7.1. `struct` — Interpret bytes as packed binary data

Source code: Lib/struct.py

This module performs conversions between Python values and C structs represented as Python `bytes` objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses Format Strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

> **Note:** By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use `standard` size and alignment instead of `native` size and alignment: see Byte Order, Size, and Alignment for details.

Several `struct` functions (and methods of `Struct`) take a *buffer* argument. This refers to objects that implement the Buffer Protocol and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.

7.1.1. Functions and Exceptions

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 bytes object containing the values *v1*, *v2*, … packed according to the format string *fmt*. 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 format string *fmt* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

`struct.`**`unpack`**(*fmt*, *buffer*)

Unpack from the buffer *buffer* (presumably packed by `pack(fmt, ...)`) according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by `calcsize()`.

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

Unpack from *buffer* starting at position *offset*, according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, minus *offset*, must be at least the size required by the format, as reflected by `calcsize()`.

`struct.`**`iter_unpack`**(*fmt*, *buffer*)

Iteratively unpack from the buffer *buffer* according to the format string *fmt*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `calcsize()`.

Each iteration yields a tuple as specified by the format string.

New in version 3.4.

`struct.`**`calcsize`**(*fmt*)

Return the size of the struct (and hence of the bytes object produced by `pack(fmt, ...)`) corresponding to the format string *fmt*.

7.1.2. Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from Format Characters, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the Byte Order, Size, and Alignment.

7.1.2.1. Byte Order, Size, and Alignment

By default, C types 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	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

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, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system.

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

Standard size depends only on the format character; see the table in the Format Characters section.

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 `'>'`.

Notes:

1. Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.
2. No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.
3. 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. See Examples.

7.1.2.2. Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The ‘Standard size’ column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value		
`c`	`char`	bytes of length 1	1	
`b`	`signed char`	integer	1	(1),(3)
`B`	`unsigned char`	integer	1	(3)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	integer	2	(3)
`H`	`unsigned short`	integer	2	(3)
`i`	`int`	integer	4	(3)
`I`	`unsigned int`	integer	4	(3)
`l`	`long`	integer	4	(3)
`L`	`unsigned long`	integer	4	(3)
`q`	`long long`	integer	8	(2), (3)
`Q`	`unsigned long long`	integer	8	(2), (3)
`n`	`ssize_t`	integer		(4)
`N`	`size_t`	integer		(4)
`e`	(7)	float	2	(5)
`f`	`float`	float	4	(5)
`d`	`double`	float	8	(5)
`s`	`char[]`	bytes		
`p`	`char[]`	bytes		
`P`	`void *`	integer		(6)

Changed in version 3.3: Added support for the `'n'` and `'N'` formats.

Changed in version 3.6: Added support for the `'e'` format.

Notes:

1. 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.

2. 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.

3. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

   Changed in version 3.2: Use of the `__index__()` method for non-integers is new in 3.2.

4. The `'n'` and `'N'` conversion codes are only available for the native size (selected as the default or with the `'@'` byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

5. For the `'f'`, `'d'` and `'e'` conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for `'f'`, `'d'` or `'e'` respectively), regardless of the floating-point format used by the platform.

6. 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.

7. The IEEE 754 binary16 “half precision” type was introduced in the 2008 revision of the IEEE 754 standard. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately `6.1e-05` and `6.5e+04` at full precision. This type is not widely supported by C compilers: on a typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format for more information.

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 length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a sin-

gle 10-byte string, while `'10c'` means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

When packing a value `x` using one of the integer formats (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), if `x` is outside the valid range for that format then `struct.error` is raised.

Changed in version 3.1: In 3.0, some of the integer formats wrapped out-of-range values and raised `DeprecationWarning` instead of `struct.error`.

The `'p'` format character encodes a "Pascal string", meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. 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 bytes.

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.

7.1.2.3. Examples

> **Note:** All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

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

```
>>> record = b'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', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

The following format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

See also:

Module `array`
: Packed binary storage of homogeneous data.

Module `xdrlib`
: Packing and unpacking of XDR data.

7.1.3. Classes

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.

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 `size`.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

Identical to the `pack_into()` function, using the compiled format.

unpack(*buffer*)

Identical to the `unpack()` function, using the compiled format. The buffer's size in bytes must equal `size`.

unpack_from(*buffer*, *offset=0*)

Identical to the `unpack_from()` function, using the compiled format. The buffer's size in bytes, minus *offset*, must be at least `size`.

iter_unpack(*buffer*)

Identical to the `iter_unpack()` function, using the compiled format. The buffer's size in bytes must be a multiple of `size`.

New in version 3.4.

format

The format string used to construct this Struct object.

size

The calculated size of the struct (and hence of the bytes object produced by the `pack()` method) corresponding to `format`.