Web scraping

PID_00256970

Laia Subirats Maté
Mireia Calvo González

Tiempo mínimo de dedicación recomendado: 5 horas

Universitat Oberta de Catalunya

Laia Subirats Maté

Ingeniera de Telecomunicaciones por la Universidad Pompeu Fabra (2008), máster en Telemática por la Universidad Politécnica de Cataluña (2009) y doctora en Informática por la Universidad Autónoma de Barcelona (2015). Desde 2009, trabaja como investigadora en Eurecat (Centro Tecnológico de Cataluña) aplicando la ciencia de datos a distintas áreas como son la salud, el medio ambiente o la educación. Desde 2016, colabora con la UOC como docente en el máster de Data Science y en el grado de Informática. Es especialista en inteligencia artificial, ciencia de datos, *eHealth* y representación del conocimiento.

Mireia Calvo González

Ingeniera de Telecomunicaciones por la Universidad Politécnica de Cataluña (2011), máster en Ingeniería Biomédica por la Universidad de Barcelona y la Universidad Politécnica de Cataluña (2014) y doctora en Procesamiento de Señales y Telecomunicaciones por la Universidad de Rennes 1 y en Ingeniería Biomédica por la Universidad Politécnica de Cataluña (2017). Desde 2012, ha trabajado como investigadora en diferentes entornos académicos, clínicos e industriales, aplicando el procesamiento de datos al estudio de diferentes enfermedades cardíacas y respiratorias. Desde 2017, colabora con la UOC como docente en el máster de Data Science.

La revisión de este recurso de aprendizaje UOC ha sido coordinada por la profesora: Isabel Guitart Hormigo (2019)

Índice

Introducción

Internet es actualmente el mayor repositorio de datos, accesibles en su mayor parte de forma gratuita, jamás recopilado. Con la aparición de la web 2.0, cuya filosofía se basa en la interoperabilidad y la colaboración en red, los usuarios pasaron a formar parte activa de dicha red, no solo utilizando internet como una herramienta de búsqueda de información, sino también como un medio para comunicarse y generar contenido y conocimiento.

Así, gracias a gran cantidad de iniciativas que promueven la compartición de datos de valor, generados tanto en entornos públicos como privados, la World Wide Web se ha convertido en una fuente inagotable de información. En el contexto de la investigación, por ejemplo, las principales revistas científicas de acceso abierto sugieren, y en muchos casos exigen, la compartición de aquellos datos utilizados en los estudios que en ellas se publican. De este modo, se promueve la inclusión de conocimiento de alta calidad en internet, estimulando la investigación colaborativa y, por tanto, fomentando el progreso hacia la búsqueda de soluciones a los problemas del momento.

Aunque en algunos casos es posible recuperar información de forma estructurada, en múltiples formatos como Comma Separated Values (CSV), JavaScript Object Notation (JSON), Extensive Markup Language (XML), Resource Description Framework (RDF), Excel Microsoft Office Open XML Format Spreadsheet (XLSX) o Extensible Stylesheet Language (XSL), a través de interfaces de programación de aplicaciones o API, la mayor parte del conocimiento en internet se encuentra integrado en la estructura y estilo de las diferentes páginas web. Es en estos casos donde la extracción de información puede convertirse en una tarea compleja y tediosa si no se manejan adecuadamente las herramientas de software actualmente disponibles que permiten simplificar y automatizar el proceso.

El ***web scraping***, que, traducido literalmente del inglés, se refiere al raspado web, permite obtener aquella información útil para un proyecto de datos que se encuentra disponible en internet.

Por ejemplo, cuando necesitamos analizar la competencia para así definir estrategias en nuestro negocio, puede ser interesante recuperar toda aquella información relacionada con productos o precios que se encuentre en su página web. Asimismo, podemos recuperar datos para hacer un estudio que permita mejorar nuestro servicio o producto. Por ejemplo, en la página web de un hospital donde se registre la lista de espera en urgencias en tiempo real, recuperar esta información de forma periódica nos puede ayudar a detectar aquellas horas más concurridas que necesitan la incorporación de personal sanitario

de refuerzo. Es por ello que, con el incremento constante de la información disponible en internet, el *web scraping* se ha convertido en una herramienta con un potencial incalculable en el dominio de la ciencia de datos y, más concretamente, en las etapas de extracción de información útil.

No obstante, se trata de una técnica que presenta diversas complejidades. Aquellas páginas web que pretendan evitar el *web scraping* podrán aplicar algunos métodos que dificulten de forma significativa la extracción de información. Además, antes de recopilar datos de una página web concreta, será importante conocer las implicaciones legales, principalmente cuando los datos obtenidos se pretendan publicar posteriormente en internet.

Asimismo, la heterogeneidad del contenido que podemos encontrar en cualquier página web (texto, tablas, imágenes, vídeos, mapas, etc.) dificultará los procesos de automatización. Y esto será solo el principio, ya que tras la recolección de bases de datos que contengan información de interés será necesario guardar, analizar y mostrar los resultados obtenidos, de forma que proporcionen nuevo conocimiento y, por tanto, valor añadido.

Previamente a la aparición del *web scraping*, los datos debían recogerse manualmente de diferentes fuentes, de forma poco eficiente, no reproducible y propensa a errores. No obstante, la ciencia de datos ha ido incorporando cada vez más procesos automatizados para la recopilación y publicación de información en línea, a través del uso de herramientas de software expandidas principalmente para las etapas de análisis, como Python, aunque también se puede realizar *web scraping* en R, como detallan Munzert y otros (2014), con el objetivo de extender estas herramientas a las fases previas a la minería de datos.

Así, para abordar los aspectos fundamentales del *web scraping*, este material didáctico se divide en ocho secciones principales. El apartado 1 incluye algunas reflexiones sobre por qué y cómo debe aplicarse esta técnica. A continuación, se detallan los primeros pasos a realizar cuando se aplica *web scraping*, utilizando el lenguaje de programación Python. En el apartado 3, se aborda la extracción de información de contenido audiovisual, y en la siguiente sección, se revisan los formatos estandarizados más comúnmente utilizados en el almacenamiento de datos generados mediante *web scraping*, así como los principales repositorios públicos en los que poder compartir dichos datos.

En los apartados 5 y 6, se revisan diferentes medidas implementadas para dificultar las tareas de *web scraping*, así como algunos métodos que permiten resolver dichos obstáculos. A continuación, se enumeran los principales aspectos legales relacionados con la extracción de datos procedentes de internet, para en el siguiente apartado citar una serie de mejores prácticas y consejos que permitan implementar un buen uso del *web scraping*. Asimismo, con el

objetivo de resaltar los potenciales beneficios del uso de esta técnica, se nombran algunos ejemplos y casos de éxito en los que el *web scraping* ha permitido extraer información de gran interés.

Por último, tras un breve resumen de los contenidos más relevantes de este material didáctico, se proponen algunos ejercicios de autoevaluación, así como sus soluciones, con los que el lector puede revisar la asimilación de los principales conceptos que aquí se presentan.

Este material didáctico se acompaña de un repositorio Github donde se incluye el código de algunos de los ejemplos proporcionados. De este modo, el lector puede descargar, probar y modificar directamente cada uno de dichos ejemplos, sin la necesidad de copiarlos manualmente.

Enlace de interés

Podéis acceder al repositorio Github en este enlace: https://github.com/datalifecicleuoc.

Objetivos

En este material didáctico se proporcionan las herramientas fundamentales que permitirán asimilar los siguientes objetivos:

1. Conocer el significado y los potenciales beneficios del *web scraping*.
2. Ser capaz de evaluar la dificultad de realizar *web scraping* en un sitio web determinado.
3. Ser capaz de realizar *web scraping* simple, utilizando Python.
4. Ser capaz de extraer contenido audiovisual de un sitio web.
5. Ser capaz de almacenar los datos obtenidos de internet en un formato interoperable.
6. Conocer los principales repositorios de bases de datos creadas a partir de información extraída mediante *web scraping*.
7. Ser capaz de buscar bases de datos disponibles para un dominio de aplicación determinado.
8. Ser capaz de solucionar los principales obstáculos implementados para evitar el *web scraping*.
9. Conocer los principales aspectos legales relacionados con el *web scraping*.
10. Conocer diferentes casos de éxito o usos prácticos del *web scraping*.

1. ¿Por qué y cómo realizar *web scraping*?

A continuación, abordaremos el porqué y el cómo realizamos *web scraping*.

1.1. ¿Por qué realizamos *web scraping*?

Lo primero que nos preguntamos entonces, cuando nos encontramos con el concepto de *web scraping*, es: ¿por qué lo realizamos?

Cuando navegamos por internet, a menudo encontramos contenido de nuestro interés que nos gustaría recuperar. Así, nos puede interesar recopilar, almacenar y analizar:

- Una lista de críticas de un sitio web sobre libros, series o películas, con el objetivo de crear un motor de recomendación, o construir un modelo predictivo que detecte aquellas críticas falsas.

- Características adicionales que enriquezcan, con información disponible en línea, una base de datos determinada. Por ejemplo, se puede añadir información meteorológica en un conjunto de datos diseñado para predecir la venta de refrescos.

- Noticias, de forma periódica, para conocer las últimas tendencias sobre un tema de interés particular.

Algunas páginas web ofrecen la posibilidad de acceder y descargar información de forma estructurada, a través de sus API (*application programming interfaces*, en inglés). Twitter, Facebook, LinkedIn y Google, por ejemplo, proporcionan este tipo de herramientas. No obstante, a pesar de que lo más conveniente es utilizar estas API cuando sea posible, no toda la información disponible en internet puede descargarse a través de las mismas.

Cuando el propietario de una página web no pone a disposición de sus usuarios herramientas de software que posibiliten la descarga de datos, ya sea por voluntad propia o por falta de recursos, el *web scraping* aparece como una alternativa para la extracción de información. Asimismo, cuando se descargan datos de diferentes páginas web que cuentan con API que no permiten la integración cohesiva de los datos, o cuando el volumen y velocidad requeridos en la descarga, o los tipos de datos y formatos que proporciona la API son insuficientes para nuestro propósito, la extracción de datos mediante *web scraping* se convierte en una necesidad.

Por lo tanto, por regla general, buscaremos en primer lugar si existe una API que nos permita extraer la información deseada. Si nuestro objetivo es, por ejemplo, recuperar una lista de tuits más recientes, dicha información puede descargarse fácilmente mediante la API de Twitter. No obstante, se pueden dar diversas situaciones en las que la extracción de datos mediante *web scraping* puede ser interesante, incluso cuando ya existe una API para tal efecto:

- Cuando existe una API, pero esta no es gratuita; mientras que el acceso a la página web sí lo es.
- Cuando la API limita el número de accesos por segundo, por día, etc.
- Cuando la API no permite recuperar toda aquella información de interés que se muestra en la página web.

Bibliografía recomendada

CEBIT d!talk, 12.06.2018, Keynote «Data Is The New Oil - The Internet revolution already started years ago and it isn't over yet», Mikko Hyppönen, Chief Research Officer, F-Secure. URL: https://www.youtube.com/watch?v=HE0RuDUy9JM.

Así, en un mundo donde, según Mikko Hypponen, «los datos son el nuevo petróleo», el *web scraping* juega un papel relevante al abrir la puerta a datos prácticamente ilimitados, que pueden ser de gran utilidad en cualquier disciplina. El acceso a información disponible en internet ha abierto nuevas fronteras para la creación de conocimiento, permitiendo realizar pronósticos de mercado en cualquier sector económico como el ocio, la restauración, la automoción, etc.; mejorar el diagnóstico médico a partir de información procedente de foros de salud; e incluso ha revolucionado el mercado del arte. En 2006, Jonathan Harris y Sep Kamvar lanzaron el proyecto «We Feel Fine» (nos sentimos bien), durante el cual crearon, a partir de la información compartida en una extensa gama de blogs, una base de datos formada con frases empezando por «I feel/I am feeling» (me siento/me estoy sintiendo). Este experimento dio lugar a una visualización de la información que se hizo muy popular, al describir cómo el mundo se sentía día tras día, minuto tras minuto. Así, independientemente del campo de interés, el *web scraping* cuenta con un potencial incalculable que, adecuadamente aplicado, puede incrementar la eficiencia de cualquier investigación o negocio.

Enlace de interés

Puede ser de vuestro interés la página web del proyecto «We Feel Fine» por Jonathan Harris y Sep Kamvar http://www.wefeelfine.org.

Además, el uso de *web scraping* aporta el valor añadido de ser fácilmente automatizable. Cuando detectamos información útil para nuestro proyecto que se encuentra disponible en internet, la recolección automática, o semiautomática, puede ser altamente recomendable si:

- Planeamos repetir la tarea en el futuro, por ejemplo, para mantener actualizada la base de datos.
- Nos interesa que terceros sean capaces de replicar nuestro proceso de recolección de datos.
- Trabajamos a menudo con datos cuyo origen se encuentra en internet.
- La extracción de los datos de interés presenta cierta complejidad.

