Capítulo 17

Serialización

Al terminar este capítulo, entenderás:

- Qué es la serialización y la des-serialización.
- Por qué es importante definir un formato de datos común.
- Cómo es la serialización binaria de datos multibyte y texto.
- Cómo utilizar `struct` para manejar datos binarios simples de tipos diferentes, como por ejemplo una cabecera de mensaje binario.

Empezamos este capítulo con una afirmación que podría aparecer en un curso de informática para niños: «La única forma en la que un computador digital procesa y almacena datos es mediante código binario». Ya, lo tenemos claro, pero eso también tiene consecuencias cuando esos datos tienen que viajar por la red.

El problema es que, salvo en unas pocas excepciones, el binario rara vez resulta suficientemente expresivo o amigable para representar información útil para las personas. Por ese motivo se utilizan distintas formas de interpretar el código binario en función del tipo de dato que se necesita: enteros, decimales, cadenas de caracteres, fechas, etc. Lo importante es recordar que, sea cual sea su representación en un lenguaje de programación de alto nivel, en la memoria o registros del computador, todo dato es a fin de cuentas una secuencia de bits.

La **serialización** es el proceso que transforma los datos que manejan los programas (enteros, cadenas, imágenes, etc.) en secuencias de bytes susceptibles de ser almacenadas en un archivo o enviadas a través de la red. La des-serialización es el proceso inverso. Existen dos tipos de serialización: binaria y textual. La binaria codifica los datos como secuencias de bytes, mientras que la textual utiliza marcado legible por humanos[1], con identificadores, espacios y comillas.

[1] El significado del adjetivo «legible» es bastante debatible según qué protocolo.

El siguiente listado es un ejemplo de serialización binaria (con pickle) y textual (con JSON) de los mismos datos:

Binaria

```
>>> import pickle
>>> data = {'width': 10, 'height': 20}
>>> pickle.dumps(data)
b'\x80\x04\x95\x1a\x00\x00\x00\x00...'
```

Textual

```
>>> import json
>>> data = {'width': 10, 'height': 20}
>>> json.dumps(data)
'{"width": 10, "height": 20}'
```

FIGURA 17.1: Serialización binaria y textual

Por supuesto existen muchos formatos o sistemas de codificación tanto binaria como textual. En este capítulo veremos algunos de los más sencillos para ilustrar el concepto con mínima complejidad.

Es buen momento para aclarar una confusión habitual con la terminología. No debes confundir «codificación» con «cifrado» (o «encriptación»). «Codificar» es simplemente aplicar una transformación que modifica el modo en que se representan los datos, pero no implica ninguna clave, ocultación u ofuscación del mensaje para conseguir confidencialidad. Por ejemplo, un mensaje codificado en *código* Morse es perfectamente entendible por cualquier persona o sistema que conozca el código, que por supuesto, es público. El cifrado pretende ocultar el contenido del mensaje, para que únicamente aquellos que conozcan la clave secreta puedan verlo. Un mensaje cifrado con Blowfish solo podrá ser descifrado por alguien que conozca la clave.

17.1. Representación, sólo eso

Una de las excepciones en las que el binario es útil directamente es la *programación de sistemas*, es decir, aquellos programas que consumen directamente servicios del SO. El binario resulta útil para manejar campos de bits, flags (banderas binarias) o máscaras, muy comunes cuando se manipulan registros de control, operaciones de E/S, etc. Por eso cualquier programador también debería manejar con soltura la representación binaria.

