Talento Digital para Chile:

Módulo 3
Programación Avanzada en Python

nodovirtual.awakelab.cl

jelou futurejob by Adalid Chile

3.6.- Contenido 6: Codificar un programa que lee y escribe archivos utilizando el lenguaje Python para resolver un problema

Objetivo de la jornada

1. Reconoce los comandos y operaciones básicas para el manejo de archivos para resolver un problema acorde al lenguaje Pyhton
2. Crea un programa Python para lectura de datos de un archivo externo
3. Crea un programa Python para escritura de datos en un archivo externo

3.6.1.- Manejando archivos con Python

3.6.1.1. Por qué necesitamos manejar archivos

El almacenamiento permanente de datos, en medios externos a la memoria del sistema, se realiza de forma estructurada en unidades básicas de almacenamiento manejadas por los sistemas operativos a las que llamamos archivos. Tarde o temprano, en desarrollo de software necesitamos trabajar con archivos, 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 archivos. Python incluye en su librería estándar, diversas herramientas para leer y escribir datos desde/a archivos.

Dentro de manejo de archivos, existe un concepto importante que se denomina serialización, el cual está directamente relacionado con los archivos. Python nos ofrece herramientas de la librería estándar para serializar datos en diversos formatos y, por lo tanto, archivos que puede leer o generar.

En término de formatos, CSV es uno de los más sencillos y populares para guardar información estructurada en archivos. Aplicaciones como MS Office y LibreOffice pueden exportar e importar datos en formato CSV, de forma que si necesitamos desarrollar una aplicación que trabaje con datos desde archivos, es muy probable que contemos con ellos en este formato.

Existen otros formatos para serializar, entre los cuales se cuentan: 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 su configuración.

3.6.1.2. Qué es un File Descriptor

Los descriptores de archivos (File Descriptors) son una función de bajo nivel para trabajar con archivos proporcionados directamente por el kernel del Sistema Operativo. Un descriptor de archivo es un número entero que identifica el archivo abierto en una tabla de archivos abiertos que mantiene el núcleo para cada proceso. Varias llamadas al sistema aceptan descriptores de archivos, pero no es conveniente trabajar con ellos, por lo general requieren búferes de ancho fijo, múltiples reintentos en ciertas condiciones y manejo manual de errores.

Los objetos de archivo son clases de Python que manejan los descriptores de archivo del Sistema Operativo para que trabajar con archivos sea más conveniente y menos propenso a errores. Por ejemplo, brindan manejo de errores, almacenamiento en búfer, lectura línea por línea y se cierran cuando se recolecta basura.

Python incorpora en su librería estándar una estructura de datos específica para trabajar con archivos. Básicamente, esta estructura referencia al archivo físico del sistema operativo. Dado que el intérprete de Python es multiplataforma, el manejo interno y a bajo nivel que se realiza del archivo es transparente para el programador. Entre las operaciones que Python permite realizar con archivos encontramos las más básicas que son: A**pertura**, **Creación**, **Lectura** y **Escritura de datos**.

3.6.1.3. Creación y Apertura de Archivos

La principal función que debemos conocer cuando trabajamos con archivos 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 archivo 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 archivos. El otro parámetro referencia el **modo** en el que va a ser abierto el archivo. Es importante tener en cuenta que Python diferencia entre dos

tipos de archivos, 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 archivo es un tipo u otro. De esta forma, si el modo del archivo 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 Visual Studio Code y añadiremos una serie de líneas de texto y lo guardaremos, por ejemplo, con el nombre **archivo.txt**. Seguidamente, ejecutaremos el intérprete de Python desde la interfaz de comandos y escribiremos la siguiente línea:

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

En nuestro ejemplo, hemos supuesto que el archivo 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 archivo. Por otro lado, no hemos indicado ningún modo, ya que, por defecto, Python emplea el valor **r** para archivos 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 archivos. Por ejemplo, supongamos que nuestro archivo se encuentra en un directorio llamado pruebas. Si estamos trabajando en Windows, nuestra línea de código para abrir el archivo sería la siguiente:

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

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

```
>>> fich = open("pruebas/archivo.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 varios strings empleando el separador adecuado para cada sistema operativo. De esta forma, sería más práctico escribir el código anterior para la apertura del archivo de la siguiente forma:

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

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

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

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 archivo permanecerá abierto, aunque explícitamente invoquemos al método **close()** para cerrar el archivo.

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

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

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

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

3.6.1.3. Lectura y Escritura de Archivos

La operación básica de escritura en un archivo se hace en Python a través del método **write()**. Si estamos trabajando con un archivo de texto, pasaremos una cadena de texto a este método como argumento principal. Supongamos que vamos a crear un nuevo archivo 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 caracter "**\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 archivo para que el intérprete vuelque el contenido al archivo físico y se pueda también cerrar su descriptor. Si necesitamos volcar texto antes de cerrar el archivo, 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 archivo.
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 archivo, es hora de pasar a la operación inversa, es decir, de la lectura desde el archivo. Para ello, Python nos ofrece tres métodos diferentes. El primero de ellos es **read()**, el cual lee el contenido de todo el archivo y devuelve un único string. El segundo en cuestión es **readline()** que se ocupa de leer línea a línea del archivo.

Por último, contamos con **readlines()** que devuelve una lista donde cada elemento corresponde a cada línea que contiene el archivo. Los siguientes ejemplos muestran cómo utilizar estos métodos y su resultado sobre el archivo 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'
```

seek() es otro de los métodos que podemos usar sobre el objeto de Python que maneja archivos y que sirve para posicionar el puntero de avance sobre un punto determinado. En nuestro ejemplo, y dado que deseamos volver al principio del archivo, 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 archivo.

Para leer todas las líneas de un archivo 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 archivo:

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

Hasta el momento, nuestros ejemplos se han referido en exclusiva a archivos de texto, pero, obviamente, es posible crear, abrir, leer y escribir en archivos binarios. La forma de llevar a cabo estas operaciones es similar a como hemos visto previamente para los archivos de texto. Por ejemplo, podemos crear un archivo binario que solo contiene una secuencia de bytes, en concreto, vamos a escribir en el archivo tres bytes representados por tres números en hexadecimal.

El código necesario para ello sería el siguiente:

```
>>> fich = open("test.bin", "bw")
<_io.BufferedWriter name='test.bin'>
>>> fich.write(b"\x33\xFA\x1E")
3
>>> fich.close()
```

El número 3 que el intérprete devuelve al escribir nuestra secuencia de bytes corresponde, efectivamente, a los tres bytes que hemos escrito en el archivo.

Recordemos, que cada número en formato hexadecimal ocupa justamente un byte. Para leer el contenido que acabamos de escribir, volveremos a abrir el archivo, esta vez en modo de solo lectura:

```
>>> fich = open("test.bin", "br")
>>> fich.read(3)
b'\x33\xFA\x1E'
```

En este caso, el parámetro pasado al método r**ead()** indica el número de bytes que deseamos leer. Si hubiéramos utilizado 1 en lugar de 3, entonces, el resultado hubiera sido **b'\x33'**. El archivo que hemos creado puede ser leído por un editor que soporte la lectura y edición de este tipo de archivos.
El método **seek()** suele ser utilizado con frecuencia para desplazarse por un archivo binario, a diferencia de los de texto, que habitualmente suelen ser leídos línea a línea. El mencionado método soporta dos tipos de parámetros diferentes, el primero de ellos indica el número de bytes que debe moverse el puntero interno de serialización del archivo y, el segundo, marca el punto de referencia desde el que debe ser movido dicho puntero. Este segundo parámetro puede tomar tres valores: **0 (comienzo del archivo)**, **1 (posición actual)** y **2 (fin de archivo)**. Por defecto, si no se indica este parámetro, el valor es **0**. De esta forma, para leer el último byte de nuestro archivo, emplearíamos las siguientes sentencias:

```
>>> fich.seek(-1, 2)
2
>>> fich.read(1)
b'\xle'
```

Cuando se emplea como punto de referencia el final del archivo, se deben indicar valores negativos para el desplazamiento, tal y como habremos podido comprobar en el ejemplo anterior de código.

Relacionado con la creación, lectura y escritura de archivos binarios se encuentra el concepto de serialización de objetos del que nos ocuparemos en el siguiente apartado.

