazley’,

’/usr/bin/csh’)

**The grp module**

Provides access to Unix group database

grp.getgrgid(gid)

grp.getgrnam(gname)

grp.getgrall()

# Return group entry for gid

# Return group entry for gname

# Get all entries

O’Reilly OSCON 2000, Advanced Python Programming, Slide 61

July 17, 2000, beazley@cs.uchicago.edu

**Other Miscellaneous Services**

**crypt**

Provides access to the Unix crypt() function.

Used to encrypt passwords

**locale**

Support for the POSIX locale functions.

**resource**

Allows a program to control and monitor its system resources

Can place limits on CPU time, file sizes, etc.

**termios**

Low-level terminal I/O handling.

For all of those vintage TTY fans.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 62

July 17, 2000, beazley@cs.uchicago.edu

**Windows and Macintosh**

**Comment**

Most of Python’s OS interfaces are Unix-centric.

However, much of this functionality is emulated on non-Unix platforms.

With a number of omissions (especially in process and user management).

**The msvcrt module**

Provides access to a number of functions in the Microsoft Visual C++ runtime.

Functions to read and write characters.

Some additional file handling (locking, modes, etc...).

But not a substitute for PythonWin.

**The macfs, macostools, and findertools modules**

Manipulation of files and applications on the Macintosh.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 63

July 17, 2000, beazley@cs.uchicago.edu

**Threads**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 64

July 17, 2000, beazley@cs.uchicago.edu

**Thread Basics**

**Background**

A running program is called a "process"

Each process has memory, list of open files, stack, program counter, etc...

Normally, a process executes statements in a single sequence of control-flow.

**Process creation with fork(),system(), popen(), etc...**

These commands create an entirely new process.

Child process runs independently of the parent.

Has own set of resources.

There is minimal sharing of information between parent and child.

Think about using the Unix shell.

**Threads**

A thread is kind of like a process (it’s a sequence of control-flow).

Except that it exists entirely inside a process and shares resources.

A single process may have multiple threads of execution.

Useful when an application wants to perform many concurrent tasks on shared data.

Think about a browser (loading pages, animations, etc.)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 65

July 17, 2000, beazley@cs.uchicago.edu

**Problems with Threads**

**Scheduling**

To execute a threaded program, must rapidly switch between threads.

This can be done by the user process (user-level threads).

Can be done by the kernel (kernel-level threads).

**Resource Sharing**

Since threads share memory and other resources, must be very careful.

Operation performed in one thread could cause problems in another.

**Synchronization**

Threads often need to coordinate actions.

Can get "race conditions" (outcome dependent on order of thread execution)

Often need to use locking primitives (mutual exclusion locks, semaphores, etc...)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 66

July 17, 2000, beazley@cs.uchicago.edu

**Python Threads**

**Python supports threads on the following platforms**

Solaris

Windows

Systems that support the POSIX threads library (pthreads)

**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.

Long-running calculations in C/C++ can block execution of all other threads.

However, 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.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 67

July 17, 2000, beazley@cs.uchicago.edu

**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.

import thread

import time

def print\_time(delay):

while 1:

time.sleep(delay)

print time.ctime(time.time())

#

Start the thread

thread.start\_new\_thread(print\_time,(5,))

Go do something else

statements

..

#

.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 68

July 17, 2000, beazley@cs.uchicago.edu

**The thread module (cont)**

**Thread termination**

Thread silently exits when the function returns.

Thread can explicitly exit by calling thread.exit() or sys.exit().

Uncaught exception causes thread termination (and prints error message).

However, 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 foo():

lk.acquire()

# Acquire the lock

critical section

lk.release()

# Release the lock

Only one thread can acquire the lock at once.

Threads block indefinitely until lock becomes available.

You might use this if two or more threads were allowed to update a shared data structure.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 69

July 17, 2000, beazley@cs.uchicago.edu

**The thread module (cont)**

**The main thread**

When Python starts, it runs as a single thread of execution.

This is called the "main thread."

On its own, it’s no big deal.

However, if you launch other threads it has some special properties.

**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).

Caveat: The keyboard-interrupt can be caught by any thread (non-deterministically).

O’Reilly OSCON 2000, Advanced Python Programming, Slide 70

July 17, 2000, beazley@cs.uchicago.edu

**The threading module**

**The threading module is a high-level threads module**

Implements threads as classes (similar to Java)

Provides an assortment of synchronization and locking primitives.

Built using the low-level thread module.

