

Fundamentos de tecnologías de Internet © EDICIONES ROBLE, S.L.

Índice

I. Introducción	3
II. Objetivos	4
III. Formatos de almacenamiento de datos en Internet	5
IV. Manipulación de documentos CSV	6
V. Manipulación de documentos JSON	12
VI. Manipulación de documentos XML	18
VII. Resumen	41
VIII. Caso práctico	42
Recursos	46
Bibliografía	46
Glosario.	46

I. Introducción

En el ámbito del análisis de datos, un elemento clave son los datos que van a ser analizados. Cuando se recuperan los datos de las fuentes de información, pueden almacenarse en diversos formatos. La diferencia entre ellos se encuentra en la forma en que se organiza la información y, por tanto, en la manera de procesar y acceder a la misma.

En general, los datos que se procesan en este ámbito son semiestructurados o sin estructura, en cantidades masivas y, a veces, presentan incorrecciones o faltan. Actualmente, existe un conjunto de formatos de datos cuyo uso se ha extendido para su distribución.

En esta unidad, se van a estudiar los principales formatos para almacenar y distribuir datos. En particular, se estudiará el formato CSV, JSON y XML. Se analizará su estructura y los métodos existentes en Python para realizar su procesamiento.

II. Objetivos

Los objetivos que los alumnos alcanzarán tras el estudio de esta unidad son:



- Conocer los principales formatos de almacenamiento de datos.
- Conocer los métodos definidos en Python para procesar cada tipo de formato de datos.
- Entender la información codificada en cada formato de datos.
- Saber codificar y diseñar un documento en cada formato para almacenar datos.
- 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 4.1., 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 4.1. Ejemplo de documento CSV.

Características

Algunas características de los archivos CSV:

- Los valores no tienen tipos, 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

Ventajas

Las principales ventajas que ofrece este formato son:

- Es simple.
- Permiten representar los datos de las hojas de cálculo.
- Puede ser visualizado por los editores de texto.

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() sobre cada línea para obtener los valores.

Inconveniente

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

El módulo CSV de Python permite leer y escribir archivos CSV. 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 4.2., se muestra un ejemplo.

```
import csv
archivoEjemplo=open("Ejemplo.csv")
ejemploLector=csv.reader(archivoEjemplo)
ejemploDatos=list(ejemploLector)
print(ejemploDatos)
```

Figura 4.2. Ejemplo de lectura de documento CSV.

Apertura y lectura

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 ser usado 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.

Acceso a valores

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 la lista de listas y representa el índice del elemento de esa lista al que se quiere acceder.

```
>>> ejemploDatos[0][0]
'04/05/2015'
>>> ejemploDatos[0][1]
'13:34'
>>> ejemploDatos[0][2]
'Manzanas'
>>> ejemploDatos[1][1]
'3:41'
>>> ejemploDatos[6][1]
'2:40'
```

Figura 4.3. Acceso a los datos.

También es posible usar el objeto Reader en un bucle for (figura 4.4.), de forma que se itera sobre las líneas del objeto. Cada línea es una lista de valores.

```
import csv
archivoEjemplo=open("Ejemplo.csv")
ejemploLector=csv.reader(archivoEjemplo)
for linea in ejemploLector:
    print("Linea #"+ str(ejemploLector.line_num)+ " "+str(linea))

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

Figura 4.4. Acceso a los datos usando un for.

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

Escritura

Para escribir datos en un archivo CSV se utiliza un objeto Writer que puede ser construido usando el método csv.writer(), tal como se ilustra en la figura 4.5.



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

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

Figura 4.5. Escritura sobre un archivo CSV.

- 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.

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 4.6.)



```
import csv
archivoSalida=open("C:\Prueba\Salida.csv", "w")
salidaEscriptor= csv.writer(archivoSalida)
salidaEscriptor.writerow(["naranjas","limones","peras","uvas"])
salidaEscriptor.writerow(["jamon","chorizo , de salamanca","queso","salchichón"])
salidaEscriptor.writerow([1,3,4,6])
archivoSalida.close()
```

Figura 4.6. Escritura de valores con comas.

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 4.7.).



