

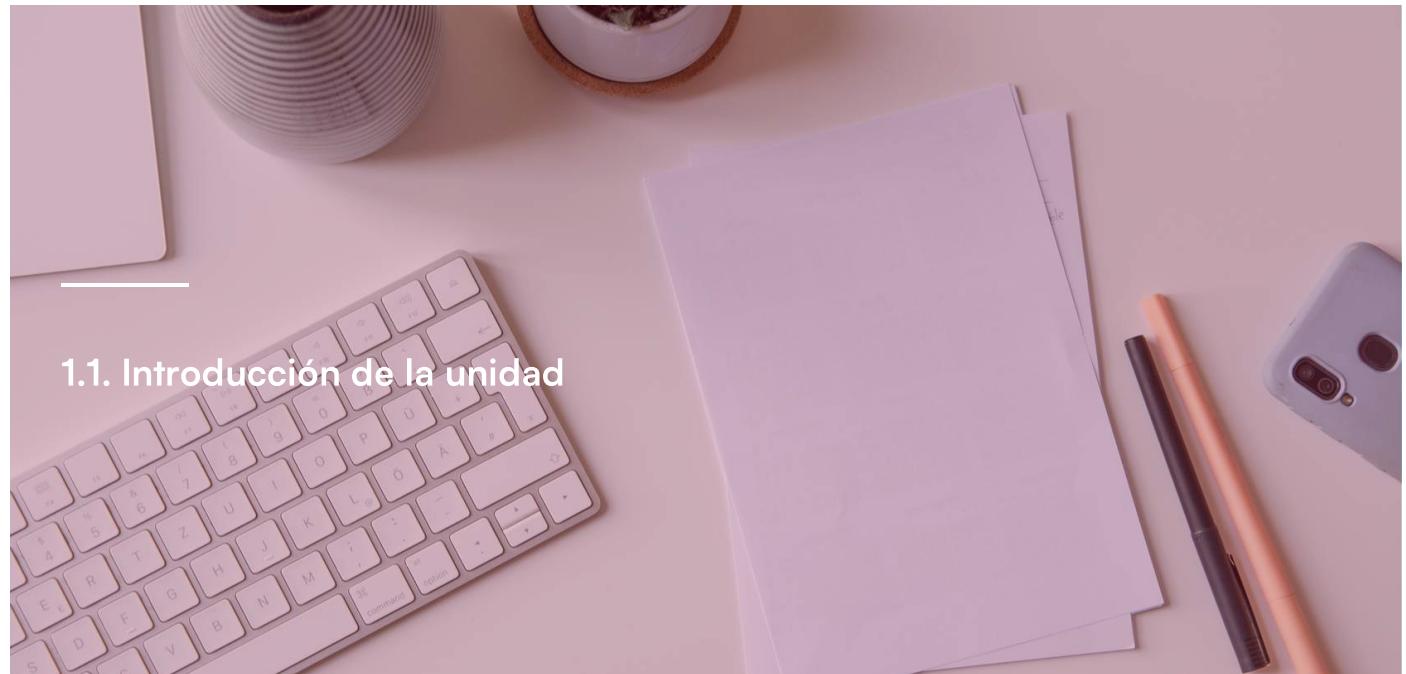


# Fundamentos de tecnología de internet



- ☰ I. Introducción
- ☰ II. Objetivos
- ☰ III. Formatos de almacenamiento de datos en internet
- ☰ IV. Manipulación de documentos CSV
- ☰ V. Manipulación de documentos JSON
- ☰ VI. Manipulación de documentos XML
- ☰ VII. Resumen
- ☰ VIII. Caso práctico con solución
- ☰ IX. Lecturas recomendadas
- ☰ X. Glosario
- ☰ XI. Bibliografía

# I. Introducción



## 1.1. Introducción de la unidad

Uno de los principales problemas con los que un analista o científico de datos se va a encontrar es **la abundante diversidad de formatos** en los que un dato puede existir.

### Recuerda:

Como se ha visto anteriormente, **los datos residen en bases de datos, donde se almacenan en formatos que facilitan su extracción mediante un lenguaje de consulta**. Cuando se utiliza un lenguaje de consulta popular como SQL, se tiene la certeza de que los datos seguirán un formato tabular muy similar entre tablas distintas.

Sin embargo, acceder directamente a una base de datos no siempre está al alcance de cualquiera, sobre todo si se está tratando de extraer datos públicos o de un tercero a cuyo sistema no se tiene acceso. En esos casos, la información se suele exponer a través de API (*application programming interfaces*) accesibles mediante cualquier cliente HTTP (como puede ser un navegador) y que pueden formatear estos datos de muchas maneras.

Dentro del abanico de formatos en un entorno digitalizado, se distinguen **tres grandes categorías**:

1

**Datos estructurados**, aquellos que tienen una estructura bien definida. Por ejemplo, una tabla de una BD relacional.

2

**Datos semiestructurados**, parte de su contenido tiene una estructura definida, como por ejemplo un objeto JSON.

3

**Datos no estructurados**, carecen de estructura para el ojo humano, aunque sí pueden ser interpretados por una aplicación. Por ejemplo, una imagen.

**CONTINUAR**

En el **ámbito del intercambio de datos a través de API**, algunos de los formatos más populares son los siguientes:

**CSV** —

Comma-Separated Values, es un formato tabular, fácilmente exportable a un sistema de hojas de cálculo como MS Excel.

**JSON** —

JavaScript Object Notation, es un formato habitual en el intercambio de información entre navegadores y servidores web y se ha hecho muy popular para otros ámbitos por su expresividad y su ligereza en cuanto a tamaño en bytes.

**XML** —

eXtensible Markup Language, es un lenguaje de marcas, similar a HTML (HyperText Markup Language), utilizado para componer la mayoría de las páginas web actuales.

A lo largo de esta unidad se analizarán estos tres formatos y se verá qué herramientas ofrece Python para trabajar con ellos.

-  Los ejercicios se incluyen en cuadernos de Jupyter Notebook (ficheros .ipynb), se recomienda descargar y ejecutar cada uno de ellos. Basta con iniciar el servicio Jupyter Notebook e importar el fichero .ipynb desde la opción *Upload*.

En el siguiente enlace, se facilita un archivo comprimido que contiene todos los ficheros (*notebooks* o cuadernos) que se necesitarán en esta unidad:



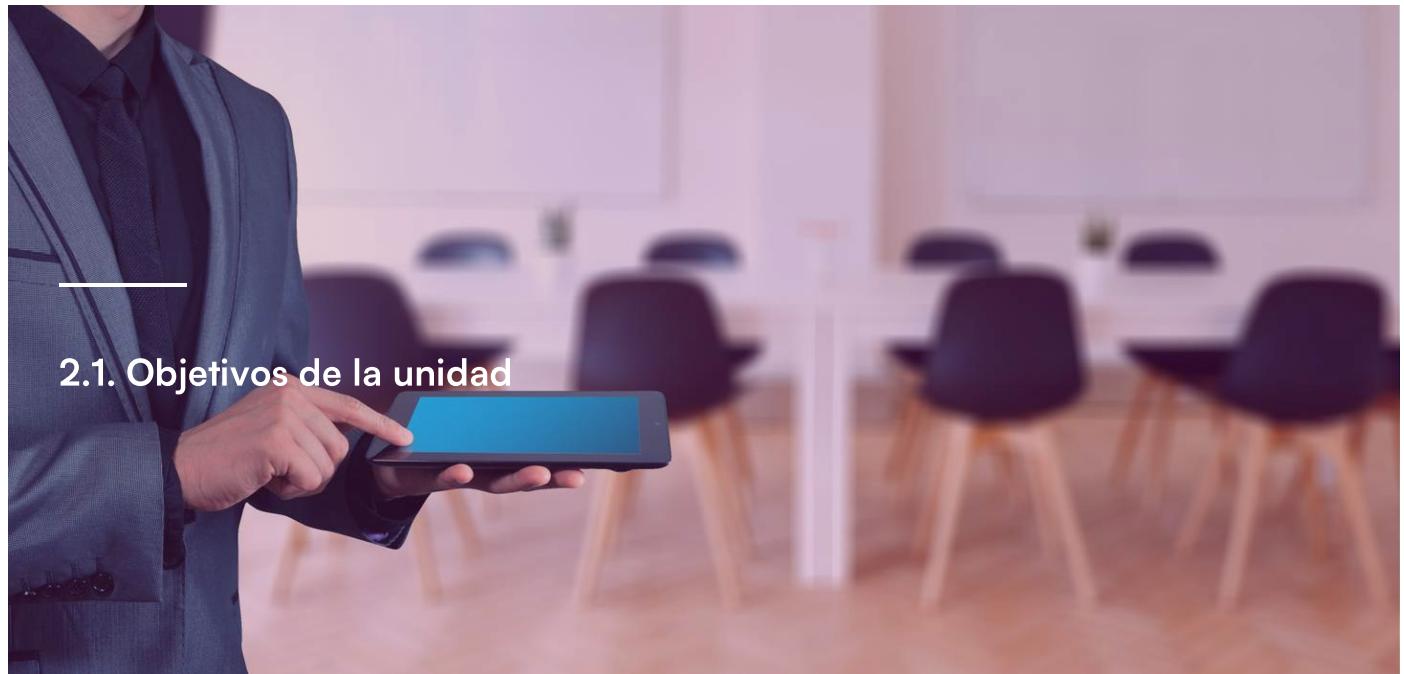
[ejercicios\\_csv\\_json\\_xml.zip](#)

373.5 KB



## II. Objetivos

---



### 2.1. Objetivos de la unidad

Los objetivos de esta unidad son:

1

Conocer los principales formatos de almacenamiento de datos.

2

Conocer los métodos definidos en Python para procesar cada tipo de formato de datos.

3

Entender la información codificada en cada formato de datos.

4

Saber codificar y diseñar un documento en cada formato para almacenar datos.

5

Saber valorar las ventajas y desventajas de usar un formato de datos.

### III. Formatos de almacenamiento de datos en internet

---

Se van a estudiar tres formatos muy utilizados para intercambiar y almacenar datos en la web:

- Archivos **CSV**.
- Documentos **JSON**.
- Documentos **XML**.

## IV. Manipulación de documentos CSV

Un archivo CSV (Comma Separated Values) es un archivo de texto plano que almacena los valores separados por comas. Los archivos se encuentran estructurados por líneas y cada línea es un conjunto de valores separados por comas. En la figura 1 se muestra un ejemplo.

04/05/2015,13:34,Manzanas,73
04/05/2015,3:41,Cerezas,85
04/06/2015,12:46,Peras,14
04/08/2015,8:59,Naranjas,52
04/10/2015,2:07,Manzanas,152
04/10/2015,18:10,Platanos,23
04/10/2015,2:40,Fresas,98

Figura 1. Ejemplo de documento CSV.

Fuente: elaboración propia.

### CARACTERÍSTICAS

### VENTAJAS

Algunas características de los archivos CSV:

- Los valores no tienen tipos, todos son cadenas.
- No tienen atributos de configuración acerca del tamaño de la fuente, color.
- No tienen imágenes o dibujos embebidos.
- Los archivos tienen extensión .csv

## CARACTERÍSTICAS

## VENTAJAS

Las principales ventajas que ofrece este formato son:

- Es simple.
- Permite representar los datos de las hojas de cálculo.
- Puede visualizarse en editores de texto.

**CONTINUAR**

Dado que los archivos CSV son archivos de texto, se podría intentar leer como una cadena y posteriormente procesarla. En este sentido, como los valores están delimitados por comas, se podría usar el método *split()* (para cadenas en lenguaje Python) sobre cada línea para obtener los valores.

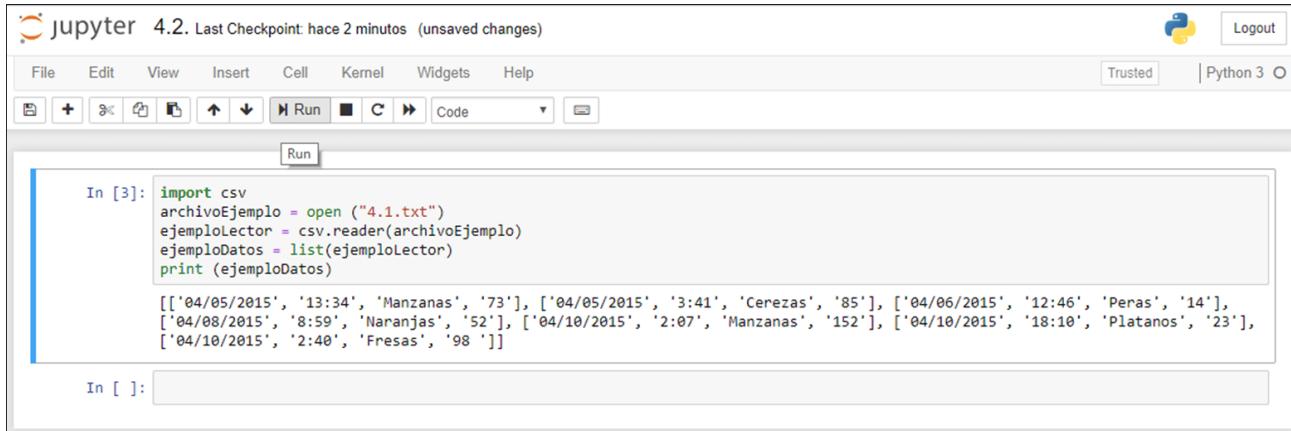
Sin embargo, no siempre las comas representan los límites de un valor, ya que los archivos CSV también tienen su propio conjunto de caracteres de escape para permitir que las comas y otros caracteres formen parte de un valor, y esos caracteres no son soportados por *split()*.