Así, el ***web scraping*** consiste en la construcción de un agente que permita descargar, analizar y organizar datos procedentes de internet, de forma automática. Gracias al uso de esta técnica, podemos diseñar un *script* que desarrolle una serie de tareas repetitivas con las que almacenar información de interés de forma estructurada y mucho más eficiente que si se realizara manualmente, acelerando el proceso y evitando errores producidos en el proceso de copiar/pegar.

En resumen, según Brody (2017), la información procedente de internet puede recuperarse, principalmente, mediante los siguientes métodos:

Bibliografía recomendada

H. Brody (2017). *The ultimate guide to web scraping*. LeanPub.

- **API**. Idealmente, se trataría del mejor método para obtener información procedente de una página web, ya que el proveedor generalmente se compromete a ofrecer los datos en un formato estándar y bien documentado. No obstante, no es habitual que los propietarios de dichas páginas dediquen recursos a crear API proporcionando datos a terceros, ya que esta tarea no suele encontrarse entre sus prioridades.

- **Web pública**. A pesar de que el *web scraping* puede ser costoso en algunos casos, permite extraer información dinámica almacenada en cualquier página web.

- **Aplicación móvil**. Dado que extraer información de una aplicación móvil suele ser más complejo que realizarlo de una página web, en aquellos casos en los que la información se encuentre en ambas plataformas, se priorizará la recuperación de los datos de la página web.

- **RSS (*rich site summary*) o *atom feeds***. Se utilizan, principalmente, con el objetivo de recibir actualizaciones de información. Son formatos bien definidos y se utilizan a menudo en blogs, servicios de suscripción, resultados de búsquedas y otras instancias en las que un conjunto de información puede actualizarse con frecuencia. Aunque no es la mejor manera de analizar todos los datos contenidos en un sitio web, permiten analizar la información más reciente publicada en el mismo, por lo que puede ser muy recomendable analizar estos *feeds* antes de realizar *web scraping*.

- **Exportación de datos en archivos**. Algunas páginas web permiten descargar la información directamente en formatos estructurados como CSV, XLSX u otro tipo de hojas de cálculo. Aunque este método ofrece la ventaja de que permite exportar datos directamente de forma muy sencilla, la principal desventaja es que estos archivos contienen información extraída en un instante de tiempo concreto, por lo que son menos dinámicos que las páginas web y, por lo tanto, pueden estar desactualizados.

1.2. ¿Cómo realizamos *web scraping*?

Aunque el *web scraping* se puede realizar mediante diferentes lenguajes de programación, en este material didáctico nos centraremos en el uso de las librerías Python Requests y BeautifulSoup, diseñadas para la extracción de contenido web.

No obstante, existen otras herramientas ampliamente utilizadas, como Scrapy (Kouzis-Loukas, 2016), que no van a ser tratadas en este material didáctico.

Bibliografía recomendada

D. Kouzis-Loukas (2016). *Learning Scrapy*. Packt Publishing.

1.2.1. Evaluación inicial

Independientemente del lenguaje utilizado, cualquier tipo de *web scraping* debe incorporar una fase previa centrada en la evaluación de los siguientes aspectos:

1) el archivo *robots.txt*,
2) el mapa del sitio web,
3) su tamaño,
4) la tecnología usada y
5) el propietario del mismo.

Bibliografía recomendada

R. Lawson (2015). *Web Scraping with Python*. Packt Publishing Ltd.

1) Archivo *robots.txt*. Es importante analizar el contenido de *robots.txt* ya que es en este archivo donde la mayor parte de páginas web indican las restricciones a tener en cuenta cuando se pretende rastrearlas. Aunque estas restricciones son solo una sugerencia y nunca una obligación, es recomendable tenerlas en cuenta, principalmente con el objetivo de reducir las posibilidades de ser bloqueados.

Enlace de interés

En el enlace siguiente se puede encontrar más información sobre este tipo de archivos: http://www.robotstxt.org.

A continuación, se muestran algunos ejemplos de restricciones o permisos definidos en un archivo *robots.txt*.

a) Exclusión de tres directorios

```
  User-agent: *
Disallow: /cgi-bin/
Disallow: /tmp/
Disallow: /~joe/
```

b) Exclusión de todos los robots

```
  User-agent: *
Disallow: /
```

c) Permiso de acceso completo a todos los robots

```
  User-agent: *
```

```
Disallow:
```

d) Permiso de acceso a un solo robot

```
  User-agent: Google
Disallow:
User-agent: *
Disallow: /
```

e) Exclusión de páginas concretas

```
  User-agent: *
Disallow: /~joe/junk.html
Disallow: /~joe/foo.html
Disallow: /~joe/bar.html
```

2) Mapa del sitio web. Examinar el mapa del sitio web (*sitemap*, en inglés) nos ayudará a localizar el contenido actualizado sin la necesidad de rastrear cada una de las páginas que lo componen. El mapa de un sitio web se describe mediante el formato Simplemaps XML y se compone, principalmente, por las siguientes etiquetas:

Enlace de interés

El formato Simplemaps XML se describe en detalle en el siguiente enlace: https://www.sitemaps.org/protocol.html.

- Etiqueta de apertura <urlset>, dentro de la que debe especificarse el espacio de nombre (estándar de protocolo).
- Cada URL debe especificarse entre las etiquetas <url> y </url>, como una etiqueta XML principal.
- Dentro de cada etiqueta primaria <url>, una entrada secundaria <loc> especifica la dirección URL.
- Etiqueta de cierre </ urlset>.

A continuación se muestra un ejemplo de *sitemap* donde se puede observar su estructura.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
<url>
<loc>http://www.example.com</loc>
<lastmod>2005-01-01</lastmod>
<changefreq>monthly</changefreq>
<priority>0.8</priority>
</url>
</urlset>
```

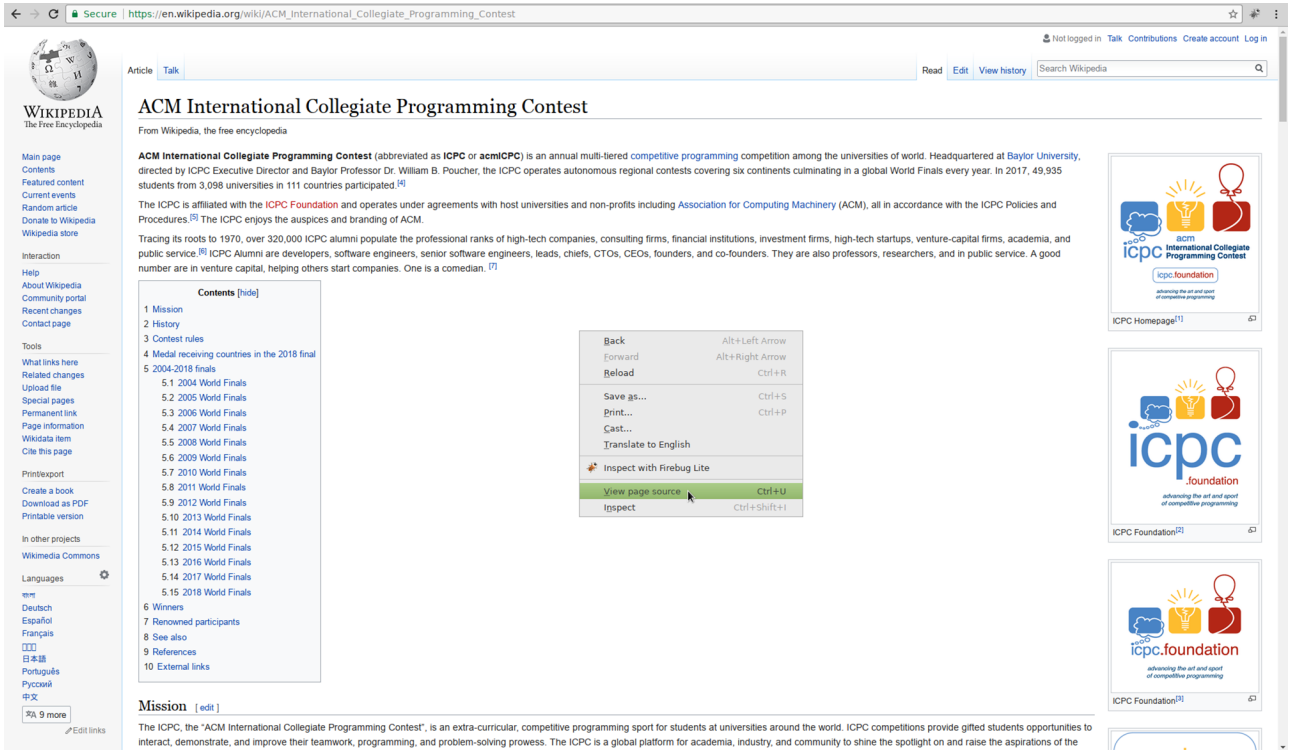
La principal ventaja de los *sitemaps* es que permiten a los motores de búsqueda, como Google o Bing, rastrear más fácilmente el sitio web. Aunque estos buscadores suelen indexar correctamente cualquier página web pequeña o mediana adecuadamente diseñada, rastrear aquellos sitios de mayor tamaño presentará ciertas complejidades, principalmente cuando se actualicen con frecuencia. Asimismo, el *sitemap* permite a los usuarios navegar más cómodamente por el sitio. Por ello, existen algunos generadores automáticos de *sitemaps*, como XML-Sitemaps.

Enlace de interés

Para más información sobre XML-Sitemaps, consultad el enlace siguiente: https://www.xml-sitemaps.com/.

Por otro lado, inspeccionar la estructura de una página web puede ser también de gran utilidad. Con este objetivo se puede clicar el botón derecho del ratón para posteriormente seleccionar la opción «View page source», tal y como se muestra en la figura 1.

Figura 1. Acceso al código fuente de la página web



Fuente: Wikipedia

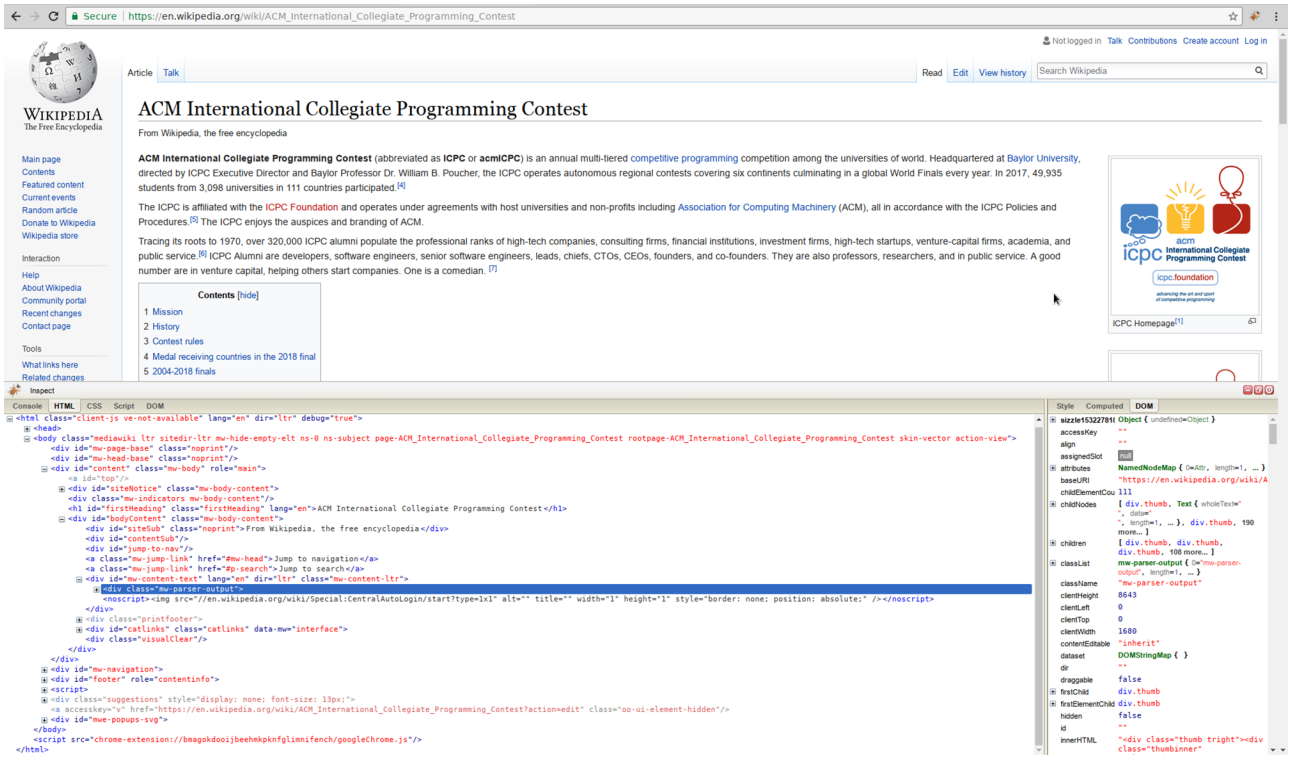
Asimismo, para inspeccionar más fácilmente el contenido de una página web, se puede añadir la extensión Firebug Lite al navegador. En la figura 2 se muestra un ejemplo donde se puede observar el contenido de una página procedente de la plataforma Wikipedia.

Según Acodemy (2015), aunque Wikipedia es una extensa fuente de información, su formato no siempre facilita la extracción de datos, por lo que analizar su estructura previamente a realizar *web scraping* acostumbra a ser muy recomendable.

Bibliografía recomendada

Acodemy (2015). *Learn Web Scraping With Python In A Day: The Ultimate Crash Course to Learning the Basics of Web Scraping With Python In No Time*. CreateSpace Independent Publishing Platform.

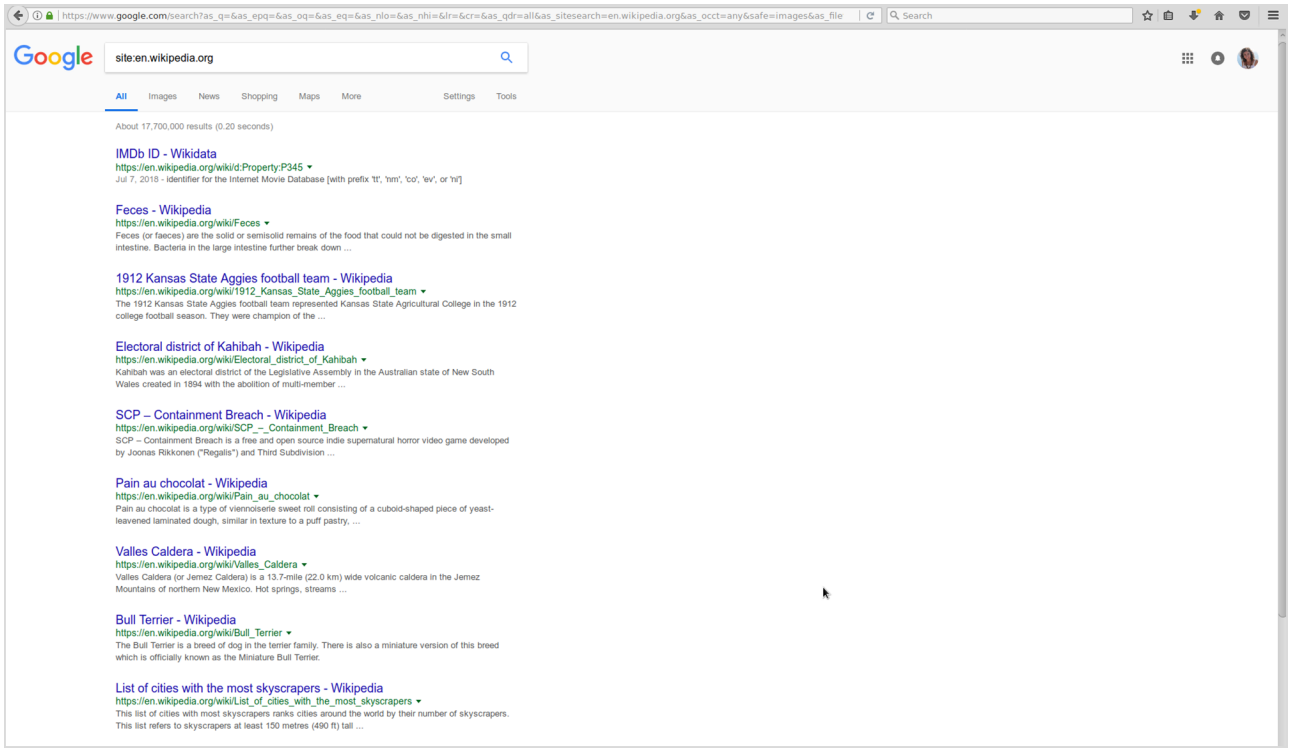
Figura 2. Análisis mediante Firebug Lite de la página web



Fuente: Wikipedia

3) Tamaño. La estimación del tamaño de una página web también afectará en la forma de realizar el rastreo. Cuando el sitio esté formado por solo un centenar de páginas, la eficiencia no será importante; pero cuando contenga más de un millón de direcciones, el uso de descargas concurrentes, en lugar de secuenciales, será relevante. Una manera rápida de verificar el tamaño de una página web se realiza mediante la búsqueda avanzada en Google, con la palabra clave `site`, del sitio web de interés. En la figura 3 se muestra el resultado de analizar el tamaño de Wikipedia, obteniendo alrededor de 17.700.000 enlaces.

Figura 3. Búsqueda avanzada del sitio Wikipedia en Google Advanced Search.



4) Tecnología. Del mismo modo, la tecnología utilizada en el diseño del sitio web condicionará el tipo de *web scraping* aplicado. Esta puede analizarse tras instalar la herramienta `builtwith`, mediante el comando `pip3 install builtwith` (o `pip install builtwith` en Python 2), para después ejecutar `builtwith.builtwith`, donde la función `builtwith` llama al sitio web que se pretende inspeccionar.

A continuación, se muestra el resultado obtenido para dicho ejemplo:

```
{'blogs': ['PHP', 'WordPress'],
'cms': ['WordPress'],
'ecommerce': ['WooCommerce'],
'font-scripts': ['Google Font API'],
'programming-languages': ['PHP'],
'web-servers': ['Nginx']}
```

5) Propietario. Finalmente, conocer el propietario de la página web que pretendemos rastrear puede ser interesante cuando, por ejemplo, este sea conocido por bloquear los procesos de *web scraping*. En ese caso, con el objetivo de evitar ser bloqueados, ajustaremos la descarga utilizando tasas más conservadoras.

Así, para conocer el propietario de la página web de interés, realizaremos las siguientes acciones:

```
pip3 install python-whois
```

```
import whois
print(whois.whois('https://www.wordpress.com'))
```

Donde la función `whois` llama al sitio web del cual se pretende conocer el propietario; por lo que para el mismo ejemplo de WordPress, el resultado obtenido se muestra a continuación:

```
#{
#"registrar": "MarkMonitor, Inc.",
#"city": null,
#"expiration_date": [
#"2020-03-03 12:13:23",
#"2020-03-03 04:13:23-08:00"
#],
#"domain_name": [
#"WORDPRESS.COM",
#"wordpress.com"
#],
#"dnssec": "unsigned",
#"name": null,
#"state": "CA",
#"status": [
#"clientDeleteProhibited https://icann.org/epp#clientDeleteProhibited",
#"clientTransferProhibited https://icann.org/epp#clientTransferProhibited",
#"clientUpdateProhibited https://icann.org/epp#clientUpdateProhibited",
#"serverDeleteProhibited https://icann.org/epp#serverDeleteProhibited",
#"serverTransferProhibited https://icann.org/epp#serverTransferProhibited",
#"serverUpdateProhibited https://icann.org/epp#serverUpdateProhibited",
#"clientUpdateProhibited (https://www.icann.org/epp#clientUpdateProhibited)",
#"clientTransferProhibited (https://www.icann.org/epp#clientTransferProhibited)",
#"clientDeleteProhibited (https://www.icann.org/epp#clientDeleteProhibited)",
#"serverUpdateProhibited (https://www.icann.org/epp#serverUpdateProhibited)",
#"serverTransferProhibited (https://www.icann.org/epp#serverTransferProhibited)",
#"serverDeleteProhibited (https://www.icann.org/epp#serverDeleteProhibited)"
#],
#"org": "Automattic, Inc.",
#"whois_server": "whois.markmonitor.com",
#"country": "US",
#"emails": [
#"abusecomplaints@markmonitor.com",
#"whoisrelay@markmonitor.com"
#],
#"zipcode": null,
#"creation_date": [
#"2000-03-03 12:13:23",
#"2000-03-03 04:13:23-08:00"
#],
```

```
#"name_servers": [
#"NS1.WORDPRESS.COM",
#"NS2.WORDPRESS.COM",
#"NS3.WORDPRESS.COM",
#"NS4.WORDPRESS.COM",
#"ns2.wordpress.com",
#"ns3.wordpress.com",
#"ns4.wordpress.com",
#"ns1.wordpress.com"
#],
#"updated_date": [
#"2017-01-12 22:53:10",
#"2017-01-13 14:26:51-08:00"
#],
#"referral_url": null,
#"address": null
#}
```

En el apartado «Resolución de obstáculos en *web scraping*» se puede encontrar más información sobre cómo resolver los principales obstáculos en *web scraping*.

1.2.2. Principales retos del *web scraping*

Aunque el *web scraping* se presenta como una herramienta con un enorme potencial al permitir el acceso a datos prácticamente ilimitados, como comenta O. Bosch (2017) en un documento oficial de la Comisión Europea, con este se presentan importantes retos que debemos tener presentes a la hora de rastrear datos de cualquier página web.

Bibliografía recomendada

O. Bosch (2017). *An introduction to web scraping, IT and Legal aspects*. <https://bit.ly/2pMUYKC>

Por un lado, será relevante determinar qué datos se desean extraer al realizar *web scraping*. Así, aunque diversos sitios web ofrecerán información de nuestro interés, deberemos analizar cuáles de estos sitios son los más adecuados en función de nuestras necesidades. Asimismo, deberemos identificar aquellas fuentes de información que cuenten con la última versión de los datos. También se debe analizar la calidad de los mismos; punto que se tratará en el apartado 8, donde se enumeran las variables a tener en cuenta a la hora de determinar la calidad de la información. En ocasiones, será necesario escoger entre el propietario de los datos y un agregador de contenidos, por lo que será interesante explorar los flujos entre sitios web.

Por otro lado, es importante recordar que internet es dinámico. A pesar de que cada sitio web cuenta con una estructura predeterminada, su contenido cambia constantemente (al hacer uso, por ejemplo, del *scroll* infinito). Por lo tanto, los *web scrapers* que implementemos deberán ser tan robustos como sea posible, teniendo en cuenta que internet es volátil y, del mismo modo que

algunas páginas pueden dejar de existir con el paso del tiempo, otras nuevas aparecerán. Así, lo más recomendable será monitorizar constantemente la información de interés.

Otro punto fundamental a tener en cuenta es el de los aspectos legales relacionados con los propietarios de los datos que pretendemos rastrear, teniendo en cuenta que la legislación puede ser específica para cada país.

Finalmente, el hecho de organizar los datos extraídos mediante *web scraping* de forma estándar puede suponer otro punto diferenciador.

2. Primeros pasos para realizar *web scraping*

Una idea clave a la hora de realizar *web scraping* es que, con el objetivo de planificar e implementar una descarga óptima de la información contenida en una página web determinada, es necesario entender cómo funciona el navegador mediante el cual se accede al contenido de dicha página.

Por ello, en este apartado, se trata en primer lugar el funcionamiento del navegador web. Posteriormente, se detalla cómo se debe descargar la página web que se desea rastrear, así como la estructura anidada que de este proceso se obtiene. Finalmente, con el objetivo de identificar la información de interés, se presentan diversas operaciones útiles a la hora de navegar por dicha estructura anidada, disponibles mediante el uso de la librería BeautifulSoup.

2.1. Funcionamiento del navegador web

Navegamos por la web todos los días, cada vez que accedemos a nuestro correo electrónico y redes sociales, consultamos las últimas noticias, realizamos alguna compra en internet, buscamos información en Wikipedia o tutoriales en YouTube sobre un tema de interés, etc. Pero ¿cómo funciona realmente el proceso en el que se accede a una página web desde nuestro navegador? Principalmente, este proceso puede resumirse en tres pasos:

1) envío de peticiones HTTP,
2) recepción de peticiones HTTP,
3) conversión de página web objetivo en estructura anidada.

En los siguientes apartados, se explican brevemente cada uno de estos pasos fundamentales a la hora de realizar *web scraping*.

2.1.1. Envío de peticiones HTTP

Como su nombre indica, el Hypertext Transfer Protocol (HTTP) es un protocolo de comunicación que permite la transferencia de información mediante documentos de tipo hipertexto, esto es, a través de internet. Así, cuando se desea acceder a una página web diseñada en lenguaje HTML (HyperText Markup Language) a través del navegador, se realiza una petición HTTP.

El siguiente fragmento de código muestra una petición HTTP realizada con el objetivo de acceder a la página web de Wikipedia:

```
GET page www.wikipedia.org HTTP/2.0
```

La figura 4 muestra la información transmitida en las cabeceras de una petición HTTP, entre las que se destacan las más relevantes:

- ***Connection***. Especifica el tipo de conexión con el servidor HTTP. Normalmente, como se muestra en el ejemplo, su valor es `keep-alive`.
- ***Accept***. Hace referencia al tipo de contenidos o ficheros aceptados como respuesta; generalmente, `text/html`.
- ***User-agent***. Contiene información sobre la petición, esto es, sobre el navegador utilizado, el sistema operativo, etc.
- ***Accept-encoding***. Especifica el tipo de codificaciones (*encodings*, en inglés) admitidas (el servidor puede comprimir la respuesta).
- ***Accept-language***. El servidor indica los idiomas aceptados.
- ***Cookie***. Otro concepto importante son las *cookies*, ya que permiten establecer preferencias que persisten a lo largo de diferentes páginas web.

Figura 4. Cabeceras de una petición HTTP

```
Accept: text/html,application/xhtml+xm...plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: ca,en-US;q=0.7,en;q=0.3
Cache-Control: max-age=0
Connection: keep-alive
Cookie: WMF-Last-Access-Global=26-Jul-...; WMF-Last-Access=26-Jul-2018
Host: www.wikipedia.org
If-Modified-Since: Mon, 23 Jul 2018 10:07:54 GMT
If-None-Match: W/"12742-571a7d1a715ba"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; W...) Gecko/20100101 Firefox/60.0
```

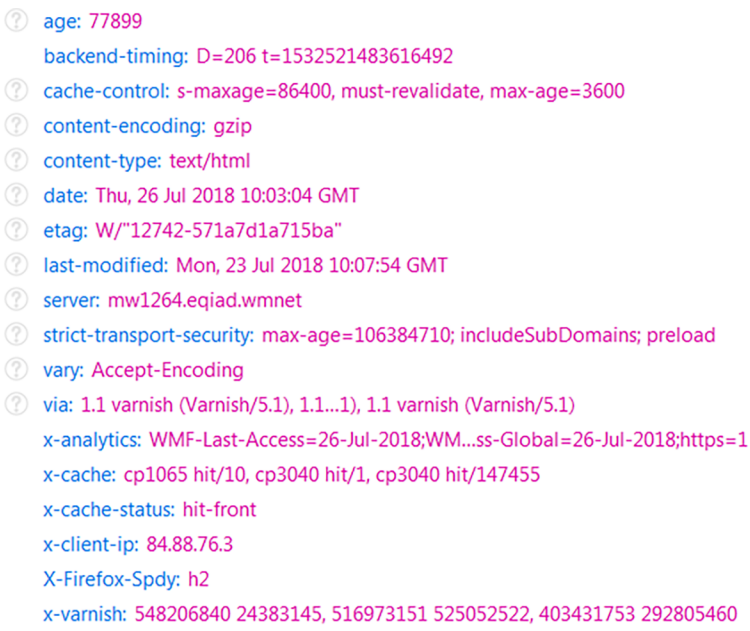
Enlace de interés

La siguiente entrada en Wikipedia incluye una lista completa de cabeceras HTTP: https://bit.ly/2PEmPIj.

2.1.2. Envío de respuestas HTTP

Una vez realizada la petición HTTP por parte del navegador, el servidor envía una respuesta de tipo HTTP/2.0 200 OK, incluyendo cabeceras HTTP de respuesta, así como un documento HTML. La figura 5 muestra un ejemplo de cabeceras de respuesta.

Figura 5. Cabeceras de respuesta HTTP



En el caso de la figura 5, el código de estado enviado por el servidor resultó ser un `200 OK`, esto es, el servidor envió una respuesta estándar para peticiones correctas. No obstante, existen muchos otros códigos de estado, detallados en Wikipedia, que pueden clasificarse en cuatro grandes grupos:

- **2XX.** Peticiones exitosas. Esta clase de código de estado indica que la petición fue recibida correctamente, entendida y aceptada. Otro ejemplo de este tipo es el `201 Created`, que indica que la petición ha sido completada y ha resultado en la creación de un nuevo recurso.

- **3XX.** Redirecciones. En este caso, el cliente debe tomar una acción adicional para completar la petición. Un ejemplo de este tipo es el código `300 Multiple Choices`, el cual indica opciones múltiples que el cliente debe seleccionar: presentando distintas opciones de formato para la visualización de vídeos, listando archivos con distintas extensiones, etc.

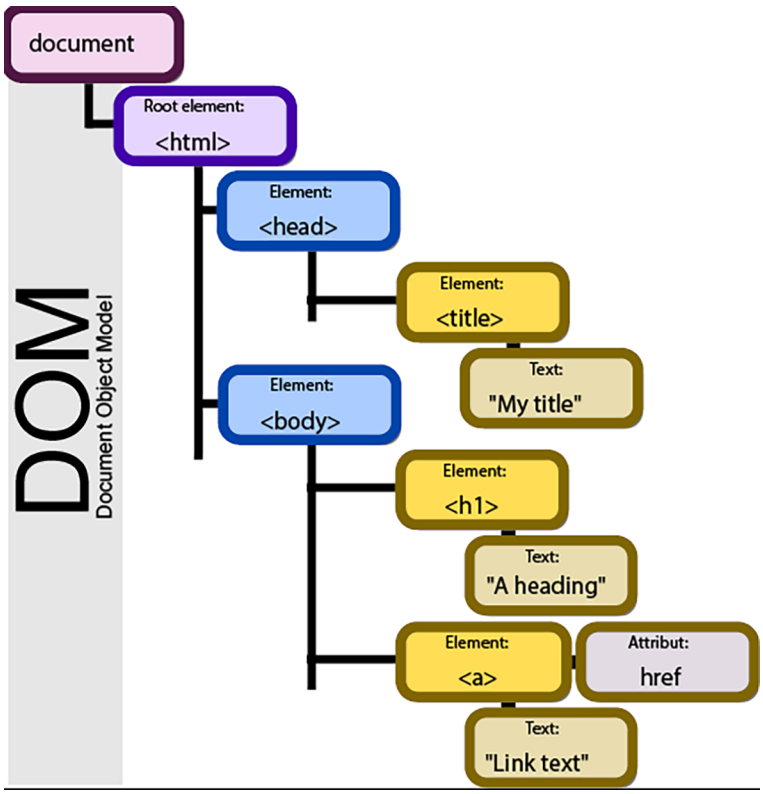
- **4XX.** Errores del cliente. La solicitud contiene una sintaxis incorrecta o no puede procesarse. Así, el código `404 Not Found` hace referencia a un recurso no encontrado y se utiliza cuando el servidor web no es capaz de encontrar la página o recurso solicitados.

- **5XX.** Errores del servidor, al completar una solicitud aparentemente válida. El código `500 Internal Server Error` es comúnmente emitido por aplicaciones empotradas en servidores web que generan contenido dinámicamente; por ejemplo, aplicaciones montadas en Tomcat, cuando se encuentran con situaciones de error ajenas a la naturaleza del servidor web.

2.1.3. Conversión de HTML a estructura anidada

Finalmente, el navegador parsea la página web objetivo para así construir una estructura anidada conocida como el *document object model* (DOM). Aunque este modelo puede ser muy complejo, establece la jerarquía anidada de cualquier sitio web. La figura 6 muestra, de forma esquemática, un ejemplo de DOM.

Figura 6. *Document object model*



Fuente:Birger Eriksson CC-BY-SA-3.0

2.2. Descarga de la página web

En *web scraping*, el primer paso a realizar es la descarga del sitio web de interés. Esto se puede realizar mediante las librerías Requests y BeautifulSoup.

En primer lugar, se debe instalar e importar la librería Requests mediante los siguientes comandos:

```
pip3 install requests
import requests
```

Posteriormente, el método `requests.get` permite recuperar la información correspondiente a la respuesta de la petición, donde `str` hace referencia a la página sobre la que queremos realizar *web scraping*.

```
page = requests.get(str)
```

Así, `page` será un objeto que, entre sus atributos más importantes, destacan:

- `page.status_code`: código HTTP devuelto por el servidor.
- `page.content`: contenido en bruto de la respuesta del servidor.

Por lo tanto, también, tras instalar e importar la librería BeautifulSoup, se debe parsear dicho contenido bruto, almacenando el resultado en un nuevo objeto, en este caso llamado `soup`:

```
pip3 install beautifulsoup4
from bs4 import BeautifulSoup
soup = BeautifulSoup(page.content)
```

A continuación, con el objetivo de obtener la estructura anidada que debemos analizar para identificar la información de interés, utilizaremos la función `prettify`. Así, si mostramos un ejemplo de uso de dicha función en un fragmento de *Alicia en el país de las maravillas*, la función `print` devuelve el siguiente resultado:

```
print(soup.prettify())
# <html>
# <head>
#  <title>#  The Dormouse's story
# </title>
# </head>
# <body>
#  <p class="title">
#  <b>
#   The Dormouse's story
#  </b>
#  </p>
#  <p class="story">
#  Once upon a time there were three little sisters; and their names were
#  <a class="sister" href="http://example.com/elsie" id="link1">
#   Elsie
#  </a>
#  ,
#  <a class="sister" href="http://example.com/lacie" id="link2">
#   Lacie
#  </a>
#  and
#  <a class="sister" href="http://example.com/tillie" id="link2">
#   Tillie
#  </a>
#  ; and they lived at the bottom of a well.
#  </p>
#  <p class="story">
```

```
#  ...
#  </p>
# </body>
# </html>
```

2.3. Tipos de objetos

En BeautifulSoup existen diversos tipos de objetos. Si tenemos en cuenta que BeautifulSoup transforma cualquier documento HTML en un árbol complejo de objetos, podemos decir que existen cuatro tipos de objetos fundamentales mediante los que se puede navegar cómodamente por la estructura anidada resultante: `Tag`, `NavigableString`, `BeautifulSoup` y `Comment`.

2.3.1. Tag

Este objeto corresponde a una etiqueta XML o HTML en el documento original. A continuación se muestra un ejemplo:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Cada tag se asocia a un nombre, accesible mediante `.name`:

```
tag.name
# u'b'
```

Asimismo, un tag puede contener un número indefinido de atributos. En el ejemplo anterior, el tag `<b id="boldest">` contiene un atributo «id» cuyo valor es «boldest». Se puede acceder a dichos atributos de la siguiente manera:

```
tag.attrs
# {u'id': 'boldest'}
```

O bien directamente:

```
tag['id']
# u'boldest'
```

Por otro lado, los atributos de un tag se pueden añadir, eliminar o modificar:

```
tag['id'] = 'verybold'
tag['another-attribute'] = 1
tag
# <b another-attribute="1" id="verybold"></b>
```

```
del tag['id']
del tag['another-attribute']
tag
# <b></b>

tag['id']
# KeyError: 'id'

print(tag.get('id'))
# None
```

Asimismo, algunos atributos pueden contener más de un valor, por lo que BeautifulSoup los representa como una lista. Es el caso de `class`, aunque otros ejemplos son `rel`, `rev`, `accept-charset`, `headers`, y `accesskey`.

```
css_soup = BeautifulSoup('<p class="body"></p>')
css_soup.p['class']
# ["body"]

css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ["body", "strikeout"]
```

2.3.2. NavigableString

En este caso, `NavigableString` corresponde a una cadena de caracteres dentro de un `tag`. La siguiente sentencia permite acceder a su contenido:

```
tag.string
# u'Extremely bold'

type(tag.string)
# <class 'bs4.element.NavigableString'>
```

Aunque este tipo de objetos no pueden modificarse, es posible reemplazar el contenido del string utilizando la función `replace_with`:

```
tag.string.replace_with("No longer bold")
tag
# <blockquote>No longer bold</blockquote>
```

2.3.3. BeautifulSoup

Este objeto representa al documento en su conjunto. Generalmente puede tratarse como un objeto de tipo `tag`, por lo que soporta el uso de la mayoría de operaciones que permiten navegar por la estructura anidada.

```
soup.name
# u'[document]'
```

2.3.4. Comment

Aunque los objetos anteriores cubren prácticamente la totalidad de la información contenida en un documento HTML o XML, puede ser necesario acceder a algunos datos de interés mediante el objeto `Comment`. Se trata de un tipo de `NavigableString`, que se muestra mediante un formato especial.

```
markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
soup = BeautifulSoup(markup)
print(soup.b.prettify())
# <b>
# <!--Hey, buddy. Want to buy a used parser?-->
# </b>

comment = soup.b.string
type(comment)
# <class 'bs4.element.Comment'>
```

2.4. Navegar por la estructura anidada

Retomemos el fragmento de *Alicia en el país de las maravillas*, con el objetivo de analizar algunos comandos de utilidad a la hora de navegar por el DOM resultante.