La codificación más básica y común es la que aplica una base numérica. Los lenguajes de programación, como las personas, utilizan la base decimal para expresar cantidades. Python permite expresar literales numéricos en varias bases. Todos los casos del siguiente ejemplo representan el número 42; y en todos los casos, si se asigna a una variable, se está creando un entero (tipo `int`) con el mismo valor. Puedes comprobarlo fácilmente en el Listado 17.1, que se está ejecutando en el modo interactivo del intérprete.

```
>>> 0b101010  # binario
42
>>> 0o52      # octal
42
>>> 0x2A      # hexadecimal
42
```

Listado 17.1: Literales numéricos en Python

Asimismo ofrece funciones para convertir entre bases: `bin()`, `oct()` y `hex()`; pero una consideración importante a tener en cuenta es que estas funciones devuelven cadenas (`str`) porque su objetivo es precisamente ofrecer diferentes *representaciones textuales* del mismo dato. Observa las comillas simples en los valores de retorno en el Listado 17.2 que indican claramente que se trata de cadenas.

```
>>> bin(42)
'0b101010'
>>> oct(42)
'0o52'
>>> hex(42)
'0x2A'
```

Listado 17.2: Conversión a representación binaria, octal y hexadecimal

Opcionalmente, el constructor de la clase `int` acepta un número expresado como cadena de caracteres pudiendo además indicar la base (incluso con bases tan exóticas como 23). Puedes verlo en el Listado 17.3.

```
>>> int('42')
42
>>> int('52', 8)
42
>>> int('1J', 23)
42
```

Listado 17.3: Especificando la base en el constructor de `int`

Insistimos: debes tener claro que 42, 052, 0x2A o 101010 no son más que diferentes representaciones del mismo dato, y que el computador lo manejará **siempre** en su forma binaria.

17.2. Los enteros de Python

El tipo `byte` es el más simple de cualquier lenguaje de programación y corresponde con una secuencia de 8 bits. Suele ser un entero sin signo, es decir, puede representar números enteros en el rango [0, 255]. Sin embargo, Python no dispone de ese tipo[2]. Python, por su naturaleza dinámica, solo

[2]No lo debes confundir con `bytes`, que es un tipo de secuencia.

tiene un tipo de datos para enteros: `int`. Estos enteros pueden ser arbitrariamente grandes puesto que el *runtime* se encarga de gestionar la memoria necesaria:

```
>>> googol = 10 ** 100
>>> type(googol)
<class 'int'>
```

Pero cuando se serializan datos en un archivo o una conexión de red, necesitamos precisar explícitamente el tamaño de los datos. Aunque Python no disponga de los tipos `byte`, `short`, `long`, etc., usaremos estos conceptos para entender los tamaños de los datos. Veremos cómo se logra todo esto en las siguientes secciones.

17.3. Caracteres

La codificación de caracteres más simple consiste en asignar un número a cada carácter del alfabeto. El código más común y veterano es ASCII, que fue creado por ANSI en 1963 como una evolución de la codificación utilizada anteriormente en telegrafía. Es un código de 7 bits (128 símbolos) que incluye los caracteres alfa-numéricos de la lengua inglesa (mayúsculas y minúsculas) y la mayoría de los signos de puntuación y tipográficos habituales. Además incluye caracteres de control para indicar salto de línea, de página, tabulador, etc. Más tarde se crearon extensiones, como por ejemplo ISO 8859-1, popularmente conocida como *latin-1* que dispone de 256 símbolos, entre los que se encuentran caracteres acentuados y otros símbolos de uso común en idiomas europeos (ß, ç, ñ, ¿, etc.), monedas (£, ¥) o signos matemáticos (±, ÷, ×, ¼, etc).

El carácter «retorno de carro» (CR, código 10 o 0x0A), pide mover el cursor a la primera columna (en el borde izquierdo), mientras que el carácter de «avance de línea» (LF, código 13 o 0x0D) pide mover el cursor en la siguiente línea. Claramente alude a las máquinas de escribir, teletipos e impresoras en los que la máquina debe situar su cabezal físico para comenzar a escribir la siguiente línea del papel. Las computadoras, por analogía, utilizaban la secuencia CR-LF para indicar la misma operación en un terminal (una pantalla). Sigue siendo de este modo en los sistemas operativos de Microsoft a día de hoy. Por contra, los creadores de los sistemas UNIX entendieron que el salto de línea sin retorno de carro no tenía sentido en un terminal y, por tanto, se utiliza únicamente el carácter CR para conseguir el mismo efecto. En muchos lenguajes de programación se representa con el carácter de control \n y se denomina «nueva línea» o EOL

