Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

18

Lectura y escritura de Ficheros

Índice de contenidos

Introducción

Y tras haber visto ya gran parte de las características de Python, ha llegado el turno de los ficheros.

Un archivo o fichero es una secuencia de datos relacionados lógicamente que pueden ser utilizados por un programa. Por lo general, un archivo se guarda en un medio de almacenamiento persistente, por ejemplo, un disco duro.

Para acceder a los archivos en un ordenador y/o una aplicación, se suele utilizar un nombre o ruta que indica dónde está almacenado el archivo en disco.

Las principales operaciones que se llevan a cabo con archivos son leer datos y escribir o añadir datos. Precisamente es lo que veremos en este tema: cómo leer y escribir en un fichero.

El objeto file

En Python, para poder leer y/o escribir en un fichero lo único que se necesita es una referencia a un objeto de tipo `file`.

Para obtener esta referencia, hay que llamar a la función predefinida `open()`. Lo más común es llamar a la función `open()` pasando dos argumentos:

- El primero de ellos es la ruta del fichero (en la que está o donde se va a crear).
- El segundo es el modo en el que se abre el fichero: lectura, escritura, ...

Normalmente, existen dos tipos de ficheros: **ficheros de texto** y **ficheros binarios**. Un fichero de texto contiene caracteres que son legibles por el ser humano y están guardados con una codificación (*ASCII*, *UTF-8*, ...). Por el contrario, un fichero binario está compuesto por un flujo de bytes y solo tienen sentido para los programas o aplicaciones para los que son creados. Un ejemplo de este tipo de archivos son las imágenes o la música. Ya vimos esto en el Tema 12, *Tipos para manejo de bytes*.

IMPORTANTE: Por defecto, la codificación utilizada para leer y escribir ficheros de texto es *utf-8*. Si se quiere utilizar otra codificación, hay que especificarla en el argumento `encoding` de la función `open()`.

Cuando se trabaja con un fichero de texto, hay que tener en cuenta que este se estructura como una secuencia de líneas. Cada una de estas líneas acaba con un carácter especial conocido como `EOL` (fin de línea). En función del sistema operativo, este carácter puede variar. Puede ser `\n` (Unix) o `\r\n` (Windows). No obstante, en Python, cuando escribimos o leemos el carácter `\n` en un fichero de texto, el propio lenguaje se encarga de convertir dicho carácter al correspondiente por el sistema operativo.

Por defecto, cuando se invoca a la función `open(path, modo)`, el fichero se abre en modo texto. Para abrir un fichero en modo binario, hay que añadir el carácter `b` al argumento `modo`.

A continuación, te muestro los diferentes modos en los que se puede abrir un fichero:

Modo	Descripción
`r`	Solo lectura. El fichero solo se puede leer. Es el modo por defecto si no se indica nada.
`w`	Solo escritura. En el fichero solo se puede escribir. Si ya existe el fichero, machaca su contenido.
`a`	Adición. En el fichero solo se puede escribir. Si ya existe el fichero, todo lo que se escriba se añadirá al final del mismo.
`x`	Como `'w'` pero si existe el fichero lanza una excepción.
`r+`	Lectura y escritura. El fichero se puede leer y escribir. La escritura comienza en el comienzo del fichero.
`w+`	Como `'r+'` pero machaca el contenido previo.

Como te he indicado, todos estos modos abren el fichero en modo texto. Su versión correspondiente para abrir el fichero en modo binario sería `rb`, `wb`, `ab`, `xb`, `rb+` y `wb+`.

Una vez que hayas terminado de trabajar con un archivo, debes cerrarlo nuevamente para liberar los recursos utilizados llamando al método `close()`.

Leer un fichero

A continuación, te voy a mostrar la estructura que se sigue para leer el contenido de un fichero:

```
f = open('hola.txt', 'r')
try:
    # Leer fichero
finally:
    f.close()
```

Una cosa a tener en cuenta es que la clase `file` es implementada como un *manejador de contexto*. Veremos en detalle qué es un manejador de contexto y cómo implementarlos al final del curso. No obstante, por ahora, piensa en un manejador de contexto como un objeto que adquiere y libera recursos al entrar y salir de un bloque de código, respectivamente.

El modo de indicar a un manejador de contexto que se entra y/o sale de un bloque de código es por medio de la sentencia `with`.

Como te decía, por el momento no te preocupes por esto. Simplemente, ten en cuenta que la estructura presentada anteriormente para leer un fichero se puede simplificar de este modo:

```
with open('hola.txt', 'r') as f:
   # Leer fichero
```

Al abrir el fichero dentro de la sentencia `with`, este se cerrará automáticamente cuando se salga de este bloque de instrucciones, incluso en caso de que se produzca alguna excepción.

IMPORTANTE: Si el fichero o ruta proporcionada no existen, la función `open()` lanzará la excepción `FileNotFoundError`.

Métodos para leer el contenido de un fichero

A continuación, te presento los diferentes métodos que existen para leer el contenido de un fichero.

Para los ejemplos, imagina un fichero `f` con el siguiente contenido:

```
Primera línea
segunda, línea
\n
tercera línea.
\n
```

read()

`f.read(n)` lee `n` caracteres del fichero y los devuelve como un string si el fichero es de tipo texto. Si el fichero es binario, lee y devuelve `n` bytes.
En caso de que no se proporcione el argumento `n`, se lee el fichero completo. Lleva cuidado en este caso porque si el archivo es muy grande, todo el contenido se lleva a memoria, lo que puede provocar errores.

```
>>> f.read(4)
'Prim'
```

readline()

Lee una línea del fichero. Cada vez que se llama a este método se lee una línea nueva que se devuelve como una cadena de caracteres.

Ten en cuenta que las líneas están delimitadas por el carácter `\n`, el cuál, es devuelto en cada invocación al método.

Cuando se alcanza el final del archivo, `readline()` devuelve una cadena vacía. También se devuelve una cadena vacía si la última línea contiene solo caracteres en blanco (`\n`, `\t`, `espacio`, ...).

```
>>> f.readline()
'Primera línea\n'
>>> f.readline()
'segunda, línea\n'
>>> f.readline()
'\n'
```

No obstante, si lo que pretendes es leer un fichero línea a línea, lo mejor es utilizar un bucle `for`. Este modo es más rápido, más eficiente en términos de memoria y el código resultante resulta mucho más simple:

```
>>> for linea in f:
...     print(linea)
```

Un truco para eliminar el carácter de fin de línea, es usar el método `rstrip()` de la clase string.

```
>>> for linea in f:
...     print(linea.rstrip())
```

readlines()

Por último, el método `readlines()` lee todo el fichero y devuelve una lista formada por cada una de las líneas del mismo.

```
>>> lineas = f.readlines()
>>> lineas[1]
'segunda, línea\n'
```

Escribir en un fichero

Para escribir en un fichero se utiliza el método `write()`. En caso de tratarse de un fichero de texto, a `write()` hay que pasarle como argumento una cadena de caracteres. Si es un fichero binario, hay que pasar una secuencia de bytes.

El método devuelve el número de elementos (caracteres o bytes) escritos en el fichero.

```
>>> f = open('ejemplo.txt', 'w')
>>> f.write('Hola\nEsto es ')
13
>>> f.write('un fichero de ejemplo')
21
>>> f.close()
```

Manejando la posición del fichero

En cualquier objeto de tipo file, es posible controlar la posición a partir de la cuál se lee o escribe en un fichero.

Para ello, hay que hacer uso de los métodos `tell()` y `seek()`.

`tell()` devuelve un número entero que indica la posición actual del objeto `file` en el archivo representado como:

- El número de bytes desde el comienzo del archivo cuando está en modo binario.
- El número de caracteres leídos cuando está en modo de texto.

`seek()` se utiliza para mover la posición actual del objeto `file` y siempre toma un punto de referencia.

Para ficheros de tipo texto o binario, el punto de referencia por defecto es desde el comienzo del archivo.

Sin embargo, para ficheros de tipo binario, esa referencia puede ser, además, desde la posición actual o desde el final del fichero.

No obstante, cualquier fichero puede colocar su posición al final llamando a `seek()` del siguiente modo:

```
f.seek(0, 2)
```

Veamos un ejemplo de uso teniendo en cuenta el fichero `ejemplo.txt` que creamos en la sección anterior:

```
>>> f = open('ejemplo.txt', 'r+')
>>> f.tell()
0
>>> f.read(5)
'Hola\n'
>>> f.seek(2)
2
>>> f.read(4)
'la\nE'
>>> f.tell()
6
```

Serializar objetos con pickle

En esta sección te voy a explicar cómo puedes serializar un objeto para guardar su estado en un fichero, de manera que pueda ser restaurado posteriormente.

Esto te permite, por ejemplo, guardar el estado de la partida en un juego, guardar la configuración de una aplicación o, simplemente, guardar datos, sin tener que usar una base de datos.

Para ello, vamos a hacer uso del módulo `pickle`, que implementa diferentes protocolos para *serializar* y *deserializar* objetos Python.

Serializar es el proceso mediante el cual una jerarquía de objetos se convierte en una secuencia de bytes (en Python, este proceso es conocido como *Pickling*). Deserializar (*unpickling*) es la operación inversa, mediante la cual una secuencia de bytes (de un archivo binario u objeto similar a bytes) se convierte nuevamente en una jerarquía de objetos.

Como te decía, el módulo `pickle` usa diferentes protocolos para realizar la serialización. Un protocolo define el formato en el que los datos son serializados.

Actualmente existen 5 protocolos, algunos de ellos son antiguos y solo dan soporte a versiones de Python 2. Los más actuales son el protocolo versión 4 y el protocolo versión 5. Ambos tienen soporte para Python 3 (el protocolo versión 4 fue introducido en Python 3.4 y es el que se usa por defecto actualmente).

¿Qué se puede serializar?

Antes de entrar en detalle en cómo serializar/deserializar objetos en Python, has de tener en cuenta que no es posible serializar cualquier objeto. A continuación, te muestro un listado de los objetos que sí pueden ser serializados:

- `None`, `True` y `False`.
- Enteros, números de coma flotante y números complejos.
- Cadenas, bytes, bytearrays.
- Tuplas, listas, conjuntos y diccionarios que contienen solo objetos serializables.
- Funciones definidas en el nivel superior de un módulo (usando `def`, no `lambda`).
- Clases definidas en el nivel superior de un módulo.
- Instancias de dichas clases tales que `__dict__` o el resultado de llamar a `__getstate__()` sea serializable.

IMPORTANTE: Si se intenta serializar un objeto que no es serializable, el intérprete lanzará la excepción PicklingError.

Cómo serializar/deserializar un objeto

Para serializar un objeto simplemente hay que llamar a la función `dump(obj, file, protocol=None, …)` del módulo `pickle`.

Los objetos que se han guardado en un archivo con `pickle.dump()` se pueden volver a utilizar en un programa utilizando la función `pickle.load(file)`. `load()` reconoce automáticamente qué formato se ha utilizado para escribir los datos.

```
import pickle

cities = ["Madrid", "Ciudad de México",
          "Buenos Aires", "Santiago"]
with open("data.pkl", "wb") as f:
    pickle.dump(cities, f)
```

Python puede volver a leer el fichero `data.pkl` en la misma sesión, en otra distinta o, simplemente, en un programa diferente:

```
import pickle

with open("data.pkl", "rb") as f:
    ciudades = pickle.load(f)
print(villes)
```

En el ejemplo anterior hemos serializado solo un objeto, es decir, una lista de capitales. Pero ¿cómo se pueden serializar múltiples objetos? La solución es fácil: hay que empaquetar los objetos en otro objeto, por lo que solo habría que serializar este último. En el siguiente ejemplo, vamos a ver cómo serializar las listas "Países" y "capitales":

```
import pickle

countries = ["España", "Mexico",
             "Argentina", "Chile"]
cities = ["Madrid", "Ciudad de México",
          "Buenos Aires", "Santiago"]
pickle_object = (countries, cities)
with open('data.pkl', 'wb') as f:
    pickle.dump(pickle_object, f)
```

Para recuperar de nuevo cada una de las listas, debemos deserializar el fichero `data.pkl` del siguiente modo:

```
import pickle

with open('data.pkl', 'rb') as f:
    (countries, cities) = pickle.load(f)
```