```
print(soup.prettify())
# <html>
# <head>
#  <title>#  The Dormouse's story
# </title>
# </head>
# <body>
#  <p class="title">
#  <b>
#   The Dormouse's story
#  </b>
#  </p>
#  <p class="story">
#  Once upon a time there were three little sisters; and their names were
#  <a class="sister" href="http://example.com/elsie" id="link1">
#   Elsie
#  </a>
#  ,
#  <a class="sister" href="http://example.com/lacie" id="link2">
#   Lacie
```

```
#  </a>
#  and
#  <a class="sister" href="http://example.com/tillie" id="link2">
#   Tillie
#  </a>
#  ; and they lived at the bottom of a well.
#  </p>
#  <p class="story">
#  ...
#  </p>
# </body>
# </html>
```

A continuación, se destacan los comandos más ampliamente utilizados a la hora de navegar por la estructura anidada:

1) `soup.title`

```
# <title>The Dormouse's story</title>
```

2) `soup.title.name`

```
# u'title'
```

3) `soup.title.string`

```
# u'The Dormouse's story'
```

4) `soup.title.parent.name`

```
# u'head'
```

5) `soup.p`

```
# <p class="title"><b>The Dormouse's story</b></p>
```

6) `soup.p['class']`

```
# u'title'
```

7) `soup.a`

```
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

8) `soup.find_all('a')`

```
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
```

```
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

9) `soup.find(id="link3")`

```
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

Asimismo, es habitual extraer todas las URL contenidas en un sitio web, asociadas a las etiquetas de tipo `<a>`, mediante la instrucción:

```
for link in soup.find_all('a'):
print(link.get('href'))
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

Finalmente, la siguiente función permite extraer la totalidad del texto contenido en una página web:

```
print(soup.get_text())
# The Dormouse's story
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
# ...
```

2.4.1. Análisis vertical

Los diferentes tags pueden contener cadenas de caracteres, así como otros tags o etiquetas. Estos elementos son lo que se conocen como los hijos (*children*, en inglés) de la etiqueta y BeautifulSoup proporciona gran cantidad de métodos que permiten navegar e iterar sobre los hijos de dicha etiqueta.

El método más sencillo para navegar por la estructura consiste en utilizar el nombre de la etiqueta de interés. Así, para acceder al `<head>` del documento HTML de interés, utilizaremos el comando `.head`:

```
head_tag = soup.head
head_tag
# <head><title>The Dormouse's story</title></head>
```

Del mismo modo, se puede ampliar la búsqueda encadenando etiquetas. En el siguiente ejemplo se muestra el primer tag `<b>` encontrado en el `<body>` del documento:

```
soup.body.b
# <b>The Dormouse's story</b>
```

Es importante destacar que este método solo devuelve el primer tag con dicho nombre, por lo que si se pretende recuperar todas las etiquetas de un mismo tipo, es necesario utilizar la función `find_all`.

Otro método para obtener los hijos de un tag consiste en utilizar la función `.contents`:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

title_tag = head_tag.contents[0]
title_tag
# <title>The Dormouse's story</title>

title_tag.contents
# [u'The Dormouse's story']
```

Asimismo, en lugar de obtener los hijos como una lista, se pueden extraer mediante el generador `.children`:

```
for child in title_tag.children:
print(child)
# The Dormouse's story
```

Tanto `.contents` como `.children` solo consideran los hijos directos de una etiqueta. Así, en el ejemplo anterior, `<head>` solo tiene como hijo directo la etiqueta `<title>`, pero esta última tiene a su vez un hijo, el string «The Dormouse's story», por lo que este string puede considerarse también descendencia de la etiqueta `<head>`. Si se quiere recuperar la totalidad de la descendencia de un tag, será necesario utilizar el método `.descendants`:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

for child in head_tag.descendants:
print(child)
# <title>The Dormouse's story</title>
# The Dormouse's story
```

De forma análoga, todo tag y string, a excepción del objeto BeautifulSoup, tiene un elemento padre (*parent*, en inglés). Así, por ejemplo, se puede acceder a la etiqueta `<head>` a partir de su hijo `<title>`, mediante la instrucción `.parent`:

```
title_tag = soup.title
title_tag
# <title>The Dormouse's story</title>

title_tag.parent
# <head><title>The Dormouse's story</title></head>
```

Finalmente, dado que `.parent` solo accede al elemento padre directo, la instrucción `.parents` permite iterar sobre toda la ascendencia. El siguiente ejemplo muestra cómo es posible navegar, mediante la función `.parents`, desde una etiqueta `<a>` contenida en un documento hasta la parte superior de dicho documento:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

for parent in link.parents:
if parent is None:
print(parent)
else:
print(parent.name)
# p
# body
# html
# [document]
# None
```

2.4.2. Análisis horizontal

Los elementos hermanos (*siblings*, en inglés) de un documento HTML son aquellos que se encuentran al mismo nivel, es decir, que son hijos de una misma etiqueta. Así, tomando el siguiente ejemplo sencillo:

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></b></a>")
print(sibling_soup.prettify())
# <html>
# <body>
#  <a>
#  <b>
#   text1
#  </b>
#  <c>
#   text2
#  </c>
#  </a>
# </body>
```

```
# </html>
```

Las etiquetas <b> y <c> se presentan como hermanas, al ser ambas elementos hijos de una etiqueta <a>.

En este caso, las funciones .next_sibling y .previous_sibling permiten navegar a través de los elementos de un sitio web que se encuentran al mismo nivel:

```
sibling_soup.b.next_sibling
# <c>text2</c>


sibling_soup.c.previous_sibling
# <b>text1</b>
```

Del mismo modo, .next_siblings y .previous_siblings permiten iterar sobre los diferentes hermanos de un tag. Así, retomando el ejemplo de *Alicia en el país de las maravillas*, a partir de un elemento <a> se puede obtener el resto:

```
for sibling in soup.a.next_siblings:
print(repr(sibling))
# u',\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u' and\n'
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
# u'; and they lived at the bottom of a well.'
# None



for sibling in soup.find(id="link3").previous_siblings:
print(repr(sibling))
# ' and\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u',\n'
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
# u'Once upon a time there were three little sisters; and their names were\n'
# None
```

2.5. Funciones principales

La siguiente lista enumera el conjunto de funciones de BeautifulSoup más utilizadas:

- **find_all**. Este método extrae todos los objetos Tag y NavigableString de la estructura analizada que coinciden con los criterios dados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findAll`.

- **find.** En este caso, la función solo devuelve el primer objeto que coincide con los criterios dados.

- **find_all_next**. Extrae todos los elementos siguientes a un objeto dado, que cumplen con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findAllNext`.

- **find_next**. Identifica el primer elemento siguiente a un objeto dado, que cumple con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findNext`.

- **find_all_previous**. Extrae todos los elementos previos a un objeto dado, que cumplen con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findAllPrevious`.

- **find_previous**. Identifica el primer elemento previo a un objeto dado, que cumple con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findPrevious`.

- **find_next_siblings**. Extrae todos los elementos hermanos siguientes de un objeto dado, que cumplen con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findNextSiblings`.

- **find_next_sibling**. Identifica el primer elemento hermano siguiente de un objeto dado, que cumple con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findNextSibling`.

- **find_previous_siblings**. Extrae todos los elementos hermanos previos de un objeto dado, que cumplen con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findPreviousSiblings`.

- **find_previous_sibling**. Identifica el primer elemento hermano previo de un objeto dado, que cumple con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findPreviousSibling`.

- **find_parents**. Extrae todos los elementos padre de un objeto dado, que cumplen con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findParents`.

Bibliografía recomendada

V. G. Nair (2014). *Getting started with BeautifulSoup*. Packt Publishing Ltd. Open Source Collaborative framework in Python. Disponible en: https://scrapy.org (accedido el 15 de marzo de 2018).

Enlace de interés

En el siguiente enlace se puede encontrar más información sobre el funcionamiento de la librería BeautifulSoup: https://bit.ly/2z01cvF>.

- **`find_parent`**. Identifica el primer elemento padre de un objeto dado, que cumple con los criterios especificados. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `findParent`.

- **`replace_with`**. Elimina un elemento y lo reemplaza por el tag o string proporcionado. Función equivalente (accesible en versiones anteriores de BeautifulSoup): `replaceWith`.

- **`wrap`**. Introduce un elemento en la etiqueta especificada.

- **`unwrap`**. Reemplaza una etiqueta con su contenido.

3. *Web scraping* de contenido gráfico y audiovisual

Además del texto procedente de un sitio web, puede ser de utilidad extraer y almacenar ciertas imágenes u otro contenido audiovisual. Para ello, se puede implementar un método que guarde dicho contenido predeterminado, dada su URL. El siguiente ejemplo almacena la imagen con URL `source_url` en la carpeta Pictures:

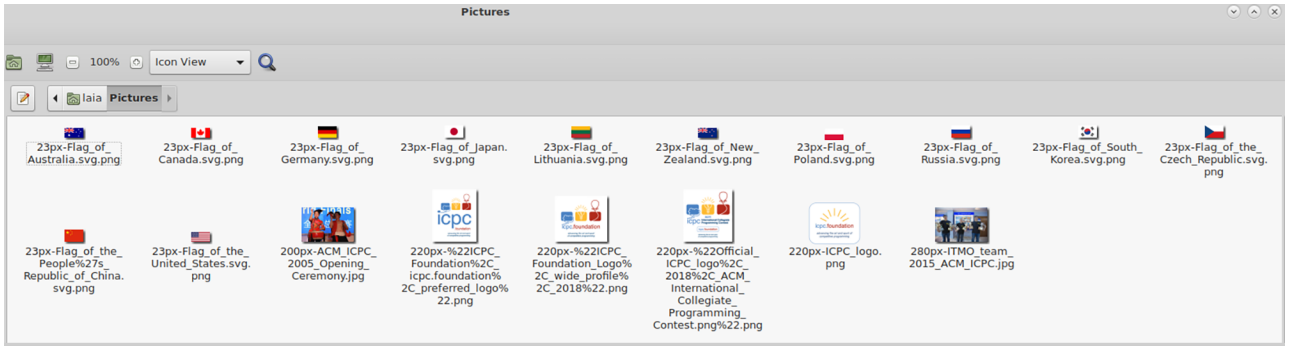
```
import requests
def load_requests(source_url):
r = requests.get(source_url, stream = True)
if r.status_code == 200:
aSplit = source_url.split('/')
ruta = "/home/user/Pictures/"+aSplit[len(aSplit)-1]
print(ruta)
output = open(ruta,"wb")
for chunk in r:
output.write(chunk)
output.close()
```

Por lo tanto, para realizar *web scraping* sobre las imágenes de un sitio web deberemos obtener sus URL para, posteriormente, almacenarlas en una carpeta, en este caso Pictures, mediante la función `load_requests` creada anteriormente.

El siguiente ejemplo muestra el resultado de extraer las imágenes contenidas en la página de Wikipedia destinada al ACM International Collegiate Programming, para posteriormente almacenarlas en la carpeta Pictures mostrada en la figura 7.

