Web scraping con Python. Extraer datos de una web. Guía de inicio de Beautiful Soup

Categoría: Tutoriales Python (https://j2logo.com/category/blog/tutoriales-python/)
avanzado (https://j2logo.com/tag/avanzado/), beautifulsoup (https://j2logo.com/tag/beautifulsoup/), python (https://j2logo.com/tag/python/), scraping (https://j2logo.com/tag/scraping/)

El web scraping es una técnica que permite extraer datos e información de una web. Este tutorial es una guía de inicio al web scraping con Python, utilizando para ello la librería *Beautiful Soup*.

Antes de comenzar, quiero resaltar que si quieres realizar *web scraping*, debes ser justo y actuar de forma legal y según las políticas que rijan cada una de las páginas de las que pretendas extraer información.

Lejos de lo que te puedas imaginar y de que pienses que el web scraping es cosa de hackers, lo cierto es que el web scraping permite, por ejemplo, llevar a cabo un análisis SEO de una web, comprobar enlaces rotos, generar el sitemap de una página, vigilar a la competencia o estar al tanto de cambios en una web. Esta técnica también puede utilizarse en las primeras fases de un proyecto de Big Data o Machine Learning, en los que datos e información juegan un papel importantísimo.

Así que, ¿estás preparad@ para aprender web scraping con Python?

Índice

Web scraping con Python

¿Por qué web scraping con Python? Fundamentalmente, por ser un lenguaje intuitivo, fácil de usar y por las librerías y herramientas que han surgido en torno al mismo para llevar a cabo esta técnica.

Yo, en función del proyecto y sus características, normalmente sigo una de estas dos alternativas:

- Utilizar las librerías *requests* y *Beautiful Soup* de manera conjunta (*es lo que veremos en este tutorial*).
- Utilizar el framework *Scrapy*. Este framework, además de hacer **scraping** (extraer información de una página) permite hacer **crawling** fácilmente (descubrir los enlaces de una web y navegar a través de ellos).

Qué es Beautiful Soup

Beautiful Soup es una librería Python que permite extraer información de contenido en formato *HTML* o *XML*. Para usarla, es necesario especificar un *parser*, que es responsable de transformar un documento *HTML* o *XML* en un árbol complejo de objetos Python. Esto permite, por ejemplo, que podamos interactuar con los elementos de una página web como si estuviésemos utilizando las herramientas del desarrollador de un navegador.

A la hora de extraer información de una web, uno de los parsers más utilizado es el parser *HTML* de *lxml*. Precisamente, será el que utilicemos en este tutorial.

A continuación, te muestro cómo instalar tanto la librería *Beautiful Soup* como el parser *lxml* utilizando el gestor de paquetes *pip*.

Para instalar **Beautiful Soup**, ejecuta el siguiente comando:

```
1.  $> pip install beautifulsoup4
```

Para instalar el parser **lxml**, ejecuta el siguiente comando:

```
1.  $> pip install lxml
```

Antes de entrar en detalle con los ejemplos, te voy a indicar los pasos que debes seguir para extraer información de una web:

1 – Identifica los elementos de la página de los que extraer la información

Las páginas web son documentos estructurados formados por una jerarquía de elementos. El primer paso para extraer información es identificar correctamente el elemento o elementos que contienen la información deseada.

Para ello, lo más fácil es abrir la página en un navegador e inspeccionar el elemento. Esto se consigue haciendo clic con el botón derecho sobre el elemento en cuestión y pulsando sobre la opción *Inspeccionar* o *Inspeccionar elemento* (depende del navegador).

Quédate con toda la información disponible asociada al elemento (como la *etiqueta*, o los atributos *id* y/o *class*) ya que, posteriormente, te hará falta para utilizarla en *Beautiful Soup*.

2 – Descarga el contenido de la página

Para ello, utiliza la librería requests (https://j2logo.com/python/python-requests-peticiones-http/). El contenido de la respuesta, el que contiene la página en *HTML*, será el que pasemos posteriormente a *Beautiful Soup* para generar el árbol de elementos y poder hacer consultas al mismo.

3 – Crear la «sopa»

El contenido de la página obtenido en el paso anterior será el que utilicemos para crear la «sopa», esto es, el árbol de objetos Python que representan al documento *HTML*.

Para ello, hay que crear un objeto de tipo `BeautifulSoup`, al cuál se le pasa el texto en formato *HTML* y el identificador del *parser* a utilizar:

```
1.  import requests
2.  from bs4 import BeautifulSoup
3.
4.  r = requests.get('http://unapagina.xyz')
5.  soup = BeautifulSoup(r.text, 'lxml')
```

4 – Busca los elementos en la «sopa» y obtén la información deseada

El último paso es hacer una búsqueda en el árbol y obtener los objetos que contienen la

Tipos de objetos de Beautiful Soup

En la sección anterior hemos visto cómo crear un objeto de tipo `BeautifulSoup`. Este objeto, que representa al árbol de objetos Python resultante de parsear el documento *HTML* de entrada, será el punto de partida para navegar a través de los elementos del árbol, así como para realizar las búsquedas necesarias en el mismo.

Para lo que resta de tutorial, vamos a tomar como referencia el siguiente documento *HTML*. Fíjate bien en su contenido porque lo tomaremos de base para explicar las operaciones fundamentales que se pueden llevar a cabo a la hora de extraer información de una web con *Beautiful Soup*.

> 🎯 Imagina que este documento *HTML* se ha obtenido al consultar una página con la librería `requests`.

```
1.  <html lang="es">
2.  <head>
```

```
    <h1>El título de la página</h1>
    <p>Este es el primer párrafo</p>
    <p>Este es el segundo párrafo</p>
    <div id="innerDiv">
        <div class="links">
            <a href="https://pagina1.xyz/">Enlace 1</a>
            <a href="https://pagina2.xyz/">Enlace 2</a>
        </div>
        <div class="right">
            <div class="links">
                <a href="https://pagina3.xyz/">Enlace 3</a>
                <a href="https://pagina4.xyz/">Enlace 4</a>
            </div>
        </div>
    </div>
    <div id="footer">
        <!-- El footer -->
        <p>Este párrafo está en el footer</p>
        <div class="links footer-links">
            <a href="https://pagina5.xyz/">Enlace 5</a>
        </div>
    </div>
</div>
</body>
</html>
```

Bien, teniendo en cuenta este contenido, paso a explicarte los distintos objetos que puedes encontrar en un árbol (o sopa) de *Beautiful Soup*. Pero antes, vamos a crear el objeto `BeautifulSoup` para que podamos *jugar* un poco.

```
from bs4 import BeautifulSoup

contenido = """
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Página de prueba</title>
</head>
<body>
<div id="main" class="full-width">
    <h1>El título de la página</h1>
    <p>Este es el primer párrafo</p>
    ...
</div>
</body>
</html>
"""

soup = BeautifulSoup(contenido, 'lxml')
```

Tag

Este objeto se corresponde con una etiqueta *HTML* o *XML*. Por ejemplo, dado el objeto `soup`, podemos acceder al objeto (tag) que representa al **título** de la página usando la etiqueta `title`:

un diccionario. Además, se puede acceder a ese diccionario por medio del atributo `attrs`:

```
>>> div_main = soup.div
>>> div_main['id']
'main'
>>> div_main.attrs
{'id': 'main', 'class': ['full-width']}
```

Como puedes apreciar, hay atributos que son simples, como el `id` y otros, como `class`, que son multivaluados.

NavigableString

Un objeto de este tipo representa a la cadena de texto que hay contenida en una etiqueta. Se accede por medio de la propiedad `string`:

```
>>> primer_parrafo = soup.p
>>> texto = primer_parrafo.string
>>> texto
'Este es el primer párrafo'
>>> type(texto)
<class 'bs4.element.NavigableString'>
```

Además, la clase `NavigableString` es clase padre de otras, entre las que se encuentran:

- `Comment`: Esta clase representa un comentario *HTML*
- `Stylesheet`: Esta clase representa un código *CSS* embebido
- `Script`: Esta clase representa un código *Javascript* embebido

Navegar a través de los elementos de Beautiful Soup

En esta sección y las dos siguientes veremos cómo podemos navegar a través de los objetos del árbol de *Beautiful Soup*.

Lo importante en este punto es que entiendas que un objeto de tipo `Tag` puede contener bien otros objetos de tipo `Tag`, bien objetos de tipo `NavegableString`.

En la sección anterior hemos visto que **es posible acceder a los objetos del árbol utilizando los nombres de las etiquetas como atributos**. Esta es la forma más simple de navegar a través del árbol.

```
>>> soup.meta['charset']
'UTF-8'
```

Además, usando los nombres de etiquetas se puede navegar hacia los elementos interiores de un objeto.

Si vuelves a examinar el documento *HTML* del ejemplo, verás que el bloque `div` que contiene los enlaces `Enlace 1` y `Enlace 2` está contenido en el bloque `div` con `id="innerDiv"` que a su vez está contenido en el bloque con `id="main"`. Por tanto, podemos acceder al mismo de la siguiente manera:

```
>>> soup.div.div.div
<div class="links">
<a href="https://pagina1.xyz/">Enlace 1</a>
<a href="https://pagina2.xyz/">Enlace 2</a>
</div>
```

Sin embargo, esta forma de acceder a los elementos puede que no sea la más adecuada en todas las situaciones. ¿Por qué? Porque **de este modo, *Beautiful Soup* solo devuelve el primer elemento que se corresponda con la etiqueta**.

Por tanto, así sería imposible acceder, por ejemplo, al bloque `div` con `id="footer"`.

Hijos y descendientes de un elemento en Beautiful Soup

Hay otras formas de acceder a los hijos de un objeto, además de a través de las etiquetas:

- **El atributo** `contents`: Devuelve una lista con todos los hijos de **primer nivel** de un objeto.
- **Usando el generador** `children`: Devuelve un iterador para recorrer los hijos de **primer nivel** de un objeto.
- **Por medio del generador** `descendants`: Este atributo devuelve un iterador que permite recorrer todos los hijos de un objeto. No importa el nivel de anidamiento.

Veamos esto con un ejemplo. Imagina que quieres acceder a todos los hijos del bloque `div` con `id="innerDiv"`.

```
>>> inner_div = soup.div.div

# contents
>>> hijos = inner_div.contents
```

```
div
div


# children
>>> hijos = inner_div.children
>>> type(hijos)
<class 'list_iterator'>
>>> for child in hijos:
...     if child.name:  # Ignoramos los saltos de línea
...         print(f'{child.name}')
div
div


# descendants
>>> hijos = inner_div.descendants
>>> for child in hijos:
...     if child.name:
...         print(f'{child.name}')
div
a
a
div
div
a
a
```

❗ **IMPORTANTE:** Los objetos de tipo texto no contienen los atributos que hacen referencia a los hijos, solo el atributo `string`.

Padres de un elemento

Además de a los hijos, es posible navegar hacia arriba en el árbol accediendo a los objetos padre de un elemento. Para ello, puedes usar las propiedades `parent` y `parents`:

- `parent` referencia al objeto padre de un elemento (`Tag` o `NavigableString`).
- `parents` es un generador que permite recorrer recursivamente todos los elementos padre de uno dado.

🎯 El padre del objeto `BeautifulSoup` es `None`.

filtros.

Beautiful Soup pone a nuestra disposición diferentes métodos para buscar elementos en el árbol. Sin embargo, dos de los principales son `find_all()` y `find()`.

Ambos métodos trabajan de forma similar. Básicamente, buscan entre los descendientes de un objeto de tipo `Tag` y recuperan todos aquellos que cumplan una serie de filtros.

Filtro por nombre de etiqueta

El filtro más básico consiste en pasar el nombre de la etiqueta a buscar como primer argumento de la función (parámetro `name`).

Imagina que quieres recuperar todos los enlaces (etiqueta `<a>`) que hay en el texto HTML del ejemplo. Se podría hacer del siguiente modo:

```
>>> enlaces = soup.find_all('a')
>>> for enlace in enlaces:
...         print(enlace)

<a href="https://pagina1.xyz/">Enlace 1</a>
<a href="https://pagina2.xyz/">Enlace 2</a>
<a href="https://pagina3.xyz/">Enlace 3</a>
<a href="https://pagina4.xyz/">Enlace 4</a>
<a href="https://pagina5.xyz/">Enlace 5</a>
```

Filtro por atributos

Además del nombre de la etiqueta, puedes especificar parámetros con nombre. Si estos no coinciden con los nombres de los parámetros de la función, serán tratados como atributos de la etiqueta entre los que filtrar.

Por ejemplo, si quisieras encontrar el bloque `div` con `id="footer"`, podrías aplicar el siguiente filtro:

```
>>> footer = soup.find_all(id='footer')
>>> print(footer)
[<div id="footer">
<!-- El footer -->
<p>Este párrafo está en el footer</p>
<div class="links footer-links">
<a href="https://pagina5.xyz/">Enlace 5</a>
</div>
</div>]
```

Filtro por clases CSS

Supón que en el documento del ejemplo quieres buscar todos los bloques `div` que tengan la clase `links`. Lo puedes hacer de la siguiente manera:

```
>>> links_divs = soup.find_all('div', class_="links")
>>> print(links_divs)
[<div class="links">
<a href="https://pagina1.xyz/">Enlace 1</a>
<a href="https://pagina2.xyz/">Enlace 2</a>
</div>,
<div class="links">
<a href="https://pagina3.xyz/">Enlace 3</a>
<a href="https://pagina4.xyz/">Enlace 4</a>
</div>,
<div class="links footer-links">
<a href="https://pagina5.xyz/">Enlace 5</a>
</div>]
```

Si te has fijado, la búsqueda devuelve también el bloque contenido en el bloque del footer, dado que `links` es una de sus clases.

¿Cómo se puede buscar por la cadena exacta del atributo `class_`? Así:

```
>>> footer_links = soup.find_all(class_="links footer-links")
>>> print(footer_links)
[<div class="links footer-links">
<a href="https://pagina5.xyz/">Enlace 5</a>
</div>]
```

Sin embargo, las variaciones de esta cadena no funcionan:

```
>>> footer_links = soup.find_all(class_="footer-links links")
>>> print(footer_links)
[]
```

No obstante, si se quiere buscar un elemento del árbol para el que coincidan dos o más clases CSS, se puede usar un selector CSS con el método `select()`:

```
>>> footer_links = soup.select('div.footer-links.links')
>>> print(footer_links)
[<div class="links footer-links">
<a href="https://pagina5.xyz/">Enlace 5</a>
</div>]
```

Limitando las búsquedas

Como puedes observar, `find_all()` devuelve una lista de objetos. Internamente, el método tiene que recorrer todos los hijos del objeto invocador en busca de aquellos que cumplan los diferentes filtros. En ocasiones, si el árbol (o documento) es muy grande, esta función puede tardar un poco. Se puede optimizar con los siguientes parámetros:

objeto o solo aquellos de primer nivel. Por defecto es `True`.

Teniendo esto en cuenta, si estamos seguros de que el documento *HTML* solo tiene un objeto específico, se puede usar la función `find()` ya que está más optimizada. Esta función se comporta de manera similar a `find_all()`, solo que devuelve un único objeto si lo encuentra.

```
>>> soup.find('title')
<title>Página de prueba</title>
```

El código anterior es equivalente al siguiente:

```
>>> soup.find_all('title', limit=1)
[<title>Página de prueba</title>]
```

Conclusiones

Has llegado al final del tutorial de introducción a *web scraping con Python*. Como en otros campos de la programación, no hay una única forma de hacer las cosas.

No obstante, sí que me gustaría darte unos tips antes de terminar:

- Como te indicaba en la introducción, sé respetuoso con los términos legales de las páginas de las que vayas a extraer información. Si no tienes permiso, no lo hagas.
- Las páginas web cambian continuamente. Es posible que un programa o script que te funcione hoy, mañana ya no lo haga. Esto implica tener que rehacer parte del trabajo. El web scraping es así.
- Ciertos sitios trabajan con frameworks Javascript que renderizan el HTML dinámicamente. Esto quiere decir que, cuando haces una petición al navegador, el contenido que descargas no es un documento HTML, sino un script. En estos casos la cosa se complica pero esto da para otro post.