struct — Binary Data Structures

Purpose: Convert between strings and binary data.

The `struct` module includes functions for converting between strings of bytes and native Python data types such as numbers and strings.

Functions versus Struct Class

A set of module-level functions is available for working with structured values, as well as the `Struct` class. Format specifiers are converted from their string format to a compiled representation, similar to the way regular expressions are handled. The conversion takes some resources, so it is typically more efficient to do it once when creating a `Struct` instance and call methods on the instance instead of using the module-level functions. All of the following examples use the `Struct` class.

Packing and Unpacking

Structs support *packing* data into strings, and *unpacking* data from strings using format specifiers made up of characters representing the type of the data and optional count and endianness indicators. Refer to the standard library documentation for a complete list of the supported format specifiers.

In this example, the specifier calls for an integer or long integer value, a two-byte string, and a floating-point number. The spaces in the format specifier are included to separate the type indicators, and are ignored when the format is compiled.

```
# struct_pack.py

import struct
import binascii

values = (1, 'ab'.encode('utf-8'), 2.7)
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)

print('Original values:', values)
print('Format string  :', s.format)
print('Uses           :', s.size, 'bytes')
print('Packed Value   :', binascii.hexlify(packed_data))
```

The example converts the packed value to a sequence of hex bytes for printing with `binascii.hexlify()`, since some of the characters are nulls.

```
$ python3 struct_pack.py

Original values: (1, b'ab', 2.7)
Format string  : I 2s f
Uses           : 12 bytes
Packed Value   : b'0100000061620000cdcc2c40'
```

Use `unpack()` to extract data from its packed representation.

```
# struct_unpack.py

import struct
import binascii

packed_data = binascii.unhexlify(b'0100000061620000cdcc2c40')

s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print('Unpacked Values:', unpacked_data)
```

Passing the packed value to `unpack()`, gives basically the same values back (note the discrepancy in the floating point value).

```
$ python3 struct_unpack.py

Unpacked Values: (1, b'ab', 2.700000047683716)
```

Endianness

By default, values are encoded using the native C library notion of *endianness*. It is easy to override that choice by providing an explicit endianness directive in the format string.

```
# struct_endianness.py
```

```
import struct
import binascii

values = (1, 'ab'.encode('utf-8'), 2.7)
print('Original values:', values)

endianness = [
    ('@', 'native, native'),
    ('=', 'native, standard'),
    ('<', 'little-endian'),
    ('>', 'big-endian'),
    ('!', 'network'),
]

for code, name in endianness:
    s = struct.Struct(code + ' I 2s f')
    packed_data = s.pack(*values)
    print()
    print('Format string  :', s.format, 'for', name)
    print('Uses           :', s.size, 'bytes')
    print('Packed Value   :', binascii.hexlify(packed_data))
    print('Unpacked Value :', s.unpack(packed_data))
```

the table below lists the byte order specifiers used by `Struct`.

Byte Order Specifiers for struct

Code	Meaning
@	Native order
=	Native standard
<	little-endian
>	big-endian
!	Network order

```
$ python3 struct_endianness.py

Original values: (1, b'ab', 2.7)

Format string  : @ I 2s f for native, native
Uses           : 12 bytes
Packed Value   : b'01000000061620000cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : = I 2s f for native, standard
Uses           : 10 bytes
Packed Value   : b'010000006162cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : < I 2s f for little-endian
Uses           : 10 bytes
Packed Value   : b'010000006162cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : > I 2s f for big-endian
```

```
Uses           : 10 bytes
Packed Value   : b'000000016162402ccccd'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : ! I 2s f for network
Uses           : 10 bytes
Packed Value   : b'000000016162402ccccd'
Unpacked Value : (1, b'ab', 2.700000047683716)
```

Buffers

Working with binary packed data is typically reserved for performance-sensitive situations or passing data into and out of extension modules. These cases can be optimized by avoiding the overhead of allocating a new buffer for each packed structure. The `pack_into()` and `unpack_from()` methods support writing to pre-allocated buffers directly.

```
# struct_buffers.py

import array
import binascii
import ctypes
import struct

s = struct.Struct('I 2s f')
values = (1, 'ab'.encode('utf-8'), 2.7)
print('Original:', values)

print()
print('ctypes string buffer')

b = ctypes.create_string_buffer(s.size)
print('Before  :', binascii.hexlify(b.raw))
s.pack_into(b, 0, *values)
print('After   :', binascii.hexlify(b.raw))
print('Unpacked:', s.unpack_from(b, 0))

print()
print('array')

a = array.array('b', b'\0' * s.size)
print('Before  :', binascii.hexlify(a))
s.pack_into(a, 0, *values)
print('After   :', binascii.hexlify(a))
print('Unpacked:', s.unpack_from(a, 0))
```

The `size` attribute of the `Struct` tells us how big the buffer needs to be.

```
$ python3 struct_buffers.py

Original: (1, b'ab', 2.7)

ctypes string buffer
Before  : b'000000000000000000000000'
After   : b'0100000061620000cdcc2c40'
Unpacked: (1, b'ab', 2.700000047683716)

array
Before  : b'000000000000000000000000'
After   : b'0100000061620000cdcc2c40'
Unpacked: (1, b'ab', 2.700000047683716)
```

See also

- Standard library documentation for struct
- Python 2 to 3 porting notes for struct
- `array` – The `array` module, for working with sequences of fixed-type values.
- `binascii` – The `binascii` module, for producing ASCII representations of binary data.
- WikiPedia: Endianness – Explanation of byte order and endianness in encoding.

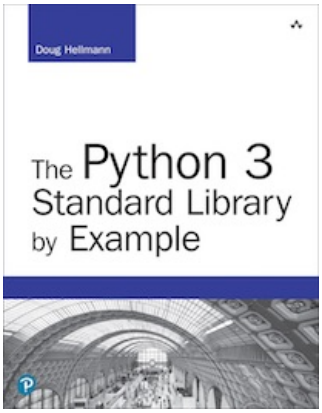
Quick Links

Functions versus Struct Class
Packing and Unpacking
Endianness
Buffers

This page was last updated 2018-12-09.

Navigation

queue — Thread-Safe FIFO Implementation
weakref — Impermanent References to Objects



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example