`!struct` --- Interpret bytes as packed binary data

- Functions and Exceptions
- Format Strings
  - Byte Order, Size, and Alignment
  - Type Codes
  - Examples
- Applications
  - Native Formats
  - Standard Formats
- Classes

-

from struct import *

struct

Source code: `Lib/struct.py`

pair: C; structures triple: packing; binary; data

This module converts between Python values and C structs represented as Python `bytes` objects. Compact `format strings <struct-format-strings>` describe the intended conversions to/from Python values. The module's functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

> [!NOTE] When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes

> which maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See `struct-alignment` for details.

Several `!struct` functions (and methods of `Struct`) take a *buffer* argument. This refers to objects that implement the `bufferobjects` 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.

Functions and Exceptions

The module defines the following exception and functions:

error

Exception raised on various occasions; argument is a string describing what is wrong.

pack(format, v1, v2, ...)

Return a bytes object containing the values *v1*, *v2*, ... packed according to the format string *format*. The arguments must match the values required by the format exactly.

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

Pack the values *v1*, *v2*, ... according to the format string *format* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument. A negative *offset* counts from the end of *buffer*.

unpack(format, buffer)

Unpack from the buffer *buffer* (presumably packed by `pack(format, ...)`) according to the format string *format*. 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`.

unpack_from(format, /, buffer, offset=0)

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

iter_unpack(format, buffer)

Iteratively unpack from the buffer *buffer* according to the format string *format*. 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.

3.4

calcsize(format)

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

Format Strings

Format strings describe the data layout when packing and unpacking data. They are built up from `type codes <type-codes>`, which specify the type of data being packed/unpacked. In addition, special characters control the `byte order, size and alignment<struct-alignment>`. Each format string consists of an optional prefix character which describes the overall properties of the data and one or more format characters which describe the actual data values and padding.

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). This behavior is chosen so that the bytes of a packed struct correspond exactly to the memory layout of the corresponding C struct. Whether to use native byte ordering and padding or standard formats depends on the application.

single: @ (at); in struct format strings single: = (equals); in struct format strings single: < (less); in struct format strings single: > (greater); in struct format strings single: ! (exclamation); in struct format strings

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

Character	Byte order	Size	Alignment
`=`	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.

> [!NOTE] The number 1023 (`0x3ff` in hexadecimal) has the following byte representations:
>
> - `03 ff` in big-endian (`>`)
> - `ff 03` in little-endian (`<`)
>
> Python example:
>
> > >>> import struct >>> struct.pack('>h', 1023) b'x03xff' >>> struct.pack('<h', 1023) b'xffx03'

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86, AMD64 (x86-64), and Apple M1 are little-endian; IBM z and many legacy architectures are big-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 type code; see the table in the `type-codes` section.

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

The form `'!'` represents the network byte order which is always big-endian as defined in IETF RFC 1700.

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 `struct-examples`.

Type Codes

Type codes (or format codes) 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		(7)
`c`	`char`	bytes of length 1	1	
`b`	`signed char`	int	1	(2)
`B`	`unsigned char`	int	1	(2)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	int	2	(2)
`H`	`unsigned short`	int	2	(2)
`i`	`int`	int	4	(2)
`I`	`unsigned int`	int	4	(2)
`l`	`long`	int	4	(2)
`L`	`unsigned long`	int	4	(2)
`q`	`long long`	int	8	(2)
`Q`	`unsigned long			
long`	int	8	(2)	
`n`	`ssize_t`	int		(2), (3)
`N`	`size_t`	int		(2), (3)
`e`	`_Float16`	float	2	(4), (6)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`Zf`	`float complex`	complex	8	(10)
`Zd`	`double complex`	complex	16	(10)

Format	C Type	Python type	Standard size	Notes
`s`	`char[]`	bytes		(9)
`p`	`char[]`	bytes		(8)
`P`	`void \*`	int		(2), (5)

3.3 Added support for the `'n'` and `'N'` formats.

3.6 Added support for the `'e'` format.

3.14 Added support for the `'F'` and `'D'` formats.

3.15 Added support for the `'Zf'` and `'Zd'` formats.

3.16 `'F'` and `'D'` formats are deprecated.

