`struct` — Interpreta bytes como paquetes de datos binarios

Código fuente: Lib/struct.py

Este módulo realiza conversiones entre valores de Python y estructuras C representadas como objetos `bytes`de Python. Se puede utilizar para el tratamiento de datos binarios almacenados en archivos o desde conexiones de red, entre otras fuentes. Utiliza Cadenas de Formato como descripciones compactas del diseño de las estructuras C y la conversión prevista a/desde valores de Python.

Nota Por defecto, el resultado de empaquetar una estructura C determinada incluye bytes de relleno para mantener la alineación adecuada para los tipos correspondientes en C; del mismo modo, la alineación se tiene en cuenta al desempaquetar. Este comportamiento se elige para que los bytes de una estructura empaquetada se correspondan exactamente con el diseño en memoria de la estructura C correspondiente. Para tratar formatos de datos que sean independientes de la plataforma u omitir bytes de relleno implícitos, utiliza el tamaño y la alineación `estándar` en lugar del `nativo`: ver Orden de Bytes, Tamaño y Alineaciónpara obtener más información.

Varias funciones `struct` (y métodos de `Struct`) toman un argumento *buffer*. Esto hace referencia a los objetos que implementan Protocolo Búfer y proporcionan un búfer de lectura o de lectura/escritura. Los tipos más comunes utilizados para ese propósito son `bytes` y `bytearray`, pero muchos otros tipos que se pueden ver como una lista de bytes implementan el protocolo de búfer, para que se puedan leer/rellenar sin necesidad de copiar a partir de un objeto `bytes`.

Funciones y Excepciones

El módulo define la siguiente excepción y funciones:

exception `struct.`**`error`**
Excepción lanzada en varias ocasiones; el argumento es una *string* que describe lo que está mal.

`struct.`**`pack`**(*`format`*, *`v1`*, *`v2`*, `...`)
Retorna un objeto bytes que contiene los valores *v1*, *v2*, … empaquetado de acuerdo con la cadena de formato *format*. Los argumentos deben coincidir exactamente con los valores requeridos por el formato.

`struct.`**`pack_into`**(*`format`*, *`buffer`*, *`offset`*, *`v1`*, *`v2`*, `...`)
Empaqueta los valores *v1*, *v2*, … de acuerdo con la cadena de formato *format* y escribe los bytes empaquetados en el búfer de escritura *buffer* comenzando en la posición *offset*. Nota: *offset* es un argumento obligatorio.

`struct.`**`unpack`**(*`format`*, *`buffer`*)
Desempaqueta del búfer *buffer* (normalmente empaquetado por `pack(format, ...)`) según la cadena de formato *format*. El resultado es una tupla incluso si contiene un solo elemento. El tamaño del búfer en bytes debe coincidir con el tamaño requerido por el formato, como se refleja en `calcsize()`.

`struct.`**`unpack_from`**(*`format`*, *`/`*, *`buffer`*, *`offset=0`*)
Desempaqueta del *buffer* comenzando en la posición *offset*, según la cadena de formato *format*. El resultado es una tupla incluso si contiene un solo elemento. El tamaño del búfer en bytes, comenzando en la posición *offset*, debe tener al menos el tamaño requerido por el formato, como se refleja en `calcsize()`.

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

Desempaqueta de manera iterativa desde el búfer *buffer* según la cadena de formato *format*. Esta función retorna un iterador que leerá fragmentos de igual tamaño desde el búfer hasta que se haya consumido todo su contenido. El tamaño del búfer en bytes debe ser un múltiplo del tamaño requerido por el formato, como se refleja en `calcsize()`.

Cada iteración produce una tupla según lo especificado por la cadena de formato.

Nuevo en la versión 3.4.

`struct.`**`calcsize`**(*`format`*)

Retorna el tamaño de la estructura (y, por lo tanto, del objeto bytes generado por `pack(format, ...)`) correspondiente a la cadena de formato *format*.