```
from bs4 import BeautifulSoup
import requests
url = 'https://en.wikipedia.org/wiki/ACM_International_Collegiate_Programming_Contest'
page = requests.get(url)
soup = BeautifulSoup(page.content)
images = []
i = 0
for img in soup.findAll('img'):
images.append(img.get('src'))
if ('static' not in images[i]):
load_requests("https:"+images[i])
i = i+1
```

Figura 7. Imágenes almacenadas en la carpeta Pictures



Fuente: Wikipedia.

La descarga de cualquier otro tipo de contenido gráfico o audiovisual se realizaría de manera similar al ejemplo anterior.

Asimismo, existen algunas librerías, como la presentada en Heydt (2018), que permiten obtener capturas de pantalla periódicas de una página web. En función del objetivo final del *web scraping* aplicado, esta información almacenada en formato gráfico puede ser de gran utilidad para el rastreo de la evolución de una página web.

Bibliografía recomendada

M. Heydt (2018). *Python Web Scraping Cookbook: Over 90 proven recipes to get you scraping with Python, microservices, Docker, and AWS*. Packt Publishing.

4. Almacenamiento y compartición de datos

Los datos obtenidos mediante *web scraping* pueden almacenarse y compartirse mediante diferentes formatos estandarizados.

En este material didáctico, se describe brevemente el proceso de creación de archivos en dos de los formatos más ampliamente utilizados: CSV y JSON. No obstante, existen muchos otros formatos estandarizados para el almacenamiento de datos, como pueden ser XML o RDF.

Bibliografía recomendada

R. Mitchell (2015). *Web Scraping with Python: Collecting Data from the Modern Web*. O'Reilly.

4.1. Creación de un archivo de datos CSV

Tras realizar *web scraping*, el formato más comúnmente utilizado para almacenar datos de tipo texto es el CSV o *comma separated values*.

El siguiente código muestra un ejemplo de creación de archivo de tipo CSV:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
spamwriter = csv.writer(csvfile, delimiter=' ',
quotechar='|', quoting=csv.QUOTE_MINIMAL)
spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

Enlace de interés

En el siguiente enlace se puede encontrar más información sobre el funcionamiento de la librería CSV: https://docs.python.org/3.4/library/csv.html.

Una vez creado el archivo *eggs.csv*, es posible leer su contenido mediante el siguiente código:

```
import csv
with open('eggs.csv', newline='') as csvfile:
spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
for row in spamreader:
print(', '.join(row))

# Spam, Spam, Spam, Spam, Spam, Baked Beans
# Spam, Lovely Spam, Wonderful Spam
```

4.2. Creación de un archivo de datos JSON

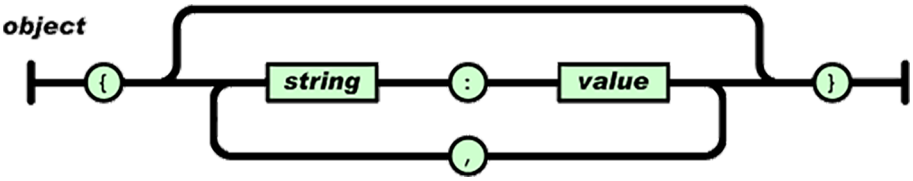
Otro tipo de datos que puede ser interesante, a la hora de almacenar la información extraída mediante *web scraping*, es el formato JSON (Javascript Object Notation), ya que se trata de un formato computacionalmente sencillo, que resulta fácil de leer, interpretar y escribir.

El formato JSON se basa en dos estructuras principales:

1) una colección de pares nombre/valor (objeto)
2) una lista ordenada de valores (*array*)

Así, un objeto, definido como un conjunto de pares nombre/valor, incluye entre llaves ('{}') cada nombre seguido de ':' y su valor; separando los diferentes pares con comas. La figura 8 muestra un esquema de su sintaxis.

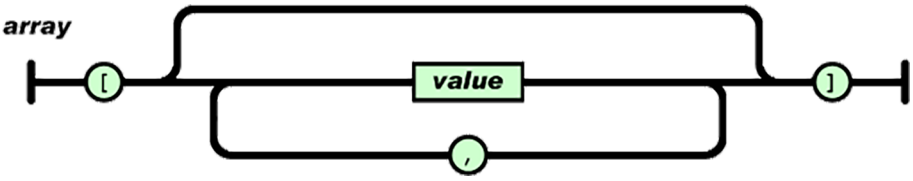
Figura 8. Sintaxis de un objeto



Fuente: JSON

Por otro lado, un *array* se define entre corchetes ('[]'); separando los diferentes valores mediante comas (ver figura 9).

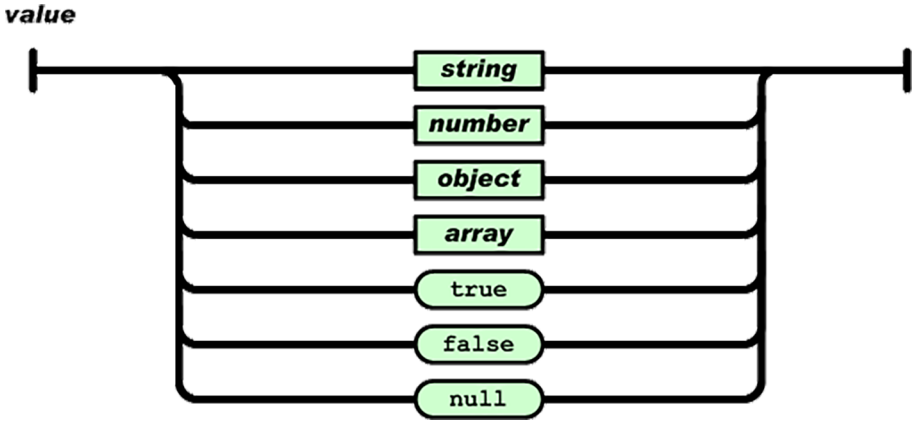
Figura 9. Sintaxis de un *array*



Fuente: JSON

Por su parte, cada valor puede ser un string, un número, un true, un false, un null, un objeto o un *array* (ver figura 10).

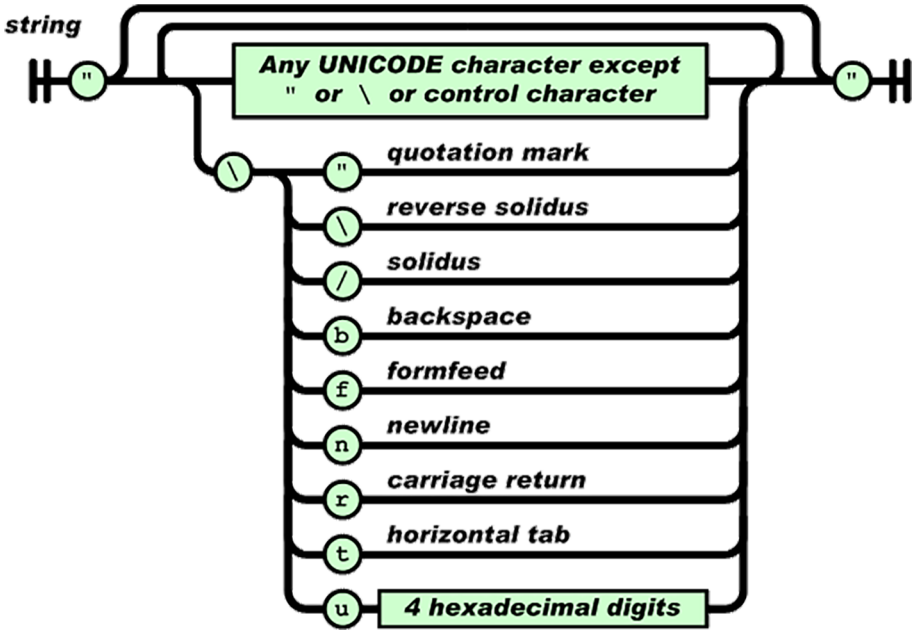
Figura 10. Sintaxis de un valor



Fuente: JSON

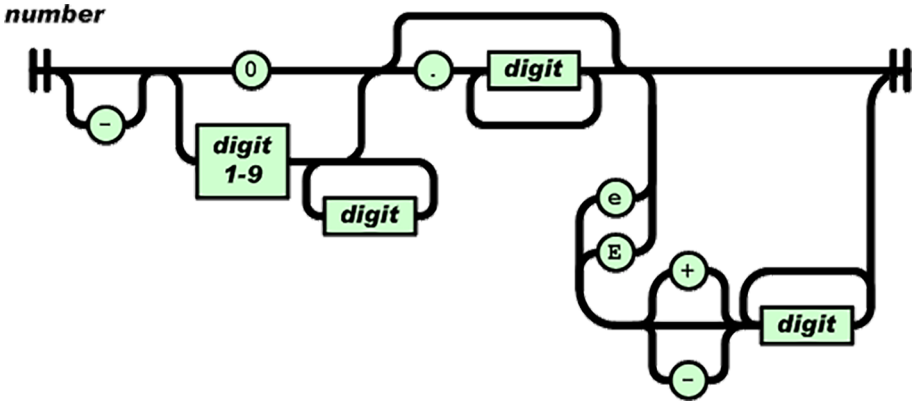
Finalmente, las figuras 11 y 12 representan las sintaxis de un string y un número, respectivamente.

Figura 11. Sintaxis de un string o cadena



Fuente: JSON

Figura 12. Sintaxis de un número



Fuente: JSON

Para aplicar este tipo de codificación se utiliza la librería JSON de Python. A continuación, se muestran algunos ejemplos de uso:

```
import json
json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
# '["foo", {"bar": ["baz", null, 1.0, 2]}]'

print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
# {"a": 0, "b": 0, "c": 0}

print(json.dumps({'4': 5, '6': 7}, sort_keys=True,
indent=4, separators=(',', ': ')))
# {
#   "4": 5,
#   "6": 7
# }
```

Finalmente, el siguiente ejemplo muestra el contenido de un archivo JSON extraído de la página web del Ajuntament de Barcelona:

Enlace de interés

Podéis consultar el archivo en este enlace: http://opendata-ajuntament.barcelona.cat.

```
{
"name" : "Carril_Bici_Construccio_GeoJson",
"type" : "FeatureCollection",
"crs" : {
"type" : "name",
"properties" : {
"name" : "EPSG:25831"
}
},
"features" : [
{
"type" : "Feature",
"geometry" : {
"type" : "LineString",
"coordinates" : [
[ 427704.83499489503, 4579666.9013711698 ],
[ 427709.81394183601, 4579642.00360063 ],
[ 427716.32587691199, 4579625.1497550504 ],
[ 427723.98780126096, 4579607.9119127104 ],
[ 427763.058417933, 4579528.6226368099 ],
[ 427780.27225003002, 4579497.1649236102 ],
[ 427781.06223390898, 4579466.1872106604 ],
[ 427899.71207724401, 4579251.9421635903 ],
[ 428209.77504851203, 4578673.2964416901 ],
```

```
[ 428242.22674152104, 4578647.8856680803 ],
[ 428341.96476299502, 4578447.6954964502 ],
[ 428357.75161059998, 4578424.7347049201 ],
[ 428549.68073765899, 4578075.81088675 ]
]
},
"properties" : {
"CODI_CAPA" : "K028",
"CODI_SUBCAPA" : "K06",
"ID" : "GL241872",
"TOOLTIP" : "Carril Bici Bidireccional pg Zona Franca"
}
},
[...]
}
```

4.3. Creación de una API

Una vez obtenidos los datos de interés mediante *web scraping*, puede ser interesante crear una API que contenga y acceda de forma amigable a dichos datos.

Bibliografía recomendada

K. Dale (2016). *Data Visualization with Python and JavaScript*. O'Reilly.

Una posible solución consiste en crear una REST API, mediante las herramientas Flask y Flask-Restful. Así, tras instalar la librería flask-restful, el siguiente código crearía una API sencilla de este tipo:

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
def get(self):
return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
app.run(debug=True)
```

Enlace de interés

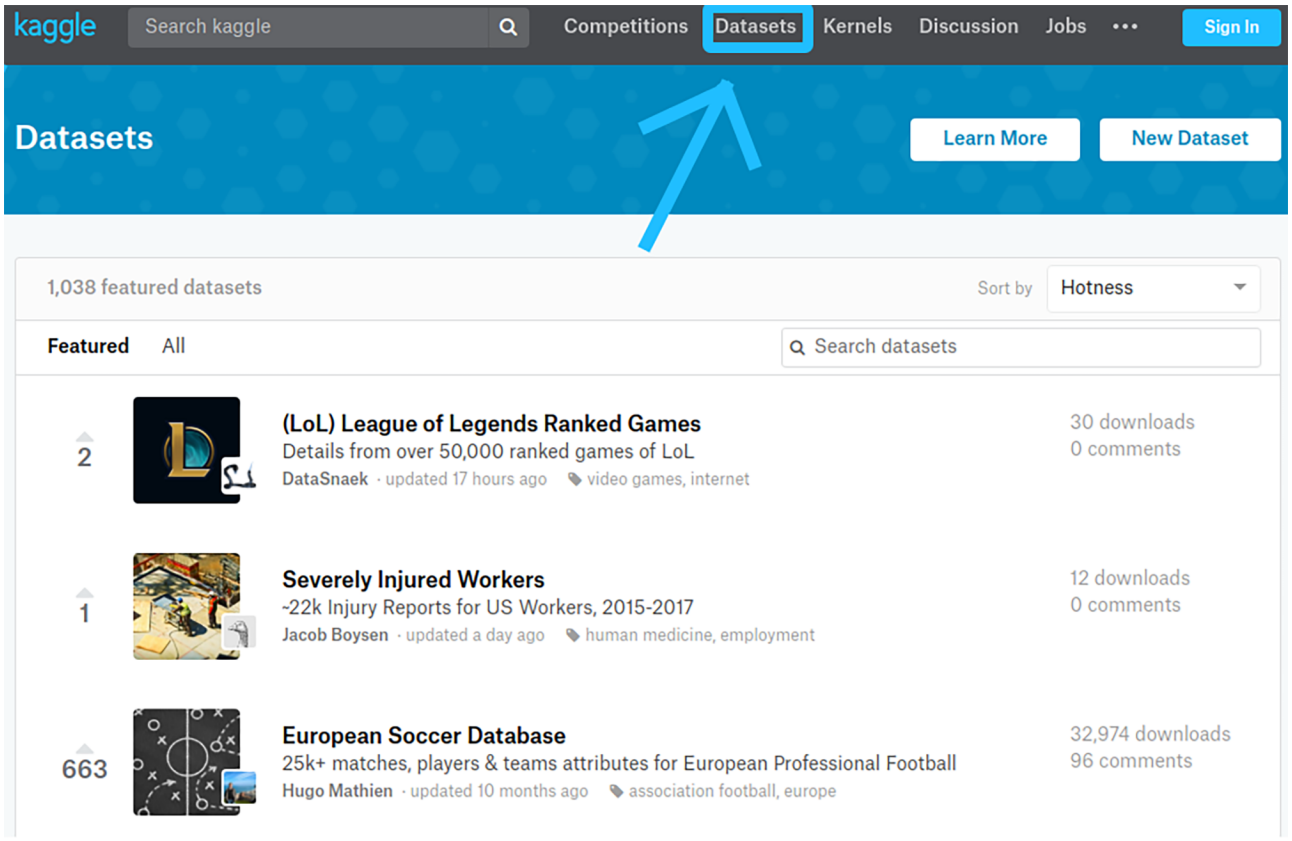
Para más información acerca de la generación de API mediante Flask, se puede consultar el siguiente enlace: https://bit.ly/2sWRCFU.