3.6.1.4. Serialización.

La serialización es un proceso mediante el cual una estructura de datos es codificada de un modo específico para su almacenamiento, que puede ser físicamente en un archivo, en una base de datos o en un buffer de memoria. El propósito de este proceso, además del almacenamiento propiamente dicho, suele ser el transporte de datos a través de una red o, simplemente para crear una copia exacta de un objeto determinado. En computación distribuida es muy común serializar objetos para su transporte entre diferentes máquinas, también es habitual emplear la serialización en protocolos como CORBA (Common Object Request Broker Architecture), que permite invocar métodos y funciones escritos en un lenguaje determinado desde

otro diferente. En general, el término de serialización es conocido como marshalling en el ámbito de las ciencias de la computación.

Diferentes lenguajes de programación emplean diferentes algoritmos para serializar. Python soporta la serialización de objetos, a través de dos módulos diferentes de su librería estándar: **pickle** y **marshal**. Dado que los algoritmos empleados por estos módulos son diferentes, es necesario escoger uno de ellos a la hora de serializar datos en Python. Al proceso de convertir un objeto cualquiera en un conjunto de bytes es llamado pickling en Python, siendo el proceso inverso llamado unpickling. Es por ello, que habitualmente, el módulo **pickle** es el más utilizado para la serialización de objetos en Python.

Debemos tener en cuenta que el módulo **marshal** serializa de una forma que no es compatible entre Python 2.x y Python 3; sin embargo, **pickle** nos garantiza la portabilidad del código entre versiones del intérprete. Por otro lado, **marshal** no puede ser utilizado para serializar las clases propias definidas por el programador, mientras que esto sí que es posible con **pickle**. Dado que el formato en el que son serializados los datos, a través de **pickle**, es específico de Python, no es posible des.serializar aquellos objetos serializados con este módulo empleando otros lenguajes de programación.

La versión 3.8 de Python incluye seis protocolos diferentes con los que puede trabajar pickle. Cada uno de ellos es identificado por un número entre 0 y 5. El primero de ellos es compatible con versiones anteriores a Python3 y serializa utilizando un formato human-readable. El segundo de los protocolos emplea un formato binario y es compatible con las primeras versiones de Python. El tercero, identificado por el número 2, fue incluido por primera vez en Python 2.3 y es mucho más eficiente que sus antecesores. Existe hasta el protocolo versión 5 incluido en Python3.8. Además, el mencionado módulo incluye dos constantes diferentes: **HIGHEST_PROTOCOL**, que se identifica a la versión más alta disponible y **DEFAULT_PROTOCOL**, que actualmente (En Python3.8) está asociada al protocolo 4. Mas detalles sobre esto puede ser consultado en: https://docs.python.org/3/library/pickle.html

Respecto a los tipos de datos que pueden ser serializados en Python con **pickle**, debemos tener en cuenta que son los siguientes:

- Aquellos que toman los valores True, False y None.
- Números enteros, complejos y reales.
- Strings, bytes y listas de bytes.
- Funciones definidas en el nivel superior de un módulo.
- Funciones integradas en el intérprete que residan en el nivel superior de un módulo.
- Tuplas, listas, conjuntos y diccionarios que contengan elementos que pertenezcan a los tipos anteriores.

- Clases definidas por el programador que contengan elementos que pertenezcan a los tipos anteriores.

Como complemento a **pickle**, Python pone a nuestra disposición otro módulo llamado **pickletools**, que contiene una serie de herramientas para analizar los datos en el formato generado al serializar con **pickle**.

Ejemplo práctico

Básicamente, para serializar objetos en Python, a través del módulo **pickle**, emplearemos dos funciones diferentes: **dump()** para serializar y **load()** para la operación inversa. Para nuestro ejemplo práctico vamos a crear una clase que representa a un alumno, tal y como muestra el siguiente código:

```
>>> class Alumno:
...     def __init__(self, nombre_completo, titulación, edad):
...         self.apellidos = nombre_completo['apellidos']
...         self.nombre = nombre_completo['nombre']
...         self.titulación = titulación
...         self.edad = edad
...     def __repr__(self):
...         return repr(self.nombre, self.apellidos, self.titulación)
```

Como puede verse, vamos a emplear strings, enteros y un diccionario como tipos de datos que contendrá la Instancia de nuestra clase. Dado que todos cumplen con las condiciones para serializar, no tendremos problema para llevar a cabo este proceso. Antes de ello, crearemos una Instancia que será la serializada y, posteriormente, deserializada:

```
>>> nombre_completo = {"apellidos": "Rodríguez", "nombre": "Lucas"}
>>> alumno = Alumno(nombre_completo, "Grado en Derecho", 21)
```