Cadenas de Formato

Las cadenas de formato son el mecanismo utilizado para especificar el diseño esperado al empaquetar y desempaquetar datos. Se crean a partir de Caracteres de formato, que especifican el tipo de datos que se empaquetan/desempaquetan. Además, hay caracteres especiales para controlar Orden de Bytes, Tamaño y Alineación.

Orden de Bytes, Tamaño y Alineación

Por defecto, los tipos C se representan en el formato nativo y el orden de bytes de la máquina, y se alinean correctamente omitiendo bytes de relleno si es necesario (según las reglas utilizadas por el compilador de C).

Como alternativa, el primer carácter de la cadena de formato se puede utilizar para indicar el orden de bytes, el tamaño y la alineación de los datos empaquetados, según la tabla siguiente:

Carácter	Orden de Bytes	Tamaño	Alineamiento
`@`	nativo	nativo	nativo
`=`	nativo	standard	none
`<`	little-endian	standard	none
`>`	big-endian	standard	none
`!`	red (= big-endian)	standard	none

Si el primer carácter no es uno de estos, se asume `'@'`.

El orden de bytes nativo es big-endian o little-endian, dependiendo del sistema host. Por ejemplo, Intel x86 y AMD64 (x86-64) son little-endian; Motorola 68000 y PowerPC G5 son big-endian; ARM e Intel *Itanium* tienen la propiedad de trabajar con ambos formatos (middle-endian). Utiliza `sys.byteorder` para comprobar la *endianness* («extremidad») de su sistema.

El tamaño y la alineación nativos se determinan mediante la expresión `sizeof` del compilador de C. Esto siempre se combina con el orden de bytes nativo.

El tamaño estándar depende únicamente del carácter de formato; ver la tabla en la sección Caracteres de formato.

Nótese la diferencia entre `'@'` y `'='` : ambos utilizan el orden de bytes nativo, pero el tamaño y la alineación de este último está estandarizado.

La forma `'!'` representa el orden de bytes en la red, el cuál siempre es big-endian tal como se define en IETF RFC 1700.

No hay manera de indicar el orden de bytes no nativo (forzar el intercambio de bytes); utiliza la elección adecuada de `'<'` o `'>'`.

Notas:

1. El relleno solo se agrega automáticamente entre los miembros sucesivos de la estructura. No se agrega ningún relleno al principio o al final de la estructura codificada.
2. No se añade ningún relleno cuando se utiliza el tamaño y la alineación no nativos, por ejemplo, con “<”, “>”, “=” y “!”.
3. Para alinear el final de una estructura con el requisito de alineación de un tipo determinado, termina el formato con el código de ese tipo con un conteo repetido de ceros. Véase Ejemplos.

Caracteres de formato

Los caracteres de formato tienen el siguiente significado; la conversión entre los valores de C y Python debe ser obvia dados sus tipos. La columna “Tamaño estándar” hace referencia al tamaño del valor empaquetado en bytes cuando se utiliza el tamaño estándar; es decir, cuando la cadena de formato comienza con uno de `'<'`, `'>'`, `'!'` o `'='`. Cuando se utiliza el tamaño nativo, el tamaño del valor empaquetado depende de la plataforma.

Formato	Tipo C	Tipo Python	Tamaño estándar	Notas
`x`	byte de relleno	sin valor		
`c`	char	bytes de longitud 1	1	
`b`	signed char	integer	1	(1), (2)
`B`	unsigned char	integer	1	(2)
(2)	_Bool	bool	1	(1)
`h`	short	integer	2	(2)
`H`	unsigned short	integer	2	(2)
`i`	int	integer	4	(2)
`I`	unsigned int	integer	4	(2)
`l`	long	integer	4	(2)
`L`	unsigned long	integer	4	(2)
`q`	long long	integer	8	(2)
`Q`	unsigned long long	integer	8	(2)
`n`	`ssize_t`	integer		(3)
`N`	`size_t`	integer		(3)
`e`	(6)	float	2	(4)
`f`	float	float	4	(4)
`d`	double	float	8	(4)

Formato	Tipo C	Tipo Python	Tamaño estándar	Notas
`s`	char[]	bytes		
`p`	char[]	bytes		
`P`	void*	integer		(5)

Distinto en la versión 3.3: Soporte añadido para los formatos `'n'` y `'N'`.

