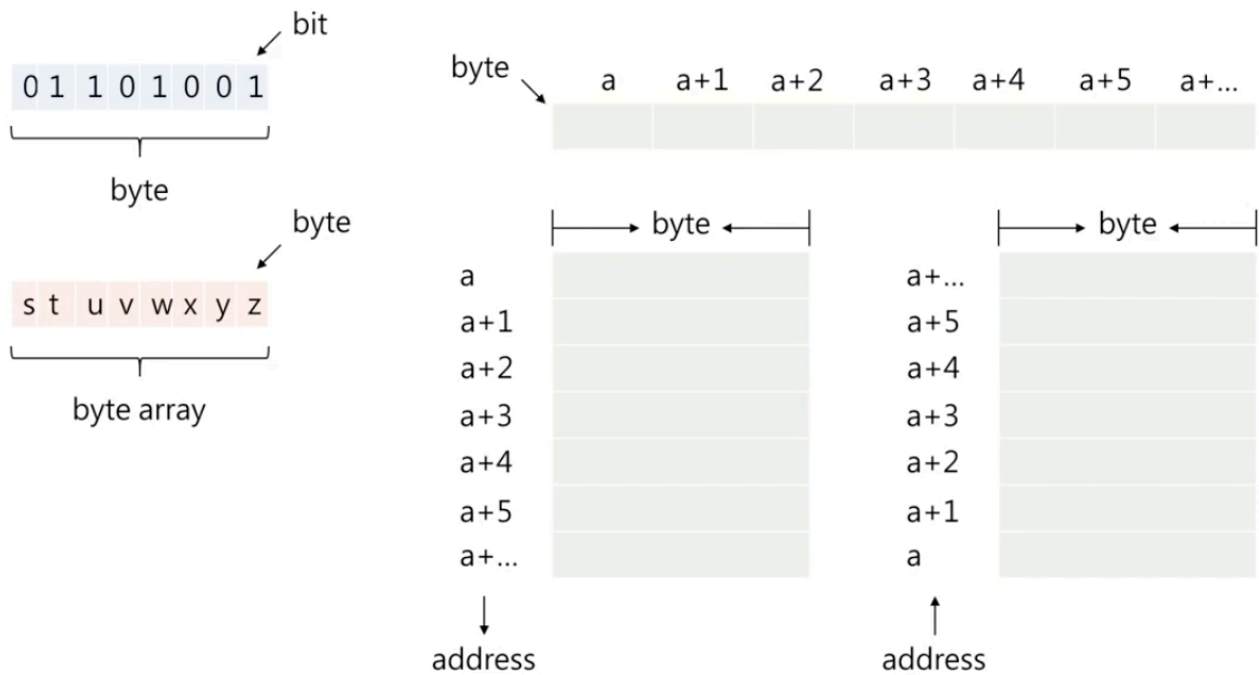


Big Endian, Little Endian and Network Byte Order

人類對於儲存數位資料方法非常情緒化的一個爭執 endianness



一般 computer science 的人在描述資料時，都是以一個一個的 bit 或是一個一個的 byte 為單位
1 byte = 8 bits

在記憶體中儲存的單位基本上也是以一個一個的 byte 為單位

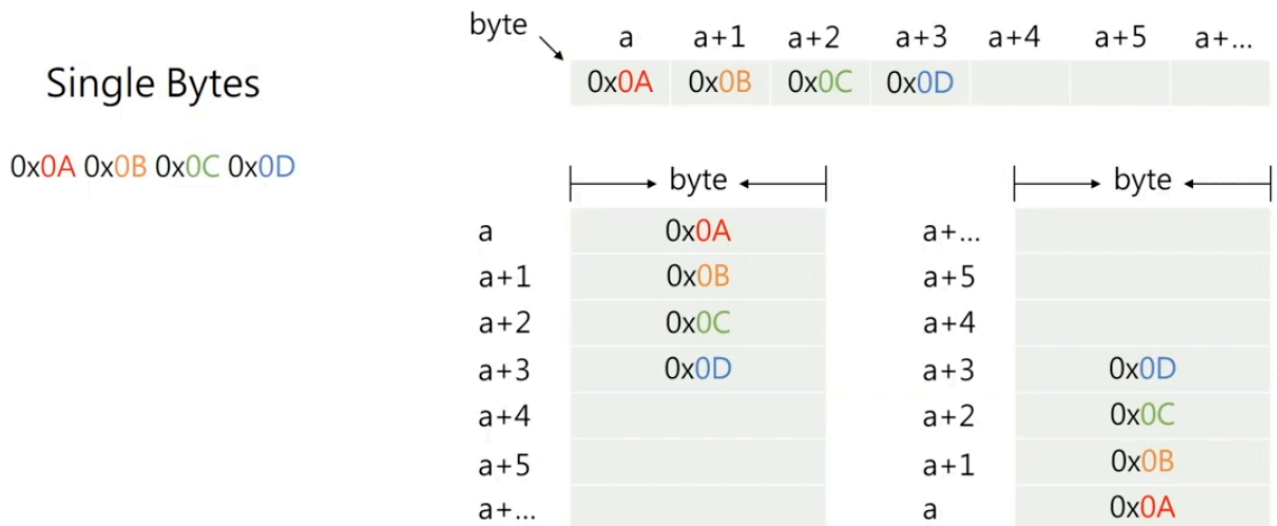
我們先把記憶體畫成連續的一格一格，每一格的寬度是一個 byte，然後我們按照記憶體位址 (address) 由小到大由左到右編排，也可以由上到下編排，或由下到上

記憶體位址由上到下，符合人的書寫習慣

記憶體位址由下到上，符合零位址比較小（低），而大位址比較高的邏輯

Big Endian, Little Endian and Network Byte Order

Single Bytes



如果我們有四個 1-byte 資料要儲存，假設這四個 1-byte 資料分別為 0x0A 0x0B 0x0C 0x0D

不要忘記一個 byte 有 8 個 bits，而一個 16 (2^4) 進位的位數只能表示 4 bits

所以一個 byte 在 16 進位來說會有兩個位數