El módulo CSV de Python permite leer y escribir archivos CSV<sup>1</sup>. Para leer datos de un archivo de este tipo, en primer lugar hay que crear un objeto *Reader*. Este objeto permite iterar sobre las líneas del archivo CSV. En la figura 2 se muestra un ejemplo. En el notebook "Ejemplo\_1.ipynb" se muestra la ejecución del programa:

<sup>1</sup>"[CSV File Reading and Writing](#)". Python.

```
import csv
archivoEjemplo = open ("4.1.txt")
ejemploLector = csv.reader(archivoEjemplo)
```

```
ejemploDatos = list(ejemploLector)
print (ejemploDatos)
```



The screenshot shows a Jupyter Notebook interface. At the top, there's a header bar with the title "Jupyter 4.2. Last Checkpoint: hace 2 minutos (unsaved changes)", a Python logo icon, and a "Logout" button. Below the header is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". To the right of the menu are "Trusted" and "Python 3" buttons. A toolbar below the menu includes icons for file operations like new, open, save, and run, along with a "Code" dropdown. The main area is titled "Run". It contains two code cells. The first cell, labeled "In [3]:", contains Python code for reading a CSV file named "4.1.txt" using the csv module. The second cell, labeled "In [ ]:", is empty. The output of the first cell is a list of lists representing the CSV data.

```
In [3]: import csv
archivoEjemplo = open ("4.1.txt")
ejemploLector = csv.reader(archivoEjemplo)
ejemploDatos = list(ejemploLector)
print (ejemploDatos)

[['04/05/2015', '13:34', 'Manzanas', '73'], ['04/05/2015', '3:41', 'Cerezas', '85'], ['04/06/2015', '12:46', 'Peras', '14'], ['04/08/2015', '8:59', 'Naranjas', '52'], ['04/10/2015', '2:07', 'Manzanas', '152'], ['04/10/2015', '18:10', 'Platanos', '23'], ['04/10/2015', '2:40', 'Fresas', '98']]
```

**Figura 2.** Ejemplo de lectura de documento CSV.

Fuente: elaboración propia.

**CONTINUAR**

En el ejemplo, para leer el archivo CSV, este se abre usando la función `open()`, como si fuera un archivo de texto normal, pero en vez de usar los métodos `read()` o `readlines()` se usa la función `csv.reader()`. Esta función devuelve un objeto de tipo Reader que puede usarse para leer el archivo.

Obsérvese que a la función `csv.reader()` no se le pasa directamente el nombre de un archivo.

Una vez que se dispone del objeto `Reader`, para acceder a los valores, se puede convertir en una lista usando el método `list()`. Este método retorna una lista de listas.

Una vez que se tiene almacenado el archivo CSV como una lista de listas, se puede acceder a un valor concreto mediante indexación sobre la lista: `ejemploDatos[x][y]` donde `x` representa una lista de listas e `y` representa el índice del elemento de esa lista al que se quiere acceder. En el notebook "Ejemplo\_2.ipynb" se muestra la ejecución del programa:

```
print (ejemploDatos[0][0])
print (ejemploDatos[0][1])
print (ejemploDatos[0][2])
print (ejemploDatos[1][1])
print (ejemploDatos[6][1])
```

The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** Jupyter 4.2. Last Checkpoint: hace 3 minutos (unsaved changes)
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Run, Stop, Cell, Kernel, Help, Trusted, Python 3.
- Code Cell (In [4]):**

```
import csv
archivoEjemplo = open ("4.1.txt")
ejemploLector = csv.reader(archivoEjemplo)
ejemploDatos = list(ejemploLector)
print (ejemploDatos[0][0])
print (ejemploDatos[0][1])
print (ejemploDatos[0][2])
print (ejemploDatos[1][1])
print (ejemploDatos[6][1])
```
- Output:**

```
04/05/2015
13:34
Manzanas
3:41
2:40
```
- Empty Cell (In [ ]):** A placeholder for the next code cell.

**Figura 3.** Acceso a los datos.

Fuente: elaboración propia.

CONTINUAR

También es posible usar el objeto *Reader* en un bucle *for* (figura 4), de forma que se itera sobre las líneas del objeto. Cada línea es una lista de valores. En el ejemplo 3.a del notebook “Ejemplo\_3.ipynb” se muestra la ejecución del programa:

```
import csv
archivoEjemplo = open ("4.1.txt")
ejemploLector = csv.reader(archivoEjemplo)
for linea in ejemploLector:
    print ('Línea #',str(ejemploLector.line_num)+" "+str(linea))
```

Jupyter 4.2. Last Checkpoint hace 4 minutos (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3 Logout

In [5]:

```
import csv
archivoEjemplo = open ("4.1.txt")
ejemploLector = csv.reader(archivoEjemplo)
for linea in ejemploLector:
    print ('Línea #',str(ejemploLector.line_num)+" "+str(linea))
```

Línea # 1 ['04/05/2015', '13:34', 'Manzanas', '73']
Línea # 2 ['04/05/2015', '3:41', 'Cerezas', '85']
Línea # 3 ['04/06/2015', '12:46', 'Peras', '14']
Línea # 4 ['04/08/2015', '8:59', 'Naranjas', '52']
Línea # 5 ['04/10/2015', '2:07', 'Manzanas', '152']
Línea # 6 ['04/10/2015', '18:10', 'Platanos', '23']
Línea # 7 ['04/10/2015', '2:40', 'Fresas', '98 ']

In [ ]:

**Figura 4.** Acceso a los datos usando un *for*.

Fuente: elaboración propia.

Mediante *print* se imprime la línea actual y su contenido. Para conseguir la línea actual, se usa el atributo *line\_num* del objeto *Reader* que contiene el número de la línea actual.

## Ten en cuenta:

El objeto *Reader* solo puede recorrerse una única vez, de forma que si se quiere volver a hacerlo habría que usar nuevamente el método *csv.reader*.

Para escribir datos en un archivo CSV, se utiliza un objeto *Writer* que puede construirse usando el método *csv.writer()*, tal como se ilustra en la figura 5. En el apartado 4.a del notebook "Ejemplo\_4.ipynb" se muestra la ejecución del programa:

```
import csv
archivoSalida = open ("Salida.csv", 'w')
salidaEscritor =
csv.writer(archivoSalida)
salidaEscritor.writerow(["naranjas","limones","peras","uvas"])
salidaEscritor.writerow(["jamon","chorizo","queso","salchichón"])
salidaEscritor.writerow([1,3,4,6])
archivoSalida.close()
```

```
1 naranjas,limones,peras,uvas
2 jamon,chorizo,queso,salchichón
3 1,3,4,6
4
```

**Figura 5.** Escritura sobre un archivo CSV.  
Fuente: elaboración propia.

**CONTINUAR**

En primer lugar, se llama a *open()* con el parámetro *w*, que indica que se abre un archivo en modo escritura. Se crea un objeto *Writer* mediante el método *csv.writer()*. A continuación, se utiliza el método *writerow()* del objeto *Writer*, que toma como argumento una lista, de manera que cada valor de la lista es almacenado como un valor delimitado por comas en el archivo CSV. El valor returned por el método *writerow()* es el número de caracteres escritos en el archivo para esa lista de valores.

**i** Téngase en cuenta que, si uno de los valores contiene comas, el módulo lo gestionará como si fuera una única cadena, almacenándolo con dobles comillas (figura 6).

En el notebook "Ejemplo\_5.ipynb" se muestra la ejecución del programa:

```
import csv
archivoSalida = open("Salida1.csv", 'w')
salidaEscritor = csv.writer(archivoSalida)
salidaEscritor.writerow(["naranjas","limones","peras","uvas"])
salidaEscritor.writerow(["jamón","chorizo, de Salamanca","queso","salchichón"])
salidaEscritor.writerow([1,3,4,6])
archivoSalida.close()
```

```
1 naranjas,limones,peras,uvas
2 jamón,"chorizo, de Salamanca",queso,salchichón
3 1,3,4,6
4
5
```

Figura 6. Escritura de valores con comas.

Fuente: elaboración propia.

Otras posibilidades son, por ejemplo, **separar los valores con otro separador diferente a la coma** o que, por ejemplo, las líneas estén, a su vez, separadas por más de un espacio (figura 7). En el notebook "Ejemplo\_6.ipynb" se muestra la ejecución del programa:

```
import csv
archivoSalida = open ("Salida1.csv", 'w')
salidaEscritor = csv.writer(archivoSalida,delimiter='\t',lineterminator='\n\n')
salidaEscritor.writerow(["naranjas","limones","peras","uvas"])
salidaEscritor.writerow(["jamón","chorizo, de Salamanca","queso","salchichón"])
salidaEscritor.writerow([1,3,4,6])
archivoSalida.close()
```

```
1 naranjas      limones peras      uvas
2 jamon        chorizo, de Salamanca     queso      salchichón
3   1   3   4   6
4
```

Figura 7. Escritura de valores con otros separadores.

Fuente: elaboración propia.

## Saber más

En el ejemplo anterior, se han modificado los atributos *delimiter* (que especifica el carácter que delimita cada valor que por defecto es una coma) y *line terminator* (que especifica el carácter que va al final de cada línea que por defecto es un único salto de línea).

CONTINUAR

## CSV CON CABECERA

Es habitual que los ficheros CSV dispongan de una cabecera (la primera línea), donde se ofrece un nombre o una breve descripción sobre lo que se espera que haya en cada columna.

Por ejemplo:

```
fecha,hora,producto,cantidad
04/05/2015,13:34,Manzanas,73
04/05/2015,3:41,Cerezas,85
04/06/2015,12:46,Peras,14
04/08/2015,8:59,Naranjas,52
```

```
04/10/2015,2:07,Manzanas,152  
04/10/2015,18:10,Platanos,23  
04/10/2015,2:40,Fresas,98
```

Para este tipo de ficheros, la librería **csv** de Python ofrece mecanismos que facilitan la lectura y escritura de estos, utilizando una lista de diccionarios como estructura de datos, en lugar de una lista de listas. Se trata de las clases **DictReader** y **DictWriter**.<sup>2</sup>

<sup>2</sup>*DictReader* y *DictWriter*. "The Python Standard Library".

- i** La principal ventaja es que se puede acceder a elementos concretos de cada fila sin necesidad de conocer su posición en el fichero CSV. Basta con referenciar la clave tal y como está definida en la cabecera.

En el apartado 3.b del notebook "Ejercicio\_3.ipnb" se muestra la ejecución de un programa de lectura mediante *DictReader*, donde para cada fila se accede a los valores de las columnas **producto** y **cantidad**:

```
import csv  
with open('4.1_cabeceras.txt', newline='') as archivo_ejemplo:  
    lector = csv.DictReader(archivo_ejemplo)  
    for linea in lector:  
        print('Producto:', linea['producto'], '- Cantidad:', linea['cantidad'])
```

```
Producto: Manzanas - Cantidad: 73  
Producto: Cerezas - Cantidad: 85  
Producto: Peras - Cantidad: 14  
Producto: Naranjas - Cantidad: 52  
Producto: Manzanas - Cantidad: 152  
Producto: Platanos - Cantidad: 23  
Producto: Fresas - Cantidad: 98
```

**Figura 8.** Lectura con DictReader.  
Fuente: elaboración propia.

---

Saber más

**De la misma manera, *DictWriter* permite escribir un fichero CSV con cabecera a partir de uno o varios diccionarios de datos.**

**CONTINUAR**

En el **apartado 4.b** del notebook “Ejercicio\_4.ipnb” se muestra un ejemplo de escritura mediante *DictWriter*:

```
import csv
with open('Salida_cabecera.csv', 'w', newline='') as archivo_salida:
    cabecera = ['fecha', 'producto', 'cantidad']
    escritor = csv.DictWriter(archivo_salida, fieldnames=cabecera)
    escritor.writeheader()
    escritor.writerow({'fecha': '2021-06-08', 'producto': 'naranjas', 'cantidad': '100'})
    escritor.writerow({'fecha': '2021-07-20', 'producto': 'peras', 'cantidad': '5'})
    escritor.writerow({'fecha': '2021-08-01', 'producto': 'manzanas', 'cantidad': '200'})
    escritor.writerow({'fecha': '2021-09-30', 'producto': 'uvas', 'cantidad': '1000'})
```

**Figura 9.** Escritura con *DictWriter*.

Fuente: elaboración propia.

Se va a considerar un programa que permita:

- Encontrar todos los archivos CSV del directorio actual.
- Leer el contenido de cada archivo.
- Escribir nuevamente el contenido saltándose la primera línea sobre un nuevo archivo CSV.

Para implementarlo:



Es necesario crear un bucle sobre una lista de todos los archivos del directorio **para saltarse aquellos que no son CSV** (figura 10). En el notebook “Ejemplo\_1\_CSV.ipynb” se muestra la ejecución del programa. Se usa el método *os.listdir()*<sup>3</sup>

para recuperar todos los archivos del directorio actual y se comprueba para cada uno de ellos si su extensión es .csv. Considérense ahora los archivos del directorio “ejercicios\_csv”<sup>4</sup>:

```
import csv, os
#Se crea un directorio para almacenar los archivos sin cabecera
os.makedirs('SinCabeceras', exist_ok=True)
# Bucle para recuperar los archivos del directorio actual
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue # Saltar los archivos que no son csv
    print('Eliminando cabeceras de' + csvFilename + '...')

Eliminando cabeceras deSacramentocrimeJanuary2006.csv...
Eliminando cabeceras deSacramentoestatetransactions.csv...
Eliminando cabeceras deSalesJan2009.csv...
Eliminando cabeceras deTechCrunchcontinentalUSA.csv...
```

Figura 10. Bucle de lectura.