4.4. Repositorios de datos

Dada la importancia de compartir datos de calidad en internet con el objetivo de contribuir con nuevo conocimiento explotable, existen repositorios públicos en los que es posible compartir aquella información obtenida mediante procesos de *web scraping*.

Aunque existen gran cantidad de repositorios de datos disponibles, los más utilizados actualmente son Kaggle), UCI Machine Learning Repository, Github y Data World. En las figuras 13, 14, 15 y 16 se muestran ejemplos respectivos de estos repositorios.

Figura 13. Ejemplo de conjuntos de datos disponibles en Kaggle



Fuente: blog oficial de Kaggle

Figura 14. Ejemplo de los conjuntos de datos disponibles en UCI Machine Learning Repository

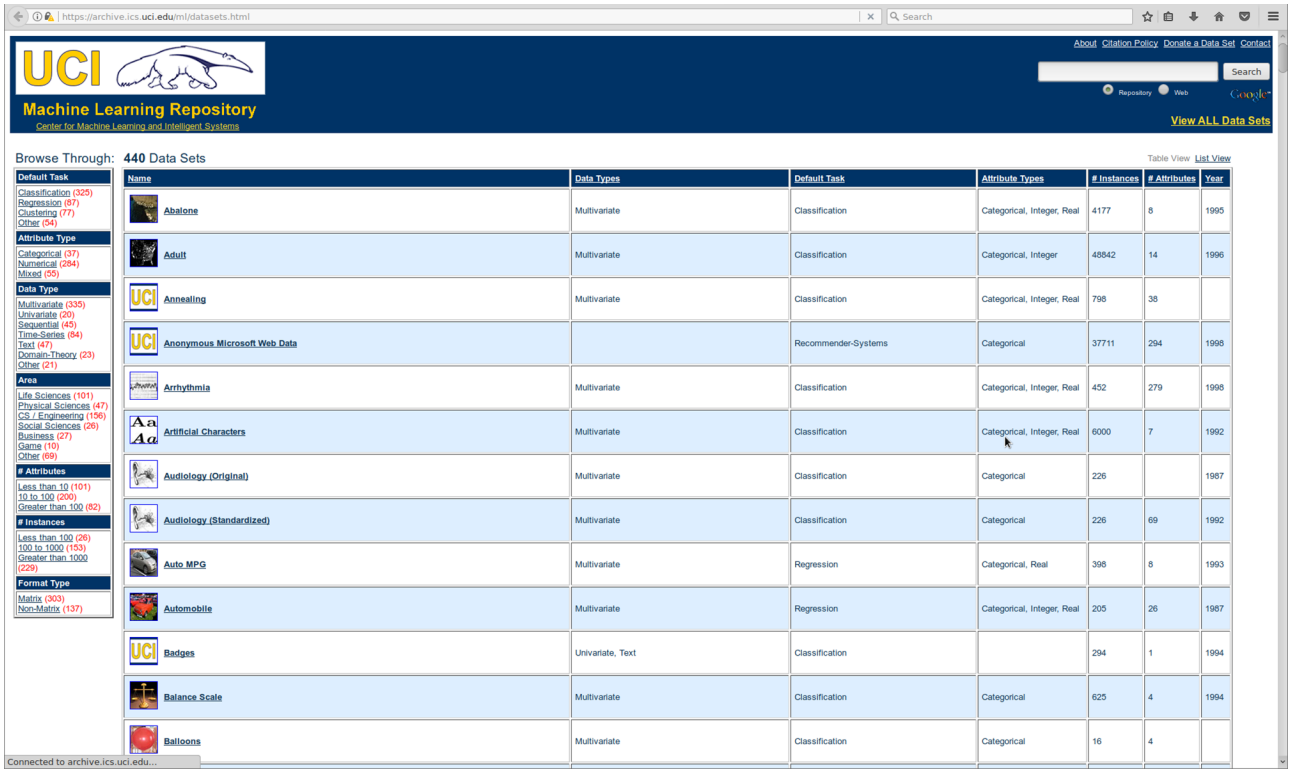
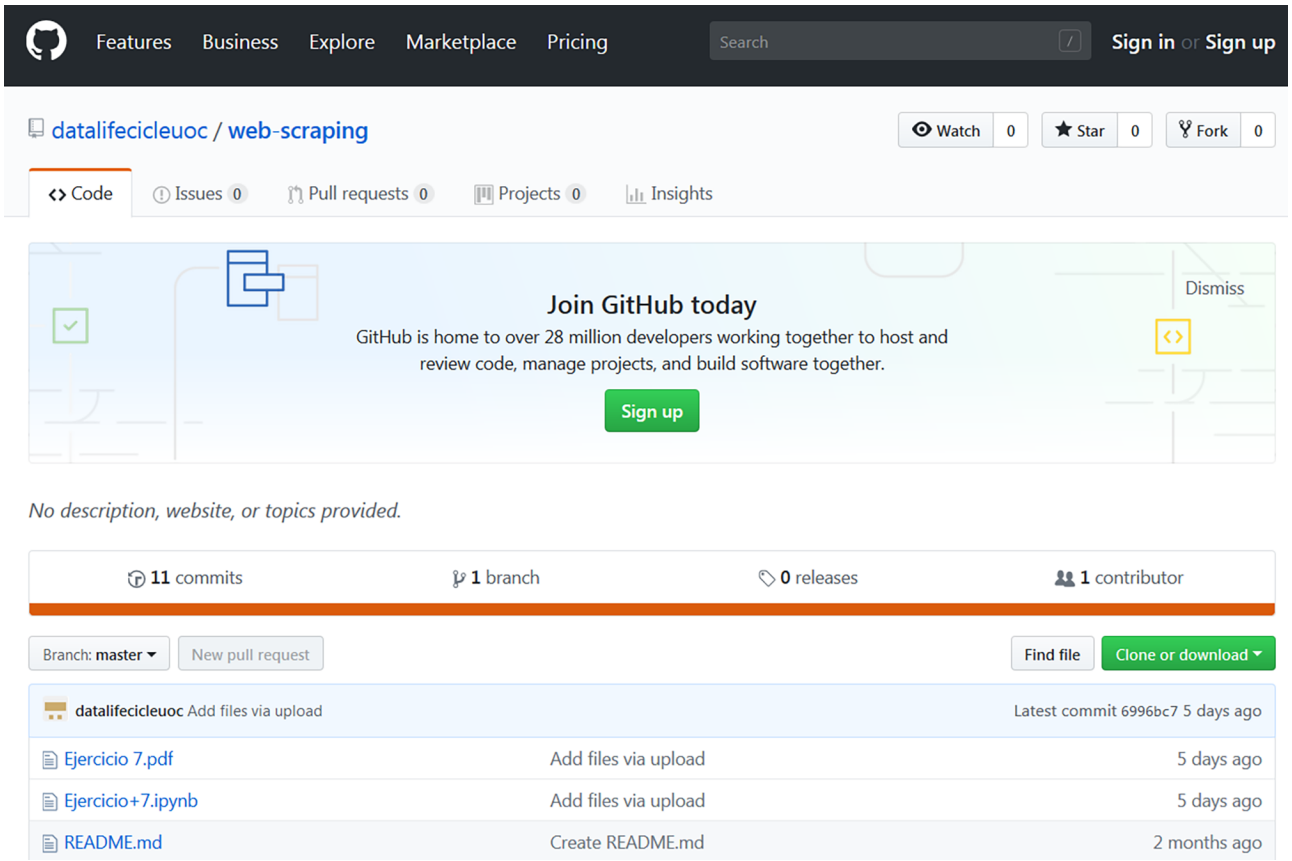


Figura 15. Ejemplo de pantalla de un repositorio Github



Fuente: Github

Figura 16. Imagen de la interfaz DataWorld

Fuente: DataWorld

5. Prevención del *web scraping*

Con el objetivo de evitar el *web scraping*, el administrador de un sitio web puede aplicar diversas medidas que permitan detener o ralentizar el uso de bots. Como consecuencia, se han desarrollado nuevas herramientas basadas en visión por computador y procesado de lenguaje natural que simulan el comportamiento humano y, por tanto, consiguen acceder al contenido web que se pretende rastrear.

Enlace de interés

Para más información sobre el concepto de web scraping, consultad el enlace siguiente: https://en.wikipedia.org/wiki/Web_scraping.

Aunque existen diversos métodos diseñados para la prevención del *web scraping*; a continuación, se listan algunos de los más habituales:

1) Bloqueo de una dirección IP, de forma manual o basado en criterios como la geolocalización.

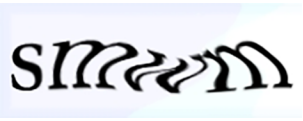
2) Deshabilitación de cualquier API o servicio web asociado al sitio.

3) Uso del archivo *robots.txt* para especificar el bloqueo de ciertos bots, como *googlebot*, o cadenas *user agent*.

4) Control del exceso de tráfico.

5) Uso de CAPTCHA (*completely automated public Turing test to tell computers and humans apart*) para verificar que quien accede al sitio es una persona real (ver figura 17). No obstante, algunos bots están diseñados para resolver ciertos patrones CAPTCHA. Asimismo, algunos sistemas de rastreo utilizan mano de obra humana para responder a estos CAPTCHA en tiempo real.

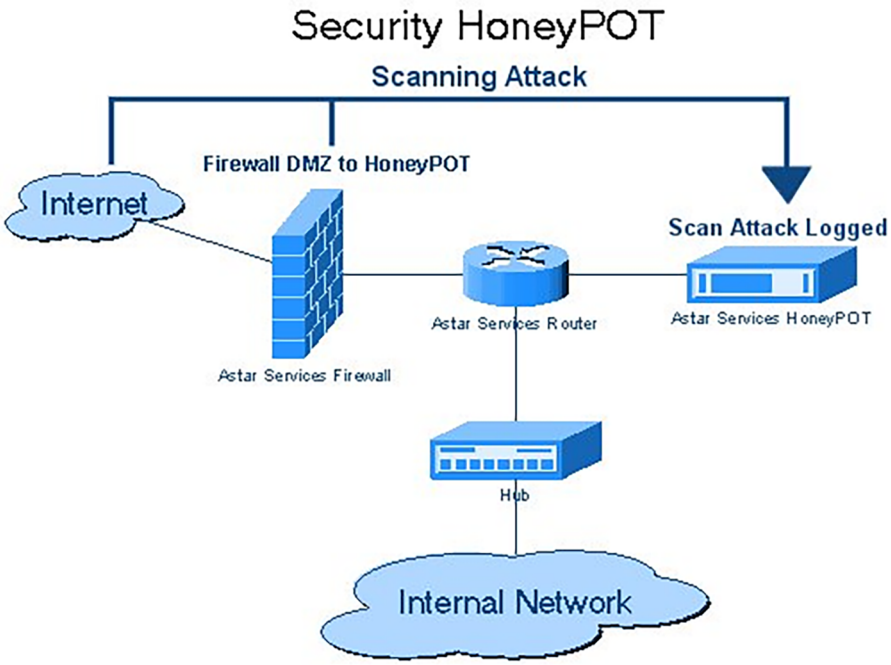
Figura17. Ejemplo de un CAPTCHA (smwm)



6) Uso de servicios comerciales anti-*web scraping*.

7) Uso de *honeypots* para identificar direcciones IP de rastreadores automatizados (ver figura 18). Un *honeypot* (tarro de miel, en castellano) es una herramienta de seguridad informática que se basa en atraer ataques con el fin de neutralizarlos.

Figura 18. Diagrama de un *honeypot*



Fuente: Wikipedia(computing)

8) Uso de CSS sprites para mostrar datos, tales como números de teléfono o direcciones de correo electrónico, ya que esto dificulta la extracción de texto por parte de los rastreadores automáticos (ver figura 19). No obstante, este método supone una pérdida de accesibilidad para los lectores de pantalla y los motores de búsqueda, así como una disminución en el rendimiento del sitio.