**Creating a new thread (as a class)**

Idea: Inherit from the "Thread" class and provide a few methods

import threading, time

class PrintTime(threading.Thread):

def \_\_init\_\_(self,interval):

threading.Thread.\_\_init\_\_(self)

self.interval = interval

# Required

def run(self):

while 1:

time.sleep(self.interval)

print time.ctime(time.time())

t = PrintTime(5)

t.start()

...

# Create a thread object

# Start it

O’Reilly OSCON 2000, Advanced Python Programming, Slide 71

July 17, 2000, beazley@cs.uchicago.edu

**The threading module (cont)**

**The Thread class**

When defining threads as classes all you need to supply is the following:

A constructor that calls threading.Thread.\_\_init\_\_(self)

A run() method that performs the actual work of the thread.

A few additional methods are also available

t.join([timeout])

t.getName()

t.setName(name)

t.isAlive()

t.isDaemon()

t.setDaemon(val)

# Wait for thread t to terminate

# Get the name of the thread

# Set the name of the thread

# Return 1 if thread is alive.

# Return daemonic flag

# Set daemonic flag

**Daemon threads**

Normally, interpreter exits only when all threads have terminated.

However, a thread can be flagged as a daemon thread (runs in background).

Interpreter really only exits when all non-daemonic threads exit.

Can use this to launch threads that run forever, but which can be safely killed.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 72

July 17, 2000, beazley@cs.uchicago.edu

**The threading module (cont)**

**The threading module provides the following synchronization primitives**

Mutual exclusion locks

Reentrant locks

Conditional variables

Semaphores

Events

**Why would you need these?**

Threads are updating shared data structures

Threads need to coordinate their actions in some manner (events).

You need to regain some programming sanity.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 73

July 17, 2000, beazley@cs.uchicago.edu

**Lock Objects**

**The Lock object**

Provides a simple mutual exclusion lock

import threading

data = [ ]

# Some data

# Create a lock

lck = threading.Lock()

def put\_obj(obj):

lck.acquire()

data.append(obj)

lck.release()

def get\_obj():

lck.acquire()

r = data.pop()

lck.release()

return r

Only one thread is allowed to acquire the lock at once

Most useful for coordinating access to shared data.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 74

July 17, 2000, beazley@cs.uchicago.edu

**RLock Objects**

**The RLock object**

A mutual-exclusion lock that allows repeated acquisition by the *same* thread

Allows nested acquire(), release() operations in the thread that owns the lock.

Only the outermost release() operation actually releases the lock.

import threading

data = [ ]

# Some data

# Create a lock

lck = threading.Lock()

def put\_obj(obj):

lck.acquire()

data.append(obj)

.

..

put\_obj(otherobj)

# Some kind of recursion

...

lck.release()

def get\_obj():

lck.acquire()

r = data.pop()

lck.release()

return r

O’Reilly OSCON 2000, Advanced Python Programming, Slide 75

July 17, 2000, beazley@cs.uchicago.edu

**Condition Variables**

**The Condition object**

Creates a condition variable.

Synchronization primitive typically used when a thread is interested in an event or state change.

Classic problem: producer-consumer problem.

#

Create data queue and a condition variable

data = []

cv = threading.Condition()

#

Consumer thread

def consume\_item():

cv.acquire()

# Acquire the lock

while not len(data):

cv.wait()

r = data.pop()

cv.release()

# Wait for data to show up

# Release the lock

return r

Producer thread

#

def produce\_item(obj):

cv.acquire()

# Acquire the lock

data.append(obj)

cv.notify()

cv.release()

# Notify a consumer

# Release the lock

O’Reilly OSCON 2000, Advanced Python Programming, Slide 76

July 17, 2000, beazley@cs.uchicago.edu

**Semaphore Objects**

**Semaphores**

A locking primitive based on a counter.

Each acquire() method decrements the counter.

Each release() method increments the counter.

If the counter reaches zero, future acquire() methods block.

Common use: limiting the number of threads allowed to execute code

sem = threading.Semaphore(5)

def fetch\_file(host,filename):

sem.acquire()

# No more than 5 threads allowed

# Decrements count or blocks if zero

.

..

blah

..

sem.release()

.

# Increment count

O’Reilly OSCON 2000, Advanced Python Programming, Slide 77

July 17, 2000, beazley@cs.uchicago.edu

**Event Objects**

**Events**

A communication primitive for coordinating threads.

One thread signals an "event"

Other threads wait for it to happen.

#

Create an event object

e = Event()

#

Signal the event

def signal\_event():

e.set()

#

def wait\_for\_event():

e.wait()

Wait for event

#

Clear event

def clear\_event():

e.clear()

Similar to a condition variable, but all threads waiting for event are awakened.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 78

July 17, 2000, beazley@cs.uchicago.edu

**Locks and Blocking**

**By default, all locking primitives block until lock is acquired**

In general, this is uninterruptible.