El siguiente paso será llevar a cabo la serialización, previamente importaremos el módulo **pickle** y posteriormente invocaremos la función **dump()**. Los datos de nuestra instancia serán serializados en un archivo binario llamado **alumnos.bin**. Efectivamente, los archivos binarios suelen ser los empleados para llevar a cabo este proceso, de ahí la relación entre este tipo de archivos y la serialización. En cuanto al código, el siguiente nos muestra cómo crear el archivo y aplicar **dump()**:

```
>>> import pickle
>>> with open(alumnos.bin", "wb") as fich:
pickle.dump(alumno, fich)
```

Para llevar a cabo el proceso inverso, es decir, la deserializacíón, es necesario invocar al método **load()**. Así pues, y siguiendo con nuestro ejemplo, para leer los datos que acabamos de serializar, basta con ejecutar las siguientes líneas de código:

```
>>> with open(alumnos.bin", "rb") as fich:
...     pickle.load(fich)
```


Es importante tener en cuenta que cuando se emplea la función **load()** para deserializar los datos de una clase, esta debe ser accesible en el mismo ámbito.

En nuestro ejemplo y dado que hemos utilizado la consola de comandos del intérprete, cumplimos con esta condición.

3.6.1.5. Archivos CSV

El formato **CSV** (Comma Separated Values) es uno de los más comunes y sencillos para almacenar una serle de valores como si de una tabla se tratara. Cada fila se representa por una línea diferente, mientras que los valores que forman una columna aparecen separados por un caracter concreto. El más común de los caracteres empleados para esta separación es la coma, de ahí el nombre del formato. Sin embargo, es habitual encontrar otros caracteres como el signo del dólar ($) o el punto y coma (;). Gracias al formato CSV es posible guardar una serle de datos, representados por una tabla, en un simple archivo de texto. Además, software para trabajar con hojas de cálculo, como, por ejemplo, Microsoft Excel, nos permiten importar y exportar datos en este formato.
Dentro de su librería estándar, Python incorpora un módulo específico para trabajar con archivos CSV, que nos permite, tanto leer datos, como escribirlos. Son dos los métodos básicos que nos posibilitan realizar estar operaciones: **reader()** y **writer()**. El primero de ellos sirve para leer los datos contenidos en un archivo CSV, mientras que el segundo nos ayudará a la escritura.

Supongamos que tenemos un sencillo archivo CSV, llamado **empleados.csv**, que contiene las siguientes líneas:

```
Martínez,Juan,Administración,Barcelona
López,María,Finanzas.Valencia
Rodríguez,Manuel.Ventas,Granada
Rojas,Ana,Dirección,Madrid
```

Cada línea de nuestro archivo identifica a un empleado, donde, el primer valor es el primer apellido, el segundo es el departamento donde trabaja y por último, el tercer valor es la ciudad en la que se encuentra. Para leer este archivo y mostrar la información indicando el nombre de cada campo y su valor asociado, bastaría con ejecutar el siguiente código:

```
>>> import csv
>>> with open('empleados.csv') as f:
...     reader = csv.reader(f)
...     for row in reader:
...         print("Apellido: {0}; Nombre: {1}; Departamento: {2};
Ciudad: {3}".format(row[0], row[1], row[2], row[3]))
...
```

```
Apellido: Martínez; Nombre: Juan; Departamento: Administración;
Ciudad: Barcelona
Apellido: López; Nombre: María; Departamento: Finanzas; Ciudad:
Valencia
Apellido: Rodríguez; Nombre: Manuel; Departamento: Ventas; Ciudad:
Granada
Apellido: Rojas; Nombre: Ana; Departamento: Dirección; Ciudad: Madrid
```

Efectivamente, el método **reader()** es de tipo iterable y nos da acceso a todas las filas del archivo, es por ello, que en nuestro ejemplo, utilizamos un loop **for** para iterar sobre el objeto devuelto. Por otro lado, cada elemento iterable es una lista que contiene tantos elementos como valores separados por el caracter de separación tenga cada línea de nuestro archivo. En caso de que empleemos un caracter de separación diferente de la coma, deberemos indicarlo a través del parámetro **delimiter**. Si cambiamos nuestro archivo **empleados.csv** y reemplazamos la coma por, por ejemplo, el símbolo del dólar, tendríamos que emplear **delimiter** de la siguiente forma:

```
>>> reader = csv.reader(f, delimiter='$')
```