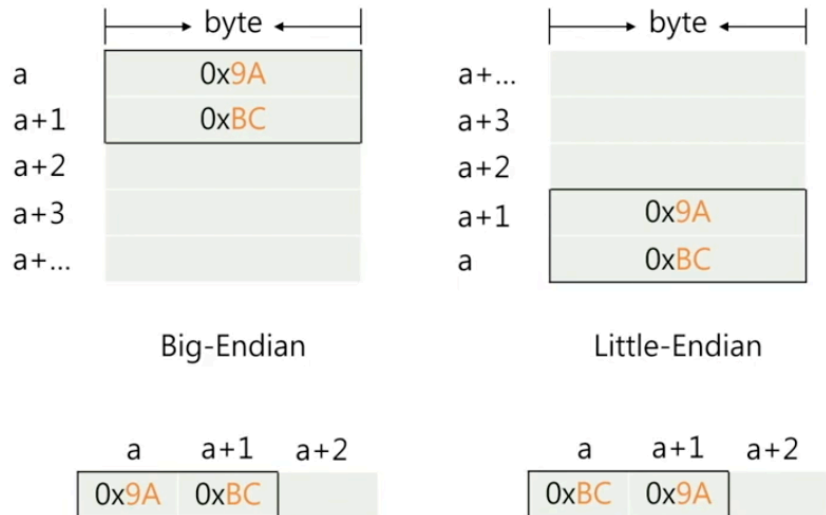
小的位址來放置 0x0A，大的位址放置 0x0D，這是一個一個分開的 byte 置放的方式

Multiple Byte

4-byte integer:
0x12 0x34 0x56 0x78

2-byte short integer:
0x9A 0xBC

1-byte char or boolean:
0xCD or 0x01



但是我們在寫程式的時候，常常會用到更長的 bytes 來儲存變數的值

Ex. 一個 integer 通常是 4-byte (32-bit)、一個 short integer 通常是 2-byte (16-bit)、一個 char 或是 boolean 通常是 1-byte

我們用一個 short 為例子：

若是其值為 0x9A 0xBC，那麼根據記憶體位址方向的不同，擺放在記憶體的方法的順序就有兩種
要注意在每一個 byte 裡面的順序仍舊是一樣的，但是這兩個 byte 的順序在這兩種方法裡面不同

1. Big endian 的方法會把 most significant 的 byte，0x9A 會放在記憶體位址小的地方
2. Little endian 的方法 把 least significant 的 byte，放在記憶體位址小的地方，也就是把 0xBC 放在位址小的地方