Los lenguajes de programación incluyen funciones elementales para manejar la conversión entre bytes (números de 8 bits) y sus caracteres equivalentes. El siguiente fragmento de código Python lo demuestra mediante las funciones `ord()` y `chr()`.

```
>>> ord('a')
97
>>> chr(97)
'a'
>>> ord('0')
48
>>> ord('\0')
0
>>> chr(0)
'\x00'
>>> ord('\n')
10
>>> ord(' ')
32
```

Listado 17.4: Conversión entre caracteres y enteros en Python

Fíjate en la **línea 5** que el código del **carácter** `\0` es 48, mientras que (**línea 7**) el carácter equivalente al código 0 es `\x00`. Es especialmente importante tener claro que los caracteres numéricos **no corresponden** con los valores que representan. También resulta digno de mención que la secuencia `\n` es *un solo carácter*, ya que la barra es lo que se conoce como un «carácter de escape», es decir, cambia el significado del siguiente carácter. En este caso significa «nueva línea», como hemos visto.

De modo similar, la secuencia `\x` indica que los siguientes dígitos deben entenderse como un código hexadecimal. La función `chr()` devuelve una secuencia de este tipo cuando no existe un carácter *imprimible* asociado al código indicado.

El siguiente listado muestra un ejemplo de la equivalencia entre una cadena y su representación numérica.

```
>>> for i in '\xd2a3\n':
...     print ord(i)
210
97
51
10
```

La cadena `\xd2a3\n` está compuesta por los caracteres `\xd2`, `a`, `3` y `\n`.

No puedes manipular el contenido de una cadena; recuerda que el tipo `str` es inmutable. Sin embargo, tenemos el tipo `bytearray`, que puede almacenar una secuencia de bytes, modificar su contenido —acepta tanto caracteres

como enteros— y permite obtener fácilmente la lista de caracteres o secuencia de enteros equivalente:

```
>>> buf = bytearray('abcd', 'ascii')
>>> buf[0] = 20
>>> buf
bytearray(b'\x14bcd')
>>> buf.decode()
'\x14bcd'
>>> bytes(buf)
b'\x14bcd'
>>> list(buf)
[20, 98, 99, 100]
```

17.4. Tipos multibyte y ordenamiento

Como sabes, un único byte solo puede representar 256 valores; obviamente casi cualquier programa o algoritmo, por simple que sea, necesita manejar enteros mayores, reales en coma flotante y otros tipos de datos que no «caben» en un byte. El más sencillo de estos tipos es el `short` o entero de 16 bits[3]. Aquí aparece una cuestión interesante: el ordenamiento de bytes (*endianness* o *byte order*) o, lo que es lo mismo, ¿en qué orden se deberían colocar en memoria los dos bytes que forman un `short`?

Dependiendo de la respuesta se distingue entre *little endian* y *big endian*. *Little endian* significa que el byte de *menor* peso[4] se coloca en la dirección más baja de memoria, mientras que en *big endian* es el byte de *mayor* peso[5]. La Figura 17.2 lo muestra para un dato de 4 bytes.

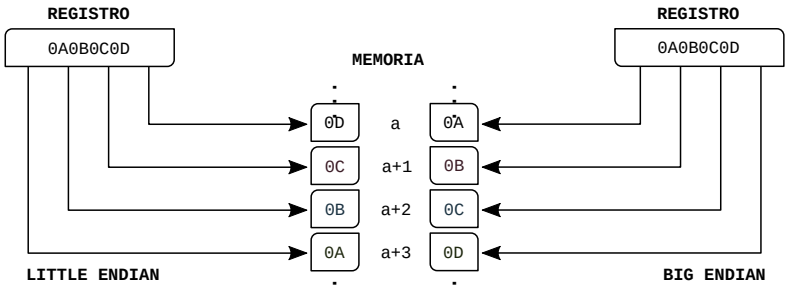

FIGURA 17.2: Ordenación de bytes

[3] `short` tampoco existe como tipo nativo en Python.
[4] el que tiene los bits menos significativos (LSB)
[5] el MSB

Para que el computador realice las operaciones (aritméticas, lógicas, etc.) que correspondan sobre el dato, es esencial que los programas manipulen la memoria de acuerdo al ordenamiento de la arquitectura.

