6 FICHEROS

INTRODUCCIÓN

Los sistemas operativos almacenan los datos de forma estructurada en unas unidades básicas de almacenamiento a las que llamamos *ficheros* o *archivos*. Tarde o temprano, los programadores necesitan trabajar con ficheros, ya sea para leer datos de los mismos o para crearlos. Es por ello que los lenguajes de programación suelen incluir librerías que contienen funciones para el tratamiento de ficheros. En Python contamos con una serie de funciones básicas, incluidas en su librería estándar, para leer y escribir datos en ficheros. A estas funciones dedicaremos el primer apartado del presente capítulo.

Una vez que aprendamos todo sobre lo básico sobre el manejo de ficheros en Python, pasaremos a adentrarnos en el concepto de *serialización*, el cual está directamente relacionado con los ficheros. Descubriremos qué herramientas nos ofrece el lenguaje y qué módulos de la librería estándar pueden ser empleados para serializar datos en ficheros. Además, continuaremos haciendo un repaso a los tres principales formatos utilizados para serializar: XML, JSON y YAML. Estos formatos no solo se usan para serializar, sino que también suelen ser empleados para guardar información estructurada y para el intercambio de la misma. Por ejemplo, son muchas las aplicaciones que utilizan el formato YAML para guardar información sobre una determinada configuración.

Dado que el formato CSV es uno de los más sencillos y populares para guardar información estructurada en ficheros, dedicaremos un apartado específico a ver cómo puede Python manejar este tipo de ficheros. Aplicaciones como *MS Office* y *LibreOffice* pueden exportar e importar datos en formato CSV, de forma que necesitamos desarrollar una aplicación que trabaje con este formato y que pueda intercambiar datos con este tipo de programas, podremos emplear Python como lenguaje de programación.

Python cuenta con un módulo específico que permite leer ficheros en formato INI. Este es muy popular en los sistemas Windows y suelen ser empleado para guardar información relativa a una determinada configuración. También en este capítulo aprenderemos lo básico para poder trabajar con ficheros del mencionado tipo.

Hoy en día es muy común trabajar con ficheros comprimidos que nos permiten almacenar más información en menos espacio. Formatos como ZIP, RAR o gzip se han convertido en formatos *de facto* y la mayoría de los sistemas operativos permiten utilizar herramientas para comprimir o descomprimir ficheros. Teniendo en cuenta estos factores, parece útil disponer de funcionalidades que permitan trabajar con este tipo de ficheros desde un lenguaje de programación. En Python contamos con varios módulos de su librería estándar que nos facilitan el trabajo, de ellas nos ocuparemos en el último apartado de este capítulo.

OPERACIONES BÁSICAS

Python incorpora en su librería estándar una estructura de datos específica para trabajar con ficheros. Básicamente, esta estructura es un *stream* que referencia al fichero físico del sistema operativo. Dado que el intérprete de Python es multiplataforma, el manejo interno y a bajo nivel que se realiza del fichero es transparente para el programador. Entre las operaciones que Python permite realizar con ficheros encontramos las más básicas que son: apertura, creación, lectura y escritura de datos.

Apertura y creación

La principal función que debemos conocer cuando trabajamos con ficheros se llama *open()* y se encuentra integrada en la librería estándar del lenguaje. Esta función devuelve un *stream* que nos permitirá operar directamente sobre el fichero en cuestión. Entre los argumentos que utiliza la mencionada función, dos son los más importantes. El primero de ellos es una cadena de texto que referencia la ruta (*path*)

del sistema de ficheros. El otro parámetro referencia el *modo* en el que va a ser abierto el fichero. Es importante tener en cuenta que Python diferencia entre dos tipos de ficheros, los de texto y los binarios. Esta diferenciación no existe como tal en los sistemas operativos, ya que son tratados de la misma forma a bajo nivel. Sin embargo, a través del *modo* podemos indicarle a Python que un fichero es un tipo u otro. De esta forma, si el *modo* del fichero es texto, los datos leídos serán considerados como un *string*, mientras que si el *modo* es binario, los datos serán tratados como *bytes*.