```
import csv
archivoSalida=open("C:\Prueba\Salida.csv", "w")
salidaEscriptor= csv.writer(archivoSalida, delimiter="\t",lineterminator='\n\n')
salidaEscriptor.writerow(["naranjas","limones","peras","uvas"])
salidaEscriptor.writerow(["jamon","chorizo , de salamanca","queso","salchichón"])
salidaEscriptor.writerow([1,3,4,6])
archivoSalida.close()
```

Figura 4.7. Escritura de valores con otros separadores.

En el ejemplo 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).

Ejemplo de programa

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:

Encontrar los ficheros CSV

Es necesario crear un bucle sobre una lista de todos los archivos del directorio para saltarse aquellos que no son CSV (figura 4.8.). Se usa el método os.listdir() para recuperar todos los archivos del directorio actual y se comprueba para cada uno de ellos si su extensión es .csv.

```
print('Eliminando cabeceras de' + csvFilename + '...')

# Leer el archivo cvs 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()
```

Figura 4.8. Bucle de lectura.

Leer el contenido

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 4.9.).

```
# 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()
```

Figura 4.9. Lectura del contenido.

Programa completo

```

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 línea
    csvRows = []
    csvFileObj = open(csvFilename)
    readerObj = csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue # Saltar primera línea
        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()

```

Figura 4.10. Ejemplo completo.

Salida del programa

La salida del programa sería la que aparece en la figura 4.11.:

```

Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014
tel) on win32
Type "copyright", "credits" or "license()" for
>>> ===== RESTART =
>>>
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
Removing header from NAICS_data_1657.csv...
Removing header from NAICS_data_1751.csv...
Removing header from NAICS_data_1814.csv...
Removing header from NAICS_data_1817.csv...
Removing header from NAICS_data_1889.csv...
Removing header from NAICS_data_1952.csv...
Removing header from NAICS_data_1973.csv...
Removing header from NAICS_data_2066.csv...
Removing header from NAICS_data_2092.csv...
Removing header from NAICS_data_2183.csv...

```

Figura 4.11. Salida del programa.

V. Manipulación de documentos JSON

Características

JSON (JavaScript Object Notation) es un formato de datos que posee 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

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 }.



Por ejemplo, considérese que 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 (figura 4.12.):

```
{
  "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"]
  }
}
```

Figura 4.12. Documento JSON

El módulo JSON

Para gestionar documentos JSON desde Python, se usa el módulo JSON que permite la traducción de datos JSON en valores de Python.

Obsérvese que:

- 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 tales como ficheros, expresiones regulares, etc.

Lectura

Para traducir una cadena que contiene datos JSON en un valor de Python, se utiliza el método json.loads(). En la figura 4.13., se muestra un ejemplo:

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

Figura 4.13. Ejemplo de lectura de cadena JSON.

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.

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. La tabla de correspondencia entre JSON y valores Python (tabla 4.2.):

JSON	Python
object	dict
array	list
object	dict
string	unicode
number(int)	int,long
number(real)	float
true	True
false	False
null	None

Tabla 4.2. Correspondencia entre JSON y Python.

Escritura

Para escribir un valor de Python como una cadena de datos JSON, se usa el método json.dumps(). En la figura 4.14., se muestra un ejemplo.

```
PythonDatos={'nombre': 'Sofia', 'matriculado':True, 'asignaturas':34, 'ID':None}
import json
JSONDatos=json.dumps(PythonDatos)
print(JSONDatos)
```

Figura 4.14. Escritura de datos JSON.

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



Python	JSON
dict	object
list, tuple	array
str.unicode	string
int,long,float	number
True	true
False	false
None	null

Tabla 4.1. Tabla de correspondencia entre Python y JSON.

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

- Leer desde teclado una ciudad.
- Llamar a la API de geocodificación de Google.
- Extraer la información en formato JSON que nos devuelve.

Para implementarlo:

Lectura de datos y geocodificación de Google

El programa toma la cadena de búsqueda que el usuario introduce por teclado y se construye una URL tomando la cadena introducida. Mediante urllib se recupera el texto en JSON que la API de geocodificación de Google devuelve (figura 4.15.):

```
import urllib
import json
serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'
while True:
    address = input('Entrar ciudad: ')
    if len(address) < 1 : break
    url = serviceurl + urllib.urlencode({'sensor':'false',
                                           'address': address})
    print ('Recuperando', url)
    uh = urllib.urlopen(url)
    data = uh.read()
    print ('Recuperados',len(data),'caracteres')
```

Figura 4.15. Recuperación de texto.

Análisis de datos e impresión

Una vez recuperados los datos JSON se analizan y se muestran (figura 4.16.):

```
try: js = json.loads(str(data))
except: js = None
if 'status' not in js or js['status'] != 'OK':
    print ('===== Fallo de recuperación =====')
    print (data)
    continue
print (json.dumps(js, indent=4))
lat = js["results"][0]["geometry"]["location"]["lat"]
lng = js["results"][0]["geometry"]["location"]["lng"]
print ('lat:',lat,'lng:',lng)
location = js['results'][0]['formatted_address']
print (location)
```

Figura 4.16. Impresión de resultados.

Programa completo

El programa completo sería (figura 4.17.):

```
import urllib.parse
import urllib.request
import json
serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'
while True:
    address = input('Entrar ciudad: ')
    if len(address) < 1 : break
    url = serviceurl + urllib.parse.urlencode({'sensor':'false', 'address': address})
    print ('Recuperando', url)
    uh = urllib.request.urlopen(url)
    data = uh.read()
    print ('Recuperados',len(data), ' caracteres')
    try: js = json.loads(data)
    except: js = None
    if "status" not in js or js['status'] != 'OK':
        print ('==== Fallo de recuperación ====')
        print (data)
        continue
    print (json.dumps(js, indent=4))
    lat = js["results"][0]["geometry"]["location"]["lat"]
    lng = js["results"][0]["geometry"]["location"]["lng"]
    print ('lat:',lat,'lng:',lng)
    location = js['results'][0]['formatted_address']
    print (location)
```

Figura 4.17. Programa completo.

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 (figura 4.18.):

```
<cuenta>
  <numero_cuenta>C-101</numero_cuenta>
  <nombre_sucursal>Centro</nombre_sucursal>
  <saldo>500</saldo>
</cuenta>
<cliente>
  <nombre_cliente>González</nombre_cliente>
  <calle_cliente>Arenal</calle_cliente>
  <ciudad_cliente>La Granja</ciudad_cliente>
</cliente>
<impositor>
  <numero_cuenta> C-101</numero_cuenta>
  <nombre_cliente>González</nombre_cliente>
</impositor>
</banco>
```

Figura 4.18. Ejemplo de documento XML.

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.

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 elementos del documento. En el primer ejemplo, el elemento <banco> era la raíz.

Atributos

Las etiquetas de los elementos pueden incluir 1 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.

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.

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.

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>]]>
```

Ejemplo completo

Para ilustrar el uso de XML, supongamos 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 (figura 4.19.):



```
<?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> José Manzaneda </nombre>
        <email> j.manzaneda@direccion.com </email>
        <email> jman@hotmail.com </email>
        <relacion enemigo_de="leire" amigo_de="ricky"/>
    </persona>
</listin>
```

Figura 4.19. Ejemplo de documento XML.

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

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. SAX es un estándar para este enfoque.

Manipulación del árbol

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

Se van a estudiar 3 herramientas de procesamiento XML:

- Procesamiento basado en SAX.

- Procesamiento basado en DOM.
- Herramienta de procesamiento específica de Python: ElementTree.

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.

Características

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 consideramos 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

El manejador ContentHandler

Para procesar usando SAX, es necesario crearse un manejador propio ContentHandler como subclase de `xml.sax.ContentHandler`. El manejador gestionará las etiquetas y atributos que se deseen del documento XML que va a ser procesado.

Métodos

El manejador proporciona un conjunto de métodos para gestionar determinados eventos que se producen en el procesamiento:

- 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.

Ejemplo completo

Se va a crear un manejador para procesar el documento de ejemplo:

- Hay que importar el paquete SAX: `import xml.sax`.
- Se crea una clase que es subclase de la clase `xml.sax.ContentHandler`.
- Se definen dentro de la clase 4 métodos:

El método init

Se definen como atributos de la clase las etiquetas y atributos del documento XML que se quieren gestionar.

```
''' def init(self):
    self.Datos=""
    self.titulo=""
    self.fecha=""
    self.autor="'''
```

El método startElement

Se indica qué acciones se quieren llevar a cabo cuando se encuentre el comienzo de un elemento. En el ejemplo, se quiere capturar el isbn del libro:

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

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.

```
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=""
```

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 imprimido por pantalla.

```
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
```



La clase completa sería (figura 4.20.):

```
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 4.20. Programa completo en SAX

campusformacion.imf.com © EDICIONES ROBLE, S.L.
IVAN GARCIA GARCIA

© EDICIONES ROBLE, S.L.

Procesamiento

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.



Una vez que se tiene definida la clase, se puede llevar a cabo el procesamiento (figura 4.21.):

```

parser=xml.sax.make_parser()
parser.setFeature(xml.sax.handler.feature_namespaces,0)
Handler=ManejadorCatalogo()
parser.setContentHandler(Handler)
parser.parse("Catalogo.xml")

****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, Pepe, Isabel
****Libro****
isbn: 0-596-10046-9
Titulo: Python en 100 paginas
Fecha: Julio 2006
Autor: Julia

```

Figura 4.21. Procesamiento.

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.

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(Elemento)`**: devuelve una lista de todos los elementos cuyo nombre ha sido proporcionado.
- **`getAttribute(Atributo)`**: devuelve el valor del atributo proporcionado como parámetro.
- **`hasAttribute(Atributo)`**: indica si un elemento tiene el atributo proporcionado como parámetro.

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

Se va a realizar el procesamiento del documento XML de ejemplo usando DOM (figura 4.22.):

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

ArboldOM=xml.dom.minidom.parse("Catalogo.xml")
catalogo=ArboldOM.documentElement
libros=catalogo.getElementsByTagName("Libro")
for libro in libros:
    print ("***Libro***")
    if libro.hasAttribute("isbn"):
        print ("isbn:",libro.getAttribute("isbn"))
    Titulo=libro.getElementsByTagName("titulo") [0]
    print ("Titulo:",Titulo.childNodes[0].data)
    Fecha=libro.getElementsByTagName("fecha") [0]
    print ("Fecha:",Fecha.childNodes[0].data)
    Autor=libro.getElementsByTagName("autor") [0]
    print ("Autor:",Autor.childNodes[0].data)
```

Figura 4.22. Procesamiento usando DOM.



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 de 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 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)

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 siguiente documento (figura 4.23.):



```
<catalogo>
    <Libro isbn="0-596-00128-2">
        <titulo>Python y XML</titulo>
        <fecha>Diciembre 2001</fecha>
        <autor>Pepito Perez</autor>
    </Libro>
    <Libro isbn="0-596-15810-6">
        <titulo>Programacion avanzada de XML</titulo>
        <fecha>Octubre 2010</fecha>
        <autor>Juan Garcia</autor>
    </Libro>
    <Libro isbn="0-596-15806-8">
        <titulo>Aprendiendo Java</titulo>
        <fecha>Septiembre 2009</fecha>
        <autor>Juan Garcia</autor>
    </Libro>
```

Figura 4.23. Ejemplo de documento XML.

Procesamiento con ElementTree

El método open()

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 (figura 4.24.).



```
from xml.etree import ElementTree  
  
f= open("Catalogo.xml", "rt")  
arbol=ElementTree.parse(f)  
print(arbol)  
  
...  
<xml.etree.ElementTree.ElementTree object at 0x0219A070>  
...
```

Figura 4.24. Procesamiento usando ElementTree.

El método iter()

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 4.25.).



```

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

```

```

catalogo {}
Libro {'isbn': '0-596-00128-2'}
titulo {}
fecha {}
autor {}
Libro {'isbn': '0-596-15810-6'}
titulo {}
fecha {}
autor {}
Libro {'isbn': '0-596-15806-8'}
titulo {}
fecha {}
autor {}
Libro {'isbn': '0-596-15808-4'}
titulo {}
fecha {}
autor {}

```

Figura 4.25. Procesamiento usando `iter()`.

Puede que solo nos interesen determinados elementos del árbol, y no en todos. Para ello, se pasa como parámetro del método `iter()` el nombre del elemento de interés (figura 4.26.).



```

from xml.etree import ElementTree
f= open("Catalogo.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=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 4.26. Procesamiento usando `iter()` con parámetros.

Iterar sobre hijos

Otra posibilidad de iterar sobre los elementos del árbol es acceder a la raíz del árbol y desde ella iterar sobre los hijos (figura 4.27.):

```

import xml.etree.ElementTree as ET
arbol=ET.parse("Catalogo.xml")
raiz=arbol.getroot()
print(raiz.tag, " ", raiz.attrib)
for hijo in raiz:
    print(hijo.tag, " ", hijo.attrib)

```

```

catalogo {}
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 4.27. Procesamiento iterando sobre elementos del árbol.

Acceso indexado

También es posible acceder a los elementos de forma indexada (figura 4.28.).

```
import xml.etree.ElementTree as ET
arbol=ET.parse("Catalogo.xml")
raiz=arbol.getroot()
print("Titulo :",raiz[0][0].text)
```

>>>
Titulo : Python y XML

Figura 4.28. Acceso indexado a los elementos.

Otros métodos

Existe otro conjunto de métodos que permiten recorrer el árbol tomando como argumento una expresión XPath que caracteriza al elemento que se está buscando:

- **find()**: recupera el primer subelemento del elemento actual encajando con la descripción dada.
- **.findall()**: recupera todos los subelementos del elemento actual encajando con la descripción dada.
- **iterfind()**: recupera todos los elementos encajando 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 4.29.:



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

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

Figura 4.29. Búsqueda de libros usando findall().

Procesamiento basado en eventos con iterparse()

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 4.30.):



```
from xml.etree.ElementTree import iterparse
for (event, element) in iterparse("Catalogo.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
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
```

Figura 4.30. Simulación del procesamiento SAX.

Modificación de documentos XML con fromstring

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 4.31.).



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

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

Figura 4.31. Uso de fromstring.

Modificación de un documento XML leído

Otra posibilidad que ofrece la API 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().

Se va a modificar el documento XML de ejemplo:

- Se va a añadir un nuevo atributo que indica el orden.
- Se va a añadir un nuevo elemento que indica la editorial.
- Se va a añadir un nuevo atributo que indica si hay ejemplares.



El resultado sería el siguiente (figura 4.32.):

```

import xml.etree.ElementTree as ET
arbol=ET.parse("Catalogo.xml")
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=i+1

arbol.write("Catalogo2.xml")

```

<catalogo>
 <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>

Figura 4.32. Modificación del documento XML.

Eliminar elementos con Element.remove()

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 4.33.).

```

import xml.etree.ElementTree as ET
arbol=ET.parse("Catalogo2.xml")
raiz=arbol.getroot()
for libro in raiz.iter("Libro"):
    orden=int(libro.get("orden"))
    if orden== 3:
        raiz.remove(libro)
arbol.write("Catalogo3.xml")

```

<catalogo>
 <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>

Figura 4.33. Eliminación de elementos.

Creación de documentos XML desde 0

También es posible la creación de documentos XML desde cero. Para ello se disponen de 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.

En el siguiente ejemplo (figura 4.34.) se va a crear un documento XML con información de un libro semejante a los ejemplos anteriores.




```

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=" ")

raiz=Element("Catalogo")
Libro=SubElement(raiz, "Libro")
Titulo=SubElement(Libro, "titulo")
Titulo.text="Python y XML"
Fecha=SubElement(Libro, "fecha")
Fecha.text="Diciembre 2001"
Autor=SubElement(Libro, "autor")
Autor.text="Pepito Perez"
print(prettify(raiz))

```




Figura 4.34. Ejemplo de creación de un documento XML.

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 4.35.).




```

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

raiz=Element("Catalogo")
Libro=SubElement(raiz, "Libro")
Titulo=SubElement(Libro, "titulo")
Titulo.text="Python y XML"
Fecha=SubElement(Libro, "fecha")
Fecha.text="Diciembre 2001"
Autor=SubElement(Libro, "autor")
Autor.text="Pepito Perez"
print(tostring(raiz))

```




Figura 4.35. Impresión del documento XML.

Añadir atributos

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.

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 4.36.).



```

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')
    reparsored = minidom.parseString(rough_string)
    return reparsored.toprettyxml(indent=" ")
    
raiz=Element("Catalogo")
raiz.set("version","1.0")
Libro=SubElement(raiz,"Libro", {"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
Titulo=SubElement(Libro,"titulo")
Titulo.text="Python y XML"
Fecha=SubElement(Libro,"fecha")
Fecha.text="Diciembre 2001"
Autor=SubElement(Libro,"autor")
Autor.text="Pepito Perez"
print(prettify(raiz))

```

Figura 4.36. Adición de varios campos y valores.

En la figura 4.37. se muestra el resultado.



```

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

```

Figura 4.37. Resultado de la creación del programa.

Añadir hijos

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 (figura 4.38.).



```

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

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

```

Figura 4.38. Uso de extend.

También se podría haber construido pasando una lista (figura 4.39.).



```

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

def prettyfy(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("Catalogo")
raiz.set("version","1.0")
Libro/SubElement(raiz,"Libro", {"orden":"1","ejemplares":"si","isbn":"0-596-00128-2",})
ultimo=Element("ultimo")
ultimo.text="Python y XML"
ultimo=Element("ultimo")
ultimo.text="Diciembre 2001"
ultimo=Element("ultimo")
ultimo.text="Pepito Perez"
ultimo.extend([ultimo,ultimo,ultimo])
Libro.extend([ultimo])
print(prettyfy(raiz))

```

Figura 4.39. Otra versión pasando una lista.

Escribir en un archivo con write()

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 maneja documentos XML muy grandes, interesa guardarlo 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 4.40.).



```

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

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

```

Figura 4.40. Almacenamiento en archivo.

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 4.41.).



```

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

def prettyfy(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("Catalogo")
raiz.set("version","1.0")
Libro=SubElement(raiz,"Libro", {"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
titulo=Element("titulo")
titulo.text="Python y XML"
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(prettyfy(raiz))

```

Figura 4.41. Nueva versión del ejemplo anterior.

Programa completo con write() y resultado

El programa se muestra en la figura 4.42.

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

raiz=Element("Catalogo")
raiz.set("version","1.0")
Libro=SubElement(raiz,"Libro", {"orden":"1","ejemplares":"si","isbn":"0-596-00128-2"})
titulo=Element("titulo")
titulo.text="Python y XML"
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 4.42. Nueva implementación.

El resultado del procesamiento se muestra en la figura 4.43.

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

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

text
Python y XMLDiciembre 2001Pepito Perez
```

Figura 4.43. Resultado del procesamiento.

VII. Resumen

En esta unidad, se han introducido los principales formatos de almacenamiento de datos y distribución desde las fuentes de información. Concretamente se han visto los formatos CSV, JSON y XML.

Para cada uno de los casos, se ha analizado cómo se organiza la información en cada uno de estos formatos y se han mostrado los métodos en Python que permiten procesar y acceder a la información que se encuentra almacenada en ellos.

En el caso de XML, se han visto tres métodos diferentes de procesamiento. Dos métodos clásicos: uno dirigido por eventos y procesamiento en árbol; y otro método de procesamiento propio de Python denominado ElementTree, que utiliza una sintaxis propia de Python y que permite simular los otros tipos de procesamiento.



En el siguiente enlace, puedes descargar un notebook que contiene el código necesario para procesar **CSV, JSON y XML**: [ENLACE DESCARGA](#)

VIII. Caso práctico

Considérese el archivo CSV PitchingPost adjunto que contiene información sobre Baseball.



[Descargar archivo PitchingPost.csv](#)

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

Crear AcumAnnos.csv

Crea un nuevo archivo CSV denominado AcumAnnos.csv que contenga la frecuencia de los años. Tendrá la estructura:

Años,Frecuencia

1980,2

1981,6

....

Crear AcumJugadores.csv

Crea un nuevo archivo CSV denominado AcumJugadores.csv que contenga la frecuencia de los jugadores. Tendrá la estructura:

Jugador,Frecuencia

bystrma01,2

carltst01,5

....

Crear Ordenado.csv

Crea un nuevo archivo CSV denominado Ordenado.csv que ordene información por el nombre del jugador.

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 éxito.')

# Función que, dada una columna, contabiliza las veces que aparece cada
# término de esa columna en el archivo "PitchingPost.csv"

def frecuency(column, fileName):

    error = False

    map = {}

    try:

        file=open("PitchingPost.csv")

    except IOError:

        error = True

        print ("Error al intentar abrir el archivo l\"PitchingPost.csv\".")

    if (error == False):

        reader=csv.reader(file)

        data=list(reader)

        for i in range(column, len(data)):

            key=str(data[i][column])

            if map.has_key(key):

                map[key]=map[key]+1

            else:

                map[key]=1
```

```
file.close()

escribe (map, fileName)

# Función que guarda el contenido del archivo "PitchingPost.csv" en una lista,
# para después guardarla ordenada en "Ordenado.csv"

def sort():

    error = False

    try:

        file=open("PitchingPost.csv")

    except IOError:

        error = True

        print ("Error al intentar abrir el archivo l\"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 l\"PitchingPost\" ordenado con éxito.")
```

```
frecuencia(1, "AcumAnnos.csv")  
frecuencia(0, "AcumJugadores.csv")  
sort()
```

Recursos

Bibliografía

- **Jones, Christopher A. y Drake Jr., Fred L.** *Python and XML*. O'Reilly Media; 2001.:
- **Marzal Varó, Andrés, Gracia Luengo, Isabel, y García Sevilla, Pedro.**: Marzal Varó, Andrés, Gracia Luengo, Isabel, y García Sevilla, Pedro. *Introducción a la programación con Python 3*. Castelló de la Plana: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions; 2014. [En línea] URL disponible en http://repositori.uji.es/xmlui/bitstream/handle/10234/102653/s93_impressora.pdf?sequence=2&isAllowed=y
- **Sarasa Cabezuelo, Antonio.** *Gestión de la Información Web*. Editorial UOC; 2016.:

Glosario.

- **CSV**: es un 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.
- **ElementTree**: es un 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.
- **Formato de datos**: es una forma determinada de almacenar la información con una estructura de organización propia.
- **Fuente de datos**: son aquellos repositorios de información donde se pueden descargar los datos en algún formato de datos específico.
- **Información desestructurada**: es la información que al almacenarla no sigue una estructura regular.
- **JSON**: es un 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.
- **Lenguaje de marcado**: es un lenguaje de etiquetas que se define usando el lenguaje XML.
- **Procesador**: es el programa que lleva a cabo el procesamiento de la información.
- **Procesamiento de la información**: consiste en realizar algún tipo de acción de modificación, acceso o transformación sobre información.
- **Procesamiento dirigido por eventos**: se trata de un 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**: se trata de un 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.

- **XML:** es un 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 base a la forma en la que se combinan las diferentes etiquetas.