Puedes comprobar de una manera muy sencilla qué tipo de ordenamiento tiene tu computadora, utilizando el módulo `struct` (más adelante lo veremos con detalle).

```
if struct.pack('H', 1) == b'\x00\x01':
    print("big endian")
else:
    print("little endian")
```

Listado 17.5: Averiguar el ordenamiento de bytes con Python.

Aunque Python ofrece una forma directa:

```
>>> sys.byteorder
'little'
```

Algo parecido ocurre también con la red. Cuando se coloca un dato multi-byte *en el cable*[6] debe elegirse un ordenamiento. Pues bien, los protocolos de la pila TCP/IP utilizan siempre ordenamiento *big-endian* [33], es decir, se envía primero el byte más significativo. Y de hecho, se le conoce como *ordenamiento de la red* (*network byte order*).

Eso significa que las arquitecturas *little-endian* (típicamente Intel y AMD) deben convertir sus datos multi-byte antes de enviarlos a la red sobre un `socket`. Para evita que el programa tenga que comprobar por sí mismo qué ordenamiento utiliza la arquitectura en la que se está ejecutando, las librerías de sockets proporcionan funciones que realizan la conversión. Nótese que en un nodo *big-endian* estas funciones no harán nada[7], pero deben usarse para conseguir que el programa sea 'portable', es decir, que se pueda ejecutar en nodos de diferente arquitectura sin ninguna modificación.

Las funciones que ofrece Python, tomadas de las llamadas al sistema POSIX homónimas, son:

- `socket.ntohs()`. Convierte un entero de 16 bits (*short*) del ordenamiento de la red al del host: ***n**etwork **to** **h**ost **s**hort*.
- `socket.ntohl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de la red al del host: ***n**etwork **to** **h**ost **l**ong*.
- `socket.htons()`. Convierte un entero de 16 bits (*short*) del ordenamiento del host al de la red: ***h**ost **to** **n**etwork **s**hort*.

[6] por analogía con la expresión en inglés *on the wire*

[7] retornan el mismo valor que se les pasa como parámetro

- `socket.htonl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de host al de la red: ***h**ost **to** **n**etwork **l**ong*.

El siguiente listado muestra unos ejemplos de uso de estas funciones en un computador *little-endian*.

```
>>> socket.htons(32)
8192
>>> socket.htonl(32)
536870912
>>> socket.htons(0)
0
```

LISTADO 17.6: Funciones de conversión de ordenamiento del módulo `socket`.

Veamos de nuevo, en hexadecimal, la primera transformación para observar mejor los 2 bytes que lo forman:

```
>>> hex(32)
'0x20'
>>> hex(socket.htons(32))
'0x2000L'
```

Se puede ver claramente que al convertir el valor `0x20` desde *big-endian* se coloca en el primer byte del entero de 16 bits. Si el computador receptor fuese *little-endian* no habría cambios.

17.5. Cadenas de caracteres y secuencias de bytes

En Python las cadenas de caracteres (de tipo `str`) utilizan Unicode. Sin embargo, este tipo de datos no se puede leer o escribir en un archivo (salvo que sea de texto), ni enviar o recibir de un socket. Todas esas operaciones requieren secuencias de bytes (de tipo `bytes`). Convertir una cadena a una secuencia de bytes requiere aplicar una codificación (un *encoding*) que establece la transformación. El siguiente listado ilustra la diferencia entre ambos tipos de datos:

```
>>> string = "hello world"
>>> type(string)
<class 'str'>
>>> sequence = bytes(string, 'ascii')
>>> type(sequence)
<class 'bytes'>
>>> string
'hello world'
>>> sequence
b'hello world'
```

LISTADO 17.7: Cadena codificada en ASCII

Python puede utilizar muchos sistemas de codificación (*encodings*) diferentes. El más habitual en los sistemas POSIX es UTF8 y también lo es para Python. UTF8 puede representar cualquier carácter Unicode. Es compatible con ASCII para su rango (127 caracteres), es decir, requiere un byte, mientras que para el resto de caracteres necesita 2 o más bytes. Por ejemplo el carácter `ñ` se codifica con 2 bytes: '0xc3b1' y `€` se codifica con 3 bytes: '0xe282ac'.

En la cadena de caracteres, cada símbolo representa un carácter, mientras que en la secuencia de bytes cada símbolo representa un byte. Y, ¿cuál es la diferencia? Parece que las líneas 8 y 10 son iguales salvo por la `b` que precede a la secuencia de bytes. Esto se debe a que este ejemplo usa codificación «ascii». ASCII codifica cada carácter con un único byte, por eso coinciden. Veamos otro ejemplo más ilustrativo:

```
1  >>> string = "ñandú"
2  >>> sequence = bytes(string, 'ascii')
3  UnicodeEncodeError: 'ascii' codec can't encode character [...]
4  >>> sequence = bytes(string, 'utf-8')
5  >>> string
6  'ñandú'
7  >>> sequence
8  b'\xc3\xb1and\xc3\xba'
9  >>> len(sequence)
10 7
```

Listado 17.8: Cadena codificada en UTF8

Esta vez, al intentar codificar la cadena «`ñandú`» con el *encoding* ASCII se produce un error (**línea 2**), porque ASCII no puede representar las letras `ñ` y `ú`. Pero UTF8 sí que puede, aunque requiere 2 bytes para esos caracteres 'no ASCII'. Por eso, la secuencia equivalente requiere 7 bytes (**línea 10**) a pesar de que la cadena solo tiene 5 caracteres. Nótese que la conversión puede lograrse utilizando los constructores de ambos tipos (`bytes` y `str`) o bien los métodos `encode()` y `decode()` respectivamente:

```
>>> bytes('ñandú', 'utf-8')
b'\xc3\xb1and\xc3\xba'
>>> 'ñandú'.encode('utf-8')
b'\xc3\xb1and\xc3\xba'
>>> str(b'\xc3\xb1and\xc3\xba', 'utf-8')
'ñandú'
>>> b'\xc3\xb1and\xc3\xba'.decode('utf-8')
'ñandú'
```

Listado 17.9: Constructores y métodos `str.encode()` y `bytes.decode()`

17.6. Empaquetado

El módulo `struct` de la librería estándar de Python puede hacer transformaciones hacia y desde binario de un modo mucho más flexible y, sobre todo, cómodo. La función `struct.pack()`[8] *serializa* datos nativos de Python generando una secuencia de bytes (tipo `bytes`) según la especificación de tamaño y ordenamiento que se le indique. Por ejemplo, el siguiente listado serializa el número 5 como un entero de 32 bits con ordenamiento *big-endian* y después como *little-endian*:

```
>>> struct.pack('>i', 5)
b'\x00\x00\x00\x05'
>>> struct.pack('<i', 5)
b'\x05\x00\x00\x00'
```

LISTADO 17.10: `struct`: alternativas de ordenamiento de un entero de 32 bits

El primer parámetro de `pack()` es la especificación de la conversión. Hay dos conjuntos de símbolos: uno para especificar ordenamiento (ver tabla La tabla 17.1) y otro para especificar formato (ver tabla 17.2).

El siguiente listado muestra el resultado de aplicar ambos ordenamientos al mismo dato en un computador *little-endian*, pero con un entero de 16 bits.

```
>>> struct.pack('>h', 5)
b'\x00\x05'
>>> struct.pack('<h', 5)
b'\x05\x00'
```

LISTADO 17.11: `struct`: alternativas de ordenamiento de un entero de 16 bits

`@`	ordenamiento nativo del computador (realiza alineamiento)
`=`	ordenamiento nativo
`<`	*little endian*
`>`	*big endian*
`!`	ordenamiento de la red (*big-endian*)

CUADRO 17.1: `struct`: especificación de ordenamiento

El Listado 17.12 muestra el resultado de aplicar los diferentes formatos al mismo dato en un computador *little-endian*.

```
>>> struct.pack('b', 5)
b'\x05'
>>> struct.pack('?', 5)
```

[8] *empaquetar*

`x`	relleno (alineado al siguiente dato)
`c`	carácter (`char`)
`b`	byte con signo
`B`	byte sin signo
`?`	booleano/char
`h`	entero de 16 bits con signo
`H`	entero de 16 bits sin signo
`i`	entero de 32 bits con signo
`I`	entero de 32 bits sin signo
`q`	entero de 64 bits con signo (nativo)
`Q`	entero de 64 bits sin signo (nativo)
`f`	`float`
`d`	`double`
`s`	cadena de caracteres (un número previo indica tamaño)
`P`	entero que puede almacenar una dirección de memoria
`q` ó `Q`	entero equivalente al `long long` de C en la misma arquitectura.

CUADRO 17.2: `struct`: especificación de formato

```
b'\x01'
>>> struct.pack('h', 5)
b'\x05\x00'
>>> struct.pack('i', 5)
b'\x05\x00\x00\x00'
>>> struct.pack('l', 5)
b'\x05\x00\x00\x00\x00\x00\x00\x00'
>>> struct.pack('f', 5)
b'\x00\x00\xa0@'
>>> struct.pack('d', 5)
b'\x00\x00\x00\x00\x00\x00\x14@'
```

LISTADO 17.12: `struct`: empaquetado en diferentes tamaños

Pero lo verdaderamente interesante de `struct` es que la cadena de formato puede especificar un número arbitrario de campos, que corresponden a parámetros sucesivos de la función `pack()`. Veamos un ejemplo empaquetando la cabecera de un mensaje ARP sobre una trama Ethernet (§ 5.3).

El valor para los campos de la trama será:

MAC destino
`FF:FF:FF:FF:FF:FF` (es una trama broadcast).

MAC origen
`C4:85:08:ED:D3:07`.

tipo `0x0806`, que corresponde con el protocolo ARP.

En el Listado 17.13 se puede ver cómo construir dicha cabecera, la secuencia de bytes que se obtiene y su equivalente numérico. La cadena de formato (`!6s6sh`) indica que debe codificarse con ordenamiento de red (`!`) y que está compuesto de dos cadenas de 6 bytes (`6s`) y un entero de 16 bits sin signo (`h`).

Lo interesante es que la secuencia resultante siempre tendrá una longitud de 14 bytes independientemente del valor de sus tres argumentos.

```
>>> header = struct.pack('!6s6sh', b'\xFF' * 6, b'\xC4\x85\x08\xED\xD3\x07', 0x0806)
>>> header
b'\xff\xff\xff\xff\xff\xff\xc4\x85\x08\xed\xd3\x07\x08\x06'
>>> list(header)
[255, 255, 255, 255, 255, 255, 196, 133, 8, 237, 211, 7, 8, 6]
```

LISTADO 17.13: `struct`: empaquetando una cabecera Ethernet

17.7. Desempaquetado

La contrapartida de `pack()` es `unpack()`. Esta función toma una cadena de formato con las misma reglas que `pack()` y una secuencia de bytes, que puede haber sido obtenida con `file.read()`, `socket.recv()` o cualquier otra función orientada a lectura de flujos (*streams*). La función `unpack()` retorna una tupla con los valores que corresponden a cada uno de los campos especificados en la cadena de formato. La función `unpack()` realiza por tanto la *des-serialización*.