Fuente: elaboración propia.

<sup>3</sup>“Miscellaneous operating system interfaces”. Python.

<sup>4</sup>“Sample CSV Data”. SpatialKey Support.



Se lee el contenido de cada archivo CSV mediante un objeto *Reader*, saltándose la primera línea, y se almacena en una variable. Para controlar la primera línea se usa el atributo *line\_num* (figura 11).

```
for row in readerObj:
    if readerObj.line_num == 1:
        continue # Saltar primera linea
    csvRows.append(row)
csvFileObj.close()
```

```
Eliminando cabeceras deSacramentocrimeJanuary2006.csv...
Eliminando cabeceras deSacramentoalestatetransactions.csv...
Eliminando cabeceras deSalesJan2009.csv...
Eliminando cabeceras deSalida.csv...
Eliminando cabeceras deSalida1.csv...
Eliminando cabeceras deTechCrunchcontinentalUSA.csv...
```

**Figura 11.** Lectura del contenido.

Fuente: elaboración propia.

**CONTINUAR**

El programa completo es el que se muestra en la figura 12, y en el notebook “Ejemplo\_2\_CSV.ipynb” se muestra la ejecución del programa. Durante esta, se crea la carpeta “SinCabeceras” en el directorio actual.

```

import csv, os
#Se crea un directorio para almacenar los archivos sin cabecera
os.makedirs('SinCabeceras', exist_ok=True)
# Bucle para recuperar los archivos del directorio actual
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue # Saltar los archivos que no son csv
    print('Eliminando cabeceras de' + csvFilename + '...')
    # Leer el archivo csv y saltarse la primera linea
    csvRows = []
    csvFileObj = open(csvFilename)
    readerObj = csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue # Saltar primera linea
        csvRows.append(row)
    csvFileObj.close()
    # Escribir la salida al archivo csv
    csvFileObj = open(os.path.join('SinCabeceras', csvFilename), 'w', newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
        csvWriter.writerow(row)
    csvFileObj.close()

```

Eliminando cabeceras deSacramentocrimeJanuary2006.csv...  
 Eliminando cabeceras deSacramentoalestatetransactions.csv...  
 Eliminando cabeceras deSalesJan2009.csv...  
 Eliminando cabeceras deTechCrunchcontinentalUSA.csv...

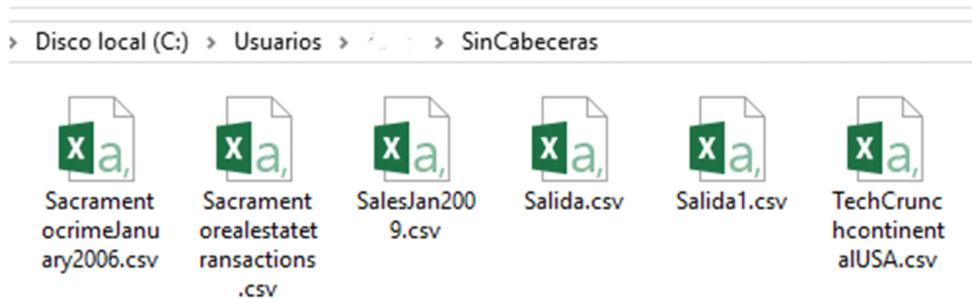


Figura 12. Ejemplo completo y nueva carpeta.  
Fuente: elaboración propia.

La **salida del programa** sería la que aparece en la figura 13:

```
Eliminando cabeceras deSacramentocrimeJanuary2006.csv...
Eliminando cabeceras deSacramentoalestatetransactions.csv...
Eliminando cabeceras deSalesJan2009.csv...
Eliminando cabeceras deSalida.csv...
Eliminando cabeceras deSalida1.csv...
Eliminando cabeceras deTechCrunchcontinentalUSA.csv...
```

Figura 13. Salida del programa.

Fuente: elaboración propia.

CONTINUAR

Un ejemplo más interesante es el siguiente: considérese un programa que retorne las direcciones del código postal que escriba el usuario. Considérese como información de entrada el archivo "SacramentocrimeJanuary2006.csv" del directorio "ejercicios\_csv".

**El programa y la salida serían lo que aparece en la figura 14**, y en el notebook "Ejemplo\_3\_CSV.ipynb" se muestra la ejecución del programa:

The screenshot shows a Jupyter Notebook interface with the following details:

- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help.
- Status Bar:** Trusted, Python 3.
- Code Cell:** In [ \* ]:

```
import csv, os
cp = {} #Diccionario de códigos postales, cp -> Llave
csvFileObj = open("C:\\\\Users\\\\Aleja\\\\Documents\\\\Info Personal\\\\Personal\\\\sigesi\\\\Work\\\\IMF - python\\\\Contenidos_M1_Big_Data\\\\Contenidos_M1_Big_Data.csv", "r")
# Leer el archivo cvs
readerObj = csv.reader(csvFileObj) |
for row in readerObj:
    if readerObj.line_num == 1:
        continue # Saltar primera linea
    lista = list(row) #convertir cada Línea en Lista
    if lista[6] in cp: #Si existe el CP en el diccionario -> concatena el valor actual con el nuevo
        cp[lista[6]] = cp[lista[6]] + "\n" + lista[1]
    else: #De lo contrario -> agrega el cp al diccionario
        cp[lista[6]] = lista[1]
csvFileObj.close()
while True:
    linea = input ("Introduce Fin para finalizar o un código postal a consultar: ")
    if (linea == "fin"):
        break
    elif linea in cp:
        print("Código postal ", linea," : ",cp[linea])
    else:
        print("El código postal ", linea," no existe en la base de datos")
```
- Output Cell:** Introduce Fin para finalizar o un código postal a consultar: fin  
El código postal fin no existe en la base de datos  
Introduce Fin para finalizar o un código postal a consultar: 2404  
Código postal 2404 : 3108 OCCIDENTAL DR  
4 PALEN CT  
47 FIRE LEAF CT  
1088 APOLLO WAY  
6571 WEATHERFORD WAY  
930 43RD AVE  
TRUXEL RD / SAGINAW CIR  
6625 VALLEY HI DR  
7769 AMHERST ST

Figura 14. Salida del programa.

Fuente: elaboración propia.



## V. Manipulación de documentos JSON

---

JSON (*JavaScript Object Notation*) es un formato de datos con las siguientes características:

- Está basado en JavaScript.
- Es independiente del lenguaje.
- Los archivos tienen extensión .json.
- Representa objetos de manera textual mediante parejas clave=valor.

### Sintaxis JSON

La sintaxis de JSON es:

- Un objeto se representa como una secuencia de parejas clave=valor encerradas entre llaves { y }.
- Las claves son cadenas de texto entre comillas "y".
- Los valores pueden ser: - Tipos básicos: cadena, número, booleano, null. - Arrays de valores: entre corchetes [ y ]. - Otros objetos JSON: entre llaves { y }.

### Ejemplo

Por ejemplo, si se quiere representar la ficha de un estudiante con sus datos personales y asignaturas matriculadas:

- Nombre="Juan Serrano Sánchez"

- DNI="4569883R"- Edad="45"

- Asignaturas matriculadas:

- Obligatorias: Álgebra, Matemáticas II, Geometría.

- Optativas: Seminario I, Seminario II, Métodos numéricos.

- Libre elección: Informática, Música.

La ficha de información se puede representar en un documento JSON de la siguiente manera:

```
{  
    "Nombre": "Juan Serrano Sanchez",  
    "DNI": "4569883R",  
    "Edad": 45,  
    "Asignaturas matriculadas":  
    {  
        "Obligatorias": ["Algebra", "Matematicas II", "Geometria"],  
        "Optativas": ["Seminario I", "Seminario II"],  
        "Libre Eleccion": ["Informatica", "Musica"]  
    }  
}
```

**CONTINUAR**

Para gestionar documentos JSON desde Python, **se usa el módulo JSON<sup>5</sup>** que permite la traducción de datos JSON en valores de Python.

<sup>5</sup> ["JSON encoder and decoder"](#). Python.

Es importante establecer de base **las siguientes convenciones** a la hora de manipular documentos JSON en Python:

- Los documentos JSON, al igual que los CSV, son eminentemente textuales. Es decir, además de tener un formato concreto, se representan y almacenan como cadenas de texto. Por tanto, cuando se habla de objetos o documentos JSON en Python, se hace referencia a cadenas de texto que contienen un documento JSON, tal que “{...}”.
- La estructura de datos Python que mejor representa un objeto JSON es el **diccionario**. Por tanto, cualquier objeto JSON se puede representar como diccionario y será la estructura de datos por defecto que la librería **JSON** utilizará en las conversiones JSON-Python.
- JSON no puede almacenar cualquier tipo de valor Python, únicamente cadenas, enteros, reales, booleanos, listas, diccionarios y el tipo None.
- JSON no puede representar objetos específicos de Python como ficheros, expresiones regulares, etc.

Para traducir una cadena que contiene datos JSON en un valor de Python, se utiliza el método *json.loads()*. En la figura 15 se muestra un ejemplo. El notebook “Ejemplo\_7.ipynb” muestra la ejecución del programa:

```
JsonDatos='{"nombre":"Sofia","matriculado":true,"asignaturas":34,"ID":null}'
import json
PythonDatos=json.loads(JsonDatos)
print (PythonDatos)
```

```
JsonDatos=' {"nombre":"Sofia","matriculado":true,"asignaturas":34,"ID":null}'
import json
PythonDatos=json.loads(JsonDatos)
print (PythonDatos)

{'nombre': 'Sofia', 'matriculado': True, 'asignaturas': 34, 'ID': None}
```

**Figura 15.** Ejemplo de lectura de cadena JSON.  
Fuente: elaboración propia.

La llamada al método *loads()* del módulo JSON permite cargar una cadena de datos JSON en valores de Python, retornando como resultado una lista donde cada elemento es un diccionario. Si se quiere acceder a los distintos elementos del diccionario, se usan los índices. La cadena JSON utiliza dobles comillas para las claves.

## Saber más

Téngase en cuenta que los valores en los diccionarios no están ordenados, por lo que los pares clave-valor pueden aparecer en orden diferente a como aparecían en la cadena original.

CONTINUAR

Para escribir un valor de Python como una cadena de datos JSON, se usa el método `json.dumps()`. En la figura 16 se muestra un ejemplo. El notebook "Ejemplo\_8.ipynb" muestra la ejecución del programa:

```
datos = {
    "nombre": "Sofia",
    "matriculado": True,
    "asignaturas": 34,
    "ID": None
}
import json
datos_json = json.dumps(datos)
print(datos_json)
```

```
datos = {
    "nombre": "Sofia",
    "matriculado": True,
    "asignaturas": 34,
    "ID": None
}
import json
datos_json = json.dumps(datos)
print(datos_json)
```

{"nombre": "Sofia", "matriculado": true, "asignaturas": 34, "ID": null}

Figura 16. Conversión de diccionario a objeto JSON.

Fuente: elaboración propia.

---

A continuación, se muestra un ejemplo de tabla de correspondencia entre los valores de Python y JSON (tabla 4.1):

<b>Python</b>	<b>JSON</b>
dict	object
list, tuple	array
str_unicode	string
int,long,float	number
True	true
False	false
None	null

Tabla 1. Tabla de correspondencia entre Python y JSON.

Fuente: elaboración propia.

The screenshot shows a web browser window with the URL `swapi.dev/api/people/1/?format=json`. The page displays a JSON object representing Luke Skywalker. The JSON structure includes his name, height, mass, hair and skin colors, eye color, birth year, gender, homeworld, films, species, vehicles, starships, and creation/edition dates. The JSON is displayed with collapsible sections indicated by arrows.

```
{
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "hair_color": "blond",
    "skin_color": "fair",
    "eye_color": "blue",
    "birth_year": "19BBY",
    "gender": "male",
    "homeworld": "http://swapi.dev/api/planets/1/",
    "films": [
        "http://swapi.dev/api/films/1/",
        "http://swapi.dev/api/films/2/",
        "http://swapi.dev/api/films/3/",
        "http://swapi.dev/api/films/6/"
    ],
    "species": [],
    "vehicles": [
        "http://swapi.dev/api/vehicles/14/",
        "http://swapi.dev/api/vehicles/30/"
    ],
    "starships": [
        "http://swapi.dev/api/starships/12/",
        "http://swapi.dev/api/starships/22/"
    ],
    "created": "2014-12-09T13:50:51.644000Z",
    "edited": "2014-12-20T21:17:56.891000Z",
    "url": "http://swapi.dev/api/people/1/"
}
```

Figura 17. Acceso a la API de “swapi.dev” desde el navegador.

Fuente: elaboración propia.

---

Para ilustrar el uso de JSON, se va a considerar el siguiente programa:

- Listar información sobre algún personaje de Star Wars.
- Leer desde teclado un número entre 1 y 10, que será el personaje que mostrar.
- Llamar a la Star Wars API swapi<sup>6</sup>.
- Extraer y mostrar la información del formato JSON que devuelve la API.

Para implementarlo:

- **Lectura de datos sobre personajes de Star Wars:** el programa toma el número que el usuario introduce por teclado y se construye una URL. Mediante *urllib*<sup>7</sup> se recupera el texto en JSON que Star Wars API devuelve; por ejemplo, el personaje 1 (figura 17) con la URL <https://swapi.dev/api/people/1/?format=json>:

---

<sup>6</sup>[SWAPI The Star Wars API](#).

<sup>7</sup>“Extensible library for opening URLs”. Python.

- **Análisis de datos e impresión.** Para recuperar los datos JSON de la URL, se debe usar el método *read* de *urllib.request.urlopen*.
- **Programa completo.** El programa completo se muestra en la figura 18. En el *notebook* “Ejemplo\_1\_json.ipynb” se muestra su ejecución.

```

import urllib
import json

serviceurl = 'https://swapi.dev/api/people/'
while True:
    numero = input('Teclea el número de personaje a consultar o cero para terminar: ')
    if int(numero) <= 0 :
        break
    else:
        url = serviceurl + numero + "?format=json"
        print ('Recuperando', url)
        request = urllib.request.Request(url)
        request.add_header('User-Agent','cheese')
        data = urllib.request.urlopen(request).read()
        print ('Recuperados',len(data),' caracteres')
        try:
            js = json.loads(data)
        except:
            js = None
        print ("Nombre: ",js['name'])
        print ("Altura: ",js['height'])
        print ("Color de cabello: ",js['hair_color'])
        print ("Color de piel: ",js['skin_color'])
        print ("Color de ojos: ",js['eye_color'])
        print ("Año de nacimiento: ",js['birth_year'])
        print ("Genero: ",js['gender'])

```

```

Teclea el número de personaje a consultar o cero para terminar: 4
Recuperando https://swapi.dev/api/people/4/?format=json
Recuperados 531 caracteres
Nombre: Darth Vader
Altura: 202
Color de cabello: none
Color de piel: white
Color de ojos: yellow
Año de nacimiento: 41.9BBY

```

**Figura 18.** Programa completo.

Fuente: elaboración propia.

**CONTINUAR**

GeolIP<sup>8</sup> es un servicio de geolocalización que se utiliza para deducir la ubicación geográfica de una persona o un objeto a partir de una dirección IP. Se trata de una base de datos de geolocalización gratuita que se actualiza periódicamente. Esta tecnología se usa a menudo para marketing geográfico, precios regionales, para personalizar los contenidos, dirigir la publicidad, gestión digital de derechos, entre otras acciones.

El servicio IP Geolocation API recibe una petición especificando la dirección IP y el formato de datos de respuesta.

URL: [http://ip-api.com/FORMATO/DIRECCION\\_IP](http://ip-api.com/FORMATO/DIRECCION_IP)

---

<sup>8</sup>Página web de IP Geolocation API.

Formatos soportados:

- JSON.
- XML.
- CSV.

En respuesta, se obtendrán los siguientes valores: city, country, countryCode, isp, lat, lon, org, query, región, regionName, status, timezone y zip.

El siguiente programa solicita al usuario una dirección IP y le muestra su ubicación geográfica: ciudad, país, código de país, entre otros datos.

El programa completo se muestra en la figura 19. En el notebook “Ejemplo\_2.json.ipynb” se muestra su ejecución:

```

import urllib
import json

serviceurl = 'http://ip-api.com/json/'
while True:
    ip = input('Teclea la dirección IP a consultar o FIN para terminar: ')
    if ip == 'FIN':
        break
    else:
        url = serviceurl + ip
        print ('Recuperando', url)
        request = urllib.request.Request(url)
        request.add_header('User-Agent', "cheese")
        data = urllib.request.urlopen(request).read()
        print ('Recuperados',len(data), ' caracteres')
        try:
            js = json.loads(data)
        except: js = None
        if ( "status" in js and js['status'] == "fail" ):
            print ("Error al obtener datos",js['message'])
        else:
            print ("City: ",js['city'])
            print ("Country: ",js['country'])
            print ("Country Code: ",js['countryCode'])
            print ("isp: ",js['isp'])
            print ("Lat: ",js['lat'])
            print ("Log: ",js['lon'])
            print ("Region: ",js['region'])
            print ("regionName: ",js['regionName'])
            print ("timezone: ",js['timezone'])

```

Teclea la dirección IP a consultar o FIN para terminar: 54.148.84.95

Recuperando <http://ip-api.com/json/54.148.84.95>

Recuperados 303 caracteres

City: Portland

Country: United States

Country Code: US

isp: Amazon.com, Inc.

Lat: 45.5235

Log: -122.676

**Figura 19.** Programa completo.

Fuente: elaboración propia.



## VI. Manipulación de documentos XML

**XML** (Extensible Markup Language) es un metalenguaje que permite definir lenguajes de marcado. Los lenguajes de marcado permiten describir la estructura de los contenidos de un documento.

Un lenguaje de marcado está formado por un conjunto de etiquetas que se encierran entre corchetes angulares, <>, y se usan en pares:

<etiqueta> y </etiqueta>

No existen conjuntos prefijados de etiquetas, se definen en cada lenguaje de marcado. Cada par de etiquetas delimita el comienzo y el final de una porción de documento a la que se refiere la etiqueta. Por ejemplo:

```
<asignatura>Bases de datos</asignatura>
```

Un documento XML **es aquel que se crea utilizando un lenguaje de marcado**. Por ejemplo (se incluye fichero 4.20.xml):

```
<banco>
  <cuenta>
    <numero_cuenta>C-101</numero_cuenta>
    <nombre_sucursal>Centro</nombre_sucursal>
    <saldo>500</saldo>
  </cuenta>
  <cliente>
    <nombre>González</nombre>
    <direccion>Calle Arenal, 2</direccion>
    <ciudad>La Granja</ciudad>
  </cliente>
  <impositor>
```

```
<numero_cuenta>C-101</numero_cuenta>
<nombre_cliente>González</nombre_cliente>
</impositor>
</banco>
```

CONTINUAR

## 6.1. Estructura básica de un documento XML

Todo documento XML está formado por:

- **Prólogo.** Consta de dos declaraciones:
  - La declaración XML que indica la versión de XML utilizada y el tipo de codificación de caracteres.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- La declaración de tipo de documento que asocia el documento a una DTD o XSD respecto a la cual el documento es conforme. DTD y XSD son dos mecanismos (metalenguajes) para definir el contenido esperado dentro de un documento XML.<sup>9</sup>

<sup>9</sup>"[What is a DTD?](#)". W3Schools.

CONTINUAR

- **Elementos.** Son un par de etiquetas de comienzo y final coincidentes, que delimitan una porción de información.

```
<título>introducción</título>
```

- Existen elementos vacíos que no encierran contenido. Se representan indistintamente como:

```
<Nombre etiqueta/> o <Nombre etiqueta> </Nombre etiqueta>
```

- Los elementos se pueden anidar. Un texto aparece en el contexto de un elemento si aparece entre la etiqueta de inicio y final de dicho elemento. Las etiquetas se anidan correctamente si toda etiqueta de inicio tiene una única etiqueta de finalización coincidente que se encuentre en el contexto del mismo elemento padre.
- Un elemento puede aparecer varias veces en un documento XML.
- El texto en un documento XML puede estar mezclado con los subelementos de otro elemento.

```
<cuenta>  
  Esta cuenta se usa muy rara vez, por no decir nunca  
  <numero_cuenta> C-102 </numero_cuenta>  
  <nombre_sucursal>Navacerrada</nombre_sucursal>  
  <saldo>400</saldo>  
</cuenta>
```

- Todo documento XML tiene un único elemento raíz que engloba al resto de los elementos del documento. En el primer ejemplo (figura 4.20), el elemento `<banco>` era la raíz.

**CONTINUAR**

- **Atributos.** Las etiquetas de los elementos pueden incluir uno o más atributos que representan propiedades de los elementos de la forma `Nombre atributo="Valor atributo"`

```
<cuenta tipo_cuenta="corriente">
```



Los atributos pueden aparecer solamente una vez en una etiqueta dada.

**CONTINUAR**

- **Comentarios.** Es un texto que se escribe entre <!-- y -->.
  - La cadena “–” no puede aparecer dentro de un comentario.
  - Los comentarios pueden aparecer en cualquier sitio salvo dentro de declaraciones, etiquetas y dentro de otros comentarios.

**CONTINUAR**

- **Espacio de nombres.** Es un mecanismo que permite especificar globalmente nombres únicos para que se usen como marcas de elementos en los documentos XML.
  - Para ello, se antepone a la etiqueta o atributo un identificador de recursos universal.
  - En el ejemplo del banco podría ser <http://www.BancoPrincipal.com>.
  - Para abreviarlo, se declaran abreviaturas del espacio de nombres mediante el atributo `xmlns`:

```
<banco xmlns:BP="http://www.BancoPrincipal.com">
  ...
  <BP:sucursal>
    <BP:nombre_sucursal>Centro</BP:nombre_sucursal>
    <BP:ciudad_sucursal>Centro</BP:ciudad_sucursal>
  </BP:sucursal>
  ...
</banco>
```

- Un documento puede tener más de un espacio de nombres declarado como parte del elemento raíz, de manera que se pueden asociar elementos diferentes con espacios de nombres distintos.
- Se puede definir un espacio de nombres predeterminado mediante el uso del atributo `xmlns` en el elemento raíz. Los elementos sin un prefijo de espacio de nombres explícito pertenecen entonces al espacio de nombres predeterminado.

---

## Saber más

**Obsérvese que a veces es necesario almacenar valores que contienen etiquetas sin que se interpreten como etiquetas XML, es decir, como texto normal. Para ello, se usa la construcción:**

```
<! [CDATA[<cuenta>...</cuenta>]]
```

**CONTINUAR**

Para ilustrar el uso de XML, supóngase que se quiere representar mediante un documento XML la siguiente información:

## Persona 1

Nombre: Roberto Casas

Email: ro.casas@direccion.com

Amigos: Leire, Pepe

## Persona 2

Nombre: Leire García

Email: le.gracia@direccion.com,le.garcia@hotmail.com

Amigos: Ricky

## Persona 3

Nombre: José Manzaneda

Email: j.manzaneda@direccion.com , jman@hotmail.com

Amigos: Ricky

Enemigos: Leire

Esta información se podría representar mediante un documento de la siguiente forma (se incluye fichero 4.21.xml):

```
<?xml version="1.0" encoding="UTF-8" ?>
<listin>
    <persona sexo="hombre" id="ricky">
        <nombre>Roberto Casas</nombre>
        <email>ro.casas@direccion.com</email>
        <relacion amigo de="leire pepe"/>
    </persona>
    <persona sexo="mujer" id="leire">
        <nombre>Leire Garcia</nombre>
        <email>le.garcia@direccion.com</email>
        <email>le.garcia@hotmail.com</email>
        <relacion amigo_de="ricky"/>
    </persona>
    <persona sexo="hombre" id="pepe">
        <nombre>Jose Manzaneda</nombre>
        <email>j.manzaneda@direccion.com</email>
        <email>jman@hotmail.com</email>
        <relacion enemigo_de="leire" amigo_de="ricky"/>
    </persona>
</listin>
```

CONTINUAR

## 6.2. Procesamiento de documentos XML usando Python

Un procesador XML permite a una aplicación acceder a los contenidos de un documento XML, así como detectar posibles errores. Hay dos enfoques para acceder a los contenidos:

### DIRIGIDO POR EVENTOS

### MANIPULACIÓN DEL ÁRBOL

El documento se procesa secuencialmente, de manera que **cada elemento reconocido activa un evento que puede dar lugar a una acción por parte de la aplicación.**<sup>10</sup> SAX es un estándar para este enfoque.

### DIRIGIDO POR EVENTOS

### MANIPULACIÓN DEL ÁRBOL

El documento se estructura como **árbol de nodos a los que se puede acceder en cualquier orden**. DOM<sup>11</sup> es un estándar para este enfoque.

<sup>10</sup>O'Reilly. "[Event Driven XML Processing](#)".

<sup>11</sup>"[Introduction to the DOM. Mozilla](#)".

Se van a estudiar tres herramientas de procesamiento XML:

- Procesamiento basado en SAX.
- Procesamiento basado en DOM.

- Herramienta de procesamiento específica de Python: ElementTree.

**CONTINUAR**

### 6.2.1. SAX (SIMPLE API FOR XML)

Es una interfaz dirigida por eventos que permite leer el contenido como una secuencia de datos e interpretar las etiquetas según se van encontrando. Se caracteriza por:

- Las partes del documento siempre se leen en orden desde el inicio al final.
- No se crea ninguna estructura de datos para representar el documento, sino que solo se analiza secuencialmente y se generan eventos, denominados eventos de análisis, que corresponden con el reconocimiento de partes de un documento.
- Por ejemplo, cuando se encuentra el inicio de un elemento se genera un evento o cuando finaliza un elemento se genera otro evento.
- Para gestionar los eventos se crean funciones controladoras para cada evento que se va a considerar, denominadas manejadores de eventos. De esta forma, cuando ocurre un evento se llama al manejador correspondiente para que realice la acción definida en dicho manejador.
- No es posible manipular información ya procesada, de manera que, si fuera necesario, habría que guardarla en una estructura de datos o volver a llamar al procesador.

Por ejemplo, si se considera el siguiente documento XML:

```
<?xml version="1.0"?>
<doc>
  <par>
    Hola Mundo
  </par>
</doc>
```

El procesamiento con SAX produciría la siguiente secuencia de eventos:

**1. inicio de documento**

**2. inicio de elemento doc**

**3. inicio de elemento par**

**4. caracteres Hola mundo**

*5. fin de elemento par*

*6. fin de elemento doc*

*7. fin documento*

CONTINUAR

---

Para procesar usando SAX, es necesario crearse un manejador propio *ContentHandler* como subclase de *xml.sax.ContentHandler*.

El manejador proporciona un conjunto de métodos para gestionar determinados eventos que se producen en el procesamiento<sup>12</sup>:

Los métodos `startDocument` y `endDocument` son llamadas al comienzo y al final del archivo XML.

Los métodos `startElement` (etiqueta, atributos) y `endElement` (etiqueta) son llamados al comienzo y al final de cada elemento. En caso de utilizar espacios de nombres, se utilizarían los métodos `startElementNS` y `endElementNS`.

El método *character* (*texto*) es llamado cuando es una cadena de texto.

`xml.sax.make_parser ([Lista de parsers])`: crea un nuevo objeto *parser*. Tiene como argumento optativo una lista de *parsers*.

`xml.sax.parse` (archivo XML, Manejador, [ManejadorErrores]): crea un **parser SAX** y lo usa para procesar el documento XML. Tiene como argumento el documento XML que va a ser procesado, el manejador de eventos y optativamente un manejador de errores.

`xml.sax.parseString (NombreCadenaXML, Manejador, [ManejadorErrores])`: crea un *parser SAX* y lo usa para procesar la cadena XML dada. Tiene como argumento la cadena XML que va a ser procesada, el manejador de eventos y optativamente un manejador de errores.

---

<sup>12</sup>[“ContentHanlder Objects”](#). Documentación oficial librería xml.sax.

CONTINUAR

A continuación, se va a realizar el siguiente programa. Para ello, se va a **crear un manejador para procesar el archivo “datos.xml”**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Datos>
    <Libro isbn="0-678-12345-9">
        <titulo>Libro 1</titulo>
        <fecha>Diciembre 2001</fecha>

        <autor>Pepito Perez</autor>
    </Libro>
    ...
</Datos>
```

Hay que importar el paquete SAX: import xml.sax.

Se crea una clase que es subclase de la clase `xml.sax.ContentHandler` (principal interfaz de llamada en SAX).

Se definen dentro de la clase cuatro métodos:

- **El método *init*.** En él se definen como atributos de la clase las etiquetas y atributos del documento XML que se quieren gestionar: datos, para almacenar el nombre del elemento actual; título; fecha y autor.
- **El método *startElement*.** Se indica qué acciones se quieren llevar a cabo cuando se encuentre el comienzo de un elemento. En el ejemplo, para la etiqueta Libro se quiere capturar el ISBN del libro.
- **El método *endElement*.** Se indica qué acciones se quieren llevar a cabo cuando se encuentre el final de un elemento. En el ejemplo, se quiere imprimir el nombre del elemento y el valor que contenía.
- **El método *characters*.** Se indica qué acciones se quieren llevar cuando se encuentre contenido textual que forma parte de un elemento. En el ejemplo, se quiere almacenar dicho contenido en un atributo de la clase que luego será impreso por pantalla por el método *endElement*.

```
import xml.sax
class ManejadorCatalogo (xml.sax.ContentHandler):
    def __init__(self):
        self.Datos=""
        self.titulo=""
        self.fecha=""
        self.autor=""

    def startElement(self,etiqueta,atributos):
        self.Datos=etiqueta
        if etiqueta=="Libro":
            print ("*****Libro*****")
            isbn=atributos["isbn"]
            print ("isbn:", isbn)

    def endElement(self,etiqueta):
        if self.Datos=="titulo":
            print ("Titulo:", self.titulo)
        elif self.Datos=="fecha":
            print ("Fecha:",self.fecha)
        elif self.Datos=="autor":
            print ("Autor:", self.autor)
        self.Datos=""

    def characters(self,contenido):
        if self.Datos=="titulo":
            self.titulo=contenido
        elif self.Datos=="fecha":
            self.fecha=contenido
        elif self.Datos=="autor":
            self.autor=contenido
```

Figura 20. Programa completo en SAX.

Fuente: elaboración propia.

El programa completo sería el de la figura 20 y la ejecución se muestra en el notebook “Ejemplo\_1\_XML\_SAX.ipynb”:

Una vez que se tiene definida la clase, se puede llevar a cabo el procesamiento:

- Se crea un objeto *parser XML*.
- Se configura el *parser*.
- Se fija el manejador de eventos.
- Se procesa el documento.

```
****Libro****
isbn: 0-596-00128-2
Titulo: Python y XML
Fecha: Diciembre 2001
Autor: Pepito Perez
****Libro****
isbn: 0-596-15810-6
Titulo: Programacion avanzada de XML
Fecha: Octubre 2010
Autor: Juan Garcia
****Libro****
isbn: 0-596-15806-8
Titulo: Aprendiendo Java
Fecha: Septiembre 2009
Autor: Juan Garcia
****Libro****
isbn: 0-596-15808-4
Titulo: Python para moviles
Fecha: Octubre 2009
Autor: Pepito Perez
****Libro****
isbn: 0-596-00797-3
Titulo: R para estadistica
Fecha: Marzo 2005
Autor: Juan
Autor: Pepe
Autor: Isabel
****Libro****
isbn: 0-596-10046-9
Titulo: Python en 100 paginas
Fecha: Julio 2006
Autor: Julia
```

**Figura 21.** Procesamiento.*Fuente:* elaboración propia.

Una vez que se tiene definida la clase, se puede llevar a cabo el procesamiento (figura 21).

## Saber más

Como información complementaria al ejercicio anterior, se ofrece el siguiente notebook con información más ampliada de SAX, “**Explicación\_SAX.ipynb**”.

Es necesario realizar el siguiente programa. Usando SAX, se va a crear un manejador para procesar las noticias RSS del archivo “**rss.xml**”.

```
<?xml version="1.0" ?>
<rss version="2.0">
    <channel>
        <title>Noticias</title>
        <link>http://misitio.com</link>
        <description>Portal de Noticias de España</description>
        <item>
            <title>Noticia 1</title>
            <link>http://misitio.com/noticia1</link>
            <description>Descripción de la noticia 1</description>
        </item>
        ...
    </channel>
</rss>
```

Es similar al ejercicio anterior:

- Hay que importar el paquete SAX: import xml.sax.
- Se crea una clase que es subclase de la clase `xml.sax.ContentHandler` (principal interfaz de llamada en SAX).
- Se definen dentro de la clase 4 (métodos).
- Realizar el procesamiento de archivo.

La clase completa sería la de la figura 22 y la ejecución se muestra en el notebook “Ejemplo\_\_2\_XML\_SAX.ipynb”

```

import xml.sax
class ManejadorCatalogo (xml.sax.ContentHandler):
    def __init__(self):
        self.Datos=""
        self.title=""          # Elemento title del elemento item
        self.link=""           # Elemento Link del elemento item
        self.description=""    # Elemento description del elemento item
    def startElement(self,etiqueta,atributos):
        self.Datos=etiqueta
    def endElement(self,etiqueta):
        if self.Datos=="title":
            print ("title:", self.title)
        elif self.Datos=="link":
            print ("link:",self.link)
        elif self.Datos=="description":
            print ("description:", self.description)
        self.Datos=""
    def characters(self,contenido):
        if self.Datos=="title":
            self.title=contenido
        elif self.Datos=="link":
            self.link=contenido
        elif self.Datos=="description":
            self.description=contenido
if ( __name__ == "__main__"):
    # Crear un XMLReader
    parser=xml.sax.make_parser()
    # Desabilitar namespaces
    parser.setFeature(xml.sax.handler.feature_namespaces,0)
    #Sobre escribir el default default ContextHandler
    Handler=ManejadorCatalogo()

    parser.setContentHandler(Handler)
    parser.parse("rss.xml")

```

title: Noticias  
link: <http://misitio.com>  
description: Portal de Noticias de España

**Figura 22.** Procesamiento.

Fuente: elaboración propia.

**CONTINUAR**

### 6.2.2. DOM (DOCUMENT OBJECT MODEL)

Para procesar un documento XML usando DOM **se debe utilizar la librería `xml.dom`**. Esta librería permite crear un objeto `minidom` que dispone de un método que procesa un documento XML dado y genera un árbol DOM.

Posteriormente, este árbol puede recorrerse para acceder a nodos concretos, manipularlos e incluso reconstruir un documento XML con los cambios que se hayan introducido.

En primer lugar, se abre el documento XML con el método `parse` del objeto `minidom` que proporciona un árbol DOM del documento. A continuación, se puede empezar a recorrer el árbol. Se accede a la raíz del árbol a través del atributo `documentElement`.

Desde la raíz del árbol, utilizando un conjunto de métodos, se puede visitar:

- `getElementsByTagName("nombre_elemento")`: devuelve una lista de todos los elementos cuyo nombre coincide con el nombre proporcionado.
- `getAttribute("nombre_atributo")`: devuelve el valor del atributo proporcionado como parámetro.
- `hasAttribute("nombre_atributo")`: indica si un elemento tiene el atributo proporcionado como parámetro. Si existe, devuelve **True**. En caso contrario, devuelve **False**.



Para acceder al contenido de cada elemento, se utiliza el atributo `data` del objeto `childNodes`.

Se va a realizar el procesamiento del documento XML “datos.xml” de ejemplo usando DOM (figura 23). La ejecución se muestra en el notebook “Ejemplo\_1\_XML\_DOM.ipynb”.

Téngase en cuenta lo siguiente:

- SAX es un procesador bastante eficiente que permite manejar documentos muy extensos en tiempo lineal y con una cantidad de memoria constante. Sin embargo, requiere un esfuerzo mayor por parte de los desarrolladores.
- DOM es más fácil de usar para los desarrolladores, pero aumenta el coste de memoria y tiempo.
- Será mejor usar SAX cuando el documento que se va a procesar no quepa en memoria o cuando las tareas sean irrelevantes con respecto a la estructura del documento (contar el número de elementos, extraer contenido de un elemento determinado).
- Será mejor usar DOM cuando se necesite realizar operaciones complejas sobre el documento, como edición o reestructuración de su contenido.

```
from xml.dom.minidom import parse
import xml.dom.minidom

ArbolDom = xml.dom.minidom.parse("datos.xml")
libros = ArbolDom.getElementsByTagName("Libro")
for libro in libros:
    isbn = libro.getAttribute("isbn")
    print("isbn:", isbn)
    titulo = libro.getElementsByTagName("titulo")[0]
    print("titulo:", titulo.firstChild.data)
    fecha = libro.getElementsByTagName("fecha")[0]
    print("fecha:", fecha.firstChild.data)
    autores = libro.getElementsByTagName("autor")
    for autor in autores:
        print("autor:", autor.firstChild.data)
    print("-----")
```

```
isbn: 0-596-00128-2
titulo: Python y XML
fecha: Diciembre 2001
autor: Pepito Perez
-----
isbn: 0-596-15810-6
titulo: Programacion avanzada de XML
fecha: Octubre 2010
autor: Juan Garcia
-----
isbn: 0-596-15806-8
titulo: Aprendiendo Java
fecha: Septiembre 2009
autor: Juan Garcia
-----
isbn: 0-596-15808-4
```

Figura 23. Procesamiento usando DOM.

Fuente: elaboración propia.

A continuación, se va a realizar el siguiente programa. Usando DOM, se va a crear un manejador para procesar las noticias RSS del archivo "rss.xml" (figura 24). La ejecución se muestra en el notebook "Ejemplo\_2\_XML\_DOM.ipynb":

```
from xml.dom.minidom import parse
import xml.dom.minidom

ArbolDom = xml.dom.minidom.parse("rss.xml")
channel = ArbolDom.getElementsByTagName("channel")[0]

items = channel.getElementsByTagName("item")
for item in items:
    title = item.getElementsByTagName("title")[0]
    print("title:", title.firstChild.data)

    link = item.getElementsByTagName("link")[0]
    print("link:", link.firstChild.data)

    description = item.getElementsByTagName("description")[0]
    print("description:", description.firstChild.data)

    print("-----")

title: Noticia 1
link: http://misitio.com/noticia1
description: Descripción de la noticia 1
-----
title: Noticia 2
link: http://misitio.com/noticia2
description: Descripción de la noticia 2
-----
title: Noticia 3
link: http://misitio.com/noticia3
description: Descripción de la noticia 3
-----
```

Figura 24. Procesamiento usando DOM.  
Fuente: elaboración propia.

CONTINUAR

### 6.2.3. Procesamiento con ELEMENTTREE

---

Es una librería estándar para procesar y crear documentos XML con características similares a DOM, ya que crea un árbol de objetos representado por la clase *ElementTree*. Sin embargo, la navegación es más ligera con un estilo específico de Python.

El árbol generado está formado por objetos “elemento” de tipo *Element* donde cada uno de ellos **dispone de un conjunto de atributos**: nombre, diccionario de atributos, valor textual y secuencia de elementos hijo.

Para los ejemplos siguientes se va a usar el archivo “datos.xml”.

Para procesar un documento, basta con **abrir el documento con el método *open()***, como si se tratara de un fichero, y usar el método *parse* de *ElementTree*.

```
from xml.etree import ElementTree
f = open ("datos.xml", 'rt')
arbol = ElementTree.parse(f)
print (arbol)
```

```
from xml.etree import ElementTree
f = open ("datos.xml", 'rt')
arbol = ElementTree.parse(f)
print (arbol)
for nodo in arbol.iter():
    print (nodo.tag, "---",nodo.text, "----" ,nodo.attrib)

<xml.etree.ElementTree.ElementTree object at 0x000001ED0CBC8408>
Datos ---
    --- {}
Libro ---
    --- {'isbn': '0-596-00128-2'}
    titulo --- Python y XML --- {}
    fecha --- Diciembre 2001 --- {}
    autor --- Pepito Perez --- {}
Libro ---
    --- {'isbn': '0-596-15810-6'}
    titulo --- Programacion avanzada de XML --- {}
    fecha --- Octubre 2010 --- {}
    autor --- Juan Garcia --- {}
Libro ---
    --- {'isbn': '0-596-15806-8'}
    titulo --- Aprendiendo Java --- {}
    fecha --- Septiembre 2009 --- {}
    autor --- Juan Garcia --- {}
Libro ---
    --- {'isbn': '0-596-15808-4'}
    titulo --- Python para moviles --- {}
    fecha --- Octubre 2009 --- {}
    autor --- Pepito Perez --- {}
Libro ---
    --- {'isbn': '0-596-00797-3'}
    titulo --- R para estadistica --- {}
    fecha --- Marzo 2005 --- {}
```

**Figura 25.** Procesamiento usando *iter()*.

Fuente: elaboración propia.

Si se quiere visitar todo el árbol, se usa el método *iter()*, que crea un generador que itera sobre todos los nodos del árbol (figura 25). La ejecución se muestra en el notebook “Ejemplo\_1\_XML\_ELEMENTTREE.ipynb”.

```
from xml.etree import ElementTree
f = open ("datos.xml", 'rt')
arbol = ElementTree.parse(f)
i=1
for nodo in arbol.iter("Libro"):
    isbn = nodo.attrib.get("isbn")
    print (nodo.tag, i, "con isbn: " ,isbn)
    i+=1

Libro 1 con isbn: 0-596-00128-2
Libro 2 con isbn: 0-596-15810-6
Libro 3 con isbn: 0-596-15806-8
Libro 4 con isbn: 0-596-15808-4
Libro 5 con isbn: 0-596-00797-3
Libro 6 con isbn: 0-596-10046-9
```

Figura 26. Procesamiento usando iter() con parámetros.

Fuente: elaboración propia.

Puede que solo interesen determinados elementos del árbol y no todos. Para ello, se pasa como parámetro del método *iter()* el nombre del elemento de interés (figura 26). La ejecución se muestra en el notebook “Ejemplo\_2\_XML\_ELEMENTTREE.ipynb”.

```
from xml.etree import ElementTree as ET
f = open ("datos.xml", 'rt')
arbol = ET.parse(f)
raiz = arbol.getroot()
for hijo in raiz:
    print (hijo.tag, "---",
          hijo.attrib)

Libro --- {'isbn': '0-596-00128-2'}
Libro --- {'isbn': '0-596-15810-6'}
Libro --- {'isbn': '0-596-15806-8'}
Libro --- {'isbn': '0-596-15808-4'}
Libro --- {'isbn': '0-596-00797-3'}
Libro --- {'isbn': '0-596-10046-9'}
```

Figura 27. Procesamiento iterando sobre elementos del árbol.

Fuente: elaboración propia.

Otra posibilidad de iterar sobre los elementos del árbol es acceder a la raíz del árbol y desde ella iterar sobre los hijos (figura 27). La ejecución se muestra en el notebook “Ejemplo\_3\_XML\_ELEMENTTREE.ipynb”.

```

from xml.etree import ElementTree as ET
f = open ("datos.xml", 'rt')
arbol = ET.parse(f)
raiz = arbol.getroot()
print ("Título: ",raiz[0][0].text, ", fecha: ",raiz[0][1].text)
print ("Título: ",raiz[1][0].text, ", fecha: ",raiz[1][1].text)
print ("Título: ",raiz[2][0].text, ", fecha: ",raiz[2][1].text)
print ("Título: ",raiz[3][0].text, ", fecha: ",raiz[3][1].text)
print ("Título: ",raiz[4][0].text, ", fecha: ",raiz[4][1].text)
print ("Título: ",raiz[5][0].text, ", fecha: ",raiz[5][1].text)

Título: Python y XML , fecha: Diciembre 2001
Título: Programacion avanzada de XML , fecha: Octubre 2010
Título: Aprendiendo Java , fecha: Septiembre 2009
Título: Python para moviles , fecha: Octubre 2009
Título: R para estadistica , fecha: Marzo 2005
Título: Python en 100 paginas , fecha: Julio 2006

```

Figura 28. Acceso indexado a los elementos.

Fuente: elaboración propia.

También es posible acceder a los elementos de forma indexada (figura 28). La ejecución se muestra en el notebook “Ejemplo\_4\_XML\_ELEMENTTREE.ipynb”.

**CONTINUAR**

Existe otro conjunto de métodos que permiten **recorrer el árbol tomando como argumento una expresión XPath<sup>13</sup>** que caracteriza al elemento que se está buscando.

<sup>13</sup>“[XPath Tutorial](#)”. W3Schools.

XPath es un estándar que implementa una sintaxis específica para definir rutas a elementos dentro de una estructura de árbol.

Funciona de manera muy similar a como se define una URL o una ruta a un fichero en un árbol de directorios.

Algunas de las funciones que la librería `xml.etree` ofrece para buscar elementos a través de su ruta XPath son:

- `find(descripción)`. Recupera el primer subelemento del elemento actual que encaja con la descripción dada.
- `findall(descripción)`. Recupera todos los subelementos del elemento actual que encajan con la descripción dada.
- `iterfind(descripción)`. Recupera todos los elementos que encajan con la descripción dada.
- `text`. Accede al contenido textual de un elemento.
- `get(atributo)`. Accede al atributo dado del elemento.

Se van a encontrar todos los títulos de los libros usando `findall()`, tal como se muestra en la figura 29. La ejecución se muestra en el notebook “Ejemplo\_5\_XML\_ELEMENTTREE.ipynb”:

```
from xml.etree import ElementTree as ET
f = open ("datos.xml", 'rt')
arbol = ET.parse(f)
i=1
for nodo in arbol.findall("./Libro/titulo"):
    print ("Título: ",i,nodo.text)
    i+=1
```

```
Título: 1 Python y XML
Título: 2 Programacion avanzada de XML
Título: 3 Aprendiendo Java
Título: 4 Python para moviles
Título: 5 R para estadistica
Título: 6 Python en 100 paginas
```

Figura 29. Búsqueda de libros usando *findall()*.

Fuente: elaboración propia.

CONTINUAR

Esta API permite realizar un procesamiento basado en eventos, al estilo de SAX, usando el método *iterparse()*.

Genera eventos *start* en las aperturas de elemento y eventos *end* en los cierres de elemento. Además, los datos pueden ser extraídos del documento durante la fase de parseo.

Ahora, se va a realizar un procesamiento similar al que se hizo con SAX (figura 30). La ejecución se muestra en el notebook “Ejemplo\_6\_XML\_ELEMENTTREE.ipynb”:

```

from xml.etree.ElementTree import iterparse

for(event, element) in iterparse ("datos.xml",("start","end")):
    if (event == "start"):
        if ( element.tag=="Libro" ):
            print("*****Libro*****")
            print("isbn:",element.attrib["isbn"])
    if (event == "end"):
        if ( element.tag=="titulo" ):
            print("Titulo:",element.text)
        if ( element.tag=="fecha" ):
            print("Fecha:",element.text)
        if ( element.tag=="autor" ):
            print("Autor:",element.text)

*****Libro*****
isbn: 0-596-00128-2
Título: Python y XML
Fecha: Diciembre 2001
Autor: Pepito Perez
*****Libro*****
isbn: 0-596-15810-6
Título: Programacion avanzada de
Fecha: Octubre 2010
Autor: Juan Garcia
*****Libro*****
isbn: 0-596-15806-8
Título: Aprendiendo Java
Fecha: Septiembre 2009
Autor: Juan Garcia
*****Libro*****
isbn: 0-596-15808-4
Título: Python para moviles
Fecha: Octubre 2009
Autor: Pepito Perez
*****Libro*****
isbn: 0-596-00797-3
Título: R para estadistica
Fecha: Marzo 2005
Autor: Juan
Autor: Pepe
Autor: Isabel
*****Libro*****
isbn: 0-596-10046-9
Título: Python en 100 paginas
Fecha: Julio 2006
Autor: Julia

```

**Figura 30.** Simulación del procesamiento SAX.

Fuente: elaboración propia.

---

```
import xml.etree.ElementTree as ET
cadena = '''<Datos>
            <Libro isbn="0-596-00128-2">
                <titulo>Python y XML</titulo>
                <fecha>Diciembre 2001</fecha>
                <autor>Pepito Perez</autor>
            </Libro>
        </Datos>'''
doc = ET.fromstring(cadena)
lista = doc.findall("Libro")
for l in lista:
    print ("isbn: ",l.get("isbn"))
    print ("Título: ",l.find("titulo").text)
    print ("Fecha: ",l.find("fecha").text)
    print ("Autor: ",l.find("autor").text)
```

```
isbn: 0-596-00128-2
Título: Python y XML
Fecha: Diciembre 2001
Autor: Pepito Perez
```

Figura 31. Uso de *fromstring*.

Fuente: elaboración propia.

También es posible procesar cadenas que representen un documento XML usando el método *fromstring*, que toma como argumento la cadena que representa el documento XML (figura 31). La ejecución se muestra en el notebook “Ejemplo\_7\_XML\_ELEMENTTREE.ipynb”.

Otra posibilidad que ofrece la librería **xml.etree** es la modificación de un documento XML que ha sido leído:

- A nivel de elemento se puede cambiar el contenido cambiando el valor de `Element.text`, añadir o modificar atributos con el método `Element.set()` y añadir nuevos hijos con el método `Element.append()`.
- A nivel de documento, se escribe el nuevo documento con el método `ElementTree.write()`.

```

<Datos>
  <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
    <titulo>Python y XML</titulo>
    <fecha>Diciembre 2001</fecha>
    <autor>Pepito Perez</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-15810-6" orden="2">
    <titulo>Programacion avanzada de XML</titulo>
    <fecha>Octubre 2010</fecha>
    <autor>Juan Garcia</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-15806-8" orden="3">
    <titulo>Aprendiendo Java</titulo>
    <fecha>Septiembre 2009</fecha>
    <autor>Juan Garcia</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-15808-4" orden="4">
    <titulo>Python para moviles</titulo>
    <fecha>Octubre 2009</fecha>
    <autor>Pepito Perez</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-00797-3" orden="5">
    <titulo>R para estadistica</titulo>
    <fecha>Marzo 2005</fecha>
    <autor>Juan</autor>
    <autor>Pepe</autor>
    <autor>Isabel</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-10046-9" orden="6">
    <titulo>Python en 100 paginas</titulo>
    <fecha>Julio 2006</fecha>
    <autor>Julia</autor>
  </editorial>Anaya</editorial></Libro>
</Datos>

```

```

from xml.etree import ElementTree as ET
f = open ("datos.xml")
arbol = ET.parse(f)
i=1
for libro in arbol.iter("Libro"):
    cadena = str(i)
    libro.set("orden",cadena)
    libro.set("ejemplares","si")
    editorial = ET.Element("editorial")
    editorial.text = "Anaya"
    libro.append(editorial)
    i+=1
arbol.write("datos2.xml")

```

**Figura 32.** Modificación del documento XML.

Fuente: elaboración propia.

El resultado sería el siguiente (figura 32). La ejecución se muestra en el notebook “Ejemplo\_8\_XML\_ELEMENTTREE.ipynb”.

```

from xml.etree import ElementTree as ET
f = open ("datos2.xml")
arbol = ET.parse(f)
raiz = arbol.getroot()
for libro in raiz.iter("Libro"):
    orden = int( libro.get("orden") )
    if ( orden == 3 ):
        raiz.remove(libro)
arbol.write("datos3.xml")

```

```

<Datos>
  <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
    <titulo>Python y XML</titulo>
    <fecha>Diciembre 2001</fecha>
    <autor>Pepito Perez</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-15810-6" orden="2">
    <titulo>Programacion avanzada de XML</titulo>
    <fecha>Octubre 2010</fecha>
    <autor>Juan Garcia</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-15808-4" orden="4">
    <titulo>Python para moviles</titulo>
    <fecha>Octubre 2009</fecha>
    <autor>Pepito Perez</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-00797-3" orden="5">
    <titulo>R para estadistica</titulo>
    <fecha>Marzo 2005</fecha>
    <autor>Juan</autor>
    <autor>Pepe</autor>
    <autor>Isabel</autor>
  </editorial>Anaya</editorial></Libro>
  <Libro ejemplares="si" isbn="0-596-10046-9" orden="6">
    <titulo>Python en 100 paginas</titulo>
    <fecha>Julio 2006</fecha>
    <autor>Julia</autor>
  </editorial>Anaya</editorial></Libro>
</Datos>

```

**Figura 33.** Eliminación de elementos.

Fuente: elaboración propia.

También es posible eliminar elementos con el método *Element.remove()*. Tomando como entrada la salida del ejemplo anterior, se van a eliminar todos los elementos de tipo “Libro” que tengan un número de orden mayor que 3 (figura 33). La ejecución se muestra en el notebook “Ejemplo\_9\_XML\_ELEMENTTREE.ipynb”:

CONTINUAR

También es posible crear documentos XML desde cero. Para ello están disponibles los siguientes métodos en la clase *Element*:

- *Element()*: crea un elemento nuevo.

- *subElement()*: añade un nuevo elemento al padre.
- *comment()*: crea un nodo que serializa el contenido usando la sintaxis de XML.

```
from xml.etree.ElementTree import Element, SubElement, Comment
from xml.etree import ElementTree
from xml.dom import minidom

def prettify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="\t")

raiz = Element("Datos")
Libro = SubElement(raiz,"Libro")
Titulo = SubElement(Libro,"titulo")
Titulo.text = "XML y Python"
Fecha = SubElement(Libro,"fecha")
Fecha.text = "Marzo, 2019"
Autor = SubElement(Libro,"autor")
Autor.text = "Pepito López"
print (prettify(raiz))

<?xml version="1.0" ?>
<Datos>
    <Libro>
        <titulo>XML y Python</titulo>
        <fecha>Marzo, 2019</fecha>
        <autor>Pepito López</autor>
    </Libro>
</Datos>
```

**Figura 34.** Ejemplo de creación de un documento XML.

Fuente: elaboración propia.

En el siguiente ejemplo (figura 34) se va a crear un documento XML con información de un libro semejante a los ejemplos anteriores. La ejecución se muestra en el notebook “Ejemplo\_10\_XML\_ELEMENTTREE.ipynb”.

```

from xml.etree.ElementTree import Element, SubElement, Comment, tostring
from xml.etree import ElementTree
from xml.dom import minidom

def prettify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="\t")

raiz = Element("Datos")
Libro = SubElement(raiz,"Libro")
Titulo = SubElement(Libro,"titulo")
Titulo.text = "XML y Python"
Fecha = SubElement(Libro,"fecha")
Fecha.text = "Marzo, 2019"
Autor = SubElement(Libro,"autor")
Autor.text = "Pepito López"
print (tostring(raiz))

b'<Datos><Libro><titulo>XML y Python</titulo><fecha>Marzo, 2019</fecha><autor>P
epito López</autor></Libro></Datos>'
```

**Figura 35.** Impresión del documento XML.

Fuente: elaboración propia.

Obsérvese que con la función definida *prettify* se consigue que las etiquetas del documento XML estén indentadas. Si no se usa, se puede generar una cadena sin indentar (figura 35). La ejecución se muestra en el notebook

"Ejemplo\_11\_XML\_ELEMENTTREE.ipynb":

En el ejemplo anterior se han creado elementos con contenido, pero en ningún caso se han añadido atributos. **Para añadir atributos a un elemento que se está creando basta con pasar como argumento del elemento o subelemento un diccionario con los atributos expresados en forma de parejas clave-valor.**

```
from xml.etree.ElementTree import Element, SubElement, Comment
from xml.etree import ElementTree
from xml.dom import minidom

def prettyify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="\t")

raiz = Element("Datos")
Libro = SubElement(raiz,"Libro",{"orden":"1","ejemplares":"si","isbn":"0-596-001
Titulo = SubElement(Libro,"titulo")
Titulo.text = "XML y Python"
Fecha = SubElement(Libro,"fecha")
Fecha.text = "Marzo, 2019"
Autor = SubElement(Libro,"autor")
Autor.text = "Pepito López"
print (prettyify(raiz))

<?xml version="1.0" ?>
<Datos>
    <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
        <titulo>XML y Python</titulo>
        <fecha>Marzo, 2019</fecha>
        <autor>Pepito López</autor>
    </Libro>
</Datos>
```

Figura 36. Adición de varios campos y valores. Fuente: elaboración propia.

Se va a modificar el código anterior para añadir atributos al elemento Libro. En concreto, se va a añadir el atributo *isbn*, *orden* y *ejemplares* (figura 36). La ejecución se muestra en el notebook “Ejemplo\_12\_XML\_ELEMENTTREE.ipynb”:

```
<?xml version="1.0" ?>
<Datos>
    <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
        <titulo>XML y Python</titulo>
        <fecha>Pepito López</fecha>
        <autor/>
    </Libro>
</Datos>
```

Figura 37. Resultado de la creación del programa.

Fuente: elaboración propia.

En la figura 37 se muestra el resultado.

**CONTINUAR**

**Se pueden añadir múltiples hijos a un elemento mediante el método `extend()`** que recibe como argumento algo que sea iterable, tal como una lista o bien otra instancia de Element.

En el caso de una instancia de Element, los hijos del elemento dado se añaden como hijos del nuevo padre. Sin embargo, el padre actual no es añadido.

Se va a reconstruir el ejemplo anterior, pero usando `extend` sobre una cadena dada (figura38). La ejecución se muestra en el notebook "Ejemplo\_13\_XML\_ELEMENTTREE.ipynb":

```

from xml.etree.ElementTree import Element,SubElement,XML
from xml.etree import ElementTree
from xml.dom import minidom

def prettyify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent=" ")

raiz = Element("Datos")
raiz.set("version","1.0")
Libro = SubElement(raiz,"Libro",{"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
hijos = XML('''<hijos><titulo>XML y Python</titulo><fecha>Diciembre 2001</fecha><autor>Pepito Perez</autor></hijos>''')
Libro.extend(hijos)
print (prettyify(raiz))

<?xml version="1.0" ?>
<Datos version="1.0">
  <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
    <titulo>XML y Python</titulo>
    <fecha>Diciembre 2001</fecha>
    <autor>Pepito Perez</autor>
  </Libro>
</Datos>

```

**Figura 38.** Uso de extend.

Fuente: elaboración propia.

---

También se podría haber construido pasando una lista (figura 39). La ejecución se muestra en el notebook “Ejemplo\_14\_XML\_ELEMENTTREE.ipynb”:

```

from xml.etree.ElementTree import Element,SubElement,XML
from xml.etree import ElementTree
from xml.dom import minidom

def prettyify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent=" ")

raiz = Element("Datos")
raiz.set("version","1.0")
Libro = SubElement(raiz,"Libro",{"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
titulo = Element("titulo")
titulo.text = "XML y Python"
fecha = Element("fecha")
fecha.text = "Diciembre 2001"
autor = Element("autor")
autor.text = "Pepito Perez"
hijos = [titulo, fecha, autor]
Libro.extend(hijos)
print (prettyify(raiz))

<?xml version="1.0" ?>
<Datos version="1.0">
<Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
<titulo>XML y Python</titulo>
<fecha>Diciembre 2001</fecha>
<autor>Pepito Perez</autor>
</Libro>
</Datos>

```

**Figura 39.** Otra versión pasando una lista.  
Fuente: elaboración propia.

CONTINUAR

En el ejemplo anterior, se ha visto que el documento XML resultante se ha mostrado como una cadena. Sin embargo, **en otros contextos en los que se manejan documentos XML muy grandes, interesa guardarlos en un archivo**. En estos casos se usará el método *write* de ElementTree.

Se va a realizar el mismo ejemplo de antes, pero ahora el resultado se almacenará en un archivo (figura 40). La ejecución se muestra en el notebook “Ejemplo\_15\_XML\_ELEMENTTREE.ipynb”:

```

from xml.etree.ElementTree import Element,SubElement,ElementTree

raiz = Element("Datos")
raiz.set("version","1.0")
Libro = SubElement(raiz,"Libro",{"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
titulo = Element("titulo")
titulo.text = "XML y Python"
fecha = Element("fecha")
fecha.text = "Diciembre 2001"
autor = Element("autor")
autor.text = "Pepito Perez"
hijos = [titulo, fecha, autor]
Libro.extend(hijos)

ElementTree(raiz).write("datos4.xml")

<Datos version="1.0">
    <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
        <titulo>XML y Python</titulo>
        <fecha>Diciembre 2001</fecha>
        <autor>Pepito Perez</autor>
    </Libro>
</Datos>

```

**Figura 40.** Almacenamiento en archivo.

Fuente: elaboración propia.

---

El método `write()` de `ElementTree` tiene un segundo argumento que sirve para controlar qué se hace con elementos que están vacíos. Existen tres posibilidades según el valor de dicho argumento:

- `xml`: genera un elemento vacío con una sola etiqueta.
- `html`: genera un elemento vacío con dos etiquetas.
- `text`: imprime solo elementos con contenido, el resto se los salta.

Siguiendo con el ejemplo anterior, se va a añadir un elemento vacío y se van a probar los tres argumentos (figura 41). La ejecución se muestra en el notebook “Ejemplo\_16\_XML\_ELEMENTTREE.ipynb”.

```

from xml.etree.ElementTree import Element, SubElement, XML
from xml.etree import ElementTree
from xml.dom import minidom

def prettyify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent=" ")

raiz = Element("Datos")
raiz.set("version", "1.0")
Libro = SubElement(raiz, "Libro", {"orden": "1", "ejemplares": "si", "isbn": "0-596-00128-2"})
titulo = Element("titulo")
titulo.text = "XML y Python"
fecha = Element("fecha")
fecha.text = "Diciembre 2001"
autor = Element("autor")
autor.text = "Pepito Perez"
hijos = [titulo, fecha, autor]
Libro.extend(hijos)

ElemVacio = SubElement(Libro, "vacio")

print (prettyify(raiz))

<?xml version="1.0" ?>
<Datos version="1.0">
  <Libro ejemplares="si" isbn="0-596-00128-2" orden="1">
    <titulo>XML y Python</titulo>
    <fecha>Diciembre 2001</fecha>
    <autor>Pepito Perez</autor>
    <vacio/>
  </Libro>
</Datos>

```

Figura 41. Nueva versión del ejemplo anterior.

Fuente: elaboración propia.

```
import sys
from xml.etree.ElementTree import Element,SubElement,ElementTree

def prettyify(elem):
    """Return a pretty-printed XML string for the Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent=" ")

raiz = Element("Datos")
raiz.set("version","1.0")
Libro = SubElement(raiz,"Libro",{"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
titulo = Element("titulo")
titulo.text = "XML y Python"
fecha = Element("fecha")
fecha.text = "Diciembre 2001"
autor = Element("autor")
autor.text = "Pepito Perez"
hijos = [titulo, fecha, autor]
Libro.extend(hijos)
ElemVacio = SubElement(Libro,"vacio")

for metodo in ["xml","html","text"]:
    print (metodo)
    ElementTree(raiz).write(sys.stdout, method=metodo)
    print("\n")
```

**Figura 42.** Nueva implementación.

Fuente: elaboración propia.

El programa se muestra en la figura 42. La ejecución se muestra en el notebook “Ejemplo\_17\_XML\_ELEMENTTREE.ipynb”:

```
xml
<Datos version="1.0"><Libro ejemplares="si" isbn="0-596-00128-2" orden="1"><titulo>XML y Python</titulo><fecha>Diciembre 2001</fecha><autor>Pepito Perez</autor><vacio /></Libro></Datos>

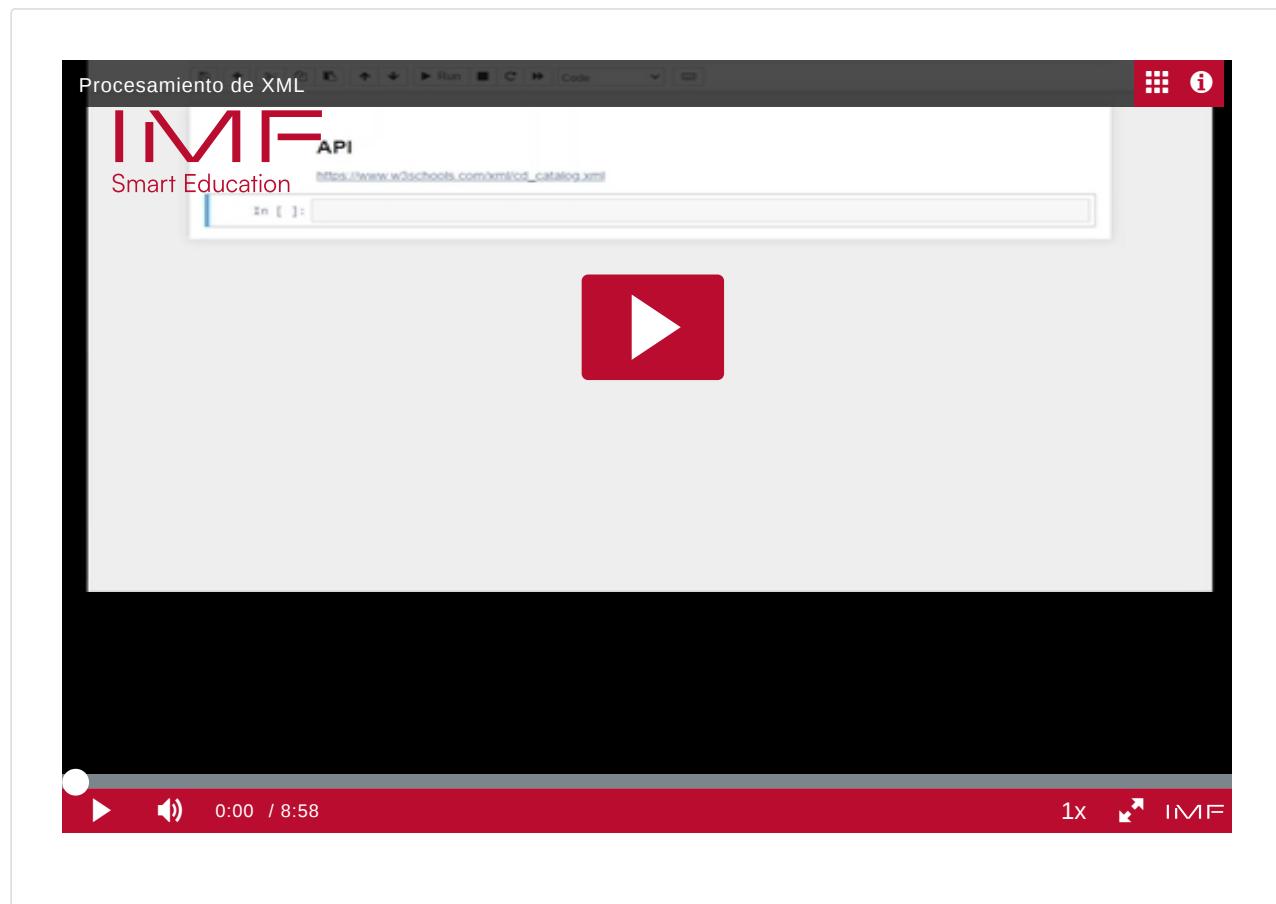
html
<Datos version="1.0"><Libro ejemplares="si" isbn="0-596-00128-2" orden="1"><titulo>XML y Python</titulo><fecha>Diciembre 2001</fecha><autor>Pepito Perez</autor><vacio></vacio></Libro></Datos>

text
XML y PythonDiciembre 2001Pepito Perez
```

Figura 43. Resultado del procesamiento.

Fuente: elaboración propia.

El resultado del procesamiento se muestra en la figura 43.





## VII. Resumen

---



**Repasa los conocimientos adquiridos en la unidad**

En esta unidad se ha explicado que existen formas muy diferentes de representar los mismos datos. Y que estas representaciones suelen depender del servicio que haya extraído esos datos.

A lo largo de su carrera profesional, analistas o científicos de datos se van a encontrar con multitud de orígenes de datos y su capacidad de adaptación y su versatilidad son fundamentales. Por tanto, dominar los formatos más habituales puede suponer un punto extra de agilidad que los haga destacar.

En esta unidad se han introducido tres formatos muy populares: CSV, JSON y XML, y las herramientas y librerías que Python ofrece para trabajar con ellos.

Para la manipulación de documentos CSV, se han visto las diferentes alternativas que ofrece la librería CSV de Python: *Reader* y *Writer* así como *DictReader* y *DictWriter*. Los primeros ofrecen mayor rendimiento para el procesamiento de grandes volúmenes de datos. Pero si los documentos CSV que se están leyendo tienen cabecera, *DictReader* facilitará mucho la lectura y extracción de datos, dando cierta información adicional acerca del significado del dato en cada columna.

De la misma manera, *DictWriter* facilitará la tarea de escribir cada dato en la columna que le corresponde, sin que exista posibilidad de equivocación (por ejemplo, que se escriban datos en columnas erróneas porque el array de entrada no esté en el orden adecuado).

Después se ha introducido el formato JSON y se ha visto que el módulo **JSON** de Python ofrece una interfaz muy simple. El hecho de que exista una relación casi de equivalencia entre un objeto JSON y un diccionario Python simplifica enormemente la interfaz de este módulo, que simplemente necesita exponer los métodos *load* y *loads* (para convertir de JSON a Dict), así como *dump* y *dumps* (para convertir de Dict a JSON). Una vez se tiene la versión *Dict* de un JSON, se puede trabajar como cualquier otro *Dict* en Python.

Se ha cerrado la unidad introduciendo el tercero, y quizás más complejo de trabajar, de los tres formatos: XML.

De los tres formatos planteados, XML es el que ofrece mayor expresividad, razón por la cual se hizo muy popular en las décadas de los ochenta y noventa como formato de serialización para el intercambio de información entre sistemas. Sin embargo, su popularidad ha ido decayendo en las dos primeras décadas del siglo XXI debido al excesivo tamaño en bytes que pueden alcanzar sus documentos. Esto plantea problemas de ineficiencia en el intercambio, cada vez más intensivo, de datos entre sistemas.

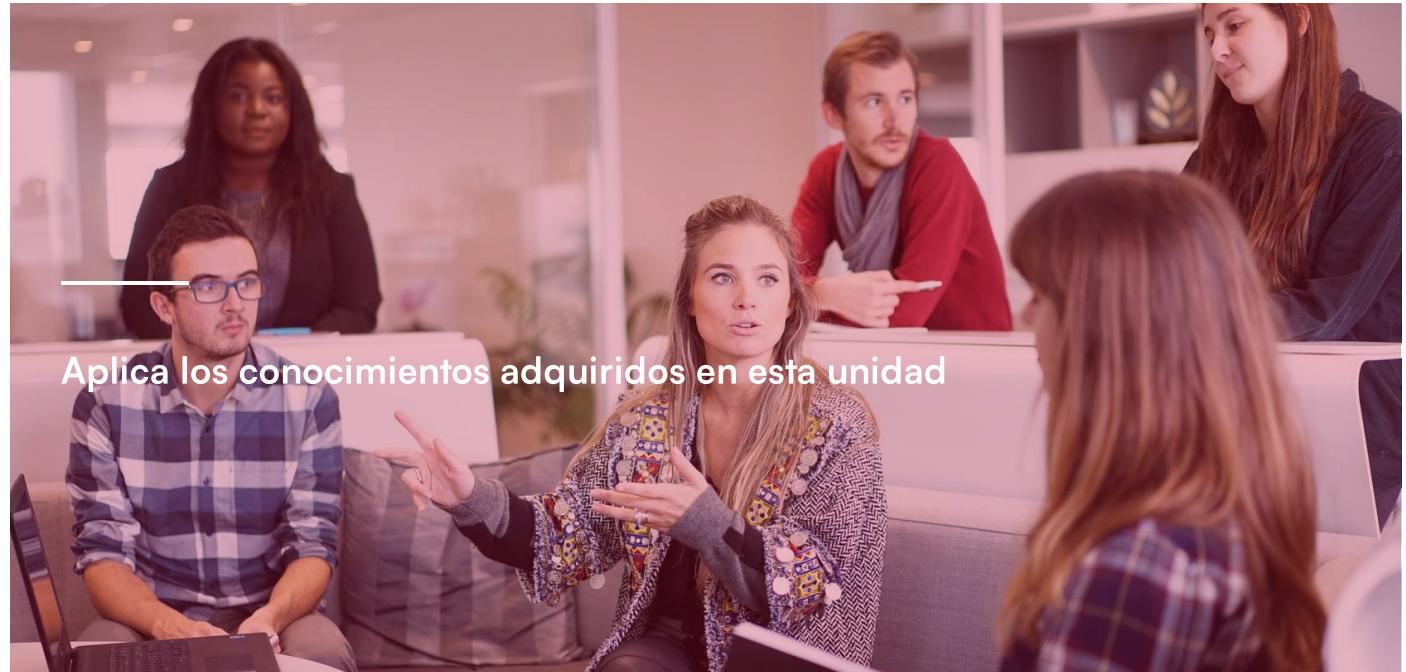
No obstante, todavía es fácil encontrarse con fuentes de datos que utilizan este formato y, por tanto, es fundamental dominarlo.

En la unidad se han introducido dos filosofías distintas a la hora de procesar un documento XML:

- **SAX**: es un procesador bastante eficiente que permite manejar documentos muy extensos en tiempo lineal. Permite leer el contenido de un XML y ejecutar acciones condicionadas a su contenido.
- **DOM**: es un procesador que construye en memoria una representación del documento, que permite su manipulación y modificación.

Asimismo, se han presentado tres módulos alternativos en Python, que siguen una de las dos filosofías mencionadas: `xml.sax` para el procesamiento en modo SAX, y `xml.dom` y `xml.etree` para el procesamiento en modo DOM.

## VIII. Caso práctico con solución



Aplica los conocimientos adquiridos en esta unidad

### ENUNCIADO

Considérese el archivo CSV “PitchingPost” adjunto, que contiene información sobre béisbol. Se puede descargar en este enlace\*.

### DATOS

Se deben realizar las funciones en Python que posibiliten las siguientes acciones:

1. Crear un nuevo archivo CSV denominado “AcumAnnos.csv” que contenga la frecuencia de los años. Tendrá la estructura:

Años,Frecuencia

1980,2

1981,6

....

2. Crear un nuevo archivo CSV denominado “AcumJugadores.csv” que contenga la frecuencia de los jugadores. Tendrá la estructura: Jugador,Frecuencia

bystroma01,2

carltst01,5

....

3. Crear un nuevo archivo CSV denominado “Ordenado.csv” que ordene información por el nombre del jugador.

\*



PitchingPost\_csv.zip

95.6 KB



**VER SOLUCIÓN**

## SOLUCIÓN

```

import csv
# Función que escribe un diccionario en un fichero CSV, con el nombre que recibe.
def escribe(map, fileName):
    outFile=open(fileName, "w")
    outWriter = csv.writer(outFile)
    for key in map:
        outWriter.writerow([key, map.get(key)])
    outFile.close()
    print ('Archivo ' + fileName + ' creado con exito.')

# Función que, dada una columna, contabiliza las veces que aparece cada
# término de esa columna en el archivo "PitchingPost.csv"
def frecuencia(column, fileName):
    error = False
    map = {}
    try:
        file=open("PitchingPost.csv")
    except IOError:
        error = True
        print ("Error al intentar abrir el archivo \"PitchingPost.csv\"")
    if (error == False):
        reader=screamer(file)
        data=list(reader)
    for i in range(1, len(data)):
        key=str(data[i][column])
        if map.get(key):
            map[key]=map[key]+1
        else:
            map[key]=1
    file.close()
    escribe (map, fileName)

def sort():
    error = False
    try:
        file=open("PitchingPost.csv")
    except IOError:
        error = True
        print ("Error al intentar abrir el archivo \"PitchingPost.csv\"")
    if (error == False):
        reader=csv.reader(file)
        lista = []
        for linea in reader:
            lista.append(linea)
        file.close()
        lista.sort()
        outFile=open("Ordenado.csv", "w")
        outWriter = csv.writer(outFile)
        for row in lista:
            outWriter.writerow(row)
        outFile.close()
        print ("Archivo \"PitchingPost\" ordenado con exito.")

frecuencia(1, "AcumAnnos.csv")

```

```
frecuency(0, "AcumJugadores.csv")  
sort()
```

La ejecución del caso práctico se muestra en el *notebook* “CASO PRÁCTICO.ipynb”, en este enlace:



### Consejo:

Se recomienda abrir el cuaderno con Jupyter Notebook y ejecutarlo paso a paso. También se facilita el archivo “Ordenado.csv”, que es la salida del caso práctico, en el siguiente enlace:



## IX. Lecturas recomendadas

---

- Python.org, [CSV File Reading and Writing](#).
- Python.org, [JSON encoder and decoder](#).
- Python.org, [Módulos de procesamiento XML](#).

## X. Glosario

---



El glosario contiene términos destacados para la comprensión de la unidad

### Fuente de datos



Repositorios de información donde se pueden descargar los datos en algún formato de datos específico.

### Formato de datos



Forma determinada de almacenar la información con una estructura de organización propia.

### Información desestructurada



Información que, al almacenarla, no sigue una estructura regular.

## **CSV** —

Formato de datos en el que la información se almacena por líneas y los valores separados usando un delimitador fijado. Es un formato que no tiene una estructura compleja.

## **JSON** —

Formato de datos en el que la información se almacena como una secuencia de pares clave-valor, así como otras estructuras como arrays. Es un formato recursivo. De manera abstracta representa una colección de valores de distintos tipos agrupados bajo un único nombre.

## **XML** —

Formato de datos que organiza la información mediante conjuntos de etiquetas –también llamados elementos o marcas– definidas por el propio usuario. La estructura lógica es definida por el propio usuario en función de la forma en la que se combinan las diferentes etiquetas.

## **Lenguaje de marcado** —

Lenguaje de etiquetas que se define usando el lenguaje XML.

## **Procesamiento de la información** —

Consiste en realizar algún tipo de acción de modificación, acceso o transformación sobre información.

## **Procesador** —

Programa que lleva a cabo el procesamiento de la información.

## **Procesamiento dirigido por eventos** —

Tipo de procesamiento sobre documentos XML que consiste en ir realizando acciones al mismo tiempo que el procesador reconoce o accede a los elementos de información. Un modelo de procesamiento dirigido por eventos es SAX.

## **Procesamiento en forma de árbol** —

Tipo de procesamiento sobre documentos XML que consiste en representar la información en forma de árbol en memoria, de manera que los procesamientos se basan en navegar por los elementos de información que se encuentran en el árbol. Un modelo de procesamiento en forma de árbol es DOM.

## **ElementTree** —

Conjunto de métodos de procesamiento de XML propios de Python que no sigue un modelo de procesamiento concreto, pero permite simular los principales modelos de procesamiento XML.

## XI. Bibliografía

---

- ["ContentHandler Objects"](#). Documentación oficial librería xml.sax.
- ["CSV File Reading and Writing"](#). Python.
- DictReader y DictWriter. ["The Python Standard Library"](#).
- ["Extensible library for opening URLs"](#). Python.
- ["Introduction to the DOM. Mozilla"](#).
- Jones, C. A. y Drake Jr, F. L. *Python and XML*. O'Reilly Media; 2001.
- ["JSON encoder and decoder"](#). Python.
- Marzal Varó, A., Gracia Luengo, I. y García Sevilla, P. ["Introducción a la programación con Python 3"](#). Castelló de la Plana: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions; 2014.
- ["Miscellaneous operating system interfaces"](#). Python.
- O'Reilly. ["Event Driven XML Processing"](#).
- ["Sample CSV Data"](#). SpatialKey Support.
- Sarasa Cabezuelo, A. *Gestión de la Información Web*. Editorial UOC; 2016.
- The Python Standard Library – ["Python 3.9.5 documentation"](#).
- ["What is a DTD?"](#). W3Schools.
- ["XPath Tutorial"](#). W3Schools.