

2 WEB SCRAPING

INTRODUCCIÓN

La Web es una fuente inagotable de información. Blogs, foros, sitios oficiales que publican datos de interés... todos ofrecen información que, recopilada de forma concienzuda, puede sernos de gran utilidad: evolución y comparativa de precios en tiendas *online*, detección de eventos, análisis de la opinión de usuarios sobre productos, crear *chatbots*, etc.

El límite es el que ponga nuestra imaginación, aunque siempre, por supuesto, dentro de la legislación vigente. Debemos recordar que algunos de estos datos pueden estar protegidos por derechos de autor, patentes, etc., así que es nuestra responsabilidad ser cuidadosos y consultar las posibilidades que ofrece la página. Además, muchos sitios web pueden llegar a bloquear nuestra IP si detectan que estamos accediendo a sus datos sin permiso.

En este sentido, hay que mencionar el concepto de *datos abiertos*, adoptado cada vez más por organizaciones privadas y públicas, que busca proporcionar datos accesibles y reutilizables, sin exigencia de permisos específicos y que logra que cada día tengamos más datos disponibles.

En todo caso, el problema es: ¿cómo “capturar” esta información?

Las páginas pueden ofrecernos información en 3 formatos principales.

La primera posibilidad es que las propias webs incluyan ficheros ya preparados en alguno de los formatos vistos en el capítulo 1 en forma de ficheros que se pueden descargar directamente. En tales casos, bastará con descargar estos ficheros (en seguida veremos cómo) para poder analizarlos.

La segunda posibilidad, muy común en las webs preparadas para consultas, es que el propio sitio web ofrezca acceso a sus datos a través de un protocolo API-REST, al que podremos hacer peticiones sobre datos concretos, que obtendremos en formato JSON o XML. Hablaremos de esta posibilidad en el capítulo siguiente.

Finalmente, la tercera posibilidad es que la información forme parte de la propia página web, en un formato muy cómodo para los usuarios “humanos” que la visitan, pero poco tratable para un programa. Por fortuna, Python incluye bibliotecas que nos ayudarán a extraer esta información, es lo que se conoce propiamente con el término *web scraping*.

Dentro de esta última posibilidad, la de tener los datos incrustados como contenido de la propia página web, tenemos a su vez dos posibilidades. En la primera, la información está disponible simplemente accediendo a la URL (dirección web) de la página. En este caso, tras descargar el contenido de la página, emplearemos una biblioteca como BeautifulSoup, que nos permita buscar la información dentro del código de la página.

En otros casos, cada vez más frecuentes, la página requerirá información por nuestra parte antes de mostrarnos los datos. Aunque sea más complicado, también podremos hacerlo mediante bibliotecas como selenium, que nos permite hacer desde un programa todo lo que haríamos desde el navegador web.

La siguiente tabla muestra esta división, que también corresponde a la estructura de este capítulo:

Fuentes Web de datos	Biblioteca Python
Ficheros incluidos en la página web	requests, csv...
API-REST	Requests
Datos que forman parte de la página	BeatifulSoup
Datos que requieren interacción	Selenium

FICHEROS INCLUIDOS EN LA PÁGINA WEB

En ocasiones las organizaciones nos ofrecen los datos ya listos para descargar, con formatos como los que hemos visto en el capítulo uno. En estos casos lo más sencillo es utilizar una biblioteca como `requests`, que permite descargar el fichero en el disco local y tratarlo a continuación, o bien, si el fichero es realmente grande hacer un tratamiento en *streaming*, sin necesidad de mantener una copia local.

URIs, URLs y URNs

Estas tres palabras aparecen a menudo cuando se habla de páginas web, e interesa que tengamos una idea precisa de su significado.

Una URI (*Uniform Resource Identifier* o identificador de recursos uniforme) es un nombre, o formalmente una cadena de caracteres, que identifica un recurso de forma accesible dentro de una red: una página web, un fichero, etc. Su forma genérica es:

schema: `[//[user[:passwd]@]host[:port]][/path][?query][#tag]`

donde las partes entre corchetes son opcionales. El esquema inicial indica la “codificación” de la URI por ejemplo “http: ”. Después vienen, opcionalmente, un nombre de usuario y su palabra clave, seguidos de un nombre del *host*, el ordenador al que nos conectamos, o su correspondiente IP y un puerto de acceso. A toda esta parte de la URI con forma `(//[usuario[:passwd]@]host[:port])` se la conoce como *autoridad*. Después viene la ruta (*path*), que es una secuencia de nombres separadas por ya sea por los símbolos “/” o “:”. Finalmente, puede incluirse una posible consulta (*query*) y una etiqueta (*tag*) para acceder a un lugar concreto del documento. Un ejemplo de URI sería:

`https://es.wikipedia.org/wiki/Alan_Turing#Turing_en_el_cine`

En esta URI, el esquema viene dado por el protocolo `https`, la autoridad corresponde con `es.wikipedia.org` (no hay usuario ni palabra clave) y la ruta es la cadena `wiki/Alan_Turing`. La URI no contiene consultas, pero sí una etiqueta final, tras el símbolo “#”: `Turing_en_el_cine`. Para ver URIs que incluyan consultas basta con que nos fijemos en la dirección que muestra nuestro navegador cuando hacemos una búsqueda en Google o en YouTube; allí está nuestra consulta tras el símbolo “?” y normalmente en forma de parejas *clave=valor*, separadas por los símbolos “&” o “:”.

Por ejemplo `https://www.youtube.com/watch?v=iSh9qg-2qKw`. En este caso, la consulta se incluye con la forma `v=codificaciónDeLaConsulta`.

Las URIs se suelen dividir en dos tipos: URLs y URNs. Las URLs (*Uniform Resource Locator* o localizador de recurso uniforme), a menudo conocidas simplemente como *direcciones web*, sirven para especificar un recurso en la red mediante su localización y una forma o protocolo que permite recuperarlo, que corresponde con la parte *schema* de la estructura general de la URI que hemos mostrado anteriormente. La barra de las direcciones de nuestro navegador normalmente muestra URLs. Los ejemplos de URIs que hemos visto en los párrafos anteriores son, en particular, URLs. Además de esquemas o protocolos `http` o `https`, en ocasiones encontraremos URLs que especifican otros, como el *File Transfer Protocol* (FTP), por ejemplo en la siguiente URL: `ftp://example.com`.

Para complicar un poco la cosa, hay que mencionar que existen páginas web que incluyen algunos caracteres como “[” o “]” que no están contemplados en la definición técnica de URI, pero sí en la de URL, así que serían URLs pero no URIs. A pesar de estos casos excepcionales, no debemos olvidar que la idea general y “oficial” es que las URLs son tipos particulares de URIs.

Finalmente, una URN (*Uniform Resource Name*, o nombre de recurso uniforme), el otro tipo de URIs, identifica un recurso, pero no tienen por qué incluir ninguna forma de localizarlos. Un ejemplo puede ser `urn:isbn:0-391-31341-0`.

Dado que en este capítulo estamos interesados en acceder a recursos, y por tanto necesitaremos su localización, usaremos a partir de ahora únicamente URLs.

Ejemplo: datos de contaminación en Madrid

`http://www.mambiente.munimadrid.es/opendata/horario.txt`

Esta URL proporciona los datos de las estaciones de medición de la calidad del aire de la ciudad de Madrid. Supongamos que queremos descargar el fichero correspondiente para analizarlo a continuación. Para ello, nos bastará con incluir la biblioteca `requests` que incluye un método para “pedir” (`get`) el fichero, que podemos grabar a continuación en nuestro disco duro local:

```
>>> import requests
>>> url = "http://www.mambiente.munimadrid.es/opendata/horario.txt"
>>> resp = requests.get(url)
>>> print(resp)
```

Tras “bajar” el fichero, con `get`, mostramos con `print` la variable `resp` que nos mostrará por pantalla el llamado *status_code*, que nos informa del resultado de la descarga:

```
<Response [200]>
```

El valor 200 indica que la página se ha descargado con éxito. En la URL:

```
https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
```

podemos encontrar información detallada acerca de los distintos *status_code*, pero a nuestros efectos baste con decir que los que empiezan con 2 suelen indicar éxito, mientras que los que comienzan por 4 o 5 indican error, como el famosísimo 404 (recurso no encontrado) o el 403 (acceso prohibido al recurso).

Por comodidad vamos a grabar el fichero descargado en un archivo local, al que llamaremos `horario.txt`. El contenido de la página está en el atributo `content` de la variable `resp`:

```
>>> path = '...carpeta de nuestro disco local...'
>>> with open(path + 'horario.txt', 'wb') as output:
    output.write(resp.content)
```

Antes de ejecutar este fragmento de código, debemos reemplazar la variable `path` por una dirección en nuestro disco local. La construcción Python `with` asegura que el fichero se cerrará automáticamente al finalizar.