單一 byte 裡面的值是按順序的，0x9A 並沒有變成 0xA9

Big endian 是按人閱讀的順序在排列的，而 little endian 的順序卻比較有邏輯

因為 most significant byte (0x9A) 的位址比較大，而 least significant byte (0xBC) 的位址比較小
所以當記憶體位址 +1 的時候，就是往資料位數較高的地方走

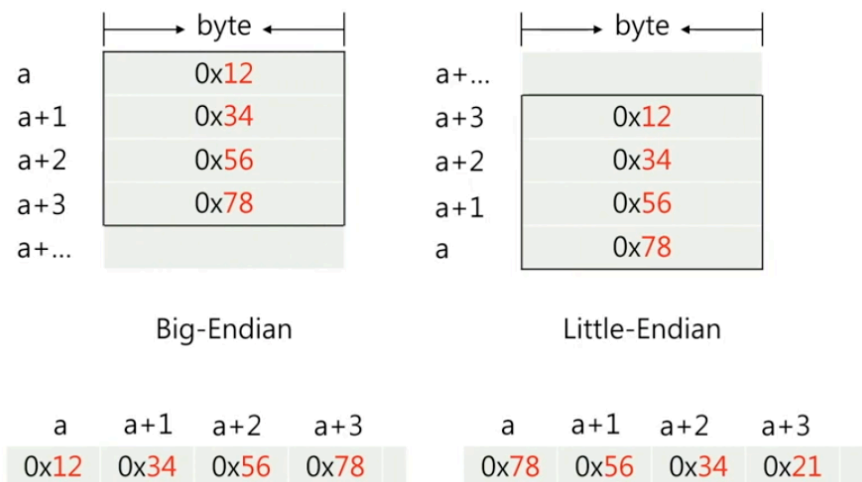
再來看一個 32-bit (4-byte) 的整數：

Multiple Byte

4-byte integer:
0x12 0x34 0x56 0x78

2-byte short integer:
0x9A 0xBC

1-byte char or boolean:
0xCD or 0x01

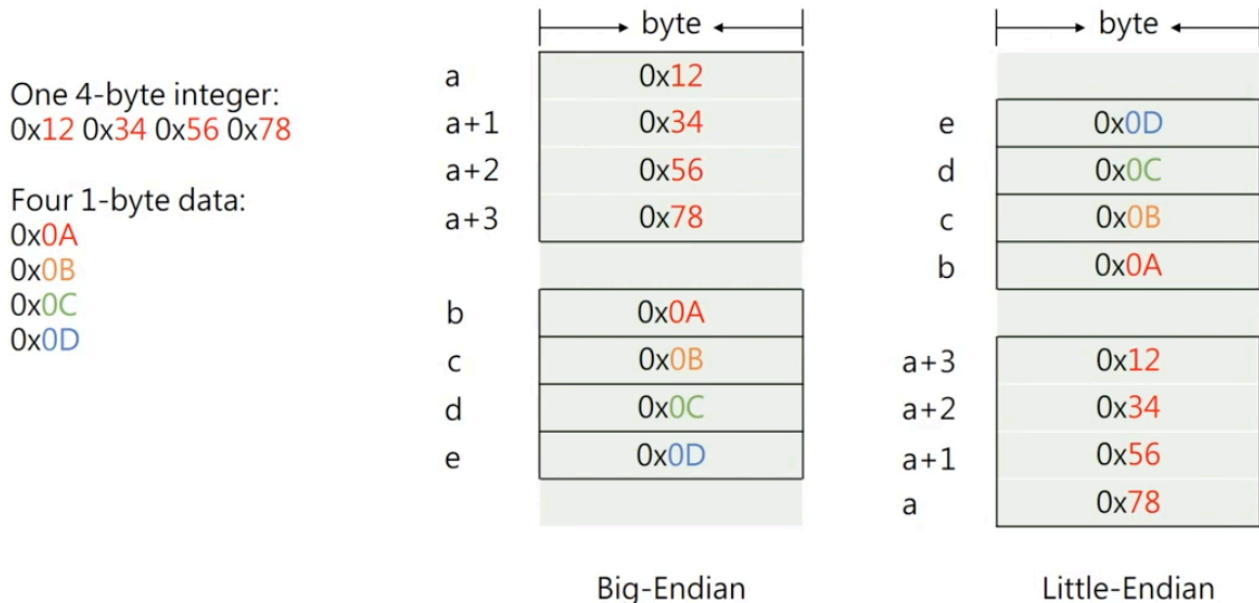


Big Endian, Little Endian and Network Byte Order

這個數字在記憶體裡面擺放的方式分別是：左邊 big endian 和右邊 little endian

這邊特別要強調的是，我們這裡說的是一次擺放一個 4-byte 的數值進入到記憶體中，這和分四次擺放各 1-byte 的數值到記憶體是不一樣的，只有一次擺放 n-byte 的值到記憶體中才有 endianness 的問題