The `array` and `ctypes <ctypes-fundamental-data-types>` modules, as well as third-party modules like numpy, use similar -- but slightly different -- type codes.

Notes:

(1)

single: ? (question mark); in struct format strings

The `'?'` conversion code corresponds to the `_Bool` type defined by C standards since C99. In standard mode, it is represented by one byte.

(2)
When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `~object.__index__` method then that method is called to convert the argument to an integer before packing.

3.2 Added use of the `~object.__index__` method for non-integers.

(3)
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.

(4)
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.

(5)
The `'P'` type code 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.

(6)
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: it's available as `_Float16` type, if the compiler supports the Annex H of the C23 standard. 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.

(7)
When packing, `'x'` inserts one NUL byte.

(8)
The `'p'` type code 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 byte 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 byte 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'` type code consumes `count` bytes, but that the `!bytes` object returned can never contain more than 255 bytes. When packing, arguments of types `bytes` and `bytearray` are accepted.

(9)
For the `'s'` type code, the count is interpreted as the length of the byte string, not a repeat count like for the other type codes; for example, `'10s'` means a single 10-byte string mapping to or from a single Python byte string, while `'10c'` means 10 separate one byte character elements (e.g., `cccccccccc`) mapping to or from ten different Python byte objects. (See `struct-examples` for a concrete demonstration of the difference.) If a count is not given, it defaults to 1. For packing, the byte 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 byte string (while `'0c'` means 0 characters). When packing, arguments of types `bytes` and `bytearray` are accepted.

(10)
For the `'Zf'` and `'Zd'` type codes, the packed representation uses the IEEE 754 binary32 and binary64 format for components of the complex number, regardless of the floating-point format used by the platform. Note that complex types are available unconditionally, despite complex types being an optional feature in C. As specified in the C11 standard, each complex type is represented by a two-element C array containing, respectively, the real and imaginary parts. The `'F'`

and `'D'` (for `'Zf'` and `'Zd'`, respectively) format characters are supported for compatibility.

A type code 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.

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.

3.1 Previously, some of the integer formats wrapped out-of-range values and raised `DeprecationWarning` instead of `struct.error`.

single: ? (question mark); in struct format strings

For the `'?'` type code, 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.

Examples

> [!NOTE] Native byte order examples (designated by the `'@'` format prefix or lack of any prefix character) may not match what the reader's machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between `'s'` and `'c'` format characters:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

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 type codes may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine:

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

The following format `'llh0l'` results in two pad bytes being added at the end, assuming the platform's 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'
```

Module `array`
Packed binary storage of homogeneous data.

Module `json`
JSON encoder and decoder.

Module `pickle`
Python object serialization.

Applications

Two main applications for the `!struct` module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (`native formats<struct-native-formats>`), and data interchange between applications using agreed upon data layout (`standard formats<struct-standard-formats>`). Generally speaking, the format strings constructed for these two domains are distinct.

Native Formats

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat type code will be needed at the end of a format string to round up to the correct byte boundary for proper alignment of consecutive chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine):

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem:

```
>>> calcsize('@llh0l')
24
```

The `'x'` type code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

Standard Formats

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be `<` or `>` (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won't work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren't guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

Classes

The `!struct` module also defines the following type:

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 module-level functions with the same format since the format string is only compiled once.

> [!NOTE] The compiled versions of the most recent format strings passed to the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single `Struct` instance.

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, starting at position *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`.

3.4

format

The format string used to construct this Struct object.

3.7 The format string type is now `str` instead of `bytes`.

size

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

3.13 The *repr()* of structs has changed. It is now:

>>> Struct('i') Struct('i')

□□□ #2
□ □□□ □□□ 2026-10-31 17:19:54 UTC
□ □□□ □□□ 2026-10-31 17:19:59 UTC