Ya tenemos el fichero en nuestro ordenador y podemos tratarlo. En nuestro ejemplo el fichero contiene los datos de contaminación del día en curso. El formato es:

- Columnas 0, 1, 2: concatenadas identifican la estación. Por ejemplo 28,079,004 identifica a la estación situada en la Plaza de España.
- Columnas 3, 4, 5: identifican el valor medido. Por ejemplo, si la columna 3 tiene el valor 12 indica que se trata de una medida de óxido de nitrógeno (las otras dos columnas son datos acerca de la medición que no vamos a usar).
- Columnas 6, 7, 8: año, mes, día de la medición, respectivamente.
- Columnas 9 – 56: indican el valor en cada hora del día. Van por parejas, donde el primer número indica la medición y el segundo es “V” si es un valor válido o “N” si no debe tenerse en cuenta. Por tanto, la columna 9 tendrá la medición a las 00 horas si la columna 10 es una “V”, la columna 11 la medición a las 01 horas si la columna 12 es una “V”, y así sucesivamente. En la práctica, el primer valor “N” corresponde con la hora actual, para la que todavía no hay medición.

Podemos usar esta información para mostrar los datos en formato de gráfica que indique la evolución de la contaminación por óxido de nitrógeno en un día cualquiera a partir de los datos de la estación situada en la Plaza de España de Madrid.

Para ello comenzamos por importar la biblioteca `csv`, ya que se trata de un fichero de texto separado por comas, y la biblioteca `matplotlib.pyplot` que nos permite hacer gráficos de forma muy sencilla, y que trataremos en más detalle en un capítulo posterior:

```
>>> import matplotlib.pyplot as plt
>>> import csv
```

Ahora abrimos el fichero y generamos el vector con los valores que corresponden a la línea de la Plaza de España para el óxido de nitrógeno. Lo que hacemos es recorrer las columnas por su posición, primero para seleccionar la línea que corresponde a la estación de la Plaza de España y para el valor 12 (óxido de nitrógeno), y, una vez localizada la línea con los datos, recorrer de la columna 9 en adelante añadiendo valores al vector `vs` hasta encontrar la primera “N” en la columna siguiente al valor:

```
>>> with open(path + 'horario.txt') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    for row in readCSV:
        if (row[0]+row[1]+row[2]=='28079004' and
            row[3]=='12'):
            plt.title("Óxido de nitrógeno: "
                      +row[8]+"/"+row[7]+"/"+row[6])
            hora = 0
            desp = 9
            vs = []
            horas = []
            while hora<=23:
                if row[desp+2*hora+1]=='V':
                    vs.append(row[desp+2*hora])
                    horas.append(hora)
                hora +=1
            plt.plot(horas, vs)
            plt.show()
```

La variable `desp` indica la columna dentro de cada fila del fichero en la que empiezan los valores medidos. En particular, para cada hora, la columna `desp+2*hora` contiene el valor medido en esa hora, que solo será válido si la columna `desp+2*hora+1` contiene una “V”. Tras recoger los valores válidos en el array `vs` y sus horas asociadas en el array `horas`, la llamada a `plot` crea la gráfica a

partir de dos vectores: uno con los valores x , que en este caso son las horas y otro, de la misma longitud, con los valores y , en este caso los valores medidos.

La figura 2-1 nos muestra un ejemplo para un día de diario. Vemos la subida en el nivel de contaminación en la hora punta de entrada, y un repunte, aunque de menor intensidad, a la hora de comer.

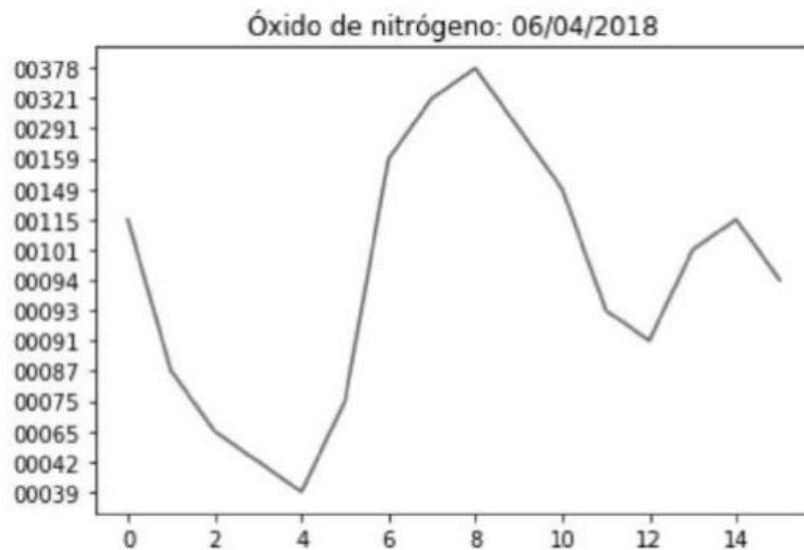


Figura 2-1. Evolución del óxido de nitrógeno entre las 0 y las 14h.

De esta forma, guardando los ficheros correspondientes a cada día podríamos calcular medias anuales, ver la evolución de la contaminación, o incluso intentar predecir los valores por anticipado.

Aunque en este caso el fichero es de reducido tamaño, en ocasiones los ficheros sobre los que queremos trabajar pueden ser realmente grandes, y puede que no nos interese descargarlos completos. En estos casos, podemos utilizar el procesamiento *perezoso*, que busca utilizar la menor información posible. Inicialmente, importamos las bibliotecas necesarias:

```
>>> import requests
>>> from contextlib import closing
>>> import csv
>>> import codecs
>>> import matplotlib.pyplot as plt
```

Llama la atención la incorporación de dos nuevas bibliotecas:

- `codecs`: utilizada para leer directamente los *strings* en formato `utf-8`.

- `contextlib`: nos permite leer directamente el valor devuelto por `requests.get`

Ahora el resto del código:

```
>>> url = "http://www.mambiente.munimadrid.es/opendata/horario.txt"
>>> with closing(requests.get(url, stream=True)) as r:
    reader = csv.reader(codecs.iterdecode(
        r.iter_lines(),
        'utf-8'),
        delimiter=',')
    for row in reader:
        ..... # igual que en el código anterior
```

Este código hace lo mismo que el anterior, pero evita:

- a) Tener que grabar el fichero en disco, gracias a la lectura directa de `request.get()`.
- b) Cargarlo completo en memoria, gracias a la opción `stream=True` de `requests.get()`.

DATOS QUE FORMAN PARTE DE LA PÁGINA

El siguiente método consiste en obtener los datos a partir de una página web. Para extraer los datos, necesitaremos conocer la estructura de la página web, que normalmente está compuesta por código *HTML*, imágenes, *scripts* y ficheros de estilo. La descripción detallada de todos estos componentes está más allá del propósito de este libro, pero vamos a ver los conceptos básicos que nos permitan realizar nuestro objetivo: hacer *web scraping*.

Lo que oculta una página web

Lo que vemos en el navegador es el resultado de interpretar el código HTML de la página. Veamos un ejemplo de una página mínima en HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      Un mini ejemplo
    </title>
  </head>
```



```
<body>
  <div id="date"> Fecha 25/03/2035 </div>
  <div id="content"> Un poco de texto </div>
</body>
</html>
```

Lo primero que vemos, es `<!DOCTYPE html>` que indica que se trata de un documento HTML, el estándar de las páginas web. A continuación, encontramos `<html>`, la etiqueta que marca el comienzo del documento, y que se cerrará al final del documento con `</html>`. En HTML, igual que en XML, los elementos tienen una estructura con la forma:

```
<elemento atributo="valor">Contenido</elemento>
```

El atributo no es obligatorio, y un mismo elemento puede incluir más de uno. Dentro del documento HTML, esto es entre la apertura `<html>` y el cierre `</html>` del elemento principal, encontramos siempre dos elementos:

- `<head> ... </head>`: describe la cabecera del documento, como puede ser su título, el autor, etc.
- `<body> ... </body>`: es el contenido en sí de la página, que es lo que nosotros deseamos examinar.

Dentro del elemento `body` habrá otros elementos, que dependerán de la página, que a su vez pueden contener otros elementos, y así sucesivamente. En nuestro pequeño ejemplo el cuerpo solo tiene dos frases, marcadas por las etiquetas `<div>` y `</div>`. Podemos grabar este código en un fichero con el nombre que deseemos, por ejemplo `mini.html`. Haciendo doble clic sobre el fichero se abrirá el navegador y veremos la (humilde) página.

Podemos cargar la página como un fichero de texto normal, y mostrarla mediante la biblioteca `BeautifulSoup`:

```
>>> from bs4 import BeautifulSoup
>>> url = r"c:\...\mini.html"
>>> with open(url, "r") as f:
    page = f.read()

>>> soup = BeautifulSoup(page, "html.parser")
>>> print(soup.prettify())
```

Para poder probar el ejemplo debemos incluir la ruta al fichero en la variable `url`. La variable `soup` contiene la página en forma de cadena de caracteres, que

convertimos en un formato interno estructurado usando la constructora BeautifulSoup, a la que indicamos como segundo elemento que se debe emplear el analizador sintáctico (parser) propio de HTML, en este caso “html.parser”. Si la página es muy compleja podemos necesitar otro analizador, como “html5lib”, más lento en el procesamiento, pero capaz de tratar páginas más complicadas.

En nuestro ejemplo el resultado del análisis se guarda en la variable soup. La última instrucción muestra el código HTML en un formato legible.

Un poco de HTML

Para hacer *web scraping* debemos identificar los elementos fundamentales que se pueden encontrar en un documento HTML. La siguiente lista no es, ni mucho menos, exhaustiva, pero sí describe los más habituales.

ELEMENTOS DE FORMATO

`...`: el texto dentro del elemento se escribirá en negrita. Por ejemplo, `negrita` escribirá **negrita**.

`<i>...</i>`: el texto dentro del elemento se escribirá en *itálica*.

`<h1>...</h1>`: el texto se escribe en una fuente mayor, normalmente para un título. También existen `<h2>...</h2>`, `<h3>...</h3>` y así hasta `<h6> ...</h6>` para títulos con fuente menor.

`<p>...</p>`: un nuevo párrafo.

`
`: salto de línea. De los pocos elementos sin etiqueta de apertura y de cierre.

`<pre>...</pre>`: preserva el formato, muy útil por ejemplo para escribir código sin temor a que algún fragmento sea interpretado como un elemento HTML.

` ` : un espacio en blanco “duro”, indica al navegador que en ese punto no puede romper la línea.

`<div>...</div>`: permite crear secciones dentro del texto con un estilo común. Suelen llevar un atributo *class* que identifica el estilo dentro de una hoja de estilos.

`...`: similar a `<div>...</div>` pero suele especificar el estilo dentro del propio elemento. Por ejemplo, para poner un texto en rojo se puede escribir: `¡alerta!`

LISTAS

...: Lista sin orden. Los elementos se especifican con **...**. Por ejemplo:

```
<ul>
<li>Uno</li>
<li>Dos</li>
<li>Tres</li>
</ul>
```

Mostrará:

- Uno
- Dos
- Tres

...: lista ordenada. Por ejemplo:

```
<ol>
<li>Uno</li>
<li>Dos</li>
<li>Tres</li>
</ol>
```

Mostrará:

1. Uno
2. Dos
3. Tres

ENLACES

...: Especifica un fragmento de texto o una imagen sobre la que se puede hacer clic. Al hacerlo, el navegador “saltará” a la dirección especificada por href, que puede ser bien una dirección web o un marcador en la propia página. Por ejemplo ** Pulsa aquí **.

IMÁGENES

****: Incluye en la página la imagen indicada en src. Se suele emplear junto con los atributos width y/o height para reescalar la imagen. Si solo se incluye uno de los dos, el otro reescalará proporcionalmente. Por ejemplo

```
<img src = "/html/images/test.png" alt="Test" width = 300 />
```

mostrará la imagen `test.png` que debe estar en la carpeta `images`, con una anchura de 300 puntos. El atributo `alt="Test"` contiene un texto que se mostrará en caso de que la imagen no se encuentre en el lugar especificado.

TABLAS

`<table>...</table>`: Las tablas son estructuras que encontraremos a menudo al hacer *web scraping*, y conviene conocer su estructura de 3 niveles:

1. El primer nivel es el de la propia tabla y viene delimitado por los *tags* `<table>...</table>`. A menudo se incluyen atributos como `border`, que especifica la forma del borde de la tabla, `cellpadding`, que indica el número de píxeles que debe haber como mínimo desde el texto de una celda o casilla hasta el borde, y `cellspacing`, que indica el espacio entre una celda y la siguiente.
2. El segundo nivel corresponde a las filas, cada fila se delimita por `<tr> ...</tr>`.
3. Finalmente, el tercer elemento es el nivel de celda o casilla, y viene delimitado por `<td>...</td>`, o en el caso de ser las celdas de cabecera por `<th>...</th>`.

Un ejemplo nos ayudará a entender mejor esta estructura:

```
<table border = "1">
  <tr>
    <td>fila 1, columna 1</td>
    <td>fila 1, columna 2</td>
  </tr>

  <tr>
    <td>fila 2, columna 1</td>
    <td>fila 2, columna 2</td>
  </tr>
</table>
```

Mostrará:

fila 1, columna 1	fila 1, columna 2
fila 2, columna 1	fila 2, columna 2

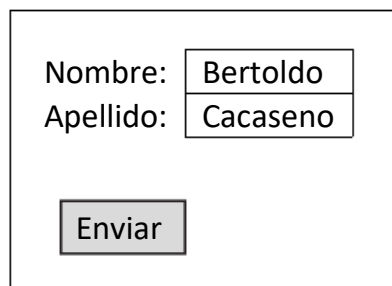
FORMULARIOS

`<form>...</form>`: Los formularios se utilizan para que el usuario pueda introducir información que será posteriormente procesada. En *web scraping* son muy importantes, porque a menudo encontraremos páginas que solo nos muestran la información como respuesta a los valores que debemos introducir nosotros (por ejemplo, tenemos que introducir previamente un usuario y una contraseña).

Los formularios ofrecen muchas posibilidades, permitiendo definir listas desplegables, varios tipos de casillas para marcar, etc. Aquí solo vamos a ver un pequeño ejemplo para conocer su aspecto general. Se trata de una sencilla página en la que va a pedir dos valores de texto, un nombre y un apellido, y tras pulsar en un botón de envío recogerá la información y la hará llegar a un módulo PHP (que no incluimos ya que escapa del ámbito de este libro).

```
<form action="/trata.php">
  Nombre:
  <input type="text" name="firstname" value="Bertoldo">
  <br>
  Apellido:
  <input type="text" name="lastname" value="Cacaseno">
  <br>
  <input type="submit" value="Enviar">
</form>
```

El aspecto de este formulario será el siguiente:



La parte fundamental y en la que debemos fijarnos son los elementos `<input>` (que, por cierto, no necesitan etiqueta de cierre `</input>`). El atributo `type` nos indica el tipo de entrada. El valor `text` de las dos primeras entradas del ejemplo, indica que se trata de un campo de texto, mientras que el tipo entrada `submit` nos indicará que se trata de un botón para enviar los datos.

ATRIBUTOS MÁS USUALES

Hay 4 atributos que podremos encontrar en la mayoría de elementos HTML:

- **id**: Identifica un elemento dentro de la página. Por tanto, los valores asociados no deben repetirse dentro de la misma página.
- **title**: indica el texto que se mostrará al dejar el cursor sobre el elemento. Por ejemplo `<h1 title="cabecera principal"> Python y Big Data </h1>` mostrará "cabecera principal" al situar el cursor sobre el texto "Python y Big Data".
- **class**: asocia un elemento con una hoja de estilo (CSS), que dará formato al elemento: tipo de fuente, color, etc.
- **style**: permite indicar el formato directamente, sin referirse a la hoja de estilo. Por ejemplo:
`<p style = "font-family:arial; color:#FF0000;"> ...texto...</p>`

Con esta información básica sobre HTML, ya estamos listos para empezar a recoger información de las páginas web.

Navegación absoluta

Como acabamos de ver, los documentos HTML, al igual que sucede en XML, siguen una estructura jerárquica. Los elementos que están directamente dentro de otros se dice que son *hijos* del elemento contenedor y *hermanos* entre sí. En nuestro miniejemplo, los elementos `<div id="date">` y `<div id="content">` son hermanos entre sí, y ambos son hijos del elemento `<body>`. Además, `<div id="date">` tiene a su vez un hijo, que en esta ocasión no es otro elemento sino un texto.

La biblioteca BeautifulSoup nos permite utilizar estos conceptos para navegar por el documento buscando la información que precisamos. Como ejemplo inicial, podemos preguntar por los hijos del documento raíz, y su tipo para hacernos una idea de lo que vamos a encontrarnos:

```
>>> hijosDoc = list(soup.children)
>>> print([type(item) for item in hijosDoc])
>>> print(hijosDoc)
```

El primer print nos indica que hay 3 hijos:

- El primero, de tipo `bs4.element.Doctype`, corresponde a la primera línea.

- El segundo, de tipo `bs4.element.NavigableString` que, como vemos en el siguiente print, es tan solo el fin de línea que se encuentra entre línea inicial y la etiqueta `<html>`, y que se representa por `\n`.
- El tercer elemento, del tipo `bs4.element.Tag`, corresponde al documento HTML en sí mismo.

Ahora podemos seleccionar el tercer elemento (índice 2) y tendremos acceso al documento HTML en sí.

```
>>> html = hijosDoc[2]
>>> print(list(html.children))
```

Repetiendo el proceso podemos ver que `html` tiene 5 hijos, aunque tres corresponden a saltos de línea, y solo 2 nos interesan: `head` y `body`. De la misma forma podemos obtener los elementos contenidos en `head` y `body`, y repetir el proceso hasta analizar toda la estructura. La figura 2-2 muestra el árbol resultante, donde por simplicidad se han eliminado los valores `\n` a partir del segundo nivel.

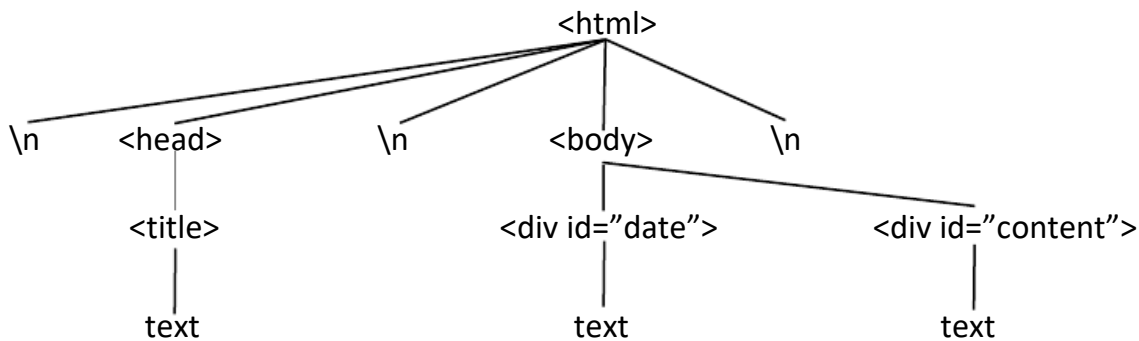


Figura 2-2. Árbol sintáctico de un documento.

En particular, supongamos que nos interesa el texto asociado al `div` con atributo `id = date`. Primero seleccionamos el elemento `body`, que es el cuarto hijo (índice Python 3) de `html`, y una vez más examinamos sus hijos:

```
>>> body = list(html.children)[3]
>>> print(list(body.children))
```

Comprobamos que `body` tiene 5 hijos: `\n`, el primer `div`, `\n`, el segundo `div`, y de nuevo `\n`. Como estamos interesados en el texto del primer `div`, accedemos a este elemento:

```
>>> divDate = list(body.children)[1]
```

Este elemento ya no tiene otros elementos tipo tag como hijos. Solo un elemento de tipo `bs4.element.NavigableString`, que corresponde al texto que buscamos. Podemos obtenerlo directamente con el método `get_text()`:

```
>>> print(divDate.get_text())
```

que muestra el valor deseado: Fecha 25/03/2035.

Navegación relativa

Para nuestro primer *web scraping* hemos recorrido el documento desde la raíz, es decir, hemos tenido que recorrer la *ruta absoluta* del elemento que buscamos. Sin embargo, en una página típica esto puede suponer un recorrido muy laborioso, implicando decenas de accesos al atributo *children*, una por nivel. Además, cualquier cambio en la estructura de la página que modifique un elemento de la ruta hará el código inservible.

Para evitar esto, *BeautifulSoup* ofrece la posibilidad de buscar elementos sin tener que explicar la ruta completa. Por ejemplo, el método `find_all()` busca todas las apariciones de un elemento a cualquier nivel y las devuelve en forma de lista:

```
>>> divs = soup.find_all("div")
```

Ahora, si sabemos que queremos mostrar el texto asociado al primer div podemos escribir:

```
>>> print(divs[0].get_text())
```

El resultado es el mismo valor ("Fecha 25/03/2035"), pero esta vez obtenido de una forma mucho más directa y sencilla. Podemos incluso reducir este código mediante el método `find`, que devuelve directamente el primer objeto del documento que representa el elemento buscado:

```
>>> print(soup.find("div").get_text())
```

Estos métodos siguen teniendo el problema de que continúan dependiendo, aunque en menor medida, de la ruta concreta que lleva hasta el elemento, ya que se basan en la posición del *tag* entre todos los del mismo tipo. Por ejemplo, si al creador de nuestra página se le ocurriera añadir un nuevo elemento div antes del que deseamos, sería este (el nuevo) el que mostraría nuestro código, ya que la posición del que buscamos habría pasado a ser la segunda (índice 1).

Para reducir esta dependencia de la posición, podemos afinar más la búsqueda apoyándonos en los valores de atributos usuales como `id` o `class`. En nuestro ejemplo, supongamos que sabemos que el elemento `div` que buscamos debe tener un `id` igual a `date`, y que este valor, como un `id`, es único en el código HTML. Entonces, podemos hacer la siguiente búsqueda, que esta vez es independiente del número de elementos `div` que pueda haber antes:

```
>>> print(soup.find("div", id="date").get_text())
```

BeautifulSoup incluye muchas más posibilidades, que se pueden encontrar en la documentación de la biblioteca. Por ejemplo, empleando el método `select()`, es posible buscar elementos que son *descendientes* de otro, donde el concepto de descendiente se refiere a los hijos, a los hijos de los hijos, y así sucesivamente.

Por ejemplo, otra forma de obtener el primer valor `div`, asegurando que está dentro, esto es, es descendiente del elemento `html` (lo que sucede siempre, pero sirva como ejemplo):

```
>>> print(soup.select("html div")[0].get_text())
```

Ahora vamos a ver una aplicación de *web scraping* estático sobre una página real.

Ejemplo: día y hora oficiales

Supongamos que estamos desarrollando una aplicación en la que en cierto punto es fundamental conocer la fecha y hora oficiales. Desde luego, podríamos obtener esta información directamente del ordenador, pero no queremos correr el riesgo de obtener un dato erróneo, ya sea por error del reloj interno o porque la hora y fecha hayan sido cambiadas manualmente.

Para solventar esto, la agencia estatal del Boletín Oficial del Estado pone a nuestra disposición la página:

https://www.boe.es/sede_electronica/informacion/hora_oficial.php

que nos da la hora oficial en la península. Para ver cómo podemos extraer la información, Introducimos esta dirección en nuestro navegador, supongamos que se trata de Google Chrome. Ahí aparecen la hora y la fecha oficiales. Para extraerla, debemos conocer la estructura de esta página en particular. Para ello, situamos el ratón sobre la hora, pulsamos el botón derecho, y del menú que aparece elegimos la opción *Inspeccionar elemento*.



Figura 2-3. Estructura de la página del BOE con la fecha oficial.

Al hacer esto el navegador se dividirá en dos partes, tal y como muestra la figura 2-3. A la izquierda nos sigue mostrando la página web, mientras que a la derecha vemos el código HTML que define la página web, el código oculto que recibe nuestro navegador y que contiene la información que deseamos.

Vemos que la fecha y la hora corresponden al texto de un elemento `span`, con atributo `class` puesto al valor `grande`, que a su vez es hijo de un elemento `p` con atributo `class = cajaHora`. Parece que lo más sencillo es localizar el elemento `span`, por ser el más cercano al dato buscado, pero debemos recordar que este elemento lo que hace es formatear cómo se muestra un texto en la caja. Puede que se utilice el mismo formato para otras partes de la página, si no en la versión actual de la página, sí en una futura. Por ello, parece mucho más seguro emplear el elemento `p` con atributo `class = cajaHora`, cuyo nombre parece definir perfectamente el contenido.

En primer lugar, como en ejemplos anteriores, cargamos la página, mediante la biblioteca `requests`:

```
>>> import requests
>>> url = "https://www.boe.es/sede_electronica/informacion/hora_oficial.php"
>>> r = requests.get(url)
>>> print(r)
```

y a continuación buscamos el elemento `p` con `class = cajaHora` y localizamos su segundo hijo (índice 1), tras comprobar que el primer hijo es un espacio en blanco (comprobación que omitimos por brevedad):

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(r.content, "html.parser")
>>> cajaHora = soup.find("p", class_="cajaHora")
>>> print(list(cajaHora.children)[1].get_text())
```

el resultado obtenido es el día y la hora oficiales según el BOE.

Fácil, ¿verdad? Por desgracia en ocasiones la cosa se complica un poco, como veremos en el siguiente apartado.

DATOS QUE REQUIEREN INTERACCIÓN

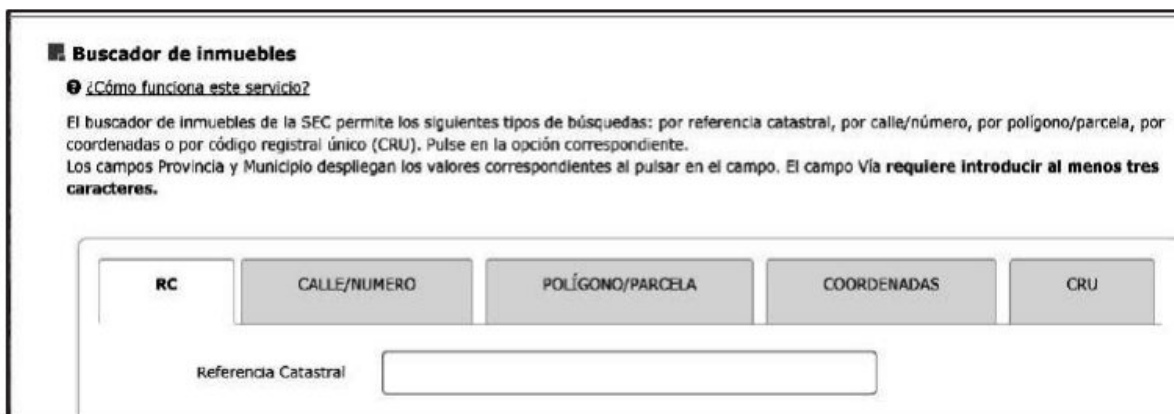
BeautifulSoup es una biblioteca excelente, y nos ayudará a recuperar, de forma cómoda, datos de aquellas páginas que nos ofrecen directamente la información buscada. La idea, como hemos visto, es sencilla: en primer lugar, nos descargamos el código HTML de la página, y a continuación navegamos por su estructura, hasta localizar la información requerida.

Sin embargo, a menudo nos encontraremos con páginas que nos solicitan información que debemos completar antes de mostrarnos el resultado deseado, e decir, que exigen cierta interacción por nuestra parte. Un caso típico son las webs que nos exigen introducir usuario y palabra clave para poder entrar y acceder así a la información que deseemos. Esto no lo podemos lograr con BeautifulSoup.

Por ejemplo, puede que queramos saber los datos catastrales (tamaño de la finca, etc.) a partir de una dirección o de unas coordenadas. En este caso, debemos consultar la página web de la sede electrónica del catastro:

<https://www1.sedecatastro.gob.es/CYCBienInmueble/OVCBusqueda.aspx>

La página nos dará la información, pero previamente nos pedirá que introduzcamos los datos tal y como se aprecia en la figura 2-4. Para lograr esta vamos a utilizar la biblioteca *Selenium*, que no fue pensada inicialmente para hacer *web scraping*, sino para hacer *web testing*, esto es, para probar automáticamente aplicaciones web.



Buscador de inmuebles

¿Cómo funciona este servicio?

El buscador de inmuebles de la SEC permite los siguientes tipos de búsquedas: por referencia catastral, por calle/número, por polígono/parcela, por coordenadas o por código registral único (CRU). Pulse en la opción correspondiente.

Los campos Provincia y Municipio despliegan los valores correspondientes al pulsar en el campo. El campo Vía requiere introducir al menos tres caracteres.

RC CALLE/NÚMERO POLÍGONO/PARCELA COORDENADAS CRU

Referencia Catastral

Figura 2-4. Buscador de inmuebles, sede electrónica del catastro.

La biblioteca iniciará una instancia de un navegador, que puede ser por ejemplo Chrome, Firefox o uno que no tenga interfaz visible, y permitirá:

- Cargar páginas.
- Hacer clic en botones.
- Introducir información en formularios.
- Recuperar información de la página cargada.

En general, Selenium nos permite realizar cualquiera de las acciones que realizamos manualmente sobre un navegador.

Selenium: instalación y carga de páginas

Para utilizar Selenium debemos, en primer lugar, instalar el propio paquete. La instalación se hace de la misma forma que cualquier paquete Python; el método estándar es tecleando en el terminal:

```
python -m pip install selenium
```

La segunda parte de la instalación es específica de Selenium. Para ello, debemos instalar el cliente del navegador que se desee utilizar. Los más comunes son:

- Firefox, cuyo cliente para Selenium es llamado *geckodriver*. Podemos descargarlo en:
<https://github.com/mozilla/geckodriver/releases>
- Google Chrome: el cliente para Selenium, llamado *chromedriver*, se puede obtener en:
<https://github.com/SeleniumHQ/selenium/wiki/ChromeDriver>

Hay más *drivers*, que podemos encontrar en la página de Selenium: Opera, Safari, Android, Edge y muchos otros.

En cualquier caso, tras descargar el fichero correspondiente debemos descomprimirlo, obteniendo el ejecutable. Aún nos queda una cosa por hacer: debemos lograr que el fichero ejecutable sea visible desde nuestra aplicación Python. Hay dos formas posibles de lograr esto.

ACCESO AL *DRIVER* A TRAVÉS DE LA VARIABLE DE ENTORNO PATH

La variable de entorno PATH es utilizada por los sistemas operativos para saber en qué lugar debe buscar ficheros ejecutables. Por tanto, una forma de hacer visible el *driver* del navegador dentro de nuestra aplicación es modificar esta variable,

haciendo que incluya la ruta hasta el ejecutable. La forma de hacer esto dependerá del sistema operativo:

- **Linux:** si nuestra consola es compatible con bash, podemos escribir:
`export PATH = $PATH:/ruta/a/la/carpeta/del/ejecutable`
- **Windows:** buscar en el inicio “Sistema”, y allí “Configuración avanzada del sistema” y “Variables de Entorno”. Entre las variables, seleccionar la variable *Path*, pulsar el botón *Editar*, y en la ventana que se abre (“Editar Variables de entorno”) pulsar *Nuevo*. Allí añadiremos la ruta donde está el fichero *.exe* que hemos descargado, pero sin incluir el nombre del fichero y sin la barra invertida (\) al final. Por ejemplo:

```
C:\Users\Bertoldo\Downloads\selenium
```

- **Mac:** si tenemos instalado *Homebrew*, bastará con teclear
`brew install geckodriver`
que además lo añade al PATH automáticamente (análogo para Chrome).

En todo caso, tras modificar la variable PATH, conviene abrir un nuevo terminal para que el cambio se haga efectivo. Ahora podemos probar a cargar una página, sustituyendo el contenido de la variable *url* por la dirección web que deseemos:

```
>>> from selenium import webdriver
>>> driver = webdriver.Chrome()
>>> url = " ..."
>>> driver.get(url)
```

Si la carpeta que contiene el *driver* ha sido añadida correctamente al PATH, el efecto de este código será:

1. Abrir una instancia del navegador Chrome.
2. Cargar la página contenida en la variable *url*. En nuestro ejemplo sugerimos cargar la página:
`https://www1.sedecatastro.gob.es/CYCBienInmueble/OVCBusqueda.asp`

ACCESO AL DRIVER INCORPORANDO LA RUTA EN EL CÓDIGO PYTHON

En ocasiones no podremos modificar el PATH, quizás por no tener permisos para hacer este tipo de cambios en el sistema. Una solución, en este caso, es incorporar la ruta al ejecutable del *driver* dentro del propio código. Para ello reemplazamos la línea `driver = webdriver.Chrome()` del ejemplo anterior por:

```
>>> import os
>>> chromedriver = "/ruta/al/exe/chromedriver.exe"
>>> os.environ["webdriver.chrome.driver"] = chromedriver
>>> driver = webdriver.Chrome(executable_path=chromedriver)
```

En este código debemos sustituir el valor de la variable `chromedriver` por la ruta al driver ejecutable. El resultado debe ser el mismo que en el caso anterior: se abrirá una nueva instancia de Chrome en la que posteriormente podremos cargar la página.

Clic en un enlace

Ya tenemos nuestra página cargada en el *driver*; podemos comprobarlo en el navegador que se ha abierto. Ahora, supongamos que deseamos conocer los datos catastrales a partir de las coordenadas (longitud y latitud). La página del catastro incluye esta opción, pero no por defecto. Por ello, antes de introducir las coordenadas debemos pulsar sobre la pestaña “COORDENADAS”. Para ver el código HTML asociado, nos situamos sobre la pestaña, hacemos clic en el botón derecho del ratón y seleccionamos “inspeccionar”.

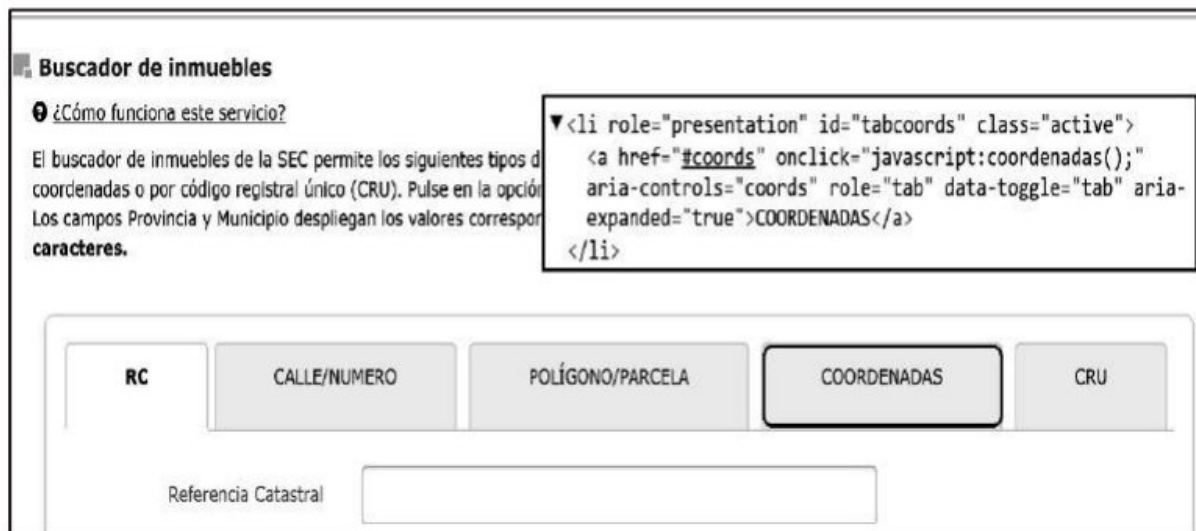


Figura 2-5. Código asociado a la pestaña COORDENADAS.

La figura 2-5 muestra el código HTML sobre la imagen de la página web (en el navegador aparecerá a la derecha, la mostramos así por simplicidad). Vemos que la palabra “COORDENADAS” aparece dentro de un elemento `<a ...> `. En HTML estos elementos son conocidos como links (enlaces). Al pulsar sobre ellos nos redirigen a otra página web, o bien a otra posición dentro de la misma página. Selenium dispone de un método que nos permite acceder a un elemento de tipo enlace a partir del texto que contiene.

```
>>> coord = driver.find_element_by_link_text("COORDENADAS")
>>> coord.click()
```

En la primera instrucción, seleccionamos el elemento de tipo link, que Selenium busca automáticamente en la página a partir del texto “COORDENADAS”. En la siguiente hacemos clic sobre este elemento, con lo que la página cambiará, mostrando algo similar a la figura 2-6.

Figura 2-6. Página del catastro que permite buscar por coordenadas.

Cómo escribir texto

Ya estamos listos para introducir las coordenadas. Para ello repetimos el mismo proceso: primero nos situamos en la casilla de una de las coordenadas (por ejemplo, latitud), pulsamos el botón derecho y elegimos “inspeccionar”. La figura 2-7 muestra el código asociado a este dato.

```
<div class="col-md-3 col-sm-3">
  <input name="ctl00$Contenido$txtLatitud" type="text" id="ctl00_Contenido_txtLatitud"
  class="form-control" onkeyup="javascript:activaBotonesCoordenadas();" oninput=
  "javascript:activaBotonesCoordenadas();" onkeypress="javascript:return
  soloDecimales(this,event);" placeholder="Ejemplo: 40.71277837">
</div>
```

Figura 2-7. Código asociado la inspección de la caja de texto “latitud”.

Vemos un elemento div que contiene un elemento input. Podemos usar el identificador (id=“ctl00_Contenido_txtLatitud”) de este elemento para seleccionarlo. De paso hacemos lo mismo con el campo de texto de la longitud:

```
>>> lat=driver.find_element_by_id("ctl00_Contenido_txtLatitud")
>>> lon=driver.find_element_by_id("ctl00_Contenido_txtLongitud")
```

De esta forma las variables `lat` y `lon` contendrán los elementos correspondientes a las cajas de texto de la latitud y de la longitud, respectivamente.

Ahora podemos introducir la latitud y la longitud utilizando el método `send_keys()`:

```
>>> latitud = "28.2723368"
>>> longitud = "-16.6600606"
>>> lat.send_keys(latitud)
>>> lon.send_keys(longitud)
```

Una nota final: en ocasiones la llamada a `send_keys()` puede devolver errores como:

```
WebDriverException: Message: unknown error: call function result
missing 'value'
(Session info: chrome=65.0.3325.181)
(Driver info: chromedriver=2.33.506120
(e3e53437346286c0bc2d2dc9aa4915ba81d9023f),platform=Windows NT
10.0.16299 x86_64)
```

Si este es el caso, tenemos un problema de compatibilidad entre nuestro navegador Chrome y el *driver*, que podremos solucionar actualizando el driver a la versión más reciente.

Pulsando botones

Aunque ya hemos introducido las coordenadas, que corresponden por cierto a la cima del Teide (Tenerife), todavía debemos pulsar el botón correspondiente para que la web del catastro haga la búsqueda y nos muestre la información deseada. Para ello repetimos, una vez más, el mismo proceso que hicimos con la pestaña “COORDENADAS”: nos situamos sobre el botón “DATOS”, pulsamos el botón derecho del ratón, y elegimos la opción “inspeccionar”. Allí veremos los datos del botón, y en particular el atributo `id`, que cuando existe es una buena forma de identificar un elemento. Por tanto, podemos seleccionar el elemento:

```
>>> datos=driver.find_element_by_id("ctl00_Contenido_btnDatos")
```

y a continuación hacer clic:


```
>>> datos.click()
```

Tras unos instantes, la web nos mostrará los datos deseados, parte de los cuales muestra la figura 2-8.

DATOS DESCRIPTIVOS DEL INMUEBLE	
Referencia catastral	38026A035000010000EI III
Localización	Polígono 35 Parcela 1 MONTE. LA OROTAVA (S.C. TENERIFE)
Clase	Rústico
Uso principal	Agrario
Superficie construida ⓘ	40 m ²
Año construcción	1999

Figura 2-8. Parte de la información mostrada para las coordenadas del ejemplo.

Es muy importante observar que tras la llamada a `datos.click()`, el contenido de la variable `driver` ha cambiado, y que ahora contendrá la nueva página web, la de los resultados. Estos “efectos laterales” pueden confundirnos a veces por lo que conviene remarcarlo: en Selenium, siempre que pulsemos sobre un enlace o un botón estaremos cambiando la página accesible desde el driver.

Si en algún momento queremos volver a la página anterior, lo más fácil es utilizar un poco de JavaScript ejecutado desde el propio driver:

```
>>> driver.execute_script("window.history.go(-1)")
```

El código indica que la página se debe sustituir por la anterior en el historial del navegador. Por cierto, ya que hemos probado este código, antes de proseguir con el capítulo conviene que hagamos:

```
>>> driver.execute_script("window.history.go(+1)")
```

para volver a la página de los resultados del catastro (figura 2-8), ya que vamos a suponer que estamos en esta página para el apartado siguiente.

Recordemos que nuestro objetivo es recopilar alguno de estos datos (por ejemplo, conocer la “clase” de terreno, o la referencia catastral). Podríamos utilizar BeautifulSoup, porque ya estamos en una página con los datos que deseamos, pero también podemos utilizar la propia biblioteca Selenium, que contiene un poderoso entorno para navegar y extraer información de páginas web.

Localizar elementos

Hasta ahora hemos empleado alguno de los métodos que tiene Selenium para localizar elementos, pero hay muchos más:

- `find_element_by_id()`: encuentra el primer elemento cuyo atributo `id` se le indica.
- `find_element_by_name()`: análogo, pero buscando un elemento con atributo `name`.
- `find_element_by_class_name()`: análogo, pero buscando el valor del atributo `class`.
- `find_element_by_xpath()`: búsqueda utilizando sintaxis XPath, que veremos en seguida.
- `find_element_by_link_text()`: primer elemento `<a...>text` según el valor `text`.
- `find_element_by_partial_link_text()`: análogo, pero usa un prefijo del texto.
- `find_element_by_tag_name()`: primer elemento con el nombre indicado.
- `find_element_by_css_selector()`: búsqueda utilizando la sintaxis CSS.

Todos estos métodos generarán una excepción `NoSuchElementException` si no se encuentra ningún elemento que cumpla la condición requerida.

Cada uno de estos métodos tiene un equivalente que encuentra no en el primer elemento, sino en *todos* los elementos que verifiquen la condición dada. El nombre de estos métodos se obtiene a partir de los métodos ya mencionados cambiando `element` por el plural `elements`. Por ejemplo, podemos usar

```
driver.find_element_by_tag_name('div')
```

para obtener el primer elemento de tipo `div`, del elemento, y

```
driver.find_elements_by_tag_name('div')
```

para obtener todos los elementos de tipo `div` en una lista. La única excepción es `find_element_by_id()`, que no tiene método “plural” porque se supone que cada identificador debe aparecer una sola vez en una página web.

XPath

XPath es un lenguaje de consultas que permite obtener información a partir de documentos XML. Selenium lo utiliza para “navegar” por las páginas HTML debido a que resulta muy sencillo de aprender y a la vez ofrece gran flexibilidad. Aquí vamos a ver los elementos básicos. Una descripción en detalle se puede consultar en

https://www.w3schools.com/xml/xpath_intro.asp

o en el documento detallado describiendo el lenguaje, más complejo, pero interesante en caso de dudas concretas:

<https://www.w3.org/TR/xpath/>

Conviene recordar que ya hemos visto métodos que nos permiten seleccionar elementos a partir de sus atributos `id`, `class` o `name`. Sin embargo, en ocasiones no dispondremos de estos elementos, y las búsquedas serán un poco más complicadas.

XPath va a permitirnos especificar una ruta a un elemento mediante consultas, que se escribirán en Selenium entre comillas. Vamos a ver algunos de los componentes que se pueden usar en esta cadena. En los ejemplos que siguen, suponemos que estamos en la página del catastro que muestra los datos para las coordenadas 28.2723368, -16.6600606, y que se muestra en la figura 2-8.

COMPONENTE “/”

Si se usa al principio de la cadena XPath hace referencia a que vamos a referirnos al comienzo del documento (camino absoluto). Por ejemplo:

```
>>> html = driver.find_element_by_xpath("/html")
>>> print(html.text)
```

Aquí “/” indica que seleccionamos el elemento `html` que cuelga de la raíz del documento. El `print()` siguiente nos muestra una versión textual del contenido del elemento. Buena parte de la flexibilidad de XPath es que permite encadenar varios pasos. Recordemos que toda página HTML comienza con un elemento `html` que tiene dos componentes (dos hijos): `head` y `body`. Podemos extraer estos componentes de la siguiente forma:

```
>>> head = driver.find_element_by_xpath("/html/head")
>>> body = driver.find_element_by_xpath("/html/body")
```

Por ejemplo, la primera instrucción se puede leer como “ve a la raíz del documento, y busca un hijo con nombre html”. Para este hijo, busca a su vez un elemento hijo con nombre head.

Recordemos que el elemento debe existir, si por ejemplo intentamos:

```
>>> otro = driver.find_element_by_xpath("/html/otro")
```

Obtendremos una excepción `NoSuchElementException`. Aquí no lo hacemos por simplicidad, pero en una aplicación real debemos utilizar estructuras `try-catch` para prevenir que el programa se “rompa” si la estructura del código HTML ha cambiado, o simplemente si introducimos un camino que no existe por error.

Un aspecto importante que debemos tener en cuenta es que cualquier búsqueda en Selenium devuelve un elemento de tipo `WebElement`

, que es un *puntero* al elemento(s) seleccionado(s). No debemos pues pensar que la variable obtenida “contiene” el elemento, sino más bien que “señala” al elemento. Esto explica que el siguiente código, aunque ciertamente extraño y poco recomendable, sea legal:

```
>>> html2 = body.find_element_by_xpath("/html")
```

Utilizamos `body` y no `driver` como punto de partida, cosa que haremos pronto para construir caminos paso a paso. Pero lo interesante es que forzamos acceder de nuevo a la raíz y tomar el elemento `html`. Esto no podría hacerse si `body` fuera realmente el cuerpo del código HTML; no podríamos acceder desde dentro de él a un elemento exterior. Pero funciona porque Selenium simplemente va a la raíz del documento al que apunta `body`, que es el documento contenido en `driver`, y devuelve el elemento `html`.

En particular esta idea de los “señaladores” nos lleva a entender que si ahora cambiamos la página contenida en el `driver`, la variable `body` dejará de tener un contenido válido, porque “apuntará” a un código que ya no existe. Por ejemplo, el siguiente código (que no recomendamos ejecutar ahora para no cambiar la página de referencia) provocará una excepción:

```
>>> driver.execute_script("window.history.go(-1)")
>>> print(body.text)
```

COMPONENTE “*”

Cuando no nos importa el nombre del elemento concreto, podemos utilizar “*”, que representa a cualquier hijo a partir del punto en el que se encuentre la ruta. Por ejemplo, para ver los nombres de todos los elementos que son hijos de body podemos escribir el siguiente código:

```
hijos = driver.find_elements_by_xpath("/html/body/*")
for element in hijos:
    print(element.tag_name)
```

Otra novedad de este ejemplo es la utilización de `find_elements_by_xpath()` en lugar de `find_element_by_xpath()` para indicar que en lugar de un elemento queremos que se devuelvan *todos* los elementos que cumplen la condición. El resultado es una lista de objetos de tipo `WebElement`, almacenada en la variable `hijos`. Para recorrer la lista utilizamos un bucle que va mostrando el nombre de cada elemento hijo (atributo `tag_name` de `WebElement`). En nuestro caso mostrará:

```
form
script
a
iframe
```

El primer elemento es un formulario, el segundo un script con código JavaScript, el tercero un enlace (representado en HTML por `a`), y el último un `iframe`, que es una estructura empleada en HTML para incrustar un documento dentro de otro.

El elemento “*” puede ser seguido por otros. Por ejemplo, supongamos que queremos saber cuántos elementos de tipo `div` son “nietos” de `body`.

```
>>> divs = driver.find_elements_by_xpath("/html/body/*/div")
>>> print(len(divs))
```

que nos mostrará el valor 4.

COMPONENTE “.”

El punto sirve para indicar que el camino sigue desde la posición actual. Resulta muy útil cuando se quiere seguir la navegación desde un “señalador”. Por ejemplo, otra forma de obtener los “nietos” de tipo `div` del elemento `body`, es partir del propio `body`:

```
>>> divs = body.find_elements_by_xpath("./*/div")
>>> print(len(divs))
```

COMPONENTE “//”

Las dos barras consecutivas se usan para saltar varios niveles. Por ejemplo, supongamos que queremos saber cuántos valores `div` son descendientes de `body`, es decir, cuántos `div` están contenidos dentro de `body` a cualquier nivel de profundidad. Podemos escribir:

```
>>> divs = driver.find_elements_by_xpath("/html/body//div")
>>> print(len(divs))
```

que devolverá el valor 108 en nuestro ejemplo. También podemos preguntar por todos los elementos `label` del documento:

```
>>> labels = driver.find_elements_by_xpath("//label")
>>> print(len(labels))
```

que muestra el valor 6.

Por ejemplo, supongamos que queremos encontrar la referencia catastral incluida en la página de nuestro ejemplo. Como siempre, nos situamos sobre ella y pulsamos el botón derecho del ratón, eligiendo *inspeccionar*. El entorno código HTML que precede a la referencia buscada es de la forma:

```
...
<div id="ctl00_Contenido_tblInmueble"
    class="form-horizontal" name="tblInmueble">
  <div class="form-group">
    <span class="col-md-4 control-label ">
      Referencia catastral
    </span>
    <div class="col-md-8 ">
      <span class="control-label black">
        <label class="control-label black text-left">
          38026A035000010000EI&nbsp;
        </label>
      </span>
    </div>
  </div>
...

```

La referencia (el valor 38026A035000010000EI), se encuentra dentro de un elemento de tipo `label`, pero como hemos visto el documento contiene 6 valores de este tipo. Lo usual en estos casos es buscar un elemento con identificador cercano, en este caso el `div` de la primera línea del código que mostramos (`id="ctl00_Contenido_tblInmueble"`), y utilizarlo de punto de partida para localizar el valor buscado:

```
>>> id = "ctl00_Contenido_tblInmueble"
>>> div = driver.find_element_by_id(id)
>>> label = div.find_element_by_xpath("//label")
>>> print(label.text)
```

Primero hacemos que la variable `div` apunte al elemento `div` con el identificador indicado. A continuación, buscamos la primera etiqueta (elemento `label`) descendiente de este elemento `div`. Finalmente mostramos el texto, que es el valor deseado:

```
38026A035000010000EI
```

En el código hemos utilizado las versiones “singulares”, es decir, `find_element()` en lugar de `find_elements()`. En el primer caso, `find_element_by_id()`, porque estamos buscando un `id`, que se supone siempre único. Y en el segundo caso, `find_element_by_xpath()`, porque estamos buscando la primera etiqueta descendiente del elemento `div`.

La ventaja del uso de `“//”` es que logramos cierta independencia del resto de la estructura, es decir la consulta seguiría funcionando siempre que el elemento con `id = "ctl00_Contenido_tblInmueble"` siga existiendo, y su primera etiqueta hija contenga el valor buscado. El único inconveniente es que aún utilizamos la posición de la etiqueta, lo que indica que, si se añadiera en el futuro otra etiqueta más arriba, sería la recién llegada la seleccionada, devolviendo información errónea.

FILTROS [...]

Los filtros son una herramienta poderosa de XPath. Permiten indicar condiciones adicionales que deben cumplir los elementos seleccionados. La forma más simple de filtrado es simplemente indicar la posición de un elemento particular. Por ejemplo, supongamos que deseamos saber qué tipo de finca es la que corresponde a esta referencia catastral. Si miramos de nuevo la figura 2-8, veremos que este valor viene en tercer lugar y corresponde a la “clase” de terreno, que en este caso es *Rústico*. Podemos entonces obtener directamente la tercera etiqueta:

```
>>> e = driver.find_elements_by_xpath(
>>>     "(//label)[position()=3]")
>>> print(e[0].text)
```

que mostrará por pantalla el texto *Rústico* tal y como deseábamos. El uso de los paréntesis alrededor de `//label` es importante, porque el operador `[]` tiene más prioridad que `“//”`, y de otra forma obtendríamos un resultado erróneo.

También observamos que, aunque hemos seleccionado un solo elemento, seguimos recibiendo una lista de `WebElement`, aunque sea de longitud unitaria. Por eso en el `print()` tenemos que usar la notación `e[0]`.

Esta aplicación de los filtros es tan usual que XPath nos permite eliminar la llamada a `position()`, dando lugar a una notación típica de los *arrays* de programación:

```
>>> e = driver.find_elements_by_xpath("(//label)[3]")
>>> print(e[0].text)
```

Es importante que señalar que en XPath *el primer elemento tiene índice 1, y no 0 como en Python*. Por eso, si queremos lograr el mismo efecto usando Python, podemos devolver todas las etiquetas y seleccionar la de índice 2, en lugar de 3 como era el caso en XPath:

```
>>> etiqs = driver.find_elements_by_xpath("//label")
>>> print(etiqs[2].text)
```

El código lógicamente mostrará el mismo valor, *Rústico*. XPath también nos permite acceder al último elemento, usando la función `last()`. En nuestro ejemplo podemos por ejemplo obtener el área de la finca, que es la última etiqueta:

```
>>> ulti = driver.find_elements_by_xpath(
    "(//label)[last()]"
>>> print(ulti[0].text)
```

que nos mostrará *“68.167.075 m2”*. Análogamente, podemos usar `last()-1` para referirnos al penúltimo elemento.

Otro de los usos más habituales de los filtros es quedarse con los elementos con valores concretos para un atributo. Esto se logra antecediendo el nombre del atributo por *“@”*:

```
>>> xpath =
    "//label[@class='control-label black text-left']"
>>> etiqs = driver.find_elements_by_xpath(xpath)
>>> print(etiqs[0].text)
```

La consulta XPath es un poco más compleja y merece ser examinada en detalle:

- En primer lugar, *“//label”* indica que buscamos elementos `label` a cualquier nivel dentro del documento.

- A continuación, el filtro `[@class='control-label black text-left']` pide que, de todas las etiquetas recolectadas, la consulta seleccione solo las que incluyan un atributo `class` que tome el valor `'control-label black text-left'`.

Como el valor de la referencia catastral es el único del documento con una etiqueta de esta clase, el fragmento de código muestra justo este valor, `38026A035000010000EI`.

Por último, veamos una forma más compleja de obtener este mismo valor, que puede presentar ciertas ventajas. Supongamos que todo lo que podemos asegurar es que:

- Sabemos que la referencia catastral será el texto de un elemento `label`.
- También sabemos que este elemento `label` será *hermano* de otro elemento de tipo `span` que tendrá como texto “Referencia catastral”.

Estas suposiciones se obtienen de la estructura del documento, y no utilizan criterios como la posición ni los atributos particulares del elemento `label`. En XPath lo podemos codificar así:

```
>>> xpath =
>>> "//*[./span/text()='Referencia catastral']//label"
>>> etiq = driver.find_element_by_xpath(xpath)
>>> print(etiq.text)
```

El resultado es, una vez más, el valor catastral. Vamos a explicar en detalle la consulta:

- Buscamos un elemento *P* de cualquier tipo y en cualquier lugar (`//*[`) tal que:
 - Tenga un hijo (`./`), de tipo `span` que incluya como texto “Referencia catastral”.
- Una vez encontrado este elemento *P*, nos quedamos con su hijo de tipo `label`.

Una nota final: cuando hayamos terminado de utilizar Selenium conviene escribir `driver.close()` para liberar recursos, especialmente si el código está dentro de un bucle que se va a repetir varias veces.

Navegadores *headless*

Trabajar desde nuestra aplicación en Selenium con un navegador abierto como en los ejemplos anteriores tiene la ventaja de que podremos ver cómo funciona la aplicación: cómo se escribe el texto, se hace clic, etc. Sin embargo, en una aplicación real normalmente no deseamos que se abra una nueva instancia del navegador; resultaría muy sorprendente para el usuario ver que de repente se le abre Chrome y empieza a cargar páginas, a rellenar datos, etc. Incluso puede suceder que lo cierre y de esa forma detenga la ejecución del programa. Si queremos evitar esto, emplearemos un navegador *headless*, nombre con el que se denominan a los navegadores que no tienen interfaz gráfica. En el caso de Chrome esto podemos lograrlo simplemente añadiendo opciones adecuadas a la hora de crear el driver:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.chrome.options import Options
>>> chrome_options = Options()
>>> chrome_options.add_argument('--headless')
>>> chrome_options.add_argument(
    '--window-size=1920x1080')
>>> driver = webdriver.Chrome(
    chrome_options=chrome_options)
```

Es especialmente llamativo el hecho de que además de indicar como argumento que el navegador debe arrancar en modo *headless*, es decir, sin interfaz gráfica, digamos también el tamaño de la ventana que debe ocupar. Es conveniente hacer esto porque, aunque no veamos el navegador, internamente sí se crea y en algunos casos puede dar error si no dispone de espacio suficiente para crear todos los componentes. En el navegador estándar esto no sucede porque se adapta automáticamente a la pantalla. En un navegador *headless*, en cambio, tenemos que indicar nosotros un tamaño suficientemente grande.

CONCLUSIONES

Aunque Python nos ofrece varias librerías como Selenium y BeautifulSoup que nos facilitan la tarea, obtener información de la web mediante *web scraping* es una tarea que requiere conocer muy bien la estructura de la página que contiene la información y que lleva tiempo de programación. Merecerá la pena sobre todo si es una tarea que vamos a realizar de forma reiterada.

En el siguiente capítulo, vamos a ver otra forma de acceder a los datos, a través de servicios proporcionados para tal fin.

REFERENCIAS

- Documentación de la biblioteca BeautifulSoup (accedido el 30/06/2018). <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- Documentación de la biblioteca Selenium (accedido el 30/06/2018). <https://www.seleniumhq.org/docs/>
- Ryan Mitchell. *Web Scraping with Python: Collecting More Data from the Modern Web*. 2ª edición. O'Reilly Media, 2018.
- Richard Lawson. *Web Scraping with Python (Community Experience Distilled)*. Packt, 2015.
- Mark Collin. *Mastering Selenium WebDriver*. Packt, 2015.
- Vineeth G. Nair. *Getting Started with Beautiful Soup*. Pack. 201