這裡畫兩張圖分別是 big-endian 和 little endian：



a 位址是一次 4-byte 的資料放一起，這就會有 endian 的差別

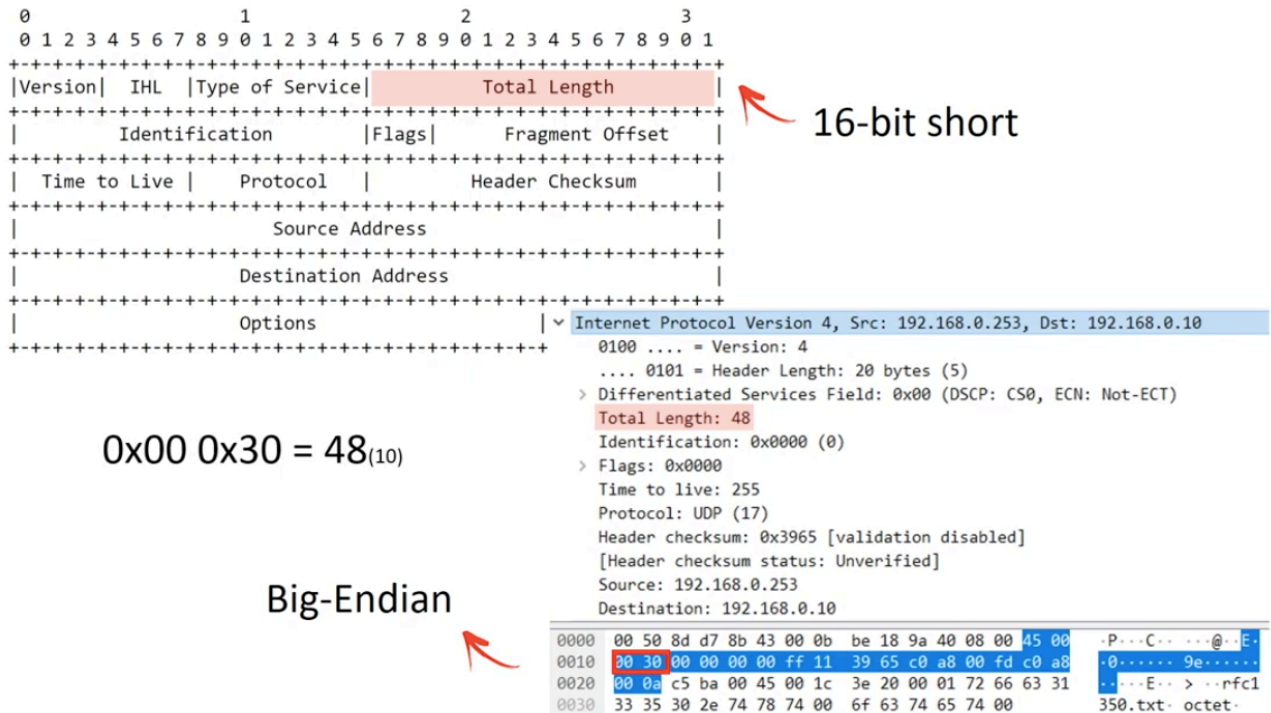
big-endian 的 a 位址是放 0x12，但是 little-endian 的 a 位址是 0x 78

如果是從位址 b 開始，分四次放 1-byte 的資料，那麼兩個 endian 就沒有差別了

這裡我用不同的實線框起每一個變數，我們每一次只放一個變數進去記憶體

Big Endian, Little Endian and Network Byte Order

網路封包的 format :



舉例來說 IP 的 header 長這樣，可以看到 total length 這一個欄位是 16-bit，因此應該是一個 short 的整數，十進位的實際值是 48，而 16 進位的封包內容看到這欄位是 0x00 0x30