**Fortunately, most primitives provide a non-blocking option**

if not lck.acquire(0):

#

lock couldn’t be acquired!

This works for Lock, RLock, and Semaphore objects

**Timeouts**

Condition variables and events provide a timeout option

cv = Condition()

.

..

cv.wait(60.0)

# Wait 60 seconds for notification

On timeout, the function simply returns. Up to caller to detect errors.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 79

July 17, 2000, beazley@cs.uchicago.edu

**The Queue Module**

**Provides a multi-producer, multi-consumer FIFO queue object**

Can be used to safely exchange data between multiple threads

q = Queue(maxsize)

q.qsize()

# Create a queue

# Return current size

# Test if empty

q.empty()

q.full()

# Test if full

q.put(item)

q.get()

# Put an item on the queue

# Get item from queue

**Notes:**

The Queue object also supports non-blocking put/get.

q.put\_nowait(item)

q.get\_nowait()

These raise the Queue.Full or Queue.Empty exceptions if an error occurs.

Return values for qsize(), empty(), and full() are approximate.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 80

July 17, 2000, beazley@cs.uchicago.edu

**Final Comments on Threads**

**Python threads are quite functional**

Can write applications that use dozens (or even hundreds) of threads

**But there are performance issues**

Global interpreter lock makes it difficult to fully utilize multiple CPUs.

You don’t get the degree of parallelism you might expect.

**Interaction with C extensions**

Common problem: I wrote a big C extension and it broke threading.

The culprit: Not releasing global lock before starting a long-running function.

**Not all modules are thread-friendly**

Example: gethostbyname() blocks all threads if nameserver down.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 81

July 17, 2000, beazley@cs.uchicago.edu

**Network Programming**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 82

July 17, 2000, beazley@cs.uchicago.edu

**Network Overview**

**Python provides a wide assortment of network support**

Low-level programming with sockets (if you want to create a protocol).

Support for existing network protocols (HTTP, FTP, SMTP, etc...)

Web programming (CGI scripting and HTTP servers)

Data encoding

**I can only cover some of this**

Programming with sockets

HTTP and Web related modules.

A few data encoding modules

**Recommended Reference**

*Unix Network Programming* by W. Richard Stevens.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 83

July 17, 2000, beazley@cs.uchicago.edu

**Network Basics: TCP/IP**

**Python’s networking modules primarily support TCP/IP**

TCP - A reliable connection-oriented protocol (streams).

UDP - An unreliable packet-oriented protocol (datagrams).

Of these, TCP is the most common (HTTP, FTP, SMTP, etc...).

**Both protocols are supported using "sockets"**

A socket is a file-like object.

Allows data to be sent and received across the network like a file.

But it also includes functions to accept and establish connections.

Before two machines can establish a connection, both must create a socket object.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 84

July 17, 2000, beazley@cs.uchicago.edu

**Network Basics: Ports**

**Ports**

In order to receive a connection, a socket must be bound to a port (by the server).

A port is a number in the range 0-65535 that’s managed by the OS.

Used to identify a particular network service (or listener).

Ports 0-1023 are reserved by the system and used for common protocols

FTP

Port 20

Port 23

Port 25

Port 80

Telnet

SMTP (Mail)

HTTP (WWW)

Ports above 1024 are reserved for user processes.

**Socket programming in a nutshell**

Server creates a socket, binds it to some well-known port number, and starts listening.

Client creates a socket and tries to connect it to the server (through the above port).

Server-client exchange some data.

Close the connection (of course the server continues to listen for more clients).

O’Reilly OSCON 2000, Advanced Python Programming, Slide 85

July 17, 2000, beazley@cs.uchicago.edu

**Socket Programming Example**

**The socket module**

Provides access to low-level network programming functions.

Example: A server that returns the current time

#

Time server program

from socket import \*

import time

s = socket(AF\_INET, SOCK\_STREAM)

s.bind(("",8888))

s.listen(5)

# Create TCP socket

# Bind to port 8888

# Start listening

while 1:

client,addr = s.accept()

# Wait for a connection

print "Got a connection from ", addr

client.send(time.ctime(time.time())) # Send time back

client.close()

**Notes:**

Socket first opened by server is not the same one used to exchange data.

Instead, the accept() function returns a new socket for this (’client’ above).

listen() specifies max number of pending connections.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 86

July 17, 2000, beazley@cs.uchicago.edu

**Socket Programming Example (cont)**

**Client Program**

Connect to time server and get current time

#

Time client program

from socket import \*

s = socket(AF\_INET,SOCK\_STREAM)

# Create TCP socket

s.connect(("makemepoor.com",8888)) # Connect to server

tm = s.recv(1024)

s.close()

print "The time is", tm

# Receive up to 1024 bytes

# Close connection

**Key Points**

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

Aside from connection process, it’s relatively straightforward.

Of course, the devil is in the details.

And are there ever a LOT of details.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 87

July 17, 2000, beazley@cs.uchicago.edu

**The socket Module**

**This is used for all low-level networking**

Creation and manipulation of sockets

General purpose network functions (hostnames, data conversion, etc...)

A direct translation of the BSD socket interface.

**Utility Functions**

socket.gethostbyname(hostname) # Get IP address for a host

socket.gethostname()

socket.ntohl(x)

socket.ntohs(x)

socket.htonl(x)

socket.htons(x)

# Name of local machine

# Convert 32-bit integer to host order

# Convert 16-bit integer to host order

# Convert 32-bit integer to network order

# Convert 16-bit integer to network order

**Comments**

Network order for integers is big-endian.

Host order may be little-endian or big-endian (depends on the machine).

O’Reilly OSCON 2000, Advanced Python Programming, Slide 88

July 17, 2000, beazley@cs.uchicago.edu

**The socket Module (cont)**

**The socket(family, type, proto) function**

Creates a new socket object.

family is usually set to AF\_INET

type is one of:

SOCK\_STREAM

SOCK\_DGRAM

SOCK\_RAW

Stream socket (TCP)

Datagram socket (UDP)

Raw socket

proto is usually only used with raw sockets

IPPROTO\_ICMP

IPPROTO\_IP

IPPROTO\_RAW

IPPROTO\_TCP

IPPROTO\_UDP

**Comments**

Currently no support for IPv6 (although its on the way).

Raw sockets only available to processes running as root.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 89

July 17, 2000, beazley@cs.uchicago.edu

**The socket Module (cont)**

**socket methods**

s.accept()

s.bind(address)

s.close()

# Accept a new connection

# Bind to an address and port

# Close the socket

s.connect(address)

s.fileno()

# Connect to remote socket

# Return integer file descriptor

# Get name of remote machine

# Get socket address as (ipaddr,port)

# Get socket options

# Start listening for connections

# Turn socket into a file object

# Receive data

s.getpeername()

s.getsockname()

s.getsockopt(...)

s.listen(backlog)

s.makefile(mode)

s.recv(bufsize)

s.recvfrom(bufsize)

s.send(string)

# Receive data (UDP)

# Send data

s.sendto(string, address) # Send packet (UDP)

s.setblocking(flag)

s.setsockopt(...)

s.shutdown(how)

# Set blocking or nonblocking mode

# Set socket options

# Shutdown one or both halves of connection

**Comments**

There are a huge variety of configuration/connection options.

You’ll definitely want a good reference at your side.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 90

July 17, 2000, beazley@cs.uchicago.edu

**The SocketServer Module**

**Provides a high-level class-based interface to sockets**

Encapsulates each protocol in a class (TCPServer, UDPServer, etc.)

Provides a series of handler classes that specify additional server behavior.

To create a network service, need to inherit from both a protocol and handler class.

**Example**

#

Simple time server

import SocketServer

import time

#

class TimeHandler(SocketServer.BaseRequestHandler):

def handle(self):

This class actually implements the server functionality

self.request.send(time.ctime(time.time()))

#

server = SocketServer.TCPServer(("",8888),TimeHandler)

server.serve\_forever()

Create the server

**Comments**

The module provides a number of specialized server and handler types.

Ex: ForkingTCPServer, ThreadingTCPServer, StreamRequestHandler, etc.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 91

July 17, 2000, beazley@cs.uchicago.edu

**Common Network Protocols**

**Modules are available for a variety of network protocols**

ftplib - FTP protocol

smtplib - SMTP (mail) protocol

nntplib - News

gopherlib - Gopher

poplib - POP3 mail server

imaplib - IMAP4 mail server

telnetlib - Telnet protocol

httplib - HTTP protocol

**Comments**

These modules are built using sockets, but operate on a very low-level.

Require a good understand of the underlying protocol.

But can be quite powerful if you know exactly what you are doing.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 92

July 17, 2000, beazley@cs.uchicago.edu

**The httplib Module**

**Implements the HTTP 1.0 protocol**

Can use to talk to a web server.

**HTTP in two bullets**

Client (e.g., a browser) sends a request to the server

GET /index.html HTTP/1.0

Connection: Keep-Alive

Host: www.python.org

User-Agent: Mozilla/4.61 [en] (X11; U; SunOS 5.6 sun4u)

[blank line]

Server responds with something like this:

HTTP/1.0 200 OK

Content-type: text/html

Content-length: 72883

Headers: blah

[

blank line]

Data

..

.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 93

July 17, 2000, beazley@cs.uchicago.edu

**The httplib Module (cont)**

**Making an HTTP connection**

import httplib

h = httplib.HTTP("www.python.org")

h.putrequest(’GET’,’/index.html’)

h.putheader(’User-Agent’,’Lame Tutorial Code’)

h.putheader(’Accept’,’text/html’)

h.endheaders()

errcode,errmsg, headers = h.getreply()

f = h.getfile() # Get file object for reading data

data = f.read()

f.close()

**Comments**

Some understanding of HTTP is required.

Only HTTP/1.0 is currently supported.

Most of the other protocol modules work in a similar manner.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 94

July 17, 2000, beazley@cs.uchicago.edu

**The urllib Module**

**A high-level interface to HTTP and FTP**

Provides a file-like object that can be used to connect to remote servers

import urllib

f = urllib.urlopen("http://www.python.org/index.html")

data = f.read()

f.close()

**Utility functions**

urllib.quote(str)

urllib.quote\_plus(str)

urllib.unquote(str)

urllib.unquote\_plus(str) # Opposite of quote\_plus()

urllib.urlencode(dict)

# Quotes a string for use in a URL

# Also replaces spaces with ’+’

# Opposite of quote()

# Turns a dictionary of key=value

# pairs into a HTTP query-string

# Produces "beazley%40cs"

Examples

urllib.quote("beazley@cs")

urllib.unquote("%23%21/bin/sh") # Produces "/bin/sh"

O’Reilly OSCON 2000, Advanced Python Programming, Slide 95

July 17, 2000, beazley@cs.uchicago.edu

**The urlparse Module**

**Functions for manipulating URLs**

URL’s have the following general format

scheme:/netloc/path;parameters?query#fragment

urlparse(urlstring) - Parses a URL into components

import urlparse

t = urlparse.urlparse("http://www.python.org/index.html")

#

Produces (’http’,’www.python.org’,’/index.html’,’’,’’,’’)

urlunparse(tuple) - Turns tuple of components back into a URL string

url = urlparse.urlunparse((’http’,’www.python.org’,’foo.html’,

’bar=spam’,’’))

Produces "http://www.python.org/foo.html?bar=spam"

#

urljoin(base, url) - Combines a base and relative URL

urlparse.urljoin("http://www.python.org/index.html","help.html")

#

Produces "http://www.python.org/help.html"

O’Reilly OSCON 2000, Advanced Python Programming, Slide 96

July 17, 2000, beazley@cs.uchicago.edu

**CGI Scripting**

**CGI Overview**

Common protocol web servers use to run external programs in response to HTTP requests.

Typical uses: forms processing, dynamic content generation

**How it works**

You write some sort of form in your HTML document

<

form method="GET" action="cgi-bin/spam.cgi">

Your name: <input type="text" name="name" size=30><p>

Your email: <input type="text" name="email" size=40><p>

<input type="submit" value="Submit"></form>

This gets translated into request with parameters

GET /cgi-bin/spam.cgi?name=Dave+Beazley&email=beazley%40cs HTTP/1.0

Web-server (e.g., Apache) launches CGI program and passes parameters

That program writes to stdout to produce the web-page.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 97

July 17, 2000, beazley@cs.uchicago.edu

**CGI Scripting**

**CGI Example**

#

print "Content-type: text/html\n"

print "<h1>Hello World</h1>

!/usr/local/bin/python

**Problem**

To do anything useful, have to receive and decode "query string" from server

Which is tedious

**The cgi module**

Provides a variety of functions for writing CGI programs.

Reading data.

Decoding query strings

Getting header information.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 98

July 17, 2000, beazley@cs.uchicago.edu

**cgi Module Example**

**Example of using CGI module**

#

!/usr/local/bin/python

import cgi

form = cgi.FieldStorage()

name = form["name"].value

email = form["email"].value

# Read query string

# Get ’name’ field from form

# Get ’email’ field from form

print "Content-type: text/html"

print

print "<H1>Hello %s. Your email is %s</h1>" % (name,email)

**Comments**

There is much more to this module than presented here.

Plus a number of security implications.

However, there are now better ways to do this sort of thing (PHP3, Zope, etc...)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 99

July 17, 2000, beazley@cs.uchicago.edu

**Miscellaneous Network Topics**

**Modules not discussed**

select - Access to the select() system call. Useful for polling.

asyncore - A framework for writing highly threaded servers based on asynchronous I/O.

BaseHTTPServer, SimpleHTTPServer, CGIHTTPServer - Framework for building web-servers.

**A few related extensions**

Fnorb - A CORBA implementation for Python.

ILU - Distributed Objects.

**A small plug**

Python is a great language for experimenting with network programming.

Can experiment interactively at the prompt.

Programs are relatively simple.

Compare to low-level network programming in C.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 100

July 17, 2000, beazley@cs.uchicago.edu

**Data Encoding**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 101

July 17, 2000, beazley@cs.uchicago.edu

**Data Encoding**

**Problem**

Data is managed in a variety of formats.

Especially in Internet applications and network protocols

**Examples**

Base64 encoding

Quoted-printable encoding

Uuencoding

MIME

HTML

XML

Binhex

Binary data structures

XDR

**Fortunately, Python has a variety of data processing modules**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 102

July 17, 2000, beazley@cs.uchicago.edu

**Base64 Encoding**

**The base64 module**

Encodes and decodes base64 encoded text

Commonly used to embed binary data in mail attachments

**Encoding**

import base64

base64.encode(inputfile,outputfile)

es = base64.encodestring(s)

# Files

# Strings

**Decoding**

import base64

base64.decode(inputfile, outputfile)

s = base64.decodestring(es)

# Files

# String

O’Reilly OSCON 2000, Advanced Python Programming, Slide 103

July 17, 2000, beazley@cs.uchicago.edu

**Uuencoding**

**The uu module**

Encodes and decodes uuencoded text

Same idea as before

**Encoding**

import uu

uu.encode(inputfile,outputfile)

**Decoding**

import uu

uu.decode(inputfile,outputfile)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 104

July 17, 2000, beazley@cs.uchicago.edu

**Quoted-Printed Encoding**

**The quopri module**

Encodes and decodes text in "quoted-printable" format

Commonly used to encode text-documents in mail messages

Yep, same general idea

**Encoding**

import quopri

quopri.encode(inputfile,outputfile)

**Decoding**

import quopri

quopri.decode(inputfile,outputfile)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 105

July 17, 2000, beazley@cs.uchicago.edu

**Binhex4 Encoding**

**The binhex module**

Encodes and decodes text in binhex format.

Commonly used to encode binary files on the Macintosh.

**Encoding**

import binhex

binhex.binhex(inputfile,outputfile)

**Decoding**

import binhex

binhex.hexbin(inputfile,outputfile)

**Note**

Macintosh resource fork ignored on non-mac systems.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 106

July 17, 2000, beazley@cs.uchicago.edu

**RFC822 Headers**

**The rfc822 module**

Used to parse RFC-822 encoded headers.

Used in e-mail and HTTP protocol.

Return-Path:

Date: Sat, 17 Jul 1999 10:18:21 -500 (CDT)

Reply-To: beazley@cs.uchicago.edu

Context-Type: text/plain; charset=US-ASCII

From: David Beazley

To: guido@cnri.reston.va.us

Subject: IPC8

Blah...

**General idea:**

Headers are parsed into a special Message object.

Can query for individual fields

O’Reilly OSCON 2000, Advanced Python Programming, Slide 107

July 17, 2000, beazley@cs.uchicago.edu

**RFC822 (cont)**

**rfc822 Example**

import rfc822

f = open("mailmessage")

m = rfc822.Message(f)

#

Extract some fields

m\_from = m["From"]

m\_to = m.getaddr("To")

m\_subject = m["Subject"]

**Selected Message operations**

m[name]

m[name] = value

m.keys()

m.values()

m.items()

m.has\_key(name)

m.getallmatchingheaders(name)

m.getaddr(name)

m.getaddrlist(name) # Get a list of email addresses

m.getdate(name)

# Return data for header name

# Add a header

# Return a list of all header names

# Return list of header values

# Return list of (header,value) pairs

# Test for existence

# Return list of all matching headers

# Return (full\_name, email) for an address field

# Get a date as a time tuple

O’Reilly OSCON 2000, Advanced Python Programming, Slide 108

July 17, 2000, beazley@cs.uchicago.edu

**Binary Data Encoding**

**The struct module**

Allows binary structures to be packed into a string

Useful if you need to interact with a binary network protocol

Or if you need to create a binary data structure for a C program.

**Packing data with pack(fmt, v1, v2, ...)**

Packs the values v1, v2, and so on according to a format string

Format codes and corresponding C datatypes

’x’

’c’

’b’

’B’

’h’

’H’

’i’

Pad byte

char

signed char

unsigned char

short

unsigned short

int

’I’ unsigned int

’l’ long

’L’ unsigned long

’f’ float

’d’ double

’s’ char[]

’P’ void \*

Example

s = struct.pack("hhii", 34, 73, 162773, 2222)

s = struct.pack("is", len(t), t)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 109

July 17, 2000, beazley@cs.uchicago.edu

**Binary Data Encoding (cont)**

**Unpacking data with unpack(fmt, string)**

Same idea in reverse.

Returns a tuple of unpacked values

t = struct.unpack("hhii",s)

a,b,c,d = struct.unpack("hhii",s)

**Data alignment and bit ordering**

First character of format string can specify encoding rules

’@’

’=’

’<’

’>’

’!’

Native byte order

Native byte order

Little endian

Big endian

Native size and alignment

Standard size and alignment

Standard size and alignment

Standard size and alignment

Standard size and alignment

Network order

Native alignment uses the size and alignment rules of the C compiler.

Standard alignment uses no padding and assumes the following sizes

short

long

double

2 bytes

4 bytes

64 bits

int

float

4 bytes

32 bits

O’Reilly OSCON 2000, Advanced Python Programming, Slide 110

July 17, 2000, beazley@cs.uchicago.edu

**Other Encoding Modules**

**xdrlib**

Encodes strings to and from Sun XDR format.

Commonly used in Remote Procedure Call (RPC)

**MIME**

The MimeWriter, multifile, mimetypes, and mimetools modules

Decoding and encoding of MIME encoded mail messages.

Basically RFC822 plus some additional encoding rules.

**htmllib**

Parsing of HTML documents

**sgmllib and xmllib**

Parsing of SGML and XML documents

Caveat: deprecated. Consult the XML-sig for more up to date work.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 111

July 17, 2000, beazley@cs.uchicago.edu

**Restricted Execution**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 112

July 17, 2000, beazley@cs.uchicago.edu

**Restricted Execution**

**Problem**

Sometimes want to run code in a restricted environment

CGI scripts

Agents

Applets

**Python solution**

rexec module - Restricted code execution

Bastion - Restricted access to objects

O’Reilly OSCON 2000, Advanced Python Programming, Slide 113

July 17, 2000, beazley@cs.uchicago.edu

**The rexec Module**

**Provides a restricted environment for code execution**

Defines a class RExec that provides a controlled execution environment

Class attributes:

RExec.nok\_builtin\_names

RExec.ok\_builtin\_modules

RExec.ok\_path

RExec.ok\_posix\_names

RExec.ok\_sys\_names

# List of prohibited built-in functions

# List of modules that can be imported

# List of directories searched on import

# List of accepted functions in os module

# List of members in sys module

Methods on an instance of RExec

r.r\_eval(code)

# Evaluate code in restricted mode

# Execute code in restricted mode

# Execute file in restricted more

r.r\_exec(code)

r.r\_execfile(filename)

A few methods which may be redefined

r.r\_import(modulename)

r.r\_open(filename,mode)

# Called whenever code imports

# Called whenever code opens a file

O’Reilly OSCON 2000, Advanced Python Programming, Slide 114

July 17, 2000, beazley@cs.uchicago.edu

**The rexec Module (cont)**

**Example**

#

Create a little restricted environment

import rexec

class AppletExec(rexec.RExec):

ok\_builtin\_modules = [’string’,’math’,’time’]

ok\_posix\_names = []

def r\_open(\*args):

#

Check filename for special cases

..

.

raise SystemError, "Go away"

r = AppletExec()

r.r\_exec(appletcode)

**Additional comments regarding restricted mode**

The interpreter runs in restricted mode if the identity of \_\_builtins\_\_ has been changed.

Restricted programs can’t access the \_\_dict\_\_ attribute of classes and instances.

Similar restrictions are placed on other objects to prevent a code from becoming priviledged.

O’Reilly OSCON 2000, Advanced Python Programming, Slide 115

July 17, 2000, beazley@cs.uchicago.edu

**The Bastion Module**

**Problem**

Sometimes a restricted program needs to access an object created in unrestricted mode

**Solution**

A Bastion

Basically just a "wrapper" that’s placed around the object.

Intercepts all attribute access with a filter function and either allows or prohibits access.

**Example**

import Bastion, StringIO

s = StringIO("")

# Create a file like object

sbast = Bastion.Bastion(s,lambda x: x in [’read’,’readline’])

sbast.readline()

sbast.write("Blah")

# Okay

# Fails. Attribute error.

**Note**

Can’t place Bastions around built-in types like files and sockets

O’Reilly OSCON 2000, Advanced Python Programming, Slide 116

July 17, 2000, beazley@cs.uchicago.edu

**C Extensions**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 117

July 17, 2000, beazley@cs.uchicago.edu

**The Final Frontier**

**Python has a lot of stuff, but sometimes you need more**

Access to special purpose libraries and applications

You have a favorite system call.

You need serious performance

**Extension Building**

Python interpreter can be extended with functions written C

This is how many of the built-in modules work.

**General Idea**

You write a C extension (using special Python API)

Compile the extension into dynamic link library (DLL)

Dynamically load the extension using ’import’

O’Reilly OSCON 2000, Advanced Python Programming, Slide 118

July 17, 2000, beazley@cs.uchicago.edu

**Example**

**Suppose you wanted to add the following C function**

/

\* Compute the greatest common divisor \*/

int gcd(int x, int y) {

int g;

g = y;

while (x > 0) {

g = x;

y = y % x;

y = g;

}

return g;

}

O’Reilly OSCON 2000, Advanced Python Programming, Slide 119

July 17, 2000, beazley@cs.uchicago.edu

**Example (cont)**

**First step: write Python "wrapper"**

#include "Python.h"

extern int gcd(int, int);

/

\* Wrapper for gcd \*/

static PyObject \*

py\_gcd(PyObject \*self, PyObject \*args) {

int x,y,g;

/

\* Get arguments \*/

if (!PyArg\_ParseTuple(args,"ii",&x,&y)) {

return NULL;

}

/

\* Call the C function \*/

g = gcd(x,y);

\* Return result \*/

return Py\_BuildValue("i",g);

/

}

O’Reilly OSCON 2000, Advanced Python Programming, Slide 120

July 17, 2000, beazley@cs.uchicago.edu

**Example (cont)**

**Step two: package into a module**

/

\* Module ’spam’

#

include "Python.h"

extern int gcd(int, int);

/

\* Wrapper for gcd \*/

static PyObject \*

py\_gcd(PyObject \*self, PyObject \*args) {

.. blah ...

.

}

/

\* Method table \*/

static PyMethodDef spammethods[] = {

{"gcd", py\_gcd, METH\_VARARGS},

NULL, NULL}

{

};

/

\* Module initialization \*/

void initspam() {

Py\_InitModule("spam",spammethods);

}

O’Reilly OSCON 2000, Advanced Python Programming, Slide 121

July 17, 2000, beazley@cs.uchicago.edu

**Example (cont)**

**Step three: Compile into a module**

Create a file called "Setup" like this

\*

shared\*

spam gcd.c spammodule.c

Copy the file Makefile.pre.in from the Python directory.

%

cp /usr/local/lib/python1.5/config/Makefile.pre.in .

Type the following

%

%

make -f Makefile.pre.in boot

make

This will (hopefully) create a shared object file with the module

O’Reilly OSCON 2000, Advanced Python Programming, Slide 122

July 17, 2000, beazley@cs.uchicago.edu

**Example (cont)**

**Step four: Use your module**

linux % **python**

Python 1.5.2 (#1, Jul 11, 1999 13:56:44) [C] on linux

Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

>

>

7

>

1

>

>> **import spam**

>> **spam.gcd(63,56)**

>> **spam.gcd(71,89)**

>>

**It’s almost too easy...**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 123

July 17, 2000, beazley@cs.uchicago.edu

**Extension Building**

**Comments**

Extension building is a complex topic.

Differences between platforms extremely problematic.

Large C libraries can be a challenge.

Complex C++ libraries can be an even greater challenge.

I have only given a small taste (it’s an entirely different tutorial)

**Resources**

Extension building API documentation (www.python.org)

The CXX extension (cxx.sourceforge.net)

SWIG (swig.sourceforge.net)

O’Reilly OSCON 2000, Advanced Python Programming, Slide 124

July 17, 2000, beazley@cs.uchicago.edu

**Conclusions**

O’Reilly OSCON 2000, Advanced Python Programming, Slide 125

July 17, 2000, beazley@cs.uchicago.edu

**Final Comments**

**This has been a whirlwind tour**

Everything covered is part of the standard Python distribution.

However, there are well over 150 standard modules in the standard library.

And we only looked at a small subset.

**Experiment!**

Python is a great language for experimentation.

Fire up the interpreter and start typing commands.

This is a great way to learn about the various modules

**For more information:**

Python Essential Reference (shameless plug)

Online documentation (www.python.org)

**Acknowledgments**

Guido van Rossum

David Ascher

Paul Dubois

O’Reilly OSCON 2000, Advanced Python Programming, Slide 126

July 17, 2000, beazley@cs.uchicago.edu