CAPÍTULO 9

Scraping

Si bien existen multitud de datos estructurados en forma de ficheros, hay otros muchos que están embebidos en páginas web y que están preparados para ser visualizados mediante un navegador.

Sin embargo, las técnicas de «scraping» nos permiten **extraer este tipo de información web** para convertirla en datos estructurados con los que poder trabajar de forma más cómoda.

Los paquetes que veremos en este capítulo bien podrían estar incluidos en otras temáticas, ya que no sólo se utilizan para «scraping».

9.1 requests

El paquete requests es uno de los paquetes más famosos del ecosistema Python. Como dice su lema «HTTP for Humans» permite realizar peticiones HTTP de una forma muy sencilla y realmente potente.[1]

```
$ pip install requests
```

9.1.1 Realizar una petición

Realizar una petición HTTP mediante *requests* es realmente sencillo:

```
>>> import requests

>>> response = requests.get('https://pypi.org')
```

Hemos ejecutado una *solicitud GET* al sitio web https://pypi.org. La respuesta se almacena en un objeto de tipo `requests.models.Response` muy rica en métodos y atributos que veremos a continuación:

```
>>> type(response)
requests.models.Response
```

[1] Foto original de portada por Frame Harirak en Unsplash.

Quizás lo primero que nos interese sea ver el contenido de la respuesta. En este sentido *requests* nos provee del atributo `text` que contendrá el **contenido html** del sitio web en cuestión como cadena de texto:

```
>>> response.text
'\n\n\n\n\n\n<!DOCTYPE html>\n<html lang="en" dir="ltr">\n  <head>\n    <meta␣
↪charset="utf-8">\n    <meta http-equiv="X-UA-Compatible" content="IE=edge">\n
↪<meta name="viewport" content="width=device-width, initial-scale=1">\n\n    <meta␣
↪name="defaultLanguage" content="en">\n    <meta name="availableLanguages" content=
↪"en, es, fr, ja, pt_BR, uk, el, de, zh_Hans, zh_Hant, ru, he, eo">\n\n    \n\n
↪<title>PyPI · The Python Package Index</title>\n    <meta name="description"␣
↪content="The Python Package Index (PyPI) is a repository of software for the␣
↪Python programming language.">\n\n    <link rel="stylesheet" href="/static/css/
↪warehouse-ltr.69ee0d4e.css">\n    <link rel="stylesheet" href="/static/css/
↪fontawesome.6002a161.css">\n    <link rel="stylesheet" href="/static/css/regular.
↪98fbf39a.css">\n    <link rel="stylesheet" href="/static/css/solid.c3b5f0b5.css">\
↪n    <link rel="stylesheet" href="/static/css/brands.2c303be1.css">\n    <link rel=
↪"stylesheet" href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400,
↪400italic,600,600italic,700,700italic%7CSource+Code+Pro:500">\n    <noscript>\n  ␣
↪  <link rel="stylesheet" href="/static/css/noscript.d4ce1e76.css">\n'
```

Nota: Se ha recortado la salida a efectos visuales.

Algo que es realmente importante en una petición HTTP es comprobar el estado de la misma. Por regla general, si todo ha ido bien, deberíamos obtener un **código 200**, pero existen muchos otros códigos de estado de respuesta HTTP:

```
>>> response.status_code
200
```

Truco: Para evitar la comparación directa con el literal 200, existe la variable `requests.codes.ok`.

9.1.2 Tipos de peticiones

Con *requests* podemos realizar peticiones mediante cualquier método HTTP[2]. Para ello, simplemente usamos el método correspondiente del paquete:

[2] Métodos de petición HTTP.

Método HTTP	Llamada
GET	`requests.get()`
POST	`requests.post()`
PUT	`requests.put()`
DELETE	`requests.delete()`
HEAD	`requests.head()`
OPTIONS	`requests.options()`

9.1.3 Usando parámetros

Cuando se realiza una petición HTTP es posible incluir parámetros. Veamos distintas opciones que nos ofrece *requests* para ello.

Query string

En una petición GET podemos incluir parámetros en el llamado «query string». Los parámetros se definen mediante un *diccionario* con nombre y valor de parámetro.

Veamos un ejemplo sencillo. Supongamos que queremos **buscar paquetes de Python** que contengan la palabra «astro»:

```
>>> payload = {'q': 'astro'}

>>> response = requests.get('https://pypi.org', params=payload)

>>> response.url
'https://pypi.org/?q=astro'
```

Truco: El atributo `url` nos devuelve la URL a la se ha accedido. Útil en el caso de paso de parámetros.

Parámetros POST

Una petición POST, por lo general, siempre va acompañada de una serie de parámetros que típicamente podemos encontrar en un formulario web. Es posible realizar estas peticiones en *requests* adjuntando los parámetros que necesitemos en el mismo formato de diccionario que hemos visto para «query string».

Supongamos un ejemplo en el que tratamos de **logearnos en la página de GIPHY** con

nombre de usuario y contraseña. Para ello, lo primero que debemos hacer es inspeccionar[3] los elementos del formulario e identificar los nombres («name») de los campos. En este caso los campos son `email` y `password`:

```
>>> url = 'https://giphy.com/login'
>>> payload = {'email': 'sdelquin@gmail.com', 'password': '1234'}

>>> response = requests.post(url, data=payload)
>>> response.status_code
403
```

Hemos obtenido un código de estado 403 indicando que el acceso está prohibido.

Envío de cabeceras

Hay veces que necesitamos modificar o añadir determinados campos en las cabeceras[4] de la petición. Su tratamiento también se realiza a base de diccionarios que son pasados al método correspondiente.

Uno de los usos más típicos de las cabeceras es el «user agent»[5] donde se especifica el tipo de navegador que realiza la petición. Supongamos un ejemplo en el que queremos especificar que **el navegador corresponde con un Google Chrome corriendo sobre Windows 10**:

```
>>> user_agent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36'
>>> headers = {'user-agent': user_agent}

>>> response = requests.get('https://pypi.org', headers=headers)

>>> response.status_code
200
```

[3] Herramientas para desarrolladores en el navegador. Por ejemplo Chrome Dev Tools.

[4] Las cabeceras HTTP permiten al cliente y al servidor enviar información adicional junto a una petición o respuesta.

[5] El agente de usuario del navegador permite que el servidor identifique el sistema operativo y las características del navegador.

9.1.4 Analizando la respuesta

A continuación analizaremos distintos **elementos que forman parte de la respuesta HTTP** tras realizar la petición.

Contenido JSON

El formato JSON es ampliamente utilizado para el intercambio de datos entre aplicaciones. Hay ocasiones en las que la respuesta a una petición viene en dicho formato. Para facilitar su tratamiento *requests* nos proporciona un método que convierte el contenido JSON a un diccionario de Python.

Supongamos que queremos tener un **pronóstico del tiempo** en Santa Cruz de Tenerife. Existen múltiples servicios online que ofrecen datos meteorológicos. En este caso vamos a usar https://open-meteo.com/. La cuestión es que *los datos que devuelve esta API son en formato JSON*. Así que aprovecharemos para convertirlos de forma apropiada:

```
>>> sc_tfe = (28.4578025, -16.3563748)
>>> params = dict(latitude=sc_tfe[0], longitude=sc_tfe[1], hourly='temperature_2m')
>>> url = 'https://api.open-meteo.com/v1/forecast'

>>> response = requests.get(url, params=params)

>>> response.url
'https://api.open-meteo.com/v1/forecast?latitude=28.4578025&longitude=-16.3563748&
↪hourly=temperature_2m'

>>> data = response.json()

>>> type(data)
dict

>>> data.keys()
dict_keys(['utc_offset_seconds', 'elevation', 'latitude', 'hourly_units', 'longitude
↪', 'generationtime_ms', 'hourly'])
```

Ahora podríamos mostrar la predicción de temperatures de una manera algo más visual. Según la documentación de la API sabemos que la respuesta contiene 168 medidas de temperatura correspondientes a todas las horas durante 7 días. Supongamos que sólo queremos **mostrar la predicción de temperaturas hora a hora para el día de mañana**:

```
>>> temperatures = data['hourly']['temperature_2m']

>>> # Las temperaturas también incluyen el día de hoy
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> for i, temp in enumerate(temperatures[24:48], start=1):
...     print(f'{temp:4.1f}', end=' ')
...     if i % 6 == 0:
...         print()
...
12.0 11.9 11.9 11.8 11.8 11.7
11.7 11.7 11.6 12.0 12.8 13.6
13.9 14.0 14.1 13.9 13.7 13.3
12.8 12.2 11.8 11.7 11.6 11.5
```

Cabeceras de respuesta

Tras una petición HTTP es posible recuperar las cabeceras que vienen en la respuesta a través del atributo `headers` como un diccionario:

```
>>> response = requests.get('https://pypi.org')
>>> response.status_code
200

>>> response.headers.get('Content-Type')
'text/html; charset=UTF-8'

>>> response.headers.get('Server')
'nginx/1.13.9'
```

Cookies

Si una respuesta contiene «cookies»[6] es posible acceder a ellas mediante el diccionario `cookies`:

```
>>> response = requests.get('https://github.com')

>>> response.cookies.keys()
['_octo', 'logged_in', '_gh_sess']

>>> response.cookies.get('logged_in')
'no'
```

Nota: Las cookies también se pueden enviar en la petición usando `requests.get(url, cookies=cookies)`.

[6] Una cookie HTTP es una pequeña pieza de datos que un servidor envía al navegador web del usuario.

Ejercicio

Utilizando el paquete *requests*, haga una petición GET a https://twitter.com y obtenga los siguientes campos:

- Código de estado.
- Longitud de la respuesta.
- Valor de la cookie `guest_id`
- Valor de la cabecera `content-encoding`

9.1.5 Descargar un fichero

Hay ocasiones en las que usamos *requests* para descargar un fichero, bien sea en texto plano o binario. Veamos cómo proceder para cada tipo.

Ficheros en texto plano

El procedimiento que utilizamos es descargar el contenido desde la url y *volcarlo a un fichero* de manera estándar:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/csv_bdsc/50155.csv'

>>> response = requests.get(url)

>>> response.status_code
200

>>> with open('data.csv', 'w') as f:
...     f.write(response.text)
...
```

Consejo: Usamos `response.text` para obtener el contenido ya que nos interesa en formato «unicode».

Podemos comprobar que el fichero se ha creado satisfactoriamente:

```
$ file data.csv
plain_text.csv: UTF-8 Unicode text, with CRLF line terminators
```

Ficheros binarios

Para descargar ficheros binarios seguimos la misma estructura que para ficheros en texto plano, pero indicando el tipo binario a la hora de escribir en disco:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/xlsx/50155.xlsx'

>>> response = requests.get(url)

>>> response.status_code
200

>>> with open('data.xlsx', 'wb') as f:
...     f.write(response.content)
...
```

Consejo: Usamos `response.content` para obtener el contenido ya que nos interesa en formato «bytes».

Podemos comprobar que el fichero se ha creado satisfactoriamente:

```
$ file data.xlsx
data.xlsx: Microsoft OOXML
```

Nombre de fichero

En los ejemplos anteriores hemos puesto el nombre de fichero «a mano». Pero podría darse la situación de necesitar el nombre de fichero que descargamos. Para ello existen dos aproximaciones en función de si aparece o no la clave «attachment» en las cabeceras de respuesta.

Podemos escribir la siguiente función para ello:

```
>>> def get_filename(response):
...     try:
...         return response.headers['Content-Disposition'].split(';')[1].split('=
↪')[1]
...     except (KeyError, IndexError):
...         return response.url.split('/')[-1]
...
```

Caso para el que no disponemos de la cabecera adecuada:

```
>>> url = 'https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf'
>>> response = requests.get(url)
>>> 'attachment' in response.headers.get('Content-Disposition')
False

>>> get_filename(response)
'pytest.pdf'
```

Caso para el que sí disponemos de la cabecera adecuada:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/csv_bdsc/45070.csv'
>>> response = requests.get(url)
>>> 'attachment' in response.headers.get('Content-Disposition')
True

>>> get_filename(response)
'45070.csv'
```

9.2 beautifulsoup

El paquete Beautiful Soup es ampliamente utilizado en técnicas de «scraping» permitiendo

«parsear»[2] principalmente código HTML.[1]

```
$ pip install beautifulsoup4
```

9.2.1 Haciendo la sopa

Para empezar a trabajar con *Beautiful Soup* es necesario construir un objeto de tipo `BeautifulSoup` que reciba el contenido a «parsear»:

```
>>> from bs4 import BeautifulSoup

>>> contents = '''
... <html lang="en">
... <head>
...     <title>Just testing</title>
... </head>
... <body>
...     <h1>Just testing</h1>
...     <div class="block">
...       <h2>Some links</h2>
...       <p>Hi there!</p>
...       <ul id="data">
...         <li class="blue"><a href="https://example1.com">Example 1</a></li>
...         <li class="red"><a href="https://example2.com">Example 2</a></li>
...         <li class="gold"><a href="https://example3.com">Example 3</a></li>
...       </ul>
...     </div>
...     <div class="block">
...       <h2>Formulario</h2>
...       <form action="" method="post">
...         <label for="POST-name">Nombre:</label>
...         <input id="POST-name" type="text" name="name">
...         <input type="submit" value="Save">
...       </form>
...     </div>
...     <div class="footer">
...       This is the footer
...       <span class="inline"><p>This is span 1</p></span>
...       <span class="inline"><p>This is span 2</p></span>
...       <span class="inline"><p>This is span 2</p></span>
...     </div>
... </body>
```

(continué en la próxima página)

[2] Analizar y convertir una entrada en un formato interno que el entorno de ejecución pueda realmente manejar.

[1] Foto original de portada por Ella Olsson en Unsplash.

(proviene de la página anterior)

```
... </html>
... '''

>>> soup = BeautifulSoup(contents, features='html.parser')
```

Atención: Importar el paquete usando `bs4`. Suele llevar a equívoco por el nombre original.

A partir de aquí se abre un abanico de posibilidades que iremos desgranando en los próximos epígrafes.

9.2.2 Localizar elementos

Una de las tareas más habituales en técnicas de «scraping» y en «parsing» de contenido es la localización de determinadas elementos de interés.

Fórmulas de localización

A continuación se muestran, mediante ejemplos, distintas fórmulas para localizar elementos dentro del DOM[3]:

- Localizar **todos los enlaces**:

  ```
  >>> soup.find_all('a')
  [<a href="https://example1.com">Example 1</a>,
   <a href="https://example2.com">Example 2</a>,
   <a href="https://example3.com">Example 3</a>]
  ```

  El primer *argumento posicional* de `find_all()` es el nombre del «tag» que queremos localizar.

- Localizar todos los **elementos con la clase** `inline`:

  ```
  >>> soup.find_all(class_='inline')
  [<span class="inline"><p>This is span 1</p></span>,
   <span class="inline"><p>This is span 2</p></span>,
   <span class="inline"><p>This is span 2</p></span>]
  ```

  Los *argumentos nominales* de `find_all()` se utilizan para localizar elementos que contengan el atributo referenciado.

[3] Document Object Model en español Modelo de Objetos del Documento.

Truco: Si el atributo a localizar tiene guiones medios (por ejemplo `aria-label`) no podremos usarlo como nombre de argumento (error sintáctico). Pero sí podemos usar un diccionario en su lugar:

```
soup.find_all(attrs={'aria-label': 'box'})
```

- Localizar todos los «divs» con la clase `footer`:

```
>>> soup.find_all('div', class_='footer')  # □ soup.find_all('div', 'footer')
[<div class="footer">
       This is the footer
       <span class="inline"><p>This is span 1</p></span>
 <span class="inline"><p>This is span 2</p></span>
 <span class="inline"><p>This is span 2</p></span>
 </div>]
```

- Localizar todos los elementos cuyo atributo `type` tenga el valor `text`:

```
>>> soup.find_all(type='text')
[<input id="POST-name" name="name" type="text"/>]
```

- Localizar todos los los `h2` que contengan el texto `Formulario`:

```
>>> soup.find_all('h2', string='Formulario')
[<h2>Formulario</h2>]
```

- Localizar todos los elementos de título `h1`, `h2`, `h3`, ... Esto lo podemos atacar usando *expresiones regulares*:

```
>>> soup.find_all(re.compile(r'^h\d+.*'))
[<h1>Just testing</h1>, <h2>Some links</h2>, <h2>Formulario</h2>]
```

- Localizar todos los «input» y todos los «span»:

```
>>> soup.find_all(['input', 'span'])
[<input id="POST-name" name="name" type="text"/>,
 <input type="submit" value="Save"/>,
 <span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]
```

- Localizar todos los párrafos que están dentro del pie de página (usando **selectores CSS**):

```
>>> soup.select('.footer p')
[<p>This is span 1</p>, <p>This is span 2</p>, <p>This is span 2</p>]
```

Nota: En este caso se usa el método `select()`.

Localizar único elemento

Hasta ahora hemos visto las funciones `find_all()` y `select()` que localizan un conjunto de elementos. Incluso en el caso de encontrar sólo un elemento, se devuelve una lista con ese único elemento.

Beautiful Soup nos proporciona la función `find()` que trata de **localizar un único elemento**. Hay que tener en cuenta dos circunstancias:

- En caso de que el elemento buscado no exista, se devuelve *None*.
- En caso de que existan múltiples elementos, se devuelve el primero.

Veamos algunos ejemplos de esto:

```
>>> soup.find('form')
<form action="" method="post">
<label for="POST-name">Nombre:</label>
<input id="POST-name" name="name" type="text"/>
<input type="submit" value="Save"/>
</form>

>>> # Elemento que no existe
>>> soup.find('strange-tag')
>>>

>>> # Múltiples "li". Sólo se devuelve el primero
>>> soup.find('li')
<li class="blue"><a href="https://example1.com">Example 1</a></li>
```

Localizar desde elemento

Todas las búsquedas se pueden realizar desde cualquier elemento preexistente, no únicamente desde la raíz del DOM.

Veamos un ejemplo de ello. Si tratamos de **localizar todos los títulos «h2»** vamos a encontrar dos de ellos:

```
>>> soup.find_all('h2')
[<h2>Some links</h2>, <h2>Formulario</h2>]
```

Pero si, previamente, nos ubicamos en el segundo bloque de contenido, sólo vamos a encontrar uno de ellos:

```
>>> second_block = soup.find_all('div', 'block')[1]

>>> second_block
<div class="block">
<h2>Formulario</h2>
<form action="" method="post">
<label for="POST-name">Nombre:</label>
<input id="POST-name" name="name" type="text"/>
<input type="submit" value="Save"/>
</form>
</div>

>>> second_block.find_all('h2')
[<h2>Formulario</h2>]
```

Otras funciones de búsqueda

Hay definidas una serie de funciones adicionales de búsqueda para cuestiones más particulares:

- Localizar los **«div» superiores** a partir de un elemento concreto:

  ```
  >>> gold = soup.find('li', 'gold')

  >>> gold.find_parents('div')
  [<div class="block">
   <h2>Some links</h2>
   <p>Hi there!</p>
   <ul id="data">
   <li class="blue"><a href="https://example1.com">Example 1</a></li>
   <li class="red"><a href="https://example2.com">Example 2</a></li>
   <li class="gold"><a href="https://example3.com">Example 3</a></li>
   </ul>
   </div>]
  ```

  Se podría decir que la función `find_all()` busca en *descendientes* y que la función `find_parents()` busca en *ascendientes*.

  También existe la versión de esta *función que devuelve un único elemento*: `find_parent()`.

- Localizar los **elementos hermanos siguientes** a uno dado:

```
>>> blue_li = soup.find('li', 'blue')

>>> blue_li.find_next_siblings()
[<li class="red"><a href="https://example2.com">Example 2</a></li>,
<li class="gold"><a href="https://example3.com">Example 3</a></li>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_next_sibling()`.

- Localizar los **elementos hermanos anteriores** a uno dado:

```
>>> gold_li = soup.find('li', 'gold')

>>> gold_li.find_previous_siblings()
[<li class="red"><a href="https://example2.com">Example 2</a></li>,
<li class="blue"><a href="https://example1.com">Example 1</a></li>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_previous_sibling()`.

- Localizar **todos los elementos a continuación** de uno dado:

```
>>> submit = soup.find('input', type='submit')

>>> submit.find_all_next()
[<div class="footer">
 This is the footer
 <span class="inline"><p>This is span 1</p></span>
 <span class="inline"><p>This is span 2</p></span>
 <span class="inline"><p>This is span 2</p></span>
 </div>,
 <span class="inline"><p>This is span 1</p></span>,
 <p>This is span 1</p>,
 <span class="inline"><p>This is span 2</p></span>,
 <p>This is span 2</p>,
 <span class="inline"><p>This is span 2</p></span>,
 <p>This is span 2</p>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_next()`.

- Localizar **todos los elementos previos** a uno dado:

```
>>> ul_data = soup.find('ul', id='data')

>>> ul_data.find_all_previous(['h1', 'h2'])
[<h2>Some links</h2>, <h1>Just testing</h1>]
```

También existe la versión de esta *función que devuelve un único elemento*: `find_previous()`.

Si hubiéramos hecho esta búsqueda usando `find_parents()` no habríamos obtenido el mismo resultado ya que los elementos de título no son elementos superiores de «ul»:

```
>>> ul_data.find_parents(['h1', 'h2'])
[]
```

Atajo para búsquedas

Dado que la función `find_all()` es la más utilizada en *Beautiful Soup* se ha implementado un atajo para llamarla:

```
>>> soup.find_all('span')
[<span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]

>>> soup('span')
[<span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]
```

Aunque uno de los preceptos del *Zen de Python* es «Explicit is better than implicit», el uso de estos atajos puede estar justificado en función de muchas circunstancias.

9.2.3 Acceder al contenido

Simplificando, podríamos decir que cada elemento de la famosa «sopa» de *Beautiful Soup* puede ser un `bs4.element.Tag` o un «string».

Para el caso de los «tags» existe la posibilidad de acceder a su contenido, al nombre del elemento o a sus atributos.

Nombre de etiqueta

Podemos conocer el nombre de la etiqueta de un elemento usando el atributo `name`:

```
>>> soup.name
'[document]'

>>> elem = soup.find('ul', id='data')
>>> elem.name
'ul'

>>> elem = soup.find('h1')
>>> elem.name
'h1'
```

Truco: Es posible modificar el nombre de una etiqueta con una simple asignación.

Acceso a atributos

Los atributos de un elemento están disponibles como claves de un diccionario:

```
>>> elem = soup.find('input', id='POST-name')

>>> elem
<input id="POST-name" name="name" type="text"/>

>>> elem['id']
'POST-name'

>>> elem['name']
'name'

>>> elem['type']
'text'
```

Exite una forma de acceder al diccionario completo de atributos:

```
>>> elem.attrs
{'id': 'POST-name', 'type': 'text', 'name': 'name'}
```

Truco: Es posible modificar el valor de un atributo con una simple asignación.

Contenido textual

Es necesario aclarar las distintas opciones que proporciona *Beautiful Soup* para acceder al contenido textual de los elementos del DOM.

Atributo	Devuelve
`text`	Una cadena de texto con todos los contenidos textuales del elemento incluyendo espacios y saltos de línea
`strings`	Un generador de todos los contenidos textuales del elemento incluyendo espacios y saltos de línea
`stripped_strings`	Un generador de todos los contenidos textuales del elemento eliminando espacios y saltos de línea
`string`	Una cadena de texto con el contenido del elemento, siempre que contenga un único elemento textual

Ejemplos:

```
>>> footer = soup.find(class_='footer')

>>> footer.text
'\n      This is the footer\n      This is span 1\nThis is span 2\nThis is span 2\n'

>>> list(footer.strings)
['\n      This is the footer\n      ',
 'This is span 1',
 '\n',
 'This is span 2',
 '\n',
 'This is span 2',
 '\n']

>>> list(footer.stripped_strings)
['This is the footer', 'This is span 1', 'This is span 2', 'This is span 2']

>>> footer.string       # El "footer" contiene varios elementos

>>> footer.span.string  # El "span" sólo contiene un elemento
'This is span 1'
```

Mostrando elementos

Cualquier elemento del DOM que seleccionemos mediante este paquete se representa con el código HTML que contiene. Por ejemplo:

```
>>> data = soup.find(id='data')

>>> data
<ul id="data">
<li class="blue"><a href="https://example1.com">Example 1</a></li>
<li class="red"><a href="https://example2.com">Example 2</a></li>
<li class="gold"><a href="https://example3.com">Example 3</a></li>
</ul>
```

Existe la posibilidad de mostrar el código HTML en formato «mejorado» a través de la función `prettify()`:

```
>>> pretty_data = data.prettify()

>>> print(pretty_data)
<ul id="data">
 <li class="blue">
  <a href="https://example1.com">
   Example 1
  </a>
 </li>
 <li class="red">
  <a href="https://example2.com">
   Example 2
  </a>
 </li>
 <li class="gold">
  <a href="https://example3.com">
   Example 3
  </a>
 </li>
</ul>
```

9.2.4 Navegar por el DOM

Además de localizar elementos, este paquete permite moverse por los elementos del DOM de manera muy sencilla.

Moverse hacia abajo

Para ir profundizando en el DOM podemos utilizar los **nombres de los «tags» como atributos del objeto**, teniendo en cuenta que si existen múltiples elementos sólo se accederá al primero de ellos:

```
>>> soup.div.p
<p>Hi there!</p>

>>> soup.form.label
<label for="POST-name">Nombre:</label>

>>> type(soup.span)
bs4.element.Tag
```

Existe la opción de obtener el **contenido (como lista) de un determinado elemento**:

```
>>> type(soup.form)  # todos los elementos del DOM son de este tipo
bs4.element.Tag

>>> soup.form.contents
['\n',
 <label for="POST-name">Nombre:</label>,
 '\n',
 <input id="POST-name" name="name" type="text"/>,
 '\n',
 <input type="submit" value="Save"/>,
 '\n']

>>> type(soup.form.contents)
list
```

Advertencia: En la lista que devuelve `contents` hay mezcla de «strings» y objetos `bs4.element.Tag`.

Si no se quiere explicitar el contenido de un elemento como lista, también es posible usar un *generador* para **acceder al mismo de forma secuencial**:

```
>>> soup.form.children
<list_iterator at 0x106643100>

>>> for elem in soup.form.children:
...     print(repr(elem))
...
'\n'
<label for="POST-name">Nombre:</label>
'\n'
<input id="POST-name" name="name" type="text"/>
'\n'
<input type="submit" value="Save"/>
'\n'
```

El atributo `contents` sólo tiene en cuenta descendientes directos. Si queremos **acceder a cualquier elemento descendiente (de manera recursiva)** tenemos que usar `descendants`:

```
>>> block = soup.find_all('div')[1]

>>> block.contents
['\n',
 <h2>Formulario</h2>,
 '\n',
 <form action="" method="post">
 <label for="POST-name">Nombre:</label>
 <input id="POST-name" name="name" type="text"/>
 <input type="submit" value="Save"/>
 </form>,
 '\n']

>>> block.descendants
<generator object Tag.descendants at 0x10675d200>

>>> list(block.descendants)
['\n',
 <h2>Formulario</h2>,
 'Formulario',
 '\n',
 <form action="" method="post">
 <label for="POST-name">Nombre:</label>
 <input id="POST-name" name="name" type="text"/>
 <input type="submit" value="Save"/>
 </form>,
 '\n',
 <label for="POST-name">Nombre:</label>,
```

(continué en la próxima página)

(proviene de la página anterior)

```
 'Nombre:',
 '\n',
 <input id="POST-name" name="name" type="text"/>,
 '\n',
 <input type="submit" value="Save"/>,
 '\n',
 '\n']
```

Importante: Tener en cuenta que `descendants` es un generador que devuelve un iterable.

Moverse hacia arriba

Para acceder al **elemento superior de otro dado**, podemos usar el atributo `parent`:

```
>>> li = soup.find('li', 'blue')

>>> li.parent
<ul id="data">
<li class="blue"><a href="https://example1.com">Example 1</a></li>
<li class="red"><a href="https://example2.com">Example 2</a></li>
<li class="gold"><a href="https://example3.com">Example 3</a></li>
</ul>
```

También podemos acceder a **todos los elementos superiores (ascendientes)** usando el generador `parents`:

```
>>> for elem in li.parents:
...     print(elem.name)
...
ul
div
body
html
[document]
```

Otros movimientos

En la siguiente tabla se recogen el resto de atributos que nos permiten movernos a partir de un elemento del DOM:

Atributo	Descripción
`next_sibling`	Obtiene el siguiente elemento «hermano»
`previous_sibling`	Obtiene el anterior elemento «hermano»
`next_siblings`	Obtiene los siguientes elementos «hermanos» (iterador)
`previous_siblings`	Obtiene los anteriores elementos «hermanos» (iterador)
`next_element`	Obtiene el siguiente elemento
`previous_element`	Obtiene el anterior elemento

Advertencia: Con estos accesos también se devuelven los saltos de línea `'\n'` como elementos válidos. Si se quieren evitar, es preferible usar las *funciones definidas aquí*.

Ejercicio

Escriba un programa en Python que obtenga de https://pypi.org datos estructurados de los «Trending projects» y los muestre por pantalla utilizando el siguiente formato:

```
<nombre-del-paquete>,<versión>,<descripción>,<url>
```

Se recomienda usar el paquete *requests* para obtener el código fuente de la página. Hay que tener en cuenta que el listado de paquetes cambia cada pocos segundos, a efectos de comprobación.

9.3 selenium

Selenium es un proyecto que permite **automatizar navegadores**. Está principalmente enfocado al testeo de aplicaciones web pero también permite desarrollar potentes flujos de trabajo como es el caso de las técnicas de *scraping*.[1]

Existen múltiples «bindings»[2] pero el que nos ocupa en este caso es el de Python:

```
$ pip install selenium
```

9.3.1 Pasos previos

Documentación

Recomendamos la documentación oficial de Selenium como punto de entrada a la librería. Eso sí, como ya hemos comentado previamente, existen adaptaciones para Python, Java, CSharp, Ruby, JavaScript y Kotlin, por lo que es conveniente fijar la pestaña de **Python** en los ejemplos de código:

Es igualmente importante manejar la documentación de la API para Python.

[1] Foto original de portada por Andy Kelly en Unsplash.
[2] Adaptación (interface) de la herramienta a un lenguaje de programación concreto.

Prerequisitos

Navegador web

Selenium necesita un **navegador web** instalado en el sistema para poder funcionar. Dentro de las opciones disponibles están Chrome, Firefox, Edge, Internet Explorer y Safari. En el caso de este documento vamos a utilizar Firefox. Su descarga e instalación es muy sencilla.

Driver

Además de esto, también es necesario disponer un **«webdriver»** que permita manejar el navegador (a modo de marioneta). Cada navegador tiene asociado un tipo de «driver». En el caso de Firefox hablamos de geckodriver. Su descarga e instalación es muy sencilla.

9.3.2 Configuración del driver

El «driver» es el manejador de las peticiones del usuario. Se trata del objeto fundamental en Selenium que nos permitirá interactuar con el navegador y los sitios web.

Inicialización del driver

Para inicializar el «driver», en su versión más simple, usaremos el siguiente código:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
```

En este momento se abrirá una ventana con el navegador Firefox.

Truco: Es posible usar otros navegadores. La elección de este documento por Firefox tiene que ver con cuestiones de uso durante los últimos años.

Capacidades del navegador

Cuando inicializamos el «driver» podemos asignar ciertas «capacidades» al navegador. Las podemos dividir en dos secciones: opciones y perfil.

Opciones del navegador

Una de las opciones más utilizadas es la capacidad de ocultar la ventana del navegador. Esto es útil cuando ya hemos probado que todo funciona y queremos automatizar la tarea:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.firefox.options import Options

>>> options = Options()
>>> options.headless = True

>>> driver = webdriver.Firefox(options=options)
```

El resto de opciones se pueden consultar en la documentación de la API[3].

Perfil del navegador

Es posible definir un perfil personalizado para usarlo en el navegador controlado por el «driver».

Como ejemplo, podríamos querer desactivar javascript en el navegador (por defecto está activado). Esto lo haríamos de la siguiente manera:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.firefox.options import Options
>>> from selenium.webdriver.firefox.firefox_profile import FirefoxProfile

>>> firefox_profile = FirefoxProfile()
>>> firefox_profile.set_preference('javascript.enabled', False)

>>> options=Options()
>>> options.profile = firefox_profile

>>> driver = webdriver.Firefox(options=options)
```

Existe una cantidad ingente de parámetros configurables en el perfil de usuario[3]. Se pueden consultar en estos dos enlaces:

- https://searchfox.org/mozilla-release/source/modules/libpref/init/all.js

[3] En este caso aplicable al navegador Firefox.

- https://searchfox.org/mozilla-release/source/browser/app/profile/firefox.js

Fichero de log

Desde la primera vez que inicializamos el «driver», se crea un fichero de log en el directorio de trabajo con el nombre `geckodriver.log`[3]. En este fichero se registran todos los mensajes que se producen durante el tiempo de vida del navegador.

Es posible, aunque no recomendable, **evitar que se cree el fichero de log** de la siguiente manera:

```
>>> import os

>>> driver = webdriver.Firefox(options=options, service_log_path=os.devnull)
```

Nota: De igual modo, con el método anterior podemos cambiar la ruta y el nombre del fichero de log.

9.3.3 Navegando

La forma de **acceder a una url** es utilizar el método `.get()`:

```
>>> driver.get('https://aprendepython.es')
```

Importante: Cuando se navega a un sitio web, Selenium espera (por defecto) a que la propiedad `document.readyState` tenga el valor `complete`. Esto no implica necesariamente que la página se haya cargado completamente, especialmente en páginas que usan mucho javascript para cargar contenido dinámicamente.

Podemos movernos **«hacia adelante» y «hacia detrás»** con:

```
>>> driver.forward()
>>> driver.back()
```

Si fuera necesario, también existe la posibilidad de **refrescar la página**:

```
>>> driver.refresh()
```

Una vez terminadas todas las operaciones requeridas, es altamente recomendable **salir del navegador** para liberar los recursos que pueda estar utilizando:

```
>>> driver.quit()
```

9.3.4 Localizando elementos

Una vez que hemos accedido a un sitio web, estamos en disposición de localizar elementos dentro del DOM[4]. El objeto `driver` nos ofrece las siguientes funciones para ello:

Acceso	Función	Localizador
Clase	`find_elements_by_class_name()`	`By.CLASS_NAME`
Selector CSS	`find_elements_by_css_selector()`	`By.CSS_SELECTOR`
Atributo ID	`find_elements_by_id()`	`By.ID`
Texto en enlace	`find_elements_by_link_text()`	`By.LINK_TEXT`
Texto en enlace (parcial)	`find_elements_by_partial_link_text()`	`By.PARTIAL_LINK_TEXT`
Atributo NAME	`find_elements_by_name()`	`By.NAME`
Nombre de etiqueta	`find_elements_by_tag_name()`	`By.TAG_NAME`
XPath	`find_elements_by_xpath()`	`By.XPATH`

Todas estas funciones tienen su correspondiente versión **para devolver un único elemento** que cumpla con el filtro especificado. En caso de que hayan varios, sólo se devolverá el primero de ellos. El nombre de estas funciones sigue el patrón en singular:

```
find_element_by_<accesor>()
```

Si te estás preguntando **para qué sirve el localizador** de la tabla anterior, es porque también existe la opción de localizar elementos mediante una función genérica:

```
>>> from selenium.webdriver.common.by import By

>>> # Estas dos llamadas tienen el mismo significado
>>> driver.find_elements_by_class_name('matraca')
>>> driver.find_elements(By.CLASS_NAME, 'matraca')
```

Veamos un ejemplo práctico de esto. Supongamos que queremos **obtener los epígrafes principales de la tabla de contenidos de «Aprende Python»**:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://aprendepython.es')
```

(continué en la próxima página)

[4] Document Object Model.

(proviene de la página anterior)

```
>>> toc = driver.find_elements_by_css_selector('div.sidebar-tree li.toctree-l1')

>>> for heading in toc:
...     print(heading.text)
...
Introducción
Entornos de desarrollo
Tipos de datos
Control de flujo
Estructuras de datos
Modularidad
Procesamiento de texto
Ciencia de datos
Scraping
```

Truco: Un poco más adelante veremos la forma de acceder a la información que contiene cada elemento del DOM.

Cada elemento que obtenemos con las funciones de localización es de tipo FirefoxWebElement:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://aprendepython.es')

>>> element = driver.find_element_by_id('aprende-python')

>>> type(element)
selenium.webdriver.firefox.webelement.FirefoxWebElement
```

9.3.5 Interacciones

Si bien el acceso a la información de un sitio web puede ser un objetivo en sí mismo, para ello podríamos usar herramientas como *requests*. Sin embargo, cuando entra en juego la interacción con los elementos del DOM, necesitamos otro tipo de aproximaciones.

Selenium nos permite hacer clic en el lugar deseado, enviar texto por teclado, borrar una caja de entrada o manejar elementos de selección, entre otros.

Clic

Para **hacer clic** utilizamos la función homónima. Veamos un ejemplo en el que accedemos a https://amazon.es y tenemos que aceptar las «cookies» haciendo clic en el botón correspondiente:

```
>>> driver = webdriver.Firefox()

>>> driver.get('https://amazon.es')

>>> accept_cookies = driver.find_element_by_id('sp-cc-accept')

>>> accept_cookies.click()
```

Inspeccionando el DOM

Una tarea inherente a las técnicas de «scraping» y a la automatización de comportamientos para navegadores web es la de **inspeccionar los elementos del DOM**. Esto se puede hacer desde las herramientas para desarrolladores que incluyen los navegadores[5].

Figura 1: Botón de «cookies» en amazon.es

Para el ejemplo anterior de Amazon en el que debemos identificar el botón para aceptar «cookies», abrimos el inspector de Firefox y descubrimos que su `id` es `sp-cc-accept`. Si no lo tuviéramos disponible habría que hacer uso de otros localizadores como «xpath» o selectores de estilo.

Ejercicio

[5] Para Firefox tenemos a disposición la herramienta Inspector.

Escriba un programa en Python que, utilizando Selenium, pulse el botón de «¡JUGAR!» en el sitio web https://wordle.danielfrg.com/. Los selectores «xpath» pueden ser de mucha ayuda.

Enviar texto

Típicamente encontraremos situaciones donde habrá que enviar texto a algún campo de entrada de un sitio web. Selenium nos permite hacer esto.

Veamos un ejemplo en el que tratamos de **hacer login sobre PyPI**:

```
>>> driver = webdriver.Firefox()

>>> driver.get('https://pypi.org/account/login/')

>>> username = driver.find_element_by_id('username')
>>> password = driver.find_element_by_id('password')

>>> username.send_keys('sdelquin')
>>> password.send_keys('1234')

>>> login_btn_xpath = '//*[@id="content"]/div/div/form/div[2]/div[3]/div/div/input'
>>> login_btn = driver.find_element_by_xpath(login_btn_xpath)

>>> login_btn.click()
```

En el caso de que queramos enviar alguna tecla «especial», Selenium nos proporciona un conjunto de símbolos para ello, definidos en selenium.webdriver.common.keys.

Por ejemplo, si quisiéramos **enviar las teclas de cursor**, haríamos lo siguiente:

```
>>> from selenium.webdriver.common.keys import Keys

>>> element.send_keys(Keys.RIGHT)  # →
>>> element.send_keys(Keys.DOWN)   # ↓
>>> element.send_keys(Keys.LEFT)   # ←
>>> element.send_keys(Keys.UP)     # ↑
```

Ejercicio

Escriba un programa en Python utilizando Selenium que, dada una palabra de 5 caracteres, permita enviar ese «string» a https://wordle.danielfrg.com/ para jugar.

Tenga en cuenta lo siguiente:

- En primer lugar hay que pulsar el botón de «¡JUGAR!».

- El elemento sobre el que enviar texto podría ser directamente el «body».
- Puede ser visualmente interesante poner un `time.sleep(0.5)` tras la inserción de cada letra.
- Una vez enviada la cadena de texto hay que pulsar ENTER.

Borrar contenido

Si queremos borrar el contenido de un elemento web editable, típicamente una caja de texto, lo podemos hacer usando el método `.clear()`.

Manejo de selects

Los elementos de selección `<select>` pueden ser complicados de manejar a nivel de automatización. Para suplir esta dificultad, Selenium proporciona el objecto `Select`.

Supongamos un ejemplo en el que modificamos el idioma de búsqueda de Wikipedia. En primer lugar hay que acceder a la web y seleccionar el desplegable del idioma de búsqueda:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.support.select import Select

>>> driver = webdriver.Firefox()
>>> driver.get('https://wikipedia.org')

>>> lang = driver.find_element_by_id('searchLanguage')
>>> lang_handler = Select(lang)
```

Ahora vamos a seleccionar el *idioma inglés* como idioma de búsqueda:

```
>>> # Selección por el índice que ocupa cada "option" (base 0)
>>> lang_handler.select_by_index(17)

>>> # Selección por el campo "value" de cada "option"
>>> lang_handler.select_by_value('en')

>>> # Selección por el texto visible de cada "option"
>>> lang_handler.select_by_visible_text('English')
```

Truco: Estas tres funciones tienen su correspondiente `deselect_by_<accesor>`.

Si queremos identificar las opciones que están actualmente seleccionadas, podemos usar los siguientes atributos:

```
>>> lang_handler.all_selected_options
[<selenium.webdriver.firefox.webelement.FirefoxWebElement (session="8612e5b7-6e66-
↪4121-8869-ffce4139d197", element="8433ffdb-a8ad-4e0e-9367-d63fe1418b94")>]

>>> lang_handler.first_selected_option
<selenium.webdriver.firefox.webelement.FirefoxWebElement (session="8612e5b7-6e66-
↪4121-8869-ffce4139d197", element="8433ffdb-a8ad-4e0e-9367-d63fe1418b94")>
```

También es posible listar todas las opciones disponibles en el elemento `select`:

```
>>> lang_handler.options
[..., ..., ...]
```

Ver también:

API del objeto Select

9.3.6 Acceso a atributos

Como ya hemos comentado, los objetos del DOM con los que trabaja Selenium son de tipo `FirefoxWebElement`. Veremos los mecanismos disponibles para poder acceder a sus atributos.

Para ejemplificar el acceso a la información de los elementos del DOM, vamos a **localizar el botón de descarga** en la web de Ubuntu:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://ubuntu.com/')

>>> # Aceptar las cookies
>>> cookies = driver.find_element_by_id('cookie-policy-button-accept')
>>> cookies.click()

>>> download_btn = driver.find_element_by_id('takeover-primary-url')
```

Nombre de etiqueta

```
>>> download_btn.tag_name
'a'
```

Tamaño y posición

Para cada elemento podemos obtener un diccionario que contiene la posición en pantalla (x, y) acompañado del ancho y del alto (todo en píxeles):

```
>>> download_btn.rect
{'x': 120.0,
 'y': 442.3999938964844,
 'width': 143.64999389648438,
 'height': 36.80000305175781}
```

Estado

Veamos las funciones disponibles para saber si un elemento se está mostrando, está habilitado o está seleccionado:

```
>>> download_btn.is_displayed()
True

>>> download_btn.is_enabled()
True

>>> download_btn.is_selected()
False
```

Propiedad CSS

En caso de querer conocer el valor de cualquier propiedad CSS de un determinado elemento, lo podemos conseguir así:

```
>>> download_btn.value_of_css_property('background-color')
'rgb(14, 132, 32)'

>>> download_btn.value_of_css_property('font-size')
'16px'
```

Texto del elemento

Cuando un elemento incluye texto en su contenido, ya sea de manera directa o mediante elementos anidados, es posible acceder a esta información usando:

```
>>> download_btn.text
'Download Now'
```

Elemento superior

Selenium también permite obtener el elemento superior que contiene a otro elemento dado:

```
>>> download_btn.parent
<selenium.webdriver.firefox.webdriver.WebDriver (session="8612e5b7-6e66-4121-8869-
↪ffce4139d197")>
```

Propiedad de elemento

De manera más genérica, podemos obtener el valor de cualquier atributo de un elemento del DOM a través de la siguiente función:

```
>>> download_btn.get_attribute('href')
'https://ubuntu.com/engage/developer-desktop-productivity-whitepaper
```

9.3.7 Esperas

Cuando navegamos a un sitio web utilizando `driver.get()` es posible que el elemento que estamos buscando no esté aún cargado en el DOM porque existan peticiones asíncronas pendientes o contenido dinámico javascript. Es por ello que Selenium pone a nuestra disposición una serie de **esperas explícitas** hasta que se cumpla una determinada condición.

Las esperas explícitas suelen hacer uso de condiciones de espera. Cada una de estas funciones se puede utilizar para un propósito específico. Quizás una de las funciones más habituales sea `presence_of_element_located()`.

Veamos un ejemplo en el que cargamos la web de Stack Overflow y esperamos a que el pie de página esté disponible:

```
>>> from selenium.webdriver.support.ui import WebDriverWait
>>> from selenium.webdriver.support import expected_conditions as EC
>>> from selenium.webdriver.common.by import By

>>> driver.get('https://stackoverflow.com')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> footer = WebDriverWait(driver, 10).until(
...     EC.presence_of_element_located((By.ID, 'footer')))

>>> print(footer.text)
STACK OVERFLOW
Questions
Jobs
Developer Jobs Directory
Salary Calculator
Help
Mobile
...
```

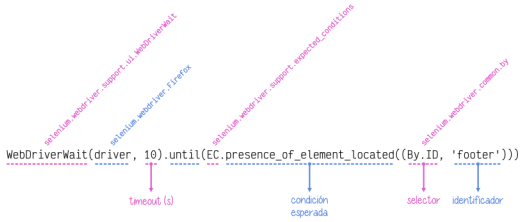

Figura 2: Anatomía de una condición de espera

Atención: En el caso de que el elemento por el que estamos esperando no «aparezca» en el DOM, y una vez pasado el tiempo de «timeout», Selenium eleva una *excepción* de tipo `selenium.common.exceptions.TimeoutException`.

9.3.8 Ejecutar javascript

Puede llegar a ser muy útil la ejecución de javascript en el navegador. La casuística es muy variada. En cualquier caso, Selenium nos proporciona el método `execute_script()` para esta tarea.

Supongamos un ejemplo en el que queremos **navegar a la web de GitHub y hacer «scroll» hasta el final de la página**:

```
>>> driver = webdriver.Firefox()
>>> driver.get('https://github.com')

>>> body = driver.find_element_by_tag_name('body')

>>> driver.execute_script('arguments[0].scrollIntoView(false)', body)
```

Cuando en la función `execute_script()` se hace referencia al array `arguments[]` podemos pasar elementos Selenium como argumentos y aprovechar así las potencialidades javascript. El primer argumento corresponde al índice 0, el segundo argumento al índice 1, y así sucesivamente.

Ejercicio

Escriba un programa en Python que permita sacar un listado de supermercados Mercadona dada una geolocalización `(lat,lon)` como dato de entrada.

Pasos a seguir:

1. Utilizar el siguiente *f-string* para obtener la url de acceso: `f'https://info.mercadona.es/es/supermercados?coord={lat}%2C{lon}'`
2. Aceptar las cookies al acceder al sitio web.
3. Hacer scroll hasta el final de la página para hacer visible el botón «Ver todos». Se recomienda usar javascript para ello.
4. Localizar el botón «Ver todos» y hacer clic para mostrar todos los establecimientos (de la geolocalización). Se recomienda una espera explícita con acceso por «xpath».
5. Recorrer los elementos desplegados `li` y mostrar el contenido textual de los elementos `h3` que hay en su interior.

Como detalle final, y una vez que compruebe que el programa funciona correctamente, aproveche para inicializar el «driver» *ocultando la ventana del navegador*.

Puede probar su programa con la localización de Las Palmas de Gran Canaria (28.1035677, -15.5319742).