我們發現這閱讀順序，和整數的表示方式是一致的，所以網路封包使用的 endian 順序應該是 big-endian，我們在網路上也會特地把這個順序叫做 network byte order

但是不同的 CPU 和硬體架構上，使用 big-endian 還是 little-endian 是爭論不休的事情

電腦常使用的 Intel CPU 是使用 little-endian

早期的麥金塔電腦使用 IBM 或 Motorola 的 CPU 是使用 big-endian

手機使用的 ARM CPU 早期是 little-endian

現在則是可以 config 成任一種 endian

由於網路封包一定是 big-endian，但是 CPU 卻不一定是哪一種，所以我們在寫封包程式的時候要特別小心

Big Endian, Little Endian and Network Byte Order

在 Python 環境下，怎麼製作出 network byte order 的封包？

1. Python 有一個內建的 module 叫做 struct
2. struct 可以幫助我們建立按我們意志順序做出來的 binary data
3. 通常這個 binary data 的 format 我們會按照 protocol message format 製作

```
struct.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.

```
struct.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 `calcsizes()`.

Character	Byte order
<	little-endian
>	big-endian
!	network (= big-endian)

```
import struct
total_length = 1
data = struct.pack('<H', total_length)
print(data)
```

b'\x01\x00'

這個 struct module 有兩個重要的 functions

1. `struct.pack(format, v1, v2, ...)`：將變數 *v1*, *v2*, ... 的值，用 *format* 的形式排列出來
2. `struct.unpack(format, buffer)`：將變數 *buffer* 的值，用 *format* 的形式解讀出來

這裡的 buffer 通常是一串 binary data，這兩個 functions 的 format 都是一個字串

使用方式如下：

1. 這個 format 字串裡，首先你會用一個字元指定你要用 big-endian (>) 或是 little-endian (<) 作為 byte 的排列順序，也可以使用 network (!) order 不過這同等於 big-endian (範例使用 little-endian (<))
2. 再來要參考一個表格，每一個英文字表示擺放一個某種型態的變數

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

```
import struct
total_length = 1
data = struct.pack('<H', total_length)
print(data)
```

b'\x01\x00'

```
import struct
total_length = 48
data = struct.pack('!H', total_length)
```

```
import binascii

print(binascii.hexlify(data))
print(struct.calcsizes('!H'))
```

b'0030'


例子：

1. 首先引入了 struct module
2. 然後指定了一個變數 total_length 為 1
3. 並且使用 little-endian (<) 排出一個 unsigned short integer (H)，也就是 16-bit，並把產生的 binary 資料放在 data 變數中
4. data 印出來：0x01 0x00 也就是數字 1 在使用 little endian 排列所製作出來的 binary data，若是使用 big endian 的話 則是會輸出 0x00 0x01

試試看製作剛剛 ip 封包的 total length：

1. format 使用的是 network (!) 以及一個 16-bit 的 short (H)
2. 不過 binary 的資料有時候印出來不好看，所以我們再使用 Python 的一個 binascii module
3. 把 data 轉成 hexadecimal 的形式印出來，也就是 0x00 0x30
4. 我們可以看到這個值和順序的確跟封包上的一樣

再來看一個例子：



```
data = struct.pack('!HHIxxxx', 127, 5000, 8888)
print(binascii.hexlify(data))

b'007f1388000022b800000000'
   127 5000 8888

ans = struct.unpack('!HHIxxxx', data)
print(ans)

(127, 5000, 8888)
```

1. format 的 '!' 表示要使用 network order
2. 接下來是兩個 16-bit 的 unsigned short integer (H)，分別帶入 127 和 5000 的數字
3. 接下來是一個 32-bit 的 unsigned int (I)，帶入 8888 的數字
4. 最後有四個 x，表示會有四個 0x00 的 padding byte 接在後面

同樣的當我們接受到別人從網路上傳來的 data 之後

我們可以利用 struct.unpack 的 function 把 data 一個一個按照 format 的形式解開
就會得到原先的內容了

Python 會把根據 format 裡英文字數量一個一個解析資料

字串的使用：

根據查表，字串是 's' 這個標記

所以我們若是要組合一個字串排列到封包裡面

你可能會想說要這樣寫 code 應該是正確的，但是實際上這樣是錯誤的

在這裡 's' 其實只表示要輸出一個字元，只有後面的第一個字元 'A' 會被轉換

```
data = struct.pack('!s', b'ABCDEFGH')
print(binascii.hexlify(data))
```

b'41'

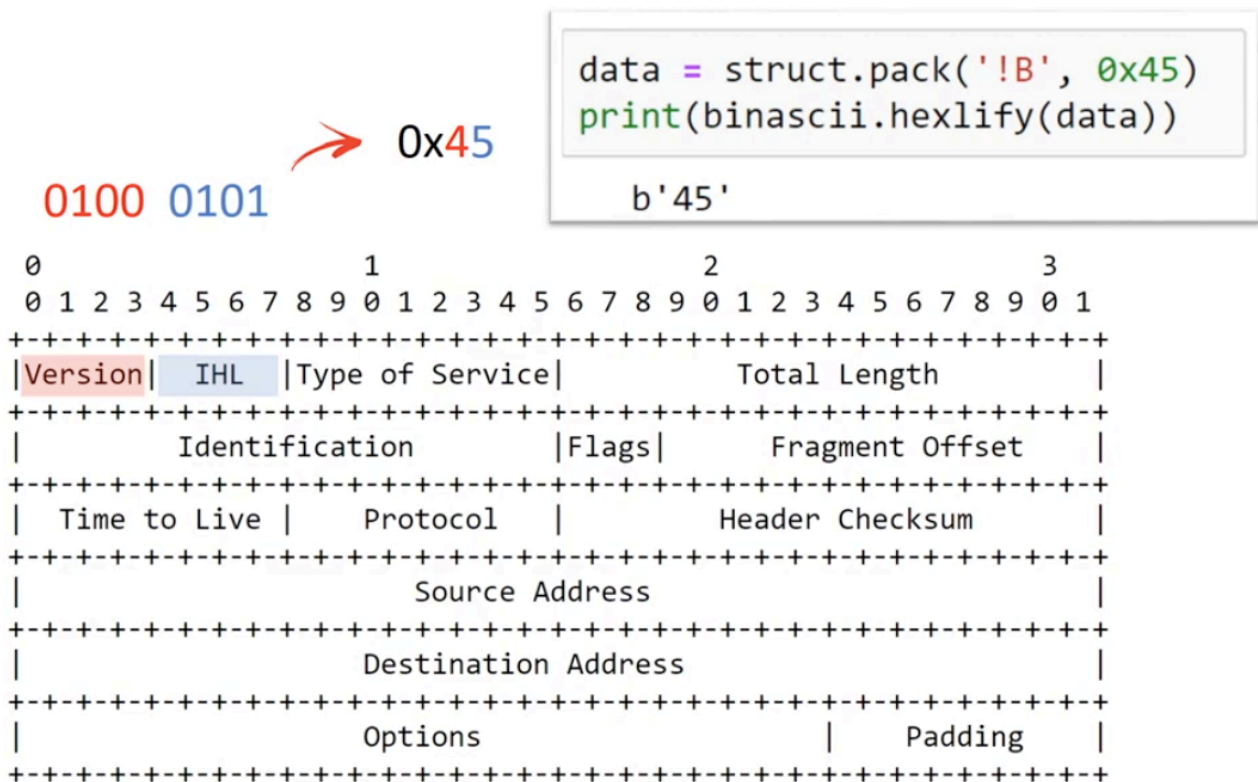
```
data = struct.pack('!8s', b'ABCDEFGH')
print(binascii.hexlify(data))
```

b'4142434445464700'

↙
In ASCII table, 'A' is 0x41.

如果要正確的輸出 8 字元的 'ABCDEFGH'，要寫 '8s' 這樣答案就會正確了

長度小於 8-bits 的資料：



我們在 layer 7 protocol 的設計裡面，很少會使用到小於一個 byte 資料 field 的設計
但是我們知道在 TCP 和 IP 裡面，有部分的欄位是小於一個 byte 的

舉例來說 IP 的第一行就有 4-bit 的 version 和 4-bit 的 IHL (header length)，
通常這兩個值是 0100 0101，也就是十六進位的 0x45

這時候你可以自己做計算把 0x45 硬寫進去成一個 unsigned char (B)，不過就要考驗你的數學計算能力了

Python 內建的 module 中並沒有一個方便的方式做這樣的轉換，你可以尋求一些第三方的套件例如 `bitstring` 或是 `bitarray`