La operación de escritura en archivos de tipo CSV se lleva a cabo a través de los métodos **writer()** y **writerow()**. Este último escribe directamente una fila en el archivo y como argumento debemos pasar una lista donde cada elemento representa cada valor de la columna correspondiente a cada fila. Volvamos a tomar un par de líneas de nuestro ejemplo anterior y creemos un nuevo archivo a partir de ellas:

```
>>> fich = open('nuevo.csv', 'w')
>>> fich_w = csv.writer(fich, delimiter='$')
>>> empleados = [['Martínez', 'Juan', 'Administración', 'Barcelona'],
['López', 'María', 'Finanzas', 'Valencia']]
>>> for empleado in empleados:
...     fich_w.writerow(empleado)
...
>>> fich.close()
```

En esta ocasión hemos creado un nuevo archivo (**nuevo.csv**) que contiene solo los datos de un par de empleados; además, el caracter de separación es el símbolo del dólar. Si abrimos el archivo recién creado por código, veremos que tiene el siguiente contenido:

```
Martinez$Juan$Administracion$Barcelona
Lopez$Maria$Finanzas$Valencia
```

Finalizamos en este punto nuestro recorrido por el tratamiento de archivos CSV con Python. El lector interesado en descubrir más sobre el módulo csv puede echar un vistazo a la página web oficial del mismo.

3.6.1.6. Archivos XML, JSON Y YAML

Con la serialización se encuentran relacionados otra serie de formatos de archivos que suelen ser utilizados para guardar datos en un determinado formato y, que pueden compartirse entre diferentes máquinas y tratarse en distintos lenguajes de programación. Entre estos formatos, destacan tres: XML, JSON y YAML. En este apartado descubriremos cómo podemos trabajar con estos formatos desde Python.

XML

Estas tres siglas vienen de eXtensible Markup Language y referencian a un lenguaje de etiquetas desarrollado por el WC3 (World Wide Web Consortium). En realidad, este lenguaje no es más que una adaptación del original SGML (Standard Generalized Markup Language) y fue diseñado con el objetivo de tener una herramienta que fuera capaz de ayudar a definir lenguajes de marcas y etiquetas, adaptados a diferentes necesidades. Además, también puede ser utilizado como un estándar para el intercambio de datos estructurados. De hecho, es esta una de las aplicaciones más usuales del formato XML. Dado que es posible definir de forma personalizada una estructura concreta de datos, estos pueden ser almacenados en un simple archivo de texto, haciendo el mismo totalmente portable entre máquinas y lenguajes de programación. En la práctica, podemos almacenar en un archivo XML desde información relativa a una configuración específica, hasta datos que habitualmente podrían almacenarse en una base de datos. Es más, comúnmente el formato XML es empleado para serializar datos y existen multitud de librerías que permiten, dada una definición de un objeto, volcarlo directamente a un archivo en este formato. En la actualidad, XML es un formato estándar de facto, empleado por multitud de aplicaciones web y de escritorio, junto con JSON (Que trataremos en la siguiente sección).

Los datos en formato XML pueden ser accedidos a través de dos interfaces diferentes: DOM (Document Object Model) y SAX (Simple API for XML). La primera de ellas se basa en utilizar y tratar los datos basándose en una estructura de árbol, del mismo modo que hemos visto para HTML, donde se definen nodos y una jerarquía que permite acceder a los diferentes elementos que forman parte de los datos contenidos en XML. Por otro lado, SAX está orientado a eventos y permite trabajar en una sección del documento XML en lugar de hacerlo en todo el conjunto, tal y como necesita DOM.

Son muchos los lenguajes de programación que ofrecen herramientas para trabajar con XML y Python se encuentra entre ellos. En concreto, el módulo **xml** de la librería estándar pone a disposición de los programadores diferentes analizadores sintácticos para leer y escribir información en formato XML. Tres son los submódulos que permiten interactuar con XML de formas diferentes. El primero de ellos es **xml.sax.handler** que utiliza el módulo SAX para analizar sintácticamente (parsear). El segundo se denomina **xml.dom.minidom** que ofrece una implementación ligera para el modelo DOM. El último de los tres es **xml.etree.ElementTree** y básicamente

ofrece la misma funcionalidad que **xml.dom**, solo que empleando una forma específica de Python para parsear el formato.

