Data Representation

"PALO ALTO, Calif. — Intel says its Pentium Pro and new Pentium II chips have a flaw that can cause computers to sometimes make mistakes but said the problems could be fixed easily with rewritten software"

from a Reuters telegram

Overview

This chapter describes a number of modules that can be used to convert between Python objects and other data representations. They are often used to read and write foreign file formats, and to store or transfer Python variables.

Binary data

Python provides several support modules that help you decode and encode binary data formats. The **struct** module can convert between binary data structures (like C structs) and Python tuples. The **array** module wraps binary arrays of data (C arrays) into a Python sequence object.

Self-describing formats

To pass data between different Python programs, you can **marshal** or **pickle** your data.

The **marshal** module uses a simple self-describing format which supports most built-in data types, including code objects. Python uses this format itself, to store compiled code on disk (in PYC files).

The **pickle** module provides a more sophisticated format, which supports user-defined classes, self-referencing data structures, and more. This module is available in two versions; the basic **pickle** module is written in Python, and is relatively slow, while **cPickle** is written in C, and is usually as fast as **marshal**.

Output formatting

This group of modules supplement built-in formatting functions like **repr**, and the **%** string formatting operator.

The **pprint** module can print almost any Python data structure in a nice, readable way (well, as readable as it can make things, that is).

The **repr** module provides a replacement for the built-in function with the same name. The version in this module applies tight limits on most things; it doesn't print more than 30 characters from each string, it doesn't print more than a few levels of deeply nested data structures, etc.

Encoded binary data

Python supports most common binary encodings, such as **base64**, **binhex** (a macintosh format), **quoted printable**, and **uu** encoding.

The array module

This module implements an efficient array storage type. Arrays are similar to lists, but all items must be of the same primitive type. The type is defined when the array is created.

Here are some simple examples. The first example creates an **array** object, and copies the internal buffer to a string through the **tostring** method:

Example: Using the array module to convert lists of integers to strings

```
# File:array-example-1.py

import array

a = array.array("B", range(16)) # unsigned char
b = array.array("h", range(16)) # signed short

print a
print repr(a.tostring())

print b
print repr(b.tostring())
```

```
array('B', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
'\000\001\002\003\004\005\006\007\010\011\012\013\014\015\016\017'
array('h', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
'\000\000\001\000\002\000\003\000\004\000\005\000\006\000\007\000
\010\000\011\000\012\000\013\000\014\000\015\000\016\000\017\000'
```

The **array** objects can be treated as ordinary lists, to some extent. You cannot concatenate arrays if they have different type codes, though.

Example: Using arrays as ordinary sequences

```
# File:array-example-2.py

import array

a = array.array("B", [1, 2, 3])

a.append(4)

a = a + a

a = a[2:-2]

print a
print repr(a.tostring())
for i in a:
    print i,
```

```
array('B', [3, 4, 1, 2])
'\003\004\001\002'
3 4 1 2
```

This module also provides a very efficient way to turn raw binary data into a sequence of integers (or floating point values, for that matter):

Example: Using arrays to convert strings to lists of integers

```
# File:array-example-3.py

import array

a = array.array("i", "fish license") # signed integer

print a
print repr(a.tostring())
print a.tolist()
```

```
array('i', [1752394086, 1667853344, 1702063717])
'fish license'
[1752394086, 1667853344, 1702063717]
```

Finally, here's how to use this module to determine the endianess of the current platform:

Example: Using the array module to determine platform endianess

```
# File:array-example-4.py

import array

def little_endian():
    return ord(array.array("i",[1]).tostring()[0])

if little_endian():
    print "little-endian platform (intel, alpha)"
else:
    print "big-endian platform (motorola, sparc)"
```

```
big-endian platform (motorola, sparc)
```

Python 2.0 and later provides a **sys.byteorder** attribute, which is set to either "**little**" or "**big**":

Example: Using the sys.byteorder attribute to determine platform endianess (Python 2.0)

```
# File:sys-byteorder-example-1.py

import sys

# available in Python 2.0 and later
if sys.byteorder == "little":
    print "little-endian platform (intel, alpha)"
else:
    print "big-endian platform (motorola, sparc)"
```

```
'big-endian platform (motorola, sparc)'
```

The struct module

This module contains functions to convert between binary strings and Python tuples. The **pack** function takes a format string and one or more arguments, and returns a binary string. The **unpack** function takes a string and returns a tuple.

Example: Using the struct module

```
# File:struct-example-1.py

import struct

# native byteorder
buffer = struct.pack("ihb", 1, 2, 3)
print repr(buffer)
print struct.unpack("ihb", buffer)

# data from a sequence, network byteorder
data = [1, 2, 3]

buffer = struct.pack("!ihb", *data)

# in Python 1.5.2 and earlier, use this instead:
# buffer = apply(struct.pack, ("!ihb",) + tuple(data))

print repr(buffer)
print struct.unpack("!ihb", buffer)
```

```
'\001\000\000\000\002\000\003'
(1, 2, 3)
'\000\000\000\001\000\002\003'
(1, 2, 3)
```

The xdrlib module

This module converts between Python data types and Sun's external data representation (XDR).

Example: Using the xdrlib module

```
# File:xdrlib-example-1.py

import xdrlib

#
# create a packer and add some data to it

p = xdrlib.Packer()
p.pack_uint(1)
p.pack_string("spam")

data = p.get_buffer()

print "packed:", repr(data)

#
# create an unpacker and use it to decode the data

u = xdrlib.Unpacker(data)

print "unpacked:", u.unpack_uint(), repr(u.unpack_string())

u.done()
```

```
packed: '\000\000\000\001\000\000\000\004spam'
unpacked: 1 'spam'
```

The XDR format is used by Sun's remote procedure call (RPC) protocol. Here's an incomplete (and rather contrived) example showing how to build an RPC request package:

Example: Using the xdrlib module to send a RPC call package

```
# File:xdrlib-example-2.py

import xdrlib

# some constants (see the RPC specs for details)
RPC_CALL = 1
RPC_VERSION = 2

MY_PROGRAM_ID = 1234 # assigned by Sun
MY_VERSION_ID = 1000
MY_TIME_PROCEDURE_ID = 9999

AUTH_NULL = 0

transaction = 1

p = xdrlib.Packer()

# send an Sun RPC call package
p.pack_uint(transaction)
p.pack_enum(RPC_CALL)
p.pack_uint(RPC_VERSION)
p.pack_uint(MY_PROGRAM_ID)
p.pack_uint(MY_VERSION_ID)
p.pack_uint(MY_TIME_PROCEDURE_ID)
p.pack_enum(AUTH_NULL)
p.pack_uint(0)
p.pack_enum(AUTH_NULL)
p.pack_uint(0)

print repr(p.get_buffer())
```

```
'\000\000\000\001\000\000\000\001\000\000\000\002\000\000\004\322
\000\000\003\350\000\000\'\017\000\000\000\000\000\000\000\000\000
\000\000\000\000\000\000\000'
```

The marshal module

This module is used to serialize data; that is, convert data to and from character strings, so that they can be stored on file or sent over a network.

Marshal uses a simple self-describing data format. For each data item, the marshalled string contains a type code, followed by one or more type specific fields. Integers are stored in little-endian order, strings as a length field followed by the string's contents (which can include null bytes), tuples as a length field followed by the objects that make up the tuple, etc.

Example: Using the marshal module to serialize data

```
# File:marshal-example-1.py

import marshal

value = (
    "this is a string",
    [1, 2, 3, 4],
    ("more tuples", 1.0, 2.3, 4.5),
    "this is yet another string"
    )

data = marshal.dumps(value)

# intermediate format
print type(data), len(data)

print "-"*50
print repr(data)
print "-"*50

print marshal.loads(data)
```

```
<type 'string'> 118
--------------------------------------------------
'(\004\000\000\000s\020\000\000\000this is a string
[\004\000\000\000i\001\000\000\000i\002\000\000\000
i\003\000\000\000i\004\000\000\000(\004\000\000\000
s\013\000\000\000more tuplesf\0031.0f\0032.3f\0034.
5s\032\000\000\000this is yet another string'
--------------------------------------------------
('this is a string', [1, 2, 3, 4], ('more tuples',
1.0, 2.3, 4.5), 'this is yet another string')
```

The marshal module can also handle code objects (it's used to store precompiled Python modules).

Example: Using the marshal module to serialize code

```
# File:marshal-example-2.py

import marshal

script = """
print 'hello'
"""

code = compile(script, "<script>", "exec")

data = marshal.dumps(code)

# intermediate format
print type(data), len(data)

print "-"*50
print repr(data)
print "-"*50

exec marshal.loads(data)
```

```
<type 'string'> 81
--------------------------------------------------
'c\000\000\000\000\001\000\000\000s\017\000\000\00
0\177\000\000\177\002\000d\000\000GHd\001\000S(\00
2\000\000\000s\005\000\000\000helloN(\000\000\000\
000(\000\000\000\000s\010\000\000\000<script>s\001
\000\000\000?\002\000s\000\000\000\000'
--------------------------------------------------
hello
```

The pickle module

This module is used to serialize data; that is, convert data to and from character strings, so that they can be stored on file or send over a network. It's a bit slower than **marshal**, but it can handle class instances, shared elements, and recursive data structures, among other things.

Example: Using the pickle module

```
# File:pickle-example-1.py

import pickle

value = (
    "this is a string",
    [1, 2, 3, 4],
    ("more tuples", 1.0, 2.3, 4.5),
    "this is yet another string"
    )

data = pickle.dumps(value)

# intermediate format
print type(data), len(data)

print "-"*50
print data
print "-"*50

print pickle.loads(data)
```

```
<type 'string'> 121
--------------------------------------------------
(S'this is a string'
p0
(lp1
I1
aI2
aI3
aI4
a(S'more tuples'
p2
F1.0
F2.3
F4.5
tp3
S'this is yet another string'
p4
tp5
.
--------------------------------------------------
('this is a string', [1, 2, 3, 4], ('more tuples',
1.0, 2.3, 4.5), 'this is yet another string')
```

On the other hand, **pickle** cannot handle code objects (but see the **copy_reg** module for a way to fix this).

By default, pickle uses a text-based format. You can also use a binary format, in which numbers and binary strings are stored in a compact binary format. The binary format usually results in smaller files.

Example: Using the pickle module in binary mode

```
# File:pickle-example-2.py

import pickle
import math

value = (
    "this is a long string" * 100,
    [1.2345678, 2.3456789, 3.4567890] * 100
    )

# text mode
data = pickle.dumps(value)
print type(data), len(data), pickle.loads(data) == value

# binary mode
data = pickle.dumps(value, 1)
print type(data), len(data), pickle.loads(data) == value
```

The cPickle module

(Optional). This module contains a faster reimplementation of the **pickle** module.

Example: Using the cPickle module

```
# File:cpickle-example-1.py

try:
    import cPickle
    pickle = cPickle
except ImportError:
    import pickle

class Sample:
    def __init__(self, value):
        self.value = value

sample = Sample(1)

data = pickle.dumps(sample)

print pickle
print repr(data)
```

```
<module 'cPickle' (built-in)>
"(i__main__\012Sample\012p1\012(dp2\012S'value'\012p3\012I1\012sb."
```

The copy_reg module

This module provides a registry that you can use to register your own extension types. The **pickle** and **copy** modules use this registry to figure out how to process non-standard types.

For example, the standard **pickle** implementation cannot deal with Python code objects, as shown by the following example:

```
# File:copy-reg-example-1.py

import pickle

CODE = """
print 'good evening'
"""

code = compile(CODE, "<string>", "exec")

exec code
exec pickle.loads(pickle.dumps(code))
```

```
good evening
Traceback (innermost last):
...
pickle.PicklingError: can't pickle 'code' objects
```

We can work around this by registering a code object handler. Such a handler consists of two parts; a *pickler* which takes the code object and returns a tuple that can only contain simple data types, and an *unpickler* which takes the contents of such a tuple as its arguments:

Example: Using the copy_reg module to enable pickling of code objects

```
# File:copy-reg-example-2.py

import copy_reg
import pickle, marshal, types

#
# register a pickle handler for code objects

def code_unpickler(data):
    return marshal.loads(data)

def code_pickler(code):
    return code_unpickler, (marshal.dumps(code),)

copy_reg.pickle(types.CodeType, code_pickler, code_unpickler)

#
# try it out

CODE = """
print "suppose he's got a pointed stick"
"""

code = compile(CODE, "<string>", "exec")

exec code
exec pickle.loads(pickle.dumps(code))
```

```
suppose he's got a pointed stick
suppose he's got a pointed stick
```

If you're transferring the pickled data across a network, or to another program, the custom unpickler must of course be available at the receiving end as well.

For the really adventurous, here's a version that allows you to pickle open file objects:

Example: Using the copy_reg module to enable pickling of file objects

```
# File:copy-reg-example-3.py

import copy_reg
import pickle, types
import StringIO

#
# register a pickle handler for file objects

def file_unpickler(position, data):
    file = StringIO.StringIO(data)
    file.seek(position)
    return file

def file_pickler(code):
    position = file.tell()
    file.seek(0)
    data = file.read()
    file.seek(position)
    return file_unpickler, (position, data)

copy_reg.pickle(types.FileType, file_pickler, file_unpickler)

#
# try it out

file = open("samples/sample.txt", "rb")

print file.read(120),
print "<here>",
print pickle.loads(pickle.dumps(file)).read()
```

```
We will perhaps eventually be writing only small
modules which are identified by name as they are
used to build larger <here>  ones, so that devices like
indentation, rather than delimiters, might become
feasible for expressing local structure in the
source language.
    -- Donald E. Knuth, December 1974
```

The pprint module

This module is a "pretty printer" for Python data structures. It's useful if you have to print non-trivial data structures to the console.

Example: Using the pprint module

```
# File:pprint-example-1.py

import pprint

data = (
    "this is a string", [1, 2, 3, 4], ("more tuples",
    1.0, 2.3, 4.5), "this is yet another string"
    )

pprint.pprint(data)
```

```
('this is a string',
 [1, 2, 3, 4],
 ('more tuples', 1.0, 2.3, 4.5),
 'this is yet another string')
```

The repr module

This module provides a version of the built-in **repr** function, with limits on most sizes (string lengths, recursion, etc).

Example: Using the repr module

```
# File:repr-example-1.py

# note: this overrides the built-in 'repr' function
from repr import repr

# an annoyingly recursive data structure
data = (
    "X" * 100000,
    )
data = [data]
data.append(data)

print repr(data)
```

```
[('XXXXXXXXXXXX...XXXXXXXXXXXXX',), [('XXXXXXXXXXXX...XXXXXXXXXX
XXX',), [('XXXXXXXXXXXX...XXXXXXXXXXXXX',), [('XXXXXXXXXXXX...XX
XXXXXXXXXX',), [('XXXXXXXXXXXX...XXXXXXXXXXXXX',), [(...), [...
]]]]]]]
```

The base64 module

The **base64** encoding scheme is used to convert arbitrary binary data to plain text. To do this, the encoder stores each group of three binary bytes as a group of four characters from the following set:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789+/
```

In addition, the = character is used for padding at the end of the data stream.

The **encode** and **decode** functions work on file objects:

Example: Using the base64 module to encode files

```
# File:base64-example-1.py

import base64

MESSAGE = "life of brian"

file = open("out.txt", "w")
file.write(MESSAGE)
file.close()

base64.encode(open("out.txt"), open("out.b64", "w"))
base64.decode(open("out.b64"), open("out.txt", "w"))

print "original:", repr(MESSAGE)
print "encoded message:", repr(open("out.b64").read())
print "decoded message:", repr(open("out.txt").read())
```

```
original: 'life of brian'
encoded message: 'bGlmZSBvZiBicmlhbg==\012'
decoded message: 'life of brian'
```

The **encodestring** and **decodestring** functions convert between strings instead. They're currently implemented as wrappers on top of **encode** and **decode**, using **StringIO** objects for input and output.

Example: Using the base64 module to encode strings

```
# File:base64-example-2.py

import base64

MESSAGE = "life of brian"

data = base64.encodestring(MESSAGE)

original_data = base64.decodestring(data)

print "original:", repr(MESSAGE)
print "encoded data:", repr(data)
print "decoded data:", repr(original_data)
```

```
original: 'life of brian'
encoded data: 'bGlmZSBvZiBicmlhbg==\012'
decoded data: 'life of brian'
```

Here's how to convert a user name and a password to a HTTP basic authentication string. Note that you don't really have to work for the NSA to be able to decode this format...

Example: Using the base64 module for basic authentication

```
# File:base64-example-3.py

import base64

def getbasic(user, password):
    # basic authentication (according to HTTP)
    return base64.encodestring(user + ":" + password)

print getbasic("Aladdin", "open sesame")
```

```
'QWxhZGRpbjpvcGVuIHNlc2FtZQ=='
```

Finally, here's a small utility that converts a GIF image to a Python script, for use with the Tkinter library:

Example: Using the base64 module to wrap GIF images for Tkinter

```
# File:base64-example-4.py

import base64, sys

if not sys.argv[1:]:
    print "Usage: gif2tk.py giffile >pyfile"
    sys.exit(1)

data = open(sys.argv[1], "rb").read()

if data[:4] != "GIF8":
    print sys.argv[1], "is not a GIF file"
    sys.exit(1)

print '# generated from', sys.argv[1], 'by gif2tk.py'
print
print 'from Tkinter import PhotoImage'
print
print 'image = PhotoImage(data="""'
print base64.encodestring(data),
print '""")'
```

```
# generated from samples/sample.gif by gif2tk.py

from Tkinter import PhotoImage

image = PhotoImage(data="""
R0lGODlhoAB4APcAAAAAAIAAAACAAICAAAAAgIAAgACAgICAgAQEBIwEBIyMBJRUlISE/L
RUBAQE
...
AjmQBFmQBnmQCJmQCrmQDNmQDvmQEBmREnkRAQEAOw==
""")
```

The binhex module

This module converts to and from the Macintosh binhex format.

Example: Using the binhex module

```
# File:binhex-example-1.py

import binhex
import sys

infile = "samples/sample.jpg"

binhex.binhex(infile, sys.stdout)
```

```
(This file must be converted with BinHex 4.0)

:#R0KEA"XC5jUF'F!2j!)!*!%%TS!N!4RdrrBrq!!%%T'58B!!3%!!!%!!3!!rpX
!3`!)"JB("J8)"`F(#3N)#J`8$3`,#``C%K-2&"dD(aiG'K`F)#3Z*b!L,#-F(#J
h+5``-63d0"mR16di-M`Z-c3brpX!3`%*#3N-#``B$3dB-L%F)6+3-[r!!"%)!)!
!J!-")J!#%3%$%3(ra!!I!!!""3'3"J#3#!%#!`3&"JF)#3S,rm3!Y4!!!J%$!`)
%!`8&"!3!!!&p!3)$!!34"4)K-8%'%e&K"b*a&$+"ND%))d+a`495dI!N-f*bJJN
```

The quopri module

This module implements quoted printable encoding, according to the MIME standard.

This encoding can be used if you need to convert text messages which mostly consists of plain US ASCII text, such as messages written in most European languages, to messages that only use US ASCII. This can be quite useful if you're sending stuff via steam-powered mail transports to people using vintage mail agents.

Example: Using the quopri module

```
# File:quopri-example-1.py

import quopri
import StringIO

# helpers (the quopri module only supports file-to-file conversion)

def encodestring(instring, tabs=0):
    outfile = StringIO.StringIO()
    quopri.encode(StringIO.StringIO(instring), outfile, tabs)
    return outfile.getvalue()

def decodestring(instring):
    outfile = StringIO.StringIO()
    quopri.decode(StringIO.StringIO(instring), outfile)
    return outfile.getvalue()

#
# try it out

MESSAGE = "å i åa ä e ö!"

encoded_message = encodestring(MESSAGE)
decoded_message = decodestring(encoded_message)

print "original:", MESSAGE
print "encoded message:", repr(encoded_message)
print "decoded message:", decoded_message
```

```
original: å i åa ä e ö!
encoded message: '=E5 i =E5a =E4 e =F6!\012'
decoded message: å i åa ä e ö!
```

As this example shows, non-US characters are mapped to an '=' followed by two hexadecimal digits. So is the '=' character itself ("=3D"), as well as whitespace at the end of lines ("=20"). Everything else looks just like before. So provided you don't use too many weird characters, the encoded string is nearly as readable as the original.

(Europeans generally hate this encoding, and strongly believe that certain US programmers deserve to be slapped in the head with a huge great fish to the jolly music of Edward German...)

The uu module

The **UU** encoding scheme is used to convert arbitrary binary data to plain text. This format is quite popular on the Usenet, but is slowly being superseded by base64 encoding.

An UU encoder takes groups of three bytes (24 bits), and converts each group to a sequence of four printable characters (6 bits per character), using characters from chr(32) (space) to chr(95). Including the length marker and line feed characters, UU encoding typically expands data by 40%.

An encoded data stream starts with a **begin** line, which also includes the file privileges (the Unix mode field, as an octal number) and the filename, and ends with an **end** line:

```
begin 666 sample.jpg
M_]C_X  02D9)1@ ! 0   0 !  #_VP!#  @&!@<&!0@'!P<)'0@*#!0-# L+
...more lines like this...
end
```

The **uu** module provides two functions, **encode** and **decode**:

encode(infile, outfile, filename) encodes data from the input file and writes it to the output file. The input and output file arguments can be either filenames or file objects. The third argument is used as filename in the **begin** field.

Example: Using the uu module to encode a binary file

```
# File:uu-example-1.py

import uu
import os, sys

infile = "samples/sample.jpg"

uu.encode(infile, sys.stdout, os.path.basename(infile))
```

```
begin 666 sample.jpg
M_]C_X  02D9)1@ ! 0   0 !  #_VP!#  @&!@<&!0@'!P<)"0@*#!0-# L+
M#!D2$P\4'1H?'AT:'!P@)"XG("(L(QP<*#<I+# Q-#0T'R<Y/3@R/"XS-#+_
MVP!# 0D)"0P+#!@-#1@R(1PA,C(R,C(R,C(R,C(R,C(R,C(R,C(R,C(R,C(R
M,C(R,C(R,C(R,C(R,C(R,C(R,C(R,C+_P  1" "  ( # 2(  A$! Q$!_\0
M'P   04! 0$! 0$         $" P0%!@<("0H+_\0 M1   @$# P($ P4%
```

decode(infile, outfile) decodes uu-encoded data from the input text file, and writes it to the output file. Again, both arguments can be either filenames or file objects.

Example: Using the uu module to decode a uu-encoded file

```
# File:uu-example-2.py

import uu
import StringIO

infile = "samples/sample.uue"
outfile = "samples/sample.jpg"

#
# decode

fi = open(infile)
fo = StringIO.StringIO()

uu.decode(fi, fo)

#
# compare with original data file

data = open(outfile, "rb").read()

if fo.getvalue() == data:
    print len(data), "bytes ok"
```

The binascii module

This module contains support functions for a number of encoding modules, including **base64**, **binhex**, and **uu**.

In 2.0 and newer, it also allows you to convert binary data to and from hexadecimal strings.

Example: Using the binascii module

```
# File:binascii-example-1.py

import binascii

text = "hello, mrs teal"

data = binascii.b2a_base64(text)
text = binascii.a2b_base64(data)
print text, "<=>", repr(data)

data = binascii.b2a_uu(text)
text = binascii.a2b_uu(data)
print text, "<=>", repr(data)

data = binascii.b2a_hqx(text)
text = binascii.a2b_hqx(data)[0]
print text, "<=>", repr(data)

# 2.0 and newer
data = binascii.b2a_hex(text)
text = binascii.a2b_hex(data)
print text, "<=>", repr(data)
```

```
hello, mrs teal <=> 'aGVsbG8sIG1ycyB0ZWFs\012'
hello, mrs teal <=> '/:&5L;&\\L(&UR<R!T96%L\012'
hello, mrs teal <=> 'D\'9XE\'mX)\'ebFb"dC@&X'
hello, mrs teal <=> '68656c6c6f2c206d7273207465616c'
```