Antes de comenzar a ver un ejemplo de código, abriremos nuestro editor de textos favorito, añadiremos una serie de líneas de texto y lo salvaremos, por ejemplo, con el nombre *fichero.txt*. Seguidamente, ejecutaremos el intérprete de Python desde la interfaz de comandos y escribiremos la siguiente línea:

```
>>> fich = open("fichero.txt")
```

En nuestro ejemplo, hemos supuesto que el fichero creado se encuentra en la misma ruta desde la que hemos lanzado el intérprete, de no ser así es necesario indicar el *path* absoluto o relativo al fichero. Por otro lado, no hemos indicado ningún modo, ya que, por defecto, Python emplea el valor *r* para ficheros de texto. Antes de continuar, debemos tener en cuenta que los sistemas operativos suelen emplear diferentes caracteres como separadores entre los directorios que forman parte de una ruta del sistema de ficheros. Por ejemplo, supongamos que nuestro fichero se encuentra en un directorio llamado *pruebas*. Si estamos trabajando en Windows, nuestra línea de código para abrir el fichero sería la siguiente:

```
>>> fich = open("pruebas\fichero.txt")
```

Sin embargo, para realizar la misma operación en Linux necesitaríamos esta otra línea de código:

```
>>> fich = open("pruebas/fichero.txt")
```

Dado que Python es multiplataforma, es deseable que el mismo código funcione con independencia del sistema operativo que lo ejecuta, pero, en nuestro ejemplo, este código es distinto. ¿Cómo resolver este problema? Es sencillo, para ello Python nos facilita una función que se encuentra integrada en el módulo *os* de su librería estándar. Se trata de *join()* y se encarga de unir varias cadenas de texto empleando el separador adecuado para cada sistema operativo. De esta forma, sería más práctico escribir el código anterior para la apertura del fichero de la siguiente forma:

```
>>> from os import path
>>> ruta_fich = path.join("pruebas", "fichero.txt")
>>> fich = open(ruta_fich)
```

Por otro lado y como complemento a la función *join()*, también existe la variable *sep*, que también pertenece al módulo *os*. Esta representa el carácter propiamente dicho que cada sistema operativo emplea para la separación entre directorios y ficheros.

Respecto al valor que se puede indicar para el modo de apertura del fichero, Python nos permite utilizar un valor determinado en función de la operación (lectura, escritura, añadir y lectura/escritura) y otro para señalar si el fichero es de texto o binario. Obviamente, ambos tipos de valores se pueden combinar, para, por ejemplo, indicar que deseamos abrir un fichero binario para solo lectura. En concreto, el valor "r" significa "solo lectura"; con "w" abriremos el fichero para poder escribir en él; para añadir datos al final de un fichero ya existente emplearemos "a" y, por último, si vamos a leer y escribir en el fichero, basta con utilizar "+". Por otro lado, con "b" señalaremos que el fichero es binario y con "t" que es de texto. Como hemos comentado previamente, por defecto, la función *open()* interpreta que vamos a abrir un fichero de texto como solo lectura. Tanto el valor para realizar una operación como para indicar el tipo de fichero debe indicarse como argumentos del parámetro *mode*. Así pues, para abrir un fichero binario para lectura y escritura basta con ejecutar el siguiente comando:

```
>>> open("data.bin", "b+")
```

Además de los mencionados parámetros de la función *open()*, existen otros que podemos utilizar. En concreto, contamos con cinco más. El primero de ellos es *buffering* y permite controlar el tamaño del *buffer* que el intérprete utiliza para leer o escribir en el fichero. El segundo parámetro es *encoding* y permite indicar el tipo de codificación de caracteres que deseamos emplear para nuestro fichero. Otro de los parámetros es *errors* que indica cómo manejar errores debidos a problemas derivados de la utilización de la codificación de caracteres. Para controlar cómo tratar los saltos de línea contamos con *newline*. Por último, *closefd* es *True* y si cambiamos su valor a *False* el descriptor de fichero permanecerá abierto aunque explícitamente invoquemos al método *close()* para cerrar el fichero.

