Beautiful Soup de *Python* para el raspado web como método para la extracción automatizada de datos

Python's Beautiful Soup for web scraping as a method for automated data extraction from websites

Rubén Alcaraz-Martínez

Como citar este artículo:

Alcaraz-Martínez, Rubén (2025). "*Beautiful Soup* de *Python* para el raspado web como método para la extracción automatizada de datos [*Python's Beautiful Soup* for web scraping as a method for automated data extraction from websites]". *Infonomy*, 3(2), e25014.
https://doi.org/10.3145/infonomy.25.014

Rubén Alcaraz-Martínez
https://orcid.org/0000-0002-7185-0227
https://directorioexit.info/ficha2806
Universitat de Barcelona
Departament de Biblioteconomia, Documentació i Comunicació Audiovisual
Melcior de Palau, 140
08014 Barcelona, España
ralcaraz@ub.edu

Resumen

Beautiful Soup es una biblioteca de *Python* pensada para la extracción, análisis y edición de datos de documentos HTML. Tras introducir diversos conceptos y tecnologías relacionados con el raspado de datos, esta guía muestra paso a paso cómo poner en marcha un entorno compatible con esta tecnología y recoge diversos ejemplos de uso para la extracción automatizada de datos de páginas web. Para ello, además de *Beautiful Soup*, se integran otros módulos de *Python* como *pandas* (*Panel Data*) y *requests*, para manejar los datos y procesar ficheros CSV y gestionar las peticiones HTTP desde *Python*, respectivamente. También se introducen algunas soluciones más avanzadas para sortear mecanismos de protección habituales.

Palabras clave
Raspado web; *Python*; *Beautiful Soup*; *Pandas*; *DataFrames*; *Selenium*; Minería de datos; Extracción de datos.

Abstract
Beautiful Soup is a *Python* library for extracting, analysing and editing data from HTML documents. After introducing various concepts and technologies related to data scraping, this guide shows step-by-step explanation on how to set up an environment compatible with this technology and includes multiple examples of automated data extraction from web pages. In addition to *Beautiful Soup*, other *Python* modules such as pandas (Panel Data) for handling data and processing CSV files, and requests for managing HTTP requests in are also integrated. Additionally, more advanced solutions are introduced to circumvent common protection mechanisms.

Keywords
Web scraping; *Python*; *Beautiful Soup*; *Pandas*; *DataFrames*; *Selenium*; Data mining; Data extraction.

1. Qué es y para qué sirve el raspado (*scraping*) de datos

El raspado de datos, en inglés denominado habitualmente *data scraping* o *web scraping*, pero también bajo otras muchas formas aunque no siempre referidas a la misma técnica específica, como *screen scraping*, *web data extraction* o, más genéricamente, como *data mining* o *web harvesting*, entre otras, es una práctica consistente en la recolección automatizada de datos de forma alternativa al uso de una API, mediante *scripts* que lanzan consultas a uno o varios servidores web (**Diouf** *et al.*, 2019) solicitando datos específicos, normalmente en forma de HTML para, posteriormente, analizarlos, extraer la información que se precisa (**Mitchell**, 2024) y presentarla en un formato estructurado (**Sarr** *et al.*, 2018) en una hoja de cálculo o base de datos (**Sirisuriya**, 2015; **Khder**, 2021).

El raspado de datos consiste en la recolección automatizada de datos de forma alternativa al uso de una API, mediante *scripts* que lanzan consultas a uno o varios servidores web, solicitando datos específicos, normalmente en forma de HTML para, posteriormente, analizarlos, extraer la información que se precisa y presentarla en un formato estructurado en una hoja de cálculo o base de datos

Frente a la posibilidad de recolectar grandes conjuntos de datos de forma manual, lo cual implica una gran cantidad de tiempo y recursos (**Diouf** *et al.*, 2019), el raspado de datos es una aproximación mucho más eficiente, precisa (**Lawson**, 2015) y de bajo coste (**Khder**, 2021).

Este proceso, con otras tecnologías y ciertas especificidades, convive con el denominado rastreo web (*web crawling*) utilizado, por ejemplo, por los buscadores en Internet para rastrear el contenido que después ofrecen a sus usuarios a través de

sus resultados de búsqueda. El rastreo web se fundamenta en técnicas que permiten navegar por la web siguiendo enlaces (**Vording**, 2021) para, posteriormente, iniciar un proceso de extracción, análisis y almacenamiento (indexación) con técnicas de raspado.