Distinto en la versión 3.6: Soporte añadido para el formato `'e'`.

Notas:

1. El código de conversión `'?'` corresponde al tipo _Bool definido por C99. Si este tipo no está disponible, se simula mediante un char. En el modo estándar, siempre se representa mediante un byte.

2. Al intentar empaquetar un no entero mediante cualquiera de los códigos de conversión de enteros, si el no entero tiene un método `__index__()`, se llama a ese método para convertir el argumento en un entero antes de empaquetar.

   Distinto en la versión 3.2: Agregado el uso del método `__index__()` para los no enteros.

3. Los códigos de conversión `'n'` y `'N'` solo están disponibles para el tamaño nativo (seleccionado como predeterminado o con el carácter de orden de bytes `'@'`). Para el tamaño estándar, puedes usar cualquiera de los otros formatos enteros que se ajusten a tu aplicación.

4. Para los códigos de conversión `'f'`, `'d'` y `'e'`, la representación empaquetada utiliza el formato IEEE 754 binary32, binary64 o binary16 (para `'f'`, `'d'` o `'e'` respectivamente), independientemente del formato de punto flotante utilizado por la plataforma.

5. El carácter de formato `'P'` solo está disponible para el orden nativo de bytes (seleccionado como predeterminado o con el carácter de orden de bytes `'@'`). El carácter de orden de bytes `'='` elige utilizar el orden little- o big-endian basado en el sistema host. El módulo *struct* no interpreta esto como un orden nativo, por lo que el formato `'P'` no está disponible.

6. El tipo IEEE 754 binary16 «half precision» se introdujo en la revisión de 2008 del IEEE 754 estándar. Tiene un bit de signo, un exponente de 5 bits y una precisión de 11 bits (con 10 bits almacenados explícitamente) y puede representar números entre aproximadamente `6.1e-05` y `6.5e+04` con total precisión. Este tipo no es ampliamente compatible con los compiladores de C: en un equipo típico, un *unsigned* short se puede usar para el almacenamiento, pero no para las operaciones matemáticas. Consulte la página de Wikipedia en el formato de punto flotante de media precisión para obtener más información.

Un carácter de formato puede ir precedido de un número de recuento que repite tantas veces el carácter. Por ejemplo, la cadena de formato `'4h'` significa exactamente lo mismo que `'hhhh'`.

Se omiten los caracteres de espacio entre formatos; sin embargo, un recuento y su formato no deben contener espacios en blanco.

Para el carácter de formato `'s'`, el recuento se interpreta como la longitud de los bytes, no un recuento de repetición como para los otros caracteres de formato; por ejemplo, `'10s'` significa una sola cadena de 10 bytes, mientras que `'10c'` significa 10 caracteres. Si no se da un recuento, el valor predeterminado es 1. Para el empaquetado, la cadena es truncada o rellenada con bytes nulos según corresponda para que se ajuste. Para desempaquetar, el objeto bytes resultante siempre tiene exactamente el número especificado de bytes. Como caso especial, `'0s'` significa una sola cadena vacía (mientras que `'0c'` significa 0 caracteres).

Al empaquetar un valor `x` utilizando uno de los formatos enteros (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), si `x` está fuera de un rango válido para ese formato, entonces se lanza la excepción `struct.error`.

Distinto en la versión 3.1: Anteriormente, algunos de los formatos enteros ajustaban los valores fuera de rango y lanzaban `DeprecationWarning` en vez de `struct.error`.