Figura 19. Ejemplo de CSS sprites

Fuente: FormGet

9) Añadir pequeñas variaciones en torno a los datos y elementos de navegación HTML/CSS requiere una mayor participación humana a la hora de inicializar un bot, por lo que, si se hace de forma efectiva, puede complicar mucho la automatización del proceso de *web scraping*.

6. Resolución de obstáculos en *web scraping*

A continuación, se resumen diferentes métodos que permiten evitar los obstáculos o trampas diseñadas para prevenir el *web scraping* en una página web.

Bibliografía recomendada

H. Brody (2017). *The ultimate guide to web scraping*. LeanPub.

6.1. Modificación del *user agent* y otras cabeceras HTTP

Como se menciona en el apartado 2.1, cuando se realiza una petición HTTP, el navegador envía una serie de cabeceras al servidor web en las que se incluye información sobre dicha petición.

Una de las cabeceras por defecto más importantes es el *user agent*, ya que contiene información sobre el software que está enviando la petición. En cualquier navegador web, este se ajusta automáticamente a valores tales como Mozilla/5.0 (Macintosh; Intel Mac OS X...). De hecho, es posible identificar qué *user agent* está utilizando nuestro navegador simplemente escribiendo «check user agent» en la barra de direcciones, como se muestra en la figura 20.

Figura 20. Búsqueda del *user agent* mediante Google

Sin embargo, de forma predeterminada, las bibliotecas utilizadas para realizar peticiones HTTP de forma automática establecen su propio *user agent* basándose en el nombre de la librería, el idioma, etc. Dado que esto evidencia el hecho de que las peticiones realizadas provengan de un *script*, en lugar de ser una persona utilizando el navegador, resulta muy recomendable reemplazar dicha cabecera para evitar ser bloqueados a la hora de realizar *web scraping*.

Para solventar este problema, el siguiente código Python muestra un ejemplo de cómo modificar algunas cabeceras HTTP, incluyendo la correspondiente al *user agent*:

```
import requests
headers = {
"Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\
*/*;q=0.8",
"Accept-Encoding": "gzip, deflate, sdch, br",
```

```
"Accept-Language": "en-US,en;q=0.8",
"Cache-Control": "no-cache",
"dnt": "1",
"Pragma": "no-cache",
"Upgrade-Insecure-Requests": "1",
"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/5\
37.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36"
}
r = requests.get("http://www.example.com", headers=headers)
```

6.2. Gestión de *logins* y *cookies* de sesión

En ocasiones, el sitio web que se pretende rastrear requiere de un inicio de sesión (*login*, en inglés) para así obtener del servidor una *cookie* de sesión que deberá acompañar a cada una de las peticiones realizadas posteriormente en el mismo sitio. En estos casos, si se intenta navegar de forma anónima, sin iniciar sesión, el servidor responderá con páginas de error o redirecciones a páginas de inicio.

Así, para gestionar el seguimiento de *cookies* configuradas por el servidor, la librería Requests cuenta con el objeto `session`, que permite agregar dichas *cookies* de forma automática a las posteriores peticiones realizadas en el mismo sitio.

El siguiente código Python muestra un ejemplo de uso:

```
import requests
session = requests.Session()
session.post("http://example.com/login", data=dict(
email="me@domain.com",
password="secret_value"
))
<!-- peticiones realizadas con session agregan automáticamente las cookies-->
r = session.get("http://example.com/protected_page")
```

6.3. Respeto del archivo robots.txt

Como se menciona en el subapartado 1.2.1, es importante verificar si la página de interés cuenta con un archivo *robots.txt* donde el propietario del sitio haya indicado las restricciones a tener en cuenta cuando se pretende rastrearlo.

Estas restricciones son solo una sugerencia y nunca una obligación, por lo que en muchos casos es posible recuperar información de páginas en las que el propietario ha expresado su voluntad de no ser rastreado. No obstante, lo más

recomendable es seguir siempre las sugerencias indicadas en *robots.txt*, con el objetivo de reducir las posibilidades de ser bloqueados y evitar problemas legales futuros.

6.4. Espaciado de peticiones HTTP

El usuario promedio toma unos segundos navegando por una página antes de intentar acceder a la siguiente, lo que desencadena otra petición HTTP en su navegador. No obstante, cuando las peticiones proceden de un *script*, este puede mandar tantas peticiones simultáneas que es capaz de saturar el servidor web. Incluso si el *web scraping* está permitido en una página web determinada, otros usuarios pueden estar navegando al mismo tiempo, por lo que es importante verificar que nuestras acciones no están colapsando el servidor web.

Por ello, una práctica habitual consiste en introducir retardos exponenciales entre peticiones consecutivas cuando se detecta algún error en la página. En lugar de reenviar una misma petición de forma prácticamente instantánea, añadir un retraso exponencialmente creciente entre peticiones proporciona al servidor web la oportunidad de recuperarse.

Otro método ampliamente utilizado consiste en calcular el tiempo que toman en completarse las diferentes peticiones para, a continuación, añadir un retardo proporcional al tiempo estimado. De este modo, si el sitio empieza a ralentizarse y las peticiones realizadas toman más tiempo en recibir una respuesta, se puede ajustar el tiempo de espera entre peticiones de forma automática.

El siguiente código muestra un ejemplo de espaciado automático entre peticiones:

```
import time
for term in ["web scraping", "web crawling", "scrape this site"]:
t0 = time.time()
r = requests.get("http://example.com/search", params=dict(query=term))

<!-- estimación del tiempo de respuesta en segundos-->
response_delay = time.time() - t0

<!-- espera de 10x, con respecto al tiempo de respuesta-->
time.sleep(10 * response_delay)
```

6.5. Uso de múltiples direcciones IP

Algunos servidores web limitan el número de peticiones recibidas por cada dirección IP en un cierto periodo de tiempo. Por ello, puede ser interesante utilizar un conjunto de direcciones IP de modo que no todas las consultas se realicen desde la misma dirección.

No obstante, si bien es posible modificar las cabeceras HTTP mencionadas en los puntos anteriores, pequeñas variaciones en la dirección IP a la hora de enviar una petición no permitirán recibir la respuesta del servidor, conteniendo la información de interés. Así, una solución consiste en utilizar un servidor proxy que enmascare el origen de la petición. Cuando se envía una petición mediante un servidor proxy, esta se transmite primero a dicho servidor, el cual realiza la petición al servidor web, de modo que recibe la respuesta y la transmite de nuevo a la máquina de origen.

Sin embargo, cabe mencionar que el uso de estos servidores suele suponer un coste adicional, que gira en torno a 40 USD por cada 100 direcciones IP, según Brody (2017).

Bibliografía recomendada

H. Brody (2017). *The ultimate guide to web scraping*. LeanPub.

6.6. Configuración de *timeouts* y otras excepciones

En un servidor web, los *timeouts* se producen cuando dicho servidor tarda mucho en devolver una respuesta, generalmente más de 30 segundos. Cuando una petición toma tanto tiempo, se asume que algo no está funcionando correctamente. Por ello, lo más recomendable en estos casos es anular la petición, esperar un poco y volver a intentarlo de nuevo.

El siguiente código muestra un ejemplo de configuración de *timeouts* mediante la librería Requests:

```
try:
<!-- esperar hasta 10 segundos-->
requests.get("http://example.com", timeout=10)
except requests.exceptions.Timeout:
pass
```

Otro problema habitual es el de las conexiones caídas (*broken connections*, en inglés). Si la conexión o el servidor se caen inesperadamente, la petición HTTP realizada se encontrará en un estado ambiguo que no devolverá una respuesta útil. Para solucionar este problema, la librería Requests permite añadir una excepción, similar a la utilizada en la gestión de *timeouts*, que evitará que el código utilizado deje de funcionar.

A continuación, se muestra un ejemplo:

```
try:
requests.get("http://example.com")
except requests.exceptions.RequestException:
pass
```

6.7. Evitar las trampas de araña

Algunos sitios generan dinámicamente su contenido de modo que pueden tener un número infinito de páginas web. Por ejemplo, si el sitio contiene un calendario con enlaces al próximo mes y año, el próximo mes también contará con un enlace al siguiente mes, y así sucesivamente. Dado que nuestro rastreador seguirá en principio cualquier enlace del sitio que no haya visto antes, se verá atrapado en una sucesión infinita de enlaces, conocida como trampa de araña (*spider trap*, en inglés).

Una manera sencilla de evitar las trampas de araña consiste en registrar la profundidad de la página, definida como el número de enlaces que se han seguido para llegar a la misma. Así, al definir previamente una profundidad máxima, cuando se alcanza este umbral, el rastreador deja de agregar enlaces a la cola.

Para implementar esta solución, se puede utilizar el código siguiente:

```
def link_crawler(..., max_depth=2):
max_depth = 2
seen = {}
…
depth = seen[url]
if depth != max_depth:
for link in links:
if link not in seen:
seen[link] = depth + 1
crawl_queue.append(link)
```

Asimismo, para deshabilitar esta opción se puede fijar la profundidad máxima (`max_depth`) a un valor negativo, de modo que la profundidad actual de la página nunca alcance ese valor.

7. Aspectos legales

El panorama legal en torno al *web scraping* se encuentra en plena evolución, por lo que muchas de las leyes vigentes presentan todavía cierta complejidad y ambigüedad cuando se aplican en nuevos escenarios surgidos durante la era digital, como aquellos relacionados con el *web scraping*.

Por ello, esta sección, basada en Vanden Broucke y Baeysens (2018), resume las principales disposiciones en las que suelen basarse los casos judiciales asociados al *web scraping*, así como algunos consejos para evitar problemas legales a la hora de embarcarse en proyectos que impliquen la extracción de información de sitios web que no sean de nuestra propiedad.

Bibliografía recomendada

S. Vanden Broucke; B. Baesens (2018). *Practical Web Scraping for Data Science*. Springer

Por un lado, en Estados Unidos, la mayoría de casos judiciales asociados al *web scraping* se han basado en algunas de las siguientes teorías de infracción o responsabilidad:

1) Incumplimiento de términos y condiciones. La mayoría de páginas web publican una serie de términos y condiciones o acuerdos de licencia de usuario que, a menudo, abordan de forma explícita el acceso a su contenido mediante rastreadores (*scrapers*, en inglés). Con ello, se pretende crear un incumplimiento de la responsabilidad contractual al establecer un contrato entre el propietario del sitio web y el *scraper*.

Sin embargo, la publicación de tales términos en un sitio web puede no ser suficiente para mostrar que un rastreador ha incumplido las condiciones, si no existe una aceptación activa por parte del mismo.

Por ello, el uso de una casilla de verificación explícita o enlace del tipo «Acepto» obliga al *web scraper* a aceptar activamente los términos. Del mismo modo, en aquellos sitios en los que es necesario iniciar sesión, la creación de una cuenta suele incluir un acuerdo explícito de los términos y condiciones.

2) Infracción de derechos de autor o marca registrada. En Estados Unidos, la legislación legal del uso justo (*fair use*, en inglés) permite el uso limitado de material protegido por derechos de autor bajo ciertas condiciones, sin el permiso explícito del titular de dichos derechos. Así, los usos con tales fines como la parodia, la crítica, los comentarios o la investigación académica se consideran uso legítimo. Sin embargo, la mayoría de usos comerciales se consideran una infracción.

3) Ley de fraude y abuso informático. Existen diversas leyes federales y estatales que prohíben el acceso a la máquina de otra persona. En resumen, estas leyes afirman que «quien accede intencionadamente a un ordenador sin autorización [...] y como resultado de tal conducta causa daño» está incumpliendo la ley.

4) Allanamiento de morada. Este término se refiere a un delito civil en el que una entidad interfiere en la propiedad personal de un individuo, causando pérdida de valor o daño. En 1999, esta ley se aplicó en un caso judicial entre Ebay y Bidder's Edge.

Enlace de interés

Para más información sobre el caso entre Ebay y Biddder's Edge, consultad el enlace siguiente: https://en.wikipedia.org/wiki/EBay_v._Bidder_%27s_Edge

5) Protocolo de exclusión de robots. Se trata de un estándar industrial que permite a una página web contar con un archivo *robots.txt* donde se proporcionan instrucciones sobre quién puede acceder al sitio y a qué páginas se puede acceder. Aunque este archivo tiene un valor legal limitado, antes de rastrear cualquier página web es aconsejable verificar si el propietario está de acuerdo, para evitar futuros problemas legales.

6) Ley de derechos de autor del milenio digital y ley CAN-SPAM. Estas leyes han sido también utilizadas en algunos casos judiciales relacionados con *web scraping*.

La primera tipifica como delito la producción y difusión de tecnología, dispositivos o servicios destinados a eludir las medidas que controlan el acceso a material protegido por derechos de autor. