8.3. `struct` — Interpret strings as packed binary data¶

This module performs conversions between Python values and C structs represented as Python strings. It uses *format strings* (explained below) as compact descriptions of the lay-out of the C structs and the intended conversion to/from Python values. This can be used in handling binary data stored in files or from network connections, among other sources.

The module defines the following exception and functions:

exception `struct.error`¶
Exception raised on various occasions; argument is a string describing what is wrong.

`struct.pack`(*fmt*, *v1*, *v2*, ...)¶
Return a string containing the values `v1, v2, ...` packed according to the given format. The arguments must match the values required by the format exactly.

`struct.pack_into`(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)¶

Pack the values `v1, v2, ...` according to the given format, write the packed bytes into the writable *buffer* starting at *offset*. Note that the offset is a required argument.

New in version 2.5.

`struct.unpack`(*fmt*, *string*)¶
Unpack the string (presumably packed by `pack(fmt, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (`len(string)` must equal `calcsize(fmt)`).

`struct.unpack_from`(*fmt*, *buffer*[, *offset=0*])¶

Unpack the *buffer* according to the given format. The result is a tuple even if it contains exactly one item. The *buffer* must contain at least the amount of data required by the format (`len(buffer[offset:])` must be at least `calcsize(fmt)`).

New in version 2.5.

`struct.calcsize`(*fmt*)¶
Return the size of the struct (and hence of the string) corresponding to the given format.

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types:

Format	C Type	Python	Notes
x	pad byte	no value	
c	char	string of length 1	
b	signed char	integer	
B	unsigned char	integer	
?	_Bool	bool	(1)
h	short	integer	
H	unsigned short	integer	
i	int	integer	
I	unsigned int	integer or long	
l	long	integer	
L	unsigned long	long	
q	long long	long	(2)
Q	unsigned long long	long	(2)
f	float	float	
d	double	float	
s	char[]	string	
p	char[]	string	
P	void *	long	

Notes:

The `'?'` conversion code corresponds to the `_Bool` type defined by C99. If this type is not available, it is simulated using a `char`. In standard mode, it is always represented by one byte.

New in version 2.6.

The `'q'` and `'Q'` conversion codes are available in native mode only if the platform C compiler supports C `long long`, or, on Windows, `__int64`. They are always available in standard modes.

New in version 2.2.

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the `'s'` format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string, while `'10c'` means 10 characters. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

The `'p'` format character encodes a "Pascal string", meaning a short variable-length string stored in a fixed number of bytes. The count is the total number of bytes stored. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to `pack()` is too long (longer than the count minus 1), only the leading count-1 bytes of the string are stored. If the string is shorter than count-1, it is padded with null bytes so that exactly count bytes in all are used. Note that for `unpack()`, the `'p'` format character consumes count bytes, but that the string returned can never contain more than 255 characters.

For the `'I'`, `'L'`, `'q'` and `'Q'` format characters, the return value is a Python long integer.

For the `'P'` format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A *NULL* pointer will always be returned as the Python integer `0`. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

For the `'?'` format character, the return value is either `True` or `False`. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be True when unpacking.

By default, C numbers are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Byte order	Size and alignment
@	native	native
=	native	standard
<	little-endian	standard
>	big-endian	standard
!	network (= big-endian)	standard

If the first character is not one of these, `'@'` is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Motorola and Sun processors are big-endian; Intel and DEC processors are little-endian.

Native size and alignment are determined using the C compiler's `sizeof` expression. This is always combined with native byte order.

Standard size and alignment are as follows: no alignment is required for any type (so you have to use pad bytes); `short` is 2 bytes; `int` and `long` are 4 bytes; `long long` (`__int64` on Windows) is 8 bytes; `float` and `double` are 32-bit and 64-bit IEEE floating point numbers, respectively. `_Bool` is 1 byte.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` is available for those poor souls who claim they can't remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the `'P'` format is not available.

Examples (all using native byte order, size and alignment, on a big-endian machine):

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
```

Hint: to align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. For example, the format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries. This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> record = 'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', s))
Student(name='raymond   ', serialnum=4658, school=264, gradelevel=8)
```

See also

Module `array`

Packed binary storage of homogeneous data.

Module `xdrlib`

Packing and unpacking of XDR data.

8.3.1. Struct Objects¶

The `struct` module also defines the following type:

class `struct.Struct`(*format*)¶

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a Struct object once and calling its methods is more efficient than calling the `struct` functions with the same format since the format string only needs to be compiled once.

New in version 2.5.

Compiled Struct objects support the following methods and attributes:

`pack`(*v1*, *v2*, ...)¶
Identical to the `pack()` function, using the compiled format. (`len(result)` will equal `self.size`.)

`pack_into`(*buffer*, *offset*, *v1*, *v2*, ...)¶
Identical to the `pack_into()` function, using the compiled format.

`unpack`(*string*)¶
Identical to the `unpack()` function, using the compiled format. (`len(string)` must equal `self.size`).

`unpack_from`(*buffer*[, *offset=0*])¶
Identical to the `unpack_from()` function, using the compiled format. (`len(buffer[offset:])` must be at least `self.size`).

`format`¶
The format string used to construct this Struct object.

`size`¶
The calculated size of the struct (and hence of the string) corresponding to `format`.

Previous topic

8.2. `re` — Regular expression operations

Next topic

8.4. `difflib` — Helpers for computing deltas

This Page

- Show Source

Navigation

- index

Last updated on Feb 26, 2010. Created using Sphinx 0.6.3.