La estructura de un documento XML puede ser muy compleja; sin embargo, para nuestros ejemplos prácticos definiremos un sencillo documento y veremos cómo puede ser accedido empleando los tres módulos diferentes que pone Python a nuestra disposición. El lector interesado en profundizar en todas y cada una de las opciones existentes de estos módulos, puede echar un vistazo a la documentación oficial de Python al respecto.

Comenzamos definiendo un documento XML con el contenido que representa a la información de varios álbumes musicales de un artista determinado:

```
<albums>
<interprete>U2</interprete>
<titulo>Achtung Baby</titulo>
<titulo>Zooropa</titulo>
<titulo>The Joshua Tree</titulo>
<genero>rock</genero>
</albums>
```

Una vez definida la estructura, podemos guardar el contenido en un nuevo archivo llamado **albums.xml.** Seguidamente, veremos cómo mostrar la información relativa a los diferentes títulos que contiene, empleando para ello los diferentes módulos de Python para análisis de XML. Comenzamos por **ElementTree**:

```
>>> from xml.etree.ElementTree import parse
>>> xml_doc = parse('albums.xml')
>>> for ele in xml_doc.findall('titulo'):
...     print(ele.text)
...
Achtung Baby
Zooropa
The Joshua Tree
```

La función **parse()** se encarga de cargar la estructura del archivo en memoria y prepararla para poder recorrerla y acceder a su información. El método **findall()** localiza todas las etiquetas que contiene el nombre indicado y, finalmente, para acceder a la información de los títulos en cuestión, utilizamos **text** que es un atributo que contiene el valor en cuestión.

De funcionalidad análoga al ejemplo anterior tenemos el siguiente código que emplea las herramientas de **minidom** para acceder e imprimir los títulos de nuestro archivo XML:

```
>>> from xml.dom.minidom import parse, Node
>>> xmltree = parse('albums.xml')
>>> for nodo in xmltree.getElementsByTagName('titulo'):
```


```
...     for nodo_hijo in nodo.childNodes:
...         if nodo_hijo.nodeType == Node.TEXT_NODE:
...             print(nodo_hijo.data)
...
Achtung Baby
Zooropa
The Joshua Tree
```

Observando el código anterior, descubriremos cómo pasando el parámetro titulo al al método **getElementsByTagName()** podemos declarar un loop para recorrer uno a uno todos los nodos que contiene que cumplen con la condición especificada. Además, anidamos otro loop para recorrer todos los hijos de estos nodos. Dentro de este último loop **for** debemos comprobar si el nodo es de tipo **texto**, en cuyo caso accederemos al atributo **data** que contiene el valor que buscamos. La constante **TEXT_NODE** representa el tipo texto que contiene una etiqueta determinada. En nuestro caso, corresponde al título en cuestión de cada disco.

El último de nuestros ejemplos muestra cómo emplear el método SAX para recorrer nuestro archivo e imprimir el título de cada álbum. Para procesar el archivo vamos a crear una clase que se encargará de responder a los diferentes eventos que se producen cuando se utiliza el parser de SAX. Así pues, nuestra clase estará definida por el siguiente código:

```
import xml.sax.handler
class AlbumSaxHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        self.in_title = False

    def startElement(self, name, attributes):
        if name == 'titulo':
            self.in title = True
    def characters(self, data):
        if self.in_title:
            print(data)

def endElement(self, name):
    if name == 'titulo':
        self.in_title = False
```

La clase **AlbumSaxHandler** hereda de una clase incluida en el módulo que Python incluye para trabajar con SAX. Los métodos **startElement()** y **endElement()** son los encargados de llevar a cabo una serie de acciones cuando se detecta el principio y el fin, respectivamente, de cada etiqueta del archivo XML. Por otro lado, **characters()** comprobará si el atributo de clase **in_title** es True, en cuyo caso imprimirá el contenido del elemento referenciado por **<titulo>**. Una vez que tenemos nuestra clase, solo nos queda invocar a **parser()**, para ello necesitaremos el siguiente código:

```
>>> import xml.sax
>>> parser = xml.sax.make_parser()
```

```
>>> sax_handler = AlbumSaxHandler()
>>> parser.setContentHandler(sax_handler)
>>> parser.parse('albums.xml')
Achtung Baby
Zooropa
The Joshua Tree
```

La elección de la técnica y módulo de Python para utilizar a la hora de parsear y trabajar con XML's depende de varios factores. Dos de los más importantes son el tamaño del archivo y el tipo de estructura que presenta. Estos tienen gran impacto sobre la capacidad de procesamiento y memoria requerida para su análisis.