El Listado 17.14 realiza la función inversa al anterior. Es decir, a partir de la secuencia de bytes devuelve una tupla con las dos direcciones MAC y el tipo de la trama. La **línea 5** simplemente corrobora que el tercer valor de la tupla (`2054`) coincide efectivamente con el valor hexadecimal `0x0806`.

```
1 >>> header
2 b'\xff\xff\xff\xff\xff\xff\xc4\x85\x08\xed\xd3\x07\x08\x06'
3 >>> struct.unpack('!6s6sh', header)
4 (b'\xff\xff\xff\xff\xff\xff', b'\xc4\x85\x08\xed\xd3\x07', 2054)
5 >>> hex(2054)
6 '0x806'
```

LISTADO 17.14: `struct`: desempaquetando una cabecera Ethernet

17.8. Formatos de serialización binaria

El módulo `struct` permite trabajar con datos binarios de un modo muy sencillo, pero es bastante limitado. Si los datos a serializar son de tamaño variable, como cadenas o secuencias de objetos, la cosa se complica bastante. Para usos más avanzados, sobre todo cuando queremos definir nuestras propias estructuras, existen librerías mucho más potentes como `construct`, Google Protocol Buffers o Apache Avro.

17.9. Serialización textual

En las capas inferiores —en realidad desde transporte hacia abajo— la gran mayoría de los protocolos utilizan serialización binaria porque normalmente eso genera mensajes más compactos y su procesamiento es más rápido y requiere menos recursos. En la capa de aplicación sin embargo, es común encontrar muchos protocolos que utilizan serialización textual.

El formato textual para codificación de datos mas usado en la actualidad con mucha diferencia es JSON. Como su nombre indica, JSON genera/reconoce cadenas de texto que replican el formato de la definición de objetos con la sintaxis de JavaScript[9]. El grán éxito de JSON se debe a que es muy legible por humanos y su procesamiento automatizado es muy sencillo, aunque costoso en recursos comparado con los formatos binarios. Hoy en día es una norma estandarizada y muchos lenguajes ofrecen soporte incluso en su librería estándar.

Desde Python, también resulta muy sencillo de manipular. El Listado 17.15 muestra como serializar la hipotética lectura de un sensor de temperatura y humedad, que incluye además un identificar y el estado de carga de su batería. El resultado de la serialización (`payload`) se puede enviar perfectamente como carga útil de un datagrama UDP.

```
>>> import json
>>> import socket

>>> data = {'sensor-id': 'temp-hum-01', 'temperature': 22.5, 'humidity': 45.2,
...         'battery': 0.85}
>>> payload = json.dumps(data)
>>> print(payload)
{"sensor-id": "temp-hum-01", "temperature": 22.5, "humidity": 45.2, "battery": 0.85}
>>> print(type(payload))
<class 'str'>

>>> with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
...     sock.sendto(payload.encode(), ('203.0.113.2', 55000))
```

LISTADO 17.15: Serialización JSON y envío de la lectura de un sensor

El código con la deserialización (Listado 17.16) muestra la recepción del mensaje su des-serialización.

```
>>> import json
>>> import socket

>>> with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
...     sock.bind(('', 55000))
...     data, addr = sock.recvfrom(1024)
```

[9]Hay mucho que puntualizar en esta frase, pero sirve como aproximación.

```
>>> data = json.loads(data.decode())
>>> print(data['temperature'])
22.5
```

LISTADO 17.16: Deserialización JSON de la lectura de un sensor

También para serialización textual hay innumerables formatos entre los que podemos destacar XML, YAML, CSV o TOML entre otros.

Y ¿qué más?

La serialización es una parte esencial de las comunicaciones en Internet, pero también una fuente inagotable de errores y confusión sobre todo entre programadores noveles. El objetivo de este capítulo ha sido aclarar los conceptos más básicos mostrando ejemplos prácticos que ilustren su funcionamiento. Quedan fuera muchos detalles técnicos, pero a partir de este punto el lector tendrá la base para profundizar por su cuenta en librerías y formatos con los que se encuentre en su día a día, que a buen seguro, serán muchos y variados.