Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

12

Tipos para manejo de bytes

Índice de contenidos

Introducción

Estamos en el último tema del bloque correspondiente a los tipos colección.

En este tema vamos a ver los tipos secuencia para manipular bytes en Python (el resto de tipos secuencia los expliqué en el Tema 9).

Tipos bytes y bytearray

A la hora de trabajar con bytes, Python (concretamente Python 3, con el que estamos trabajando) pone a nuestra disposición los tipos `bytes` y `bytearray`. Son muy similares, con la principal diferencia de que `bytes` es un tipo *inmutable* y `bytearray` es *mutable*.

Entendiendo qué son los bytes

El único tipo de información que entiende realmente un ordenador es aquella que está formada por secuencias de unos y ceros. Un 0 o un 1 se conoce comúnmente como *bit*. Una secuencia de 8 ceros y/o unos es un *byte*. Por tanto, se podría decir que un *byte* se puede representar con un entero cuyo rango válido de valores va de 0 a 255.

Además de los tipos básicos (`int`, `float`, `bool` o `str`, entre otros), en programación es necesario un tipo para manipular secuencias de bytes. ¿Por qué? Porque, por ejemplo, las imágenes son secuencias de bytes, los archivos son secuencias de bytes, la información que se envía a través de sockets son secuencias de bytes o la comunicación con los dispositivos y periféricos se realiza a través de secuencias de bytes.

Una cosa muy importante que debes tener en cuenta es que una secuencia de bytes solamente es entendible por el ordenador. Más concretamente, por el programa que manipula dicha secuencia, ya que la secuencia puede representar un número (con signo o sin signo), una cadena de texto, un comando, un objeto, etc. A esta forma de entender la secuenciad de *bytes* se conoce como *codificación.*

bytes vs str

En relación con el apartado anterior, vamos a ver a continuación la diferencia entre un objeto de tipo `bytes` y un objeto de tipo `str`:

- Los objetos de tipo `bytes` son secuencias de bytes, mientras que las cadenas son secuencias de caracteres.
- Los objetos de tipo `bytes` son legibles solo por la máquina (o programa), las cadenas de caracteres son legibles por los humanos.
- Dado que los objetos de tipo `bytes` son legibles por el ordenador, pueden almacenarse directamente en el disco. Por su parte, las cadenas de caracteres necesitan indicar la codificación antes de que se puedan almacenar.

Los tipos bytes y bytearray en Python

Una vez aclarado qué son los bytes y para qué se utilizan, te explicaré los tipos `bytes` y `bytearray` que implementa Python

Formalmente, el tipo `bytes` es una **secuencia inmutable de bytes**. Puedes pensar en un objeto de tipo `bytes` como si fuera una secuencia inmutable de enteros, en el que cada elemento de la secuencia solo puede tomar un valor comprendido en el rango entre 0 y 255 (ambos inclusive).

Por su parte, el tipo `bytearray` es la versión mutable del tipo `bytes`.

Una particularidad de los números comprendidos entre 0 y 127 es que se corresponden con los códigos de los caracteres en codificación *ASCII*. Esto hace que muchos objetos de tipo `bytes` (o `bytearray`) se puedan ver como cadenas de caracteres en dicha codificación. De hecho, en Python, cuando se muestran por pantalla estos objetos, lo que se visualiza son los caracteres *ASCII*. Además, esto permite inicializar un objeto de tipo `bytes` a partir de una cadena formada únicamente por caracteres *ASCII*. Por todo ello, las clases `bytes` y `bytearray` ofrecen métodos muy parecidos a los de la

clase `str` para cuando se trabaja con datos compatibles con esta codificación.

Cómo crear un objeto bytes

Un primer modo de crear un objeto de tipo `bytes` es usando un literal de bytes, que es muy similar a una cadena de caracteres solo que antepone el carácter `b` a las comillas.

```
>>> b = b'Esto son bytes'
>>> b
b'Esto son bytes'
```

Fíjate que, al mostrar la secuencia de bytes, la cadena está precedida por el carácter `b` para diferenciarlo de un string.

IMPORTANTE: Solo se permiten caracteres ASCII en literales de bytes. Cualquier valor binario superior a 127 debe ingresarse en bytes literales utilizando la secuencia de escape apropiada.

La otra forma de crear un objeto de tipo `bytes` es usando el constructor de la clase `bytes([source[, encoding[, errors]]])`:

- `source`: origen para inicializar la secuencia de bytes. Puede ser:
  - **Un string**: Convierte la cadena a bytes. Es obligatorio indicar el parámetro `encoding`.
  - **Un entero**: Crea una secuencia de bytes de la longitud indicada, todos ellos sin inicializar.
  - Un objeto que implemente la interfaz *buffer*.
  - **Un iterable**: Crea una secuencia de bytes de tamaño igual al *iterable*. Los elementos del *iterable* deben ser enteros comprendidos entre 0 y 255.
- `encoding`: Si `source` es un *string*, la codificación de la cadena.
- `errors`: Si `source` es un *string*, la acción a realizar cuando la codificación falla.

Cómo crear un objeto bytearray

Para crear un objeto de tipo `bytearray` se debe utilizar el constructor de la clase `bytearray([source[, encoding[, errors]]])`. El significado de los parámetros es el mismo que para la clase `bytes`.

Operaciones sobre los tipos bytes y bytearray

Al tratarse de tipos secuencia, `bytes` y `bytearray` implementan las operaciones comunes de estos tipos de datos, como *pertenencia*, *concatenación*, *longitud* o acceso a un elemento mediante *índice*. Además, `bytearray` implementa las operaciones de los tipos secuenciales *mutables*.

NOTA: ¡Cuidado a la hora de acceder a los elementos mediante índices! Imagina que `b` es un objeto de tipo bytes y que `i` y `j` son índices. Pues `b[i]` devuelve un entero, mientras que `b[i:j]` devuelve una secuencia de bytes. Esto no ocurría así para el tipo `str`, que siempre devolvía un string.

Equivalencias de datos

Como 2 dígitos hexadecimales se corresponden precisamente con un byte, los números hexadecimales son un formato comúnmente utilizado para describir datos binarios.

En Python, se puede crear una secuencia de bytes a partir de un literal que contiene valores hexadecimales con el carácter de escape `\x`.

Por ejemplo, el carácter `E` en *ASCII* tiene el código hexadecimal 45. Esto implica que las siguientes secuencias de bytes sean equivalentes:

```
>>> b1 = bytes(b'E')
>>> b2 = bytes(b'\x45')
>>> b1
b'E'
>>> b2
b'E'
>>> b1 == b2
True
```

Además, las clases `bytes` y `bytearray` disponen de métodos para leer una cadena de dígitos hexadecimales y convertirlos a bytes y hacer el paso contrario:

```
# fromhex ignora los espacios en blanco
>>> b = bytes.fromhex('2e   45')
>>> b
b'.E'

>>> b.hex()
'2e45'
```

Por otro lado, como un objeto de tipo `bytes` o `bytearray` es una secuencia de números enteros, podemos obtener su representación numérica pasando un objeto de cualquiera de estos tipos como argumento del constructor `list()`. Siguiendo con el ejemplo anterior:

```
>>> list(b)
[46, 69]
```

Codificar y decodificar secuencias de bytes

Para convertir una cadena de caracteres a bytes, es necesario indicar una codificación. Esto hará que los caracteres cuyo código sea mayor que 127 se conviertan a los bytes correspondientes.

Podemos convertir una cadena de caracteres a bytes de dos formas diferentes:

```
# 1. Pasando un string al constructor
# de la clase bytes
>>> b = bytes('Pingüino', 'utf-8')
>>> b
b'Ping\xc3\xbcino'

# 2. Llamando al método encode de
# la clase str
>>> b2 = 'Pingüino'.encode('latin1')
>>> b2
b'Ping\xfcino'
```

Fíjate bien que en los ejemplos anteriores he utilizado dos codificaciones diferentes.

La operación contraria, convertir una secuencia de bytes a cadena de caracteres, se consigue llamando al método `decode()` de las clases `bytes` y `bytearray`. Siguiendo con el ejemplo anterior:

```
>>> b.decode('utf-8')
'Pingüino'
>>> b2.decode('latin1')
'Pingüino'
```

Otras operaciones

Además de las operaciones anteriores, las clases `bytes` y `bytearray` implementan una serie de métodos muy similares a los de la clase `str`. Sin embargo, algunos de estos métodos solo funcionan correctamente cuando el flujo de bytes se corresponde exclusivamente con caracteres *ASCII*.

Entre estos métodos se encuentran, por ejemplo, contar el número de apariciones de un byte en la secuencia o comprobar si una secuencia de bytes comienza por un prefijo de terminado.