Hasta el momento hemos hablado de abrir ficheros, utilizando para ello diferentes parámetros. Pero ¿cómo podemos crear un fichero en nuestro sistema de archivos desde Python? Basta con aplicar el modo de *escritura* y pasar el nombre del nuevo fichero, tal y como muestra el siguiente ejemplo:

```
>>> f_nuevo = open("nuevo.txt", "w")
```

Debemos tener en cuenta que, pasando el valor *w* sobre un fichero que ya existe, este será sobrescrito, borrando su contenido.

Ahora que ya sabemos cómo abrir y crear un fichero, es hora de aprender cómo leer y escribir datos.

Lectura y escritura

La operación básica de escritura en un fichero se hace en Python a través del método *write()*. Si estamos trabajando con un fichero de texto, pasaremos una cadena de texto a este método como argumento principal. Supongamos que vamos a crear un nuevo fichero de texto añadiendo una serie de líneas, bastaría con ejecutar las siguientes líneas de código:

```
>>> fich = open("texto.txt", "w")
>>> fich.write("Primera línea\n")
14
>>> fich.write("Segunda línea\n")
14
>>> fich.close()
```

El carácter "\n" sirve para indicar que deseamos añadir un retorno de carro, es decir, crear una nueva línea. Después de realizar las operaciones de escritura, debemos cerrar el fichero para que el intérprete vuelque el contenido al fichero físico y se pueda también cerrar su descriptor. Si necesitamos volcar texto antes de cerrar el fichero, podemos hacer uso del método *flush()* y, posteriormente, podemos seguir empleando *write()*, sin olvidar finalmente invocar a *close()*. En nuestro ejemplo, observaremos cómo, después de cada operación de escritura, aparece un número. Este indica el número de bytes que han sido escritos en el fichero.

Adicionalmente, el método *writelines()* escribe una serie de líneas leyéndolas desde una lista. Por ejemplo, si deseamos emplear este método en lugar de varias llamadas a *write()*, podemos sustituir el código anteriormente mostrado por este otro:

```
>>> lineas = ["Primera línea\n", "Segunda línea\n"]
>>> fich.writelines(lineas)
```

Ahora que ya tenemos texto en nuestro fichero, es hora de pasar a la operación inversa a la escritura, hablamos de la lectura desde el fichero. Para ello, Python nos ofrece tres métodos diferentes. El primero de ellos es *read()*, el cual lee el contenido de todo el fichero y devuelve un única cadena de texto. El segundo en cuestión es *readline()* que se ocupa de leer línea a línea del fichero. Por último, contamos con *readlines()* que devuelve una lista donde cada elemento corresponde a cada línea

que contiene el fichero. Los siguientes ejemplos muestran cómo utilizar estos métodos y su resultado sobre el fichero que hemos creado previamente:

```
>>> fich = open("texto.txt", "w")
>>> fich.read()
'Primera línea\nSegunda línea\n'
>>> fich.seek(0)
0
>>> fich.readlines(fich)
['Primera línea\n', 'Segunda línea\n']
>>> fich.seek(0)
0
>>> fich.readline()
'Primera línea\n'
```

Seguro que al lector no le ha pasado desapercibido el uso de un nuevo método. Efectivamente, *seek()* es otro de los métodos que podemos usar sobre el objeto de Python que maneja ficheros y que sirve para posicionar el puntero de avance sobre un punto determinado. En nuestro ejemplo, y dado que deseamos volver al principio del fichero, hemos empleado el valor *0*. Debemos tener en cuenta que cuando se hace una operación de escritura, Python maneja un puntero para saber en qué posición deberá ser escrita la siguiente línea con el objetivo de no sobrescribir nada. Sin embargo, este puntero avanza de forma automática y el método *seek()* sirve para situar el mencionado puntero en una determinada posición del fichero.

Para leer todas las líneas de un fichero no es necesario emplear *seek()* tal y como muestra el ejemplo anterior de código. Existe un método más sencillo basado en emplear un *iterator* para ello:

```
>>> for line in open("texto.txt"):
….      print(line)
…
Primera línea
Segunda línea
```

Además, también es práctica habitual emplear *with* para leer todas las líneas de un fichero:

```
>>> with open("texto.txt") as fich:
…       print(fich.read())
…
Primera línea
Segunda línea
```