El carácter de formato `'p'` codifica una «cadena de Pascal», lo que significa una cadena de longitud variable corta almacenada en un número *fijo de bytes*, dado por el recuento. El primer byte almacenado es el valor mínimo entre la longitud de la cadena o 255. A continuación se encuentran los bytes de la cadena. Si la cadena pasada a `pack()` es demasiado larga (más larga que la cuenta menos 1), solo se almacenan los bytes iniciales `count-1` de la cadena. Si la cadena es más corta que `count-1`, se rellena con bytes nulos para que se utilicen exactamente los bytes de recuento en total. Tenga en cuenta que para `unpack()`, el carácter de formato `'p'` consume bytes `count`, pero que la cadena retornada nunca puede contener más de 255 bytes.

Para el carácter de formato `'?'`, el valor retornado es `True` o `False`. Al empaquetar, se utiliza el valor verdadero del objeto del argumento. Se empaquetará 0 o 1 en la representación *bool* nativa o estándar, y cualquier valor distinto de cero será `True` al desempaquetar.

Ejemplos

Nota Todos los ejemplos asumen un orden de bytes, tamaño y alineación nativos con una máquina big-endian.

Un ejemplo básico de empaquetado/desempaquetado de tres enteros:

```
>>> 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
```

Los campos desempaquetados se pueden nombrar asignándolos a variables o ajustando el resultado en una tupla con nombre:

```
>>> 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)
```

El orden de los caracteres de formato puede tener un impacto en el tamaño ya que el relleno necesario para satisfacer los requisitos de alineación es diferente:

```
>>> 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
```

El siguiente formato `'llh0l'` especifica dos bytes de relleno al final, suponiendo que los tipos *longs* están alineados en los límites de 4 bytes:

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

Esto solo funciona cuando el tamaño y la alineación nativos tienen efecto; el tamaño estándar y la alineación no imponen ninguna alineación.

Ver también

Módulo `array`
Almacenamiento binario empaquetado de datos homogéneos.

Módulo `xdrlib`
Empaquetar y desempaquetar datos XDR.

Clases

El módulo `struct` también define el siguiente tipo:

class `struct.`**`Struct`**(*format*)

Retorna un nuevo objeto Struct que escribe y lee datos binarios según la cadena de formato *format*. Crear un objeto Struct una vez y llamar a sus métodos es más eficaz que llamar a las funciones `struct`con el mismo formato, ya que la cadena de formato solo se compila una vez en ese caso.

Nota Las versiones compiladas de las cadenas de formato más recientes pasadas a `Struct` y las funciones de nivel de módulo se almacenan en caché, por lo que los programas que utilizan solo unas pocas cadenas de formato no necesitan preocuparse por volver a usar una sola instancia `Struct`.

Los objetos Struct compilados admiten los siguientes métodos y atributos:

`pack`(*v1*, *v2*, ...)

Idéntico a la función `pack()`, utilizando el formato compilado. (`len(result)` será igual a `size`.)

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

Idéntico a la función `pack_into()`, utilizando el formato compilado.

`unpack`(*buffer*)

Idéntico a la función `unpack()`, utilizando el formato compilado. El tamaño del búfer en bytes debe ser igual a `size`.

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

Idéntico a la función `unpack_from()`, utilizando el formato compilado. El tamaño del búfer en bytes, comenzando en la posición *offset*, debe ser al menos `size`.

`iter_unpack`(*buffer*)

Idéntico a la función `iter_unpack()`, utilizando el formato compilado. El tamaño del búfer en bytes debe ser un múltiplo de `size`.

Nuevo en la versión 3.4.

`format`

Cadena de formato utilizada para construir este objeto Struct.

Distinto en la versión 3.7: El tipo de cadena de formato es ahora `str` en lugar de `bytes`.

`size`

El tamaño calculado de la estructura (y, por lo tanto, del objeto bytes generado por el método `pack()`) correspondiente a `format`.