JSON

El formato JSON se ha convertido en uno de los más populares para la serialización e intercambio de datos, sobre todo en aplicaciones web que utilizan AJAX. Recordemos que esta técnica permite intercambiar datos entre el navegador cliente y el servidor sin necesidad de recargar la página. JSON son las siglas en inglés de JavaScript Object Notation y fue definido dentro de uno de los estándares (ECMA-262) en los que está basado el lenguaje JavaScript. Es en este lenguaje donde, a través de una simple función **eval()**, podemos crear un objeto directamente a través de una cadena de texto en formato JSON. Este factor ha contribuido significativamente a que sean muchos los servicios web que utilizan JSON para intercambiar datos a través de AJAX, ya que el análisis y procesamiento de datos es muy rápido. Incluso, son muchas las que han sustituido el XML por el JSON a la hora de intercambiar datos entre cliente y
servidor.

Para estructurar la información que contiene el formato, se utilizan las llaves **{}** y se definen pares de nombres y valor separados entre ellos por dos puntos. De esta forma, un sencillo ejemplo en formato JSON, para almacenar la información de un empleado, sería el siguiente:

```
{"apellidos": "Fernández Rojas", "nombre": "José Luis",
"departamento": "Finanzas", "ciudad": "Madrid"}
```

JSON puede trabajar con varios tipos de datos como valores; en concreto, admite cadenas de caracteres, números, booleanos, arrays y null. La mayoría de los enguajes de programación modernos incluyen API y/o librerías que permiten trabajar con datos en formato JSON. En Python disponemos de un módulo llamado **json** que forma parte de la librería estándar del lenguaje. Básicamente, este método cuenta con dos funciones principales, una para codificar y otra para descodificar. El objetivo de ambas funciones es traducir entre una string con formato JSON y objetos de Python. Obviamente, utilizando las funciones de archivos de Python podemos leer y escribir archivos que contengan datos en este formato. No obstante, debemos tener en cuenta que las funciones contenidas en **json** no permiten trabajar directamente con archivos, sino con strings.

Volviendo a nuestro ejemplo de archivo XML, los mismos datos pueden ser codificados en formato JSON de la siguiente forma:

```
{"albums": {"titulos": ["Achtung Baby", "Zooropa", "The Joshua
Tree"], "genero": "Rock"}}
```

La anterior cadena puede ser salvada en un archivo llamado **albums.json**, siendo este el que utilizaremos para nuestros posteriores ejemplos.

Para leer los datos de nuestro nuevo archivo, basta con ejecutar las siguientes líneas de código:

```
>>> import json
>>> fich = open('albums.json')
>>> line = fich.readline()
>>> fich.close()
>>> data = json.loads(line)
```

La variable **data** contendrá un diccionario de Python que se corresponde con la estructura de nuestro archivo JSON. Así pues, para obtener los títulos de nuestros álbumes basta con ejecutar:

```
>>> data['albums'] ['titulos']
['Achtung Baby', 'Zooropa', 'The Joshua Tree']
```

Realizar el paso inverso es sencillo, partiendo del nuevo diccionario data, podemos directamente invocar a **dumps()**, pasando como argumento nuestro diccionario, de esta forma obtendremos una cadena de texto JSON, que posteriormente, puede ser guardada en un archivo. El siguiente código realizaría todas estas operaciones, incluyendo la modificación de la cadena origina, añadiendo un nuevo título:

```
>>> data['albums']['titulos'].append("Rattle and Hum")
>>> cad = json.dumps(data)
>>> new_fich = open('albums_new.json', 'w')
>>> new_fich.writeline(cad)
>>> new_fich.close()
```

Si abrimos el nuevo archivo **albums_new.json** y echamos un vistazo a su contenido, encontraremos el siguiente texto en formato JSON:

```
{"albums": {"títulos": ["Achtung Baby", "Zooropa", "The Joshua Tree",
"Rattle and Hum"], "genero": "Rock"}}
```

Además del módulo **json**, también existen otros módulos para trabajar con JSON. Uno de ellos es **simplejson**, que no forma parte de la librería estándar pero que puede ser instalado fácilmente.

YAML

El último de los formatos de archivos relacionados con la serialización de datos es YAML, acrónimo de YAML Ain't Markup Language. Este formato fue diseñado con el objetivo de disponer de un formato de texto para serializar datos que fuese sencillo y fácil de leer para las personas. A pesar de no ser tan popular como JSON y XML, es interesante conocer este formato, ya que puede ser procesado rápidamente y resulta muy fácil de leer. El conocido framework web Ruby on Rails emplea el formato YAML para definir determinados archivos de configuración, como es, por ejemplo, el que declara los datos para la conexión a las diferentes bases de datos de cada entorno.

Básicamente, YAML utiliza pares clave-valor para almacenar y estructurar los datos. Además, la indentaclón es empleada para separar datos que pertenecen a otros de una jerarquía superior. Un ejemplo de una cadena YAML podría ser la siguiente, donde se definen tres entornos diferentes y cada uno de ellos cuenta con una dirección IP y puerto diferentes:

```
desarrollo:
    IP: 127.0.0.1
    puerto: 8000
staging:
    IP: 192.168.1.2
    puerto: 8002
producción:
    IP: 192.168.1.2
    puerto: 8003
```

Python no dispone de ningún módulo en su librería estándar para trabajar con archivos YAML; sin embargo, existen varias librerías de terceros para ello. Una de las más populares es PyYAML. Su instalación puede ser llevada a cabo directamente a través de su código fuente, el cual puede ser descargado desde la correspondiente página web (https://pyyaml.org/). En cuanto descarguemos el archivo zip con el código fuente, pasaremos a descomprimirlo.

Seguidamente, desde la línea de comandos y desde el directorio creado al descomprimir, ejecutaremos el siguiente comando:

```
python setup.py install
```

En cuanto termine la ejecución del comando anterior, tendremos disponible un nuevo módulo para Python llamado **yaml**. Esto implica que, desde el intérprete de Python, podemos invocar al nuevo módulo con la siguiente sentencia:

```
>>> import yaml
```

Para la parte práctica partiremos de la cadena YAML que hemos definido previamente, creando un nuevo archivo denominado **servidores.yml**. De forma similar al módulo **json**, **yaml** cuenta con dos funciones principales, una para leer una

cadena de texto a través de un archivo y transformarla en un diccionario de Python, y otra que realiza la operación Inversa, es decir, obtiene una cadena de texto a partir de un diccionario de Python. La primera de ellas se denomina **load()** y la segunda **dump()**. Así pues, para leer nuestro archivo YAML, basta con ejecutar la siguiente línea de código:

```
>>> data = yaml.load(open('servidores.yml'))
```

La variable **data** contiene ahora un diccionario donde las claves son desarrollo, **staging** y **producción**. Además, cada una de estas claves da acceso a otro diccionario cuyas claves son **IP** y **puerto** que da a su vez acceso a los valores de nuestro archivo. Dada esta estructura, para, por ejemplo, acceder al puerto del entorno de producción, basta con ejecutar la siguiente línea:

```
>>> data['producción']['puerto']
8003
```

Por otro lado, la estructura puede ser modificada y crear con ella un nuevo archivo YAML. Supongamos que necesitamos añadir un nuevo entorno llamado pruebas, junto con los datos de un nuevo servidor. En primer lugar, crearíamos un nuevo diccionario y lo asignaríamos a data, tal y como muestra el código a continuación:

```
>>> data['pruebas'] = {'IP': '192.168.2.8', 'puerto': 8004}
```

Seguidamente, invocamos a la función **dump()** pasando como argumento nuestro diccionario original:

```
>>> yaml.dump(data)
'desarrollo: {IP: 192.168.2.8, puerto: 8004}\nproduccion: {IP:
192.168.1.2, puerto: 8003}\npruebas: {IP: 192.168.1.8, puerto:
8004}\n'\nstaging: {iP: 192.168.1.2, puerto: 8002}\n'
```

Ahora podemos guardar nuestra cadena YAML en un nuevo archivo:

```
>>> fich = open('new_servidores', 'w')
>>> fich.write(yaml.dump(data))
>>> fich.close()
```

3.6.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/