El raspado de datos es una técnica útil y necesaria cuando se pretende recoger de forma eficiente grandes cantidades de datos disponibles en la Web. En algunos casos, la disponibilidad de una API hará innecesaria estas técnicas, aunque, en ocasiones, incluso disponiendo de una, ciertos límites pueden resolverse mediante técnicas alternativas como las que las herramientas de raspado permiten ejecutar (**Khder**, 2021).

El raspado de datos es una técnica estrechamente vinculada a la minería de datos. El proceso de raspado se entiende como una frase previa o inicial, a partir de la cual se obtienen datos de diversas fuentes sobre los cuales se aplicarán, posteriormente, técnicas estadísticas avanzadas.

El proceso de raspado se lleva a cabo a través de 3 fases bien diferenciadas (**Khder**, 2021):

- Fase de obtención, en la que se accede a la página en cuestión a través del protocolo HTTP y se obtiene una respuesta en formato HTML.
- Fase de extracción, en la que, mediante expresiones regulares, consultas *XPath* o bibliotecas de análisis sintáctico (*parseadores*), se extraen ciertos datos.
- Fase de transformación, en la que los datos obtenidos en la fase anterior se transforman en un formato estructurado para su presentación en forma de tabla, ser almacenados en un fichero o base de datos, etc.

Este trabajo introduce el concepto de raspado de datos (*scraping*), enumera algunas tecnologías disponibles y se centra en el caso específico de la biblioteca de *Python Beautiful Soup*. Se muestra el proceso de creación de un primer *script* para obtener, de un conjunto de páginas web, sus títulos, hacer un recuento de la extensión (cantidad de palabras) del contenido y saber si implementan o no *Schema.org*. A partir de estos estos ejemplos sería posible extraer el contenido de otras etiquetas HTML o valorar la presencia de otros tipos de *scripts*, etiquetas o contenidos específicos. Con el resultado se genera un fichero CSV estructurado para su posterior análisis y explotación.

2. Métodos y tecnologías

Antes de la existencia de bibliotecas modernas como *Beautiful Soup* o *Jsoup*, la extracción de datos de sitios web se realizaba mediante técnicas más rudimentarias, entre las que destaca el uso de expresiones regulares (*regex*) junto a herramientas como *grep*, integradas en líneas de comandos. Esto permitía buscar patrones específicos dentro del código para obtener y procesar esos datos (**Sirisuriya**, 2015). Otra aproximación popular es la centrada en métodos basados en la tecnología *XPath* (*XML Path Language*), un lenguaje pensado para construir expresiones que recorren y procesan documentos XML (**Grasso** *et al.*, 2013). En la actualidad, diversos lenguajes de programación como *Java*, *JavaScript* o *Python* cuentan con bibliotecas y otras

herramientas que facilitan la programación de este tipo de soluciones. Algunos ejemplos representativos son *Jsoup*[1], *Jaunt*[2] o *StormCrawler*[3] (*Java*), *Cheerio*[4] y *Apify*[5] (*JavaScript Node.js*), *Beautiful Soup*[6] o *Lxml*[7] (*Python*).

En el mercado tecnológico también es posible encontrar herramientas específicas preparadas para el raspado de datos de sitios web sin necesidad de conocer ningún lenguaje de programación. Algunos ejemplos son *ParseHub*[8], *Octoparse*[9] o *WebHarvy*[10]. También existen extensiones para navegadores como *Data Miner*[11] o *Agenty*[12].

En ámbitos específicos como el del SEO, *Screaming Frog*[13] es una de las soluciones más populares para el análisis en lote de múltiples páginas de un mismo dominio. Estrictamente hablando, no se trata de una herramienta de *scraping*, sino de rastreo web (*web crawling*) especializada en análisis SEO con algunas funciones de extracción de datos que podrían asemejarse al raspado de datos, pero mediante *XPath*, selectores CSS o expresiones regulares.

3. Usos

Las tecnologías relacionadas con el raspado son muy versátiles y se aplican a una importante cantidad de sectores y en situaciones muy diversas (**Diouf** *et al*., 2019). En este sentido, más allá del ámbito académico, otros contextos en los que estas tecnologías son protagonistas son algunos caracterizados por su volatilidad, como los servicios centrados en el monitoreo, comparación o análisis de precios (*e-commerce*, sector inmobiliario, turismo, seguros...), la recopilación de reseñas, ofertas de empleo, la personalización de servicios, el periodismo de datos, el análisis de redes sociales, la acumulación de datos personales disponibles en internet con fines poco éticos o ilícitos, o la creación de sitios web espejo con fines lucrativos o vinculados a técnicas como el *phishing*.

En el ámbito del SEO y, específicamente en la generación de contenido, las técnicas de raspado orientadas a la recopilación de diversos textos sobre una temática específica se complementan con las técnicas de reescritura o rotación (*text spinning*), mediante las cuales se genera un nuevo contenido "original" a partir de las fuentes raspadas. En ocasiones, también se someten a un proceso de traducción, facilitando así que estas prácticas pasen más desapercibidas y superen algoritmos como *Panda* (**Maheshwari**; **Ali**, 2013). Se tratan, todas ellas, de técnicas vinculadas al denominado *black hat SEO* (**Alcaraz**, 2023).

4. Consideraciones éticas y legales

La minería de datos está avalada desde un punto de vista legal en Europa para las universidades y centros de investigación, así como para las instituciones responsables de la conservación del patrimonio cultural y sus socios tecnológicos, a través de la *Directiva 2019/790* sobre los derechos de autor y derechos afines en el mercado único digital, y el correspondiente decreto que transpone esa directiva al marco jurídico español (*Real decreto-ley 24/2021*), siempre y cuando su finalidad sea la investigación científica no comercial. Esta excepción aplica a cualquier otra persona o empresa siempre y cuando los datos que se busca recopilar no estén protegidos por derechos de autor o cuando el método aplicable no implique actos de

reproducción o esta esté contemplada en la excepción obligatoria aplicable a los actos de reproducción provisional establecida en el artículo 5, apartado 1 de la *Directiva 2001/29/CE*.

Si el raspado sobrecarga o daña un servidor, el responsable puede ser demandado. Aunque no siempre es fácil probar estos daños ante un tribunal (**Krotov** *et al.*, 2020), en la última década se han dado producido diversos litigios entre empresas. Por ejemplo, en 2012, *Craigslist*, un sitio web de anuncios clasificados, demandó a *3Taps*, una empresa que raspaba sus anuncios para ofrecerlos a través de una interfaz alternativa. *Craigslist* bloqueó explícitamente las IP desde las cuales *3Taps* realizaba el raspado, pero esta ignoró estos bloqueos llegando a sobrecargar los servidores de la primera como consecuencia de las peticiones masivas de información. En esa ocasión, los tribunales fallaron a favor de *Craigslist*. Otro caso que llegó a los tribunales es el de *Facebook* contra *Power Ventures* en 2017, una empresa que raspaba la red social para recolectar datos de los usuarios y publicarlos en sitios web de terceros. Un aspecto interesante de la sentencia es que el tribunal llegó a equiparar este tipo de raspado de datos con ataques *DDoS*[14] por su capacidad para colapsar los servidores de una compañía.

Al hacer *scraping* es importante ser responsable y evitar sobrecargar los servidores de las páginas que se pretende raspar. En este sentido, algunas buenas prácticas son: usar una API específica preferentemente si se dispone de ella, respetar el fichero robots.txt, evitar raspar sitios con políticas contrarias a esta práctica, hacer peticiones con intervalos para evitar saturar el servidor y evitar raspados masivos.

5. *Beautiful Soup* y otras bibliotecas de *Python* para el raspado de datos

En los últimos años, *Python* se ha erigido como uno de los lenguajes de programación más populares y demandados, superando a otros como *Java*, *JavaScript*, *C++* o *PHP* los cuales habían capitalizado estos rankings en décadas anteriores (**Cass**, 2024). De hecho, en 2024, por primera vez, *Python* se sitúo como el lenguaje de programación más utilizado en *GitHub* por delante de *JavaScript*, que había liderado esta clasificación durante los diez años anteriores (*GitHub Staff*, 2024). Su potencia, versatilidad, legibilidad y facilidad de aprendizaje (**Khder**, 2021) son algunas de las razones de su popularidad.

En 2024, por primera vez, *Python* se sitúo como el lenguaje de programación más utilizado en *GitHub* por delante de *JavaScript*, que había liderado esta clasificación durante los diez años anteriores. Su potencia, versatilidad, legibilidad y facilidad de aprendizaje son algunas de las razones de su popularidad

Python desempeña también un papel fundamental en el entrenamiento de modelos de inteligencia artificial gracias a lo anteriormente expuesto y al hecho de contar con una importante cantidad de bibliotecas especializadas como *TensorFlow*, *Keras* (creación y entrenamiento de redes neuronales) o *Pytorch* (visión por computador y procesamiento del lenguaje natural) (**Vasilev** *et al.*, 2019).

Beautiful Soup es una biblioteca pensada para la extracción, análisis y edición de datos disponibles a partir del árbol DOM (*Document Object Model*)[15] de un documento en formato HTML o XML (**Khder**, 2021). Una característica muy interesante de *Beautiful Soup* es que es compatible con diversos *parseadores* que permiten corregir errores (etiquetas mal anidadas, sin cerrar, atributos sin comillas…) en la estructura HTML. Algunos ejemplos son *html.parser*, *lxml* o *html5lib*.

5.1. Preparación del entorno e instalación de las bibliotecas necesarias

Los pasos que continúan muestran el proceso de preparación del entorno e instalación de las bibliotecas necesarias para ejecutar *scripts* que utilizan *Beautiful Soup* en un entorno *Ubuntu 24.04.2 LTS* (*Linux*). Antes de proceder a la instalación de los componentes necesarios, se recomienda la ejecución de los siguientes comandos para actualizar la lista de paquetes disponibles y descargar e instalar las versiones más recientes de los paquetes ya instalados, respectivamente:

sudo apt update
sudo apt upgrade

Una vez actualizado, se procederá a desplegar un entorno capaz de ejecutar *scripts* escritos en *Python*. Para ello es necesario un intérprete de este lenguaje de programación, un gestor de paquetes y ciertas herramientas esenciales de desarrollo. Con el siguiente comando instalamos la versión 3 de *Python*:

sudo apt install python3

Una vez instalado es posible comprobarlo con:

python3 --version

Si todo ha ido bien, debería aparecer algo como:

Python 3.12.3

A continuación, instalamos el gestor de paquetes de *Python*, *PIP*:

sudo apt install python3-pip

Una vez instalado *Python* y el gestor de paquetes, es posible trabajar en el entorno global o en un entorno virtual. Es decir, un directorio específico que contiene una copia aislada de *Python* y *PIP*, junto al resto de los paquetes que precisemos en cada proyecto. Trabajar en entornos virtuales es una buena práctica, ya que los paquetes que se instalen en cada uno de ellos no afectarán al entorno *Python* global del sistema (aislamiento). Además, mediante esta aproximación, cada proyecto podrá tener sus propias versiones de las bibliotecas que precise, evitando conflictos con otros proyectos (compatibilidad). Finalmente, trabajar con entornos virtuales también facilita la portabilidad del proyecto. Los entornos virtuales presentan una estructura como la que se muestra en la figura 1, donde el directorio */bin* almacena los *scripts* y ejecutables, el directorio */lib* las bibliotecas instaladas, y *pyvenv.cfg* es el fichero de configuración del entorno. Adicionalmente, si se utilizan determinadas bibliotecas de *Python* que dependen de código *C/C++* (por ejemplo, *Pandas* que veremos más adelante) se generará el directorio */include*. Por otro lado, el directorio */lib64* es un enlace simbólico al directorio */lib* que permite distinguir entre bibliotecas de 32 y 64

bits. Toda esta estructura se genera automáticamente tras seguir los pasos que se muestran a continuación.

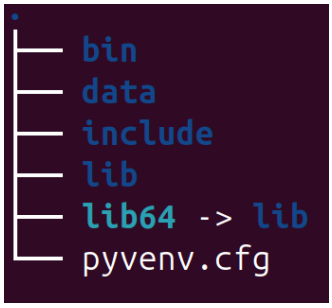

Figura 1. Estructura de un entorno virtual en *Python*.

Para trabajar en un entorno virtual, será necesario instalar el paquete *python3.12-venv* con:

sudo apt install python3.12-venv

A continuación, utilizamos el siguiente comando para crear el entorno, donde "mi_entorno" es el nombre del directorio o entorno virtual aislado:

python3 -m venv mi_entorno

Una vez creado, utilizamos el siguiente comando para activar el entorno virtual, es decir, utilizar los programas y bibliotecas del entorno en lugar de los del sistema:

source mi_entorno/bin/activate

A partir de este momento el *prompt* de la terminal cambia para indicar que nos encontramos dentro del entorno virtual (figura 2).

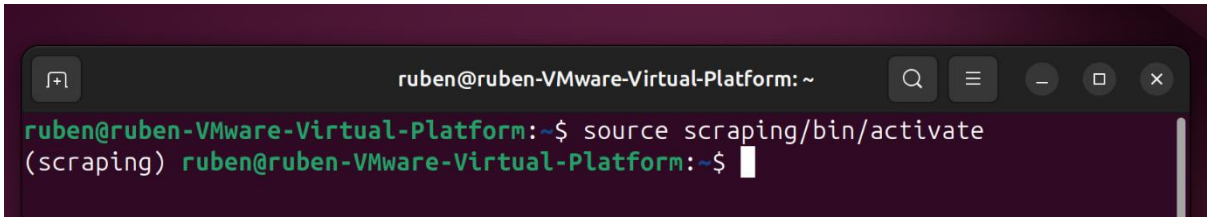

Figura 2. Terminal dentro de un entorno virtual con el nombre *scraping*.

Cada vez que reiniciemos la máquina y necesitemos volver a trabajar en el entorno virtual, será necesario activarlo con el comando anterior.

Ya dentro del entorno virtual, instalamos *Beautiful Soup* y algunas otras bibliotecas que serán necesarias para la gestión de datos tabulares desde ficheros CSV, XSLX o JSON y realizar peticiones HTTP. Concretamente, utilizaremos *Pandas* y *Requests*. *Pandas*[16] (*Panel Data*) es una biblioteca para el manejo y análisis de datos utilizada ampliamente, que en este caso en particular utilizaremos para procesar ficheros CSV.

Por otro lado, la biblioteca *Requests*[17] facilita la gestión de peticiones HTTP desde *Python*, lo que permite descargar contenido desde páginas web para, posteriormente, analizarlo con *Beautiful Soup* o *Pandas*.

Con el siguiente comando, podemos instalar las tres bibliotecas necesarias:

pip3 install pandas requests beautifulsoup4

5.2. Scripts para la extracción de datos

Nota previa: El código de ejemplo que se muestra y comenta en este y en el siguiente apartado se puede descargar desde *Figshare*[18].

Como ejemplo planteamos una situación habitual. Disponemos de una lista de páginas web en formato CSV (una tabla con una sola columna) de las cuales queremos obtener cierta información. Por ejemplo, el valor de la etiqueta *<title>* de cada una de ellas, revisar si cuentan o no con una etiqueta o contenido específico dentro de su código fuente. Por ejemplo, si utilizan *Schema.org*, y hacer un recuento de palabras en el contenido para conocer su extensión total.

Para crear el *script* generamos un fichero de texto plano con extensión *.py*. En él comenzaremos importando las dependencias o bibliotecas necesarias para llevar a cabo el raspado.

```
import pandas as pd
import requests
from bs4 import BeautifulSoup
```

En el ejemplo anterior, importamos las bibliotecas *pandas* y *requests* completas. Además, en el caso de *Pandas* se le asigna un alias, el cual permite utilizar esta biblioteca usando la abreviatura “pd” en lugar de tener que escribir una y otra vez el nombre completo (también se podría hacer con *requests*). Por otro lado, en el caso de bs4 (*Beautiful Soup*), al tratarse de una biblioteca muy grande, tan sólo importamos los paquetes que contienen los módulos y clases que necesitaremos: *BeautifulSoup*. Con esto es posible llamar a las funciones y clases necesarias directamente, lo cual ahorra tiempo y espacio en el código, así como permite gestionar mejor el consumo de memoria y rendimiento en el arranque.

Una vez importados todos los paquetes necesarios, el siguiente paso es leer el fichero CSV. Para ello, *Pandas* cuenta con la función *read_csv()*, la cual convierte el fichero en un *DataFrame*, es decir, una estructura tabular (filas y columnas). Si en lugar de un fichero CSV, tenemos un fichero *Excel*, podemos utilizar la función *read_excel()*.

```
df = pd.read_csv(“data.csv”)
```

En el fragmento de código anterior, se declara una variable (*df*) en la que se almacena el *DataFrame* resultado de procesar el fichero “data.csv”. Para declarar una función es necesario utlizar la sintaxis *biblioteca.función*. En el caso anterior, *pd* es el alias para referirnos a la biblioteca *Pandas* y *read_csv* la función. Si el fichero que se necesita procesar se encuentra en un directorio diferente, entonces será necesario indicar la ruta completa. Por ejemplo:

```
df = pd.read_csv(“/home/user/datos/data.csv”)
```

A continuación, se define una función (*get_page_data*), que podría tener otro nombre cualquiera y que recibirá un URL como parámetro:

def get_page_data(url):

El código la función se muestra en la figura 3 y se comenta a continuación.

```
def get_page_data(url):
    try:
        response = requests.get(url, timeout=5, headers={'User-Agent': 'Mozilla/5.0'})
        soup = BeautifulSoup(response.text, "html.parser")

        title = soup.title.string.strip() if soup.title else "No Title"

        words = soup.get_text().split()
        word_count = len(words)

        schema_used = "Yes" if soup.find_all("script", {"type": "application/ld+json"}) else "No"

        return title, word_count, schema_used

    except Exception as e:
        return "Error", "Error", "Error"
```

Figura 3. *Script* para recolectar los títulos, contar la extensión en palabras y comprobar si el sitio utiliza o no *Schema.org*.

En el código anterior, se utiliza *try* una estructura de control de errores que permite ejecutar un bloque de código y, en el caso de que se produzca un error o excepción, manejarlo de forma controlada, en lugar de que se produzca una detención brusca.

A continuación, se realiza la petición a cada URL mediante la función *requests.get()* la cual se ejecuta con los siguientes parámetros:

- url: con el valor de cada url disponible en el CSV procesado anteriormente.
- timeout=5: es el tiempo de espera para la respuesta. Si es superior a 5, devolverá error.
- headers={'User-Agent': 'Mozilla/5.0'}: se pasa un agente de usuario (navegador) para simular que el proceso lo está realizando una persona desde un navegador web real con el objetivo de evitar mecanismos básicos de bloqueo de peticiones sin un *user-agent* explícito.

La respuesta obtenida se almacena en la variable *response*.

A continuación, el contenido HTML de la página (disponible en *response.text*) se pasa como parámetro a la función *BeautifulSoup()* con el objetivo de poder navegar y buscar dentro de cada uno de ellos. El segundo parámetro para esta función es *html.parser*. Un *parser* (o analizador) es un motor que interpreta y convierte el contenido HTML en un formato que *Python* puede procesar. El objetivo es convertir el contenido HTML en un árbol de elementos fácil de manipular. Con el parámetro utilizado invocamos el *parser* HTML básico incluido en la biblioteca estándar de *Python*.

En la siguiente línea, se define una nueva variable (*title*). Dentro se declara la función *title* para buscar la etiqueta homónima en el código HTML. La función *string* extrae sólo el texto de la etiqueta y la función *strip()* es un método que elimina los espacios en blanco al principio y al final. A continuación, se utiliza un condicional *if* para manejar aquellos casos en los que no exista la etiqueta. En todos ellos, el *script* devolverá el texto "No Title".

Para calcular la extensión del texto disponible en la página, se utiliza la función *get_text()* de *Beautiful Soup* la cual se encarga de extraer todo el texto visible eliminando las etiquetas HTML. Al resultado se le aplica la función *split()* para dividir el texto en palabras. Esta tarea por defecto la lleva a cabo a partir de los espacios en blanco y saltos de línea. El resultado es un *array* (lista) con todas las palabras. Por ejemplo, si se encuentra un párrafo como:

> * Bienvenido a Infonomy, la revista académica de metodologías y divulgación científica *

En primer lugar, eliminar las etiquetas HTML para obtener:

> *Bienvenido a Infonomy, la revista académica de metodologías y divulgación científica*

Finalmente, generará un *array* como el siguiente:

> *['Bienvenido', 'a', 'Infonomy,', 'la', 'revista', 'académica', 'de', 'metodologías', 'y', 'divulgación', 'científica']*

El resultado anterior (*array* con la lista de palabras) se pasa como parámetro a la función de *Python len()* para calcular su longitud. El resultado (un valor numérico) se almacena en la variable *word_count*.

Para valorar la presencia o no de *Schema.org* sería posible aproximarse de diferentes formas. En el ejemplo, se utiliza el método *find_all()* de *Beautiful Soup* para buscar todas las etiquetas con el selector que se pasa como primer parámetro (en este caso, "script") y que además tengan el atributo y valor "type=application/ld+json". El resultado es una lista que se somete a un condicional *if else*, para devolver como resultado *Yes* o *No*, según si esa etiqueta se encuentra en el código o no.

Finalmente, mediante la palabra clave *return*, finaliza la ejecución de la función y se devuelven los valores contenidos en las variables que se han ido declarando en pasos anteriores con sus respectivos resultados. Datos que se guardan en el *DataFrame* de *Pandas* para exportarlos a un fichero CSV.

El bloque final (*except Exception as e*) es parte de un bloque *try-except* de *Python* que se utiliza para manejar errores. Si no se da ningún error, se devuelven los datos extraídos. Si se dan errores, en las columnas afectadas se devuelve la palabra "Error". Esto permite que el *script* vaya procesando URLs sin detenerse, aunque se encuentre con uno o más errores.

Llegados a este punto es necesario trabajar con el *DataFrame*. *df["..."]* es un *DataFrame* y su contenido, el nombre de sus columnas. En el ejemplo que estamos comentando: *Title, Word Count* y *Schema.org Used*. Para cada fila del *DataFrame* se toma la columna URL y se llama a la función *get_page_data()* convirtiendo la tupla devuelta (*pd.Series()*) con los tres datos (título, número de palabras y valor Yes o No según si usa o no *Schema.org*) en una nueva fila del *DataFrame*. El código se muestra en la figura 4.

```
df[["Title", "Word Count", "Schema.org Used"]] = df["URL"].apply(lambda url: pd.Series(get_page_data(url)))
```

Figura 4. Código para la creación del *DataFrame* con *Pandas*.

En el código anterior *lambda* es una forma de definir funciones anónimas (sin nombre) en una sola línea. Se usa cuando se precisa una función corta y simple sin la necesidad de definirla con *def* (funciones con nombre que se reutilizan en varias partes del código).

Finalmente, tan sólo falta guardar el resultado en un archivo CSV con *to_csv()* (figura 5).

```
df.to_csv("results.csv", index=False, encoding="utf-8")
```

Figura 5. Exportación del *DataFrame* a un fichero CSV.

to_csv() es un método de los *DataFrame* de *Pandas* que permite guardar los datos en un fichero CSV. Como parámetros, pasamos el nombre del fichero ("datos.csv"), indicamos que no queremos guardar la columna índice del dataframe (una columna con un valor autonumérico para cada fila) y el formato de codificación de caracteres (utf-8).
Para ejecutar el código, escribimos el comando *python* seguido del nombre del fichero que contiene el script:
python mi_script.py
El código completo anterior se muestra en la figura 6.

5.3. Métodos para sortear mecanismos de protección

En ocasiones, algunos sitios web implementan mecanismos de protección para evitar el raspado de datos o algunos tipos de ataques que también pueden interceptar las técnicas que abordamos en este trabajo. Actualmente, se dispone de algunos métodos complementarios que permiten sortearlos. A continuación, se muestran algunos ejemplos de menor a mayor complejidad.
Una opción simple consiste en agregar una pausa antes de hacer la petición. Para ello, es posible utilizar *time.sleep()*, una función de *Python* que pausa la ejecución del

programa durante un número específico de segundos. Por ejemplo, *time.sleep(5)*, espera 5 segundos antes de cada petición.

```
import pandas as pd
import requests
from bs4 import BeautifulSoup

df = pd.read_csv("data.csv")

def get_page_data(url):
    try:
        response = requests.get(url, timeout=5, headers={'User-Agent': 'Mozilla/5.0'})
        soup = BeautifulSoup(response.text, "html.parser")

        title = soup.title.string.strip() if soup.title else "No Title"

        words = soup.get_text().split()
        word_count = len(words)

        schema_used = "Yes" if soup.find_all("script", {"type": "application/ld+json"}) else "No"

        return title, word_count, schema_used

    except Exception as e:
        return "Error", "Error", "Error"

df[["Title", "Word Count", "Schema.org Used"]] = df["URL"].apply(lambda url: pd.Series(get_page_data(url)))

df.to_csv("results.csv", index=False, encoding="utf-8")
```

Figura 6. Código de ejemplo para raspar los títulos, calcular la extensión del texto disponible en toda la página y valorar la presencia de *scripts* de tipo *ld+json*.

Como algunas páginas web sólo muestran su contenido completo tras ejecutar su código *JavaScript*, la biblioteca *requests_html*, a diferencia de la utilizada anteriormente (*requests*), permite no sólo hacer peticiones HTTP, sino que también habilita la renderización de páginas que usan JavaScript gracias a un mini navegador basado en la tecnología de *Chromium*[19].

Como tercera opción ante sitios que restringen el raspado, cabe la posibilidad de utilizar *Selenium*[20], una biblioteca para controlar navegadores desde *Python*, con un navegador como Chrome en modo *headless* (en segundo plano). Esto permite interactuar con las páginas simulando que el proceso lo está llevando a cabo un humano. El uso de *Selenium* implica instalar esta biblioteca e instalar también, por ejemplo, *ChromeDriver*, que es el controlador que permite a Selenium automatizar navegadores basados en *Chromium*.

6. Disponibilidad del código

El código que se ha mostrado como ejemplo, así como otros casos de uso más complejos y ejemplos en los que se aplican algunas de estas últimas aproximaciones para sortear mecanismos de protección se recoge en *Figshare*[18].

7. Notas

[1] *https://jsoup.org*
[2] *https://jaunt-api.com*
[3] *https://stormcrawler.apache.org*

[4] *https://cheerio.js.org*
[5] *https://apify.com*
[6] *https://www.crummy.com/software/BeautifulSoup*
[7] *https://lxml.de*
[8] *https://www.parsehub.com*
[9] *https://www.octoparse.com*
[10] *https://www.webharvy.com*
[11] *https://dataminer.io*
[12] *https://agenty.com*
[13] *https://www.screamingfrog.co.uk/seo-spider*
[14] *DDoS* (*Distributed Denial-of-Service*) en español, ataque distribuido de denegación de servicio, es un tipo de ataque informático malintencionado consistente en un intento explícito de impedir el uso legítimo de un servicio disponible en Internet. En este tipo de ataques se despliegan múltiples máquinas encargadas de enviar un flujo de paquetes a la víctima, sobrecargando su infraestructura.
[15] El DOM es un estándar del *W3C* que define como acceder programáticamente a un documento HTML. Define: los elementos HTML como objetos, las propiedades de todos los elementos HTML, los métodos para acceder a todos los elementos HTML, los eventos asociados a todos los elementos HTML. Es, en definitiva, una forma de acceder, modificar, añadir o eliminar elementos HTML y su contenido asociado.
[16] *https://pandas.pydata.org*
[17] *https://pypi.org/project/requests*
[18] *https://doi.org/10.6084/m9.figshare.28614665.v1*
[19] *https://www.chromium.org*
[20] *https://www.selenium.dev*

8. Referencias

Alcaraz-Martínez, Rubén (2023) "Black hat SEO y otras técnicas poco éticas: evolución y situación actual". *Infonomy*, v. 1, n. 1.
https://doi.org/10.3145/infonomy.23.008

Cass, Stephen (2024). The top programming languages 2024. *IEEE Spectrum*.
https://spectrum.ieee.org/top-programming-languages-2024

Diouf, Rabiyatou; **Sarr, Edouard**; **Sall, Ousmane**; **Birregah, Babiga**; **Bousso, Mamadou**; **Mbaye, Sény Ndiaye** (2019). Web scraping: state-of-the-art and areas of application. In: *IEEE International Conference on Big Data (Big Data)*, pp. 6040-6042.
https://doi.org/10.1109/BigData47090.2019.9005594

GitHub Staff (2024). Octoverse: AI leads Python to top language as the number of global developers surges. *Octoverse*.
https://github.blog/news-insights/octoverse/octoverse-2024

Grasso, Giovanni; **Furche, Tim**; **Schallhart, Christian** (2013). "Effective web scraping with OXPath". In: *Proceedings of the 22nd International Conference on World Wide Web*, pp. 23-26.
https://doi.org/10.1145/2487788.2487796

Khder, Moaiad-Admad (2021). Web scraping or web crawling: state of art, techniques, approaches and application. *International journal of advances in soft computing & its applications*, v. 13, n. 3, pp. 144-168.
http://dx.doi.org/10.15849/IJASCA.211128.11

Krotov, Vlad; **Johnson, Leigh**; **Silva, Leiser** (2020). Tutorial: legality and ethics of web scraping. *Communications of the Association for Information Systems*, n. 47.
https://doi.org/10.17705/1CAIS.04724

Lawson, Richard (2015). *Web scraping with Python: scrape data from any website with the power of Python*. Packt Publishing.

Maheshwari, Manish; **Ali, Roohi** (2013). Evolution of search engine optimization and investigating the effect of Panda update into it. *International journal of scientific & engineering research*, v. 4, n. 12, pp. 2045-2053.

Mitchell, Ryan (2024). *Web scraping with Python: collecting more data from the modern web*. O'Reilly.

Sarr, Edouard-Ngor; **Sall, Ousmane**; **Diallo, Aminata** (2018). FactExtract: automatic collection and aggregation of articles and journalistic factual claims from online newspaper. In: *Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pp. 336-341.
http://dx.doi.org/10.1109/SNAMS.2018.8554421

Sirisuriya, D. S. (2015). A comparative study on web scraping. In: *Proceedings of 8th International Research Conference*, pp. 135-140.
http://ir.kdu.ac.lk/handle/345/1051

Thomas, David-Mathew; **Mathur, Sandeep** (2019). Data analysis by web scraping using Python. In: *3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 450-454.
https://doi.org/10.1109/ICECA.2019.8822022

Vasilev, Ivan; **Slater, Daniel**; **Spacagna, Gianmario**; **Roelants, Peter**; **Zocca, Valentino** (2019). *Python deep learning: exploring deep learning techniques and neural network architectures with PyTorch, Keras and TensorFlow*. Packt Publishing.

Vording, Robbin (2021). Harvesting unstructured data in heterogenous business environments; exploring modern web scraping technologies.
https://purl.utwente.nl/essays/85663