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

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.

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.

A Very Brief Tour of Python

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
```

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"
```

Variables and Expressions

Expressions

- Standard mathematical operators work like other languages:

```
3 + 5
3 + (5*4)
3 ** 2
'Hello' + 'World'
```

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.

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.

Conditionals

elif statement

```
if a == '+':
    op = PLUS
elif a == '-':
    op = MINUS
elif a == '*':
    op = MULTIPLY
else:
    op = UNKNOWN
```

- Note: There is no `switch` statement.

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"
```

Basic Types (Numbers and Strings)

Numbers

```
a = 3              # Integer
b = 4.5            # Floating point
c = 517288833333L  # Long integer (arbitrary precision)
d = 4 + 3j         # Complex (imaginary) number
```

Strings

```
a = ’Hello’                  # Single quotes
b = "World"                  # 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"""
```

Basic Types (Lists)

Lists of Arbitrary Objects

```
a = [2, 3, 4]                   # A list of integers
b = [2, 7, 3.5, "Hello"]        # A mixed list
c = []                          # An empty list
d = [2, [a,b]]                  # A list containing a list
e = a + b                       # Join two lists
```

List Manipulation

```
x = a[1]                        # Get 2nd element (0 is first)
y = b[1:3]                      # Return a sublist
z = d[1][0][2]                  # Nested lists
b[0] = 42                       # Change an element
```

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]                        # Element access. x = 3
y = f[1:3]                      # Slices. y = (3,4)
z = h[1][1]                     # Nesting. z = 4
```

Comments

- Tuples are like lists, but size is fixed at time of creation.
- Can’t replace members (said to be "immutable")

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"          # Set an element
if c.has_key("directory"):      # Check for presence of an member
    d = c[’directory’]
else:
    d = None

d = c.get("directory",None)     # Same thing, more compact
```

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

# Print characters one at a time
for c in "Hello World":
    print c

# Loop over a range of numbers
for i in range(0,100):
    print i
```

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
```

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()
```

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
>>>
```

Files

The open() function

```
f = open("foo","w")       # Open a file for writing
g = open("bar","r")       # Open a 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 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))
```

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.

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)
```

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.

String Processing

The string module

Various string processing functions

```
string.atof(s)             # Convert to float
string.atoi(s)             # Convert to integer
string.atol(s)             # Convert to long
string.count(s,pattern)    # Count occurrences of pattern in s
string.find(s,pattern)     # Find pattern in s
string.split(s, sep)       # 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"
```

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
\d*             # Match any number decimal digits
[a-zA-Z]+       # Match a sequence of one or more letters
```

The re module

- Provides regular expression pattern matching and replacement.
- Details follow.

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
{m,n}	Match m to n repetitions
{m,n}?	Match m to n repetitions, few as possible
[...]	Match a set of characters
[^...]	Match characters not in set
A | B	Match A or B
(...)	Match regex in parenthesis as a group

Regular Expressions

Special characters

```
\number     Matches text matched by previous group
\A          Matches start of string
\b          Matches empty string at beginning or end of word
\B          Matches empty string not at begin or end of word
\d          Matches any decimal digit
\D          Matches any non-digit
\s          Matches any whitespace
\S          Matches any non-whitespace
\w          Matches any alphanumeric character
\W          Matches characters not in \w
\Z          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
```

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*)'    # My pattern
r = re.compile(pat)      # Compile it
m = r.match(s)           # See if string s matches
if m:
    # Yep, it matched
    ...
else:
    # Nope.
    ...
```

The re Module (cont)

Regular Expression Objects

- Objects created by re.compile() have these methods

```
r.search(s [,pos [,endpos]])   # Search for a match
r.match(s [,pos [,endpos]])    # Check string for match
r.split(s)                     # Split on a regex match
r.findall(s)                   # Find all matches
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.

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)    # Return text matched for group n
m.start(n)    # Return starting index for group n
m.end(n)      # Return ending index for group n
```

The re Module (cont)

Matching Example

```
import re

r = re.compile(r’(\d+)\.(\d*)’)
m = r.match("42.37")
a = m.group(0)        # Returns ’42.37’
b = m.group(1)        # Returns ’42’
c = m.group(2)        # Returns ’37’
print m.start(2)      # 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

Working with Files

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
"w"      Open for writing (truncating to zero length)
"a"      Open for append
"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.

File Objects

File Methods

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

```
f.read([n])              # Read at most n bytes
f.readline([n])          # Read a line of input with max length of n
f.readlines()            # Read all input and return a list of lines
f.write(s)               # Write string s
f.writelines(ls)         # Write a list of strings
f.close()                # Close a file
f.tell()                 # Return current file pointer
f.seek(offset [,where]) # Seek to a new position
                         #    where = 0:  Relative to start
                         #    where = 1:  Relative to current
                         #    where = 2:  Relative to end
f.isatty()               # Return 1 if interactive terminal
f.flush()                # Flush output
f.truncate([size])       # Truncate file to at most size bytes
f.fileno()               # Return integer file descriptor
```

File Objects

File Attributes

- The following attributes provide additional file information

```
f.closed                # Set to 1 if file object has been closed
f.mode                  # 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.

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")
```

File and Path Manipulation

os.path - Functions for portable path manipulation

```
abspath(path)          # Returns the absolute pathname of a path
basename(path)         # Returns filename component of path
dirname(path)          # Returns directory component of path
normcase(path)         # Normalize capitalization of a name
normpath(path)         # Normalize a pathname
split(path)            # Split path into (directory, file)
splitdrive(path)       # Split path into (drive, pathname)
splitext(path)         # Split path into (filename, suffix)
expanduser(path)       # Expands ~user components
expandvars(path)       # Expands environment vars ’$name’ or ’${name}’
join(p1,p2,...)        # Join pathname components
```

Examples

```
abspath("../foo")             # Returns "/home/beazley/blah/foo"
basename("/usr/bin/python")   # Returns "python"
dirname("/usr/bin/python")    # Returns "/usr/bin"
normpath("/usr/./bin/python") # Returns "/usr/bin/python"
split("/usr/bin/python")      # Returns ("/usr/bin","python")
splitext("index.html")        # Returns ("index",".html")
```

File Tests

os.path - Functions for portable filename inquires

```
exists(path)           # Test for existence
isabs(path)            # Return 1 if path is an absolute pathname
isfile(path)           # Return 1 if path is a regular file
isdir(path)            # Return 1 if path is a directory
islink(path)           # Return 1 if path is a symlink
ismount(path)          # Return 1 if path is a mountpoint
getatime(path)         # Get access time
getmtime(path)         # Get modification time
getsize(path)          # 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.

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.

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`	`Open file for reading`
`O_WRONLY`	`Open file for writing`
`O_RDWR`	`Open file for read/write`
`O_APPEND`	`Append to the end of the file`
`O_CREAT`	`Create file if it doesn’t exit`
`O_NONBLOCK`	`Don’t block on open,read, or write.`
`O_TRUNC`	`Truncate to zero length`
`O_TEXT`	`Text mode (Windows)`
`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)
```

Low-Level I/O operations

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

```
os.close(fd)                      # Close a file
os.dup(fd)                        # Duplicate file descriptor fd
os.dup2(oldfd,newfd)              # Duplicate oldfd to newfd
os.fdopen(fd [,mode [,bufsize]])  # Create a file object from an fd
os.fstat(fd)                      # Return file status for fd
os.fstatvfs(fd)                   # Return file system info for fd
os.ftruncate(fd,size)             # Truncate file to given size
os.lseek(fd,pos,how)              # Seek to new position
                                  #    how = 0: beginning of file
                                  #    how = 1: current position
                                  #    how = 2: end of file

os.read(fd,n)                     # Read at most n bytes
os.write(fd,str)                  # Write data in str
```

Notes

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

Low-level File and Directory Manipulation

The os module also contains functions manipulating files and directories

```
os.access(path,accessmode)     # Checks access permissions on a file
os.chmod(path,mode)            # Change file permissions
os.chown(path,uid,gid)         # Change owner and group permissions
os.link(src,dst)               # Create a hard link
os.listdir(path)               # Return a list of names in a directory
os.mkdir(path [,mode])         # Create a directory
os.remove(path)                # Remove a file
os.rename(src,dst)             # Rename a file
os.rmdir(path)                 # Remove a directory
os.stat(path)                  # Return file information
os.statvfs(path)               # Return filesystem information
os.symlink(src,dst)            # Create a symbolic link
os.unlink(path)                # Remove a file (same as remove)
os.utime(path,(atime,mtime))   # Change access and modification times
```

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.

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()
```

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.

Object Serialization and Persistence

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

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)      # file is an open file object
p.dump(obj)                   # Dump object
```

- To unserialize, you 'unpickle' an object

```
p = pickle.Unpickler(file)    # file is an open file
obj = p.load()                # 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.

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

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")         # Open a 'shelf'
d['foo'] = 42                   # Save data
x = d['bar']                    # Retrieve data
```

- Shelf operations

```
d[key] = obj                    # Store an object
obj = d[key]                    # Retrieve an object
del d[key]                      # Delete an object
d.has_key(key)                  # Test for existence of key
d.keys()                        # Return a list of all keys
d.close()                       # Close the shelf
```

Comments

- Keys must be strings.
- Data can be any object serializable with pickle.

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"        # Store a value
s = d["spam"]           # Retrieve a value
del d["name"]           # Delete a value
d.close()               # 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).

Operating System Services

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.

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)         # Change current working directory
os.getcwd()            # Get current working directory
os.umask(mask)         # Change umask setting. Returns previous umask
```

User and group identification

```
os.getegid()           # Get effective group id
os.geteuid()           # Get effective user id
os.getgid()            # Get group id
os.getuid()            # Get user id
os.setgid(gid)         # Set group id
os.setuid(uid)         # Set user id
```

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)            # Execute process, use default path
os.execvpe(path,args, env)
os.wait([pid)]                   # Wait for child process
os.waitpid(pid,options)          # Wait for change in state of child
os.system(command)               # Execute a system command
os._exit(n)                      # Exit immediately with status n.
```

Canonical Example

```
import os
pid = os.fork()         # Create child
if pid == 0:
    # Child process
    os.execvp("ls", ["ls","-l"])
else:
    os.wait()           # Wait for child
```

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

```
(o,i) = popen2.popen2("wc")
i.write(data)         # Write to child’s input
i.close()
result = o.read()     # Get child’s output
o.close()
```

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).

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
```

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)                  # Schedules a SIGALRM signal
signal.pause()                      # Go to sleep until signal
signal.getsignal(signalnum)         # Get signal handler
```

Supported signals (platform specific)

```
SIGABRT      SIGFPE      SIGKILL    SIGSEGV    SIGTTOU
SIGALRM      SIGHUP      SIGPIPE    SIGSTOP    SIGURG
SIGBUS       SIGILL      SIGPOLL    SIGTERM    SIGUSR1
SIGCHLD      SIGINT      SIGPROF    SIGTRAP    SIGUSR2
SIGCLD       SIGIO       SIGPWR     SIGTSTP    SIGVTALRM
SIGCONT      SIGIOT      SIGQUIT    SIGTTIN    SIGWINCH
SIGXCPU      SIGXFSZ
```

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
```

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.

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
time.localtime(secs)   # Convert time to local time (returns a tuple).
time.gmtime(secs)      # Convert time to GMT (returns a tuple)
time.asctime(tuple)    # Creates a string representing the time
time.ctime(secs)       # Create a string representing local time
time.mktime(tuple)     # Convert time tuple to seconds
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)
```

Getting User and Group Information

The pwd module

- Provides access to the Unix password database

```
pwd.getpwuid(uid)       # Returns passwd entry for uid
pwd.getpwname(login)    # Returns passwd entry for login
pwd.getpwall()          # 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)      # Return group entry for gid
grp.getgrnam(gname)    # Return group entry for gname
grp.getgrall()         # Get all entries
```

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.

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.

Threads

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.)

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...)

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.

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
...
```

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.

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).

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)    # Required
           self.interval = interval
     def run(self):
           while 1:
                time.sleep(self.interval)
                print time.ctime(time.time())

t = PrintTime(5)    # Create a thread object
t.start()           # Start it
...
```

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])      # Wait for thread t to terminate
t.getName()            # Get the name of the thread
t.setName(name)        # Set the name of the thread
t.isAlive()            # Return 1 if thread is alive.
t.isDaemon()           # Return daemonic flag
t.setDaemon(val)       # 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.

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.

Lock Objects

The Lock object

- Provides a simple mutual exclusion lock

```
import threading
data = [ ]                    # Some data
lck = threading.Lock()        # Create a 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.

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
lck = threading.Lock()       # Create a 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
```

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()          # Wait for data to show up
    r = data.pop()
    cv.release()            # Release the lock
    return r
# Producer thread
def produce_item(obj):
    cv.acquire()           # Acquire the lock
    data.append(obj)
    cv.notify()            # Notify a consumer
    cv.release()           # Release the lock
```

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)     # No more than 5 threads allowed
def fetch_file(host,filename):
    sem.acquire()                # Decrements count or blocks if zero
    ...
    blah
    ...
    sem.release()                # Increment count
```

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()

# Wait for event
def wait_for_event():
    e.wait()

# Clear event
def clear_event():
    e.clear()
```

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

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.

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)      # Create a queue
q.qsize()               # Return current size
q.empty()               # Test if empty
q.full()                # Test if full
q.put(item)             # Put an item on the queue
q.get()                 # 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.

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.

Network Programming

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.

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.

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
Telnet         Port 23
SMTP (Mail)    Port 25
HTTP (WWW)     Port 80
```

- 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).

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)    # Create TCP socket
s.bind(("",8888))                   # Bind to port 8888
s.listen(5)                         # 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.

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)                    # Receive up to 1024 bytes
s.close()                            # Close connection
print "The time is", tm
```

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.

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()           # Name of local machine
socket.ntohl(x)                # Convert 32-bit integer to host order
socket.ntohs(x)                # Convert 16-bit integer to host order
socket.htonl(x)                # Convert 32-bit integer to network order
socket.htons(x)                # 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).

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          Stream socket (TCP)
SOCK_DGRAM           Datagram socket (UDP)
SOCK_RAW             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.

The socket Module (cont)

socket methods

```
s.accept()                 # Accept a new connection
s.bind(address)            # Bind to an address and port
s.close()                  # Close the socket
s.connect(address)         # Connect to remote socket
s.fileno()                 # Return integer file descriptor
s.getpeername()            # Get name of remote machine
s.getsockname()            # Get socket address as (ipaddr,port)
s.getsockopt(...)          # Get socket options
s.listen(backlog)          # Start listening for connections
s.makefile(mode)           # Turn socket into a file object
s.recv(bufsize)            # Receive data
s.recvfrom(bufsize)        # Receive data (UDP)
s.send(string)             # Send data
s.sendto(string, address)  # Send packet (UDP)
s.setblocking(flag)        # Set blocking or nonblocking mode
s.setsockopt(...)          # Set socket options
s.shutdown(how)            # 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.

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
# This class actually implements the server functionality
class TimeHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.send(time.ctime(time.time()))

# Create the server
server = SocketServer.TCPServer(("",8888),TimeHandler)
server.serve_forever()
```

Comments

- The module provides a number of specialized server and handler types.
- Ex: ForkingTCPServer, ThreadingTCPServer, StreamRequestHandler, etc.

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.

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
...
```

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.

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)        # Quotes a string for use in a URL
urllib.quote_plus(str)   # Also replaces spaces with ’+’
urllib.unquote(str)      # Opposite of quote()
urllib.unquote_plus(str) # Opposite of quote_plus()
urllib.urlencode(dict)   # Turns a dictionary of key=value
                         # pairs into a HTTP query-string
```

- Examples

```
urllib.quote("beazley@cs")      # Produces "beazley%40cs"
urllib.unquote("%23%21/bin/sh") # Produces "/bin/sh"
```

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"
```

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.

CGI Scripting

CGI Example

```
#!/usr/local/bin/python
print "Content-type: text/html\n"
print "<h1>Hello World</h1>
```

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.

cgi Module Example

Example of using CGI module

```
#!/usr/local/bin/python
import cgi
form = cgi.FieldStorage()         # Read query string
name = form["name"].value         # Get ’name’ field from form
email = form["email"].value       # 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...)

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.

Data Encoding

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

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)     # Files
es = base64.encodestring(s)             # Strings
```

Decoding

```
import base64
base64.decode(inputfile, outputfile)    # Files
s = base64.decodestring(es)             # String
```

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)
```

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)
```

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.

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

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]              # Return data for header name
m[name] = value      # Add a header
m.keys()             # Return a list of all header names
m.values()           # Return list of header values
m.items()            # Return list of (header,value) pairs
m.has_key(name)      # Test for existence
m.getallmatchingheaders(name)    # Return list of all matching headers
m.getaddr(name)      # Return (full_name, email) for an address field
m.getaddrlist(name)  # Get a list of email addresses
m.getdate(name)      # Get a date as a time tuple
```

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'	Pad byte	'I'	unsigned int
'c'	char	'l'	long
'b'	signed char	'L'	unsigned long
'B'	unsigned char	'f'	float
'h'	short	'd'	double
'H'	unsigned short	's'	char[]
'i'	int	'P'	void *

- Example

```
s = struct.pack("hhii", 34, 73, 162773, 2222)
s = struct.pack("is", len(t), t)
```

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 size and alignment
'='        Native byte order     Standard size and alignment
'<'        Little endian         Standard size and alignment
'>'        Big endian            Standard size and alignment
'!'        Network order         Standard size and alignment
```

- Native alignment uses the size and alignment rules of the C compiler.
- Standard alignment uses no padding and assumes the following sizes

```
short    2 bytes          int      4 bytes
long     4 bytes          float    32 bits
double   64 bits
```

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.

Restricted Execution

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

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     # List of prohibited built-in functions
RExec.ok_builtin_modules    # List of modules that can be imported
RExec.ok_path               # List of directories searched on import
RExec.ok_posix_names        # List of accepted functions in os module
RExec.ok_sys_names          # List of members in sys module
```

- Methods on an instance of RExec

```
r.r_eval(code)             # Evaluate code in restricted mode
r.r_exec(code)             # Execute code in restricted mode
r.r_execfile(filename)     # Execute file in restricted more
```

- A few methods which may be redefined

```
r.r_import(modulename)     # Called whenever code imports
r.r_open(filename,mode)    # Called whenever code opens a file
```

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.

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()             # Okay
sbast.write("Blah")          # Fails. Attribute error.
```

Note

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

C Extensions

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’

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;
}
```

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);
}
```

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);
}
```

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

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
>>> import spam
>>> spam.gcd(63,56)
7
>>> spam.gcd(71,89)
1
>>>
```

It’s almost too easy...

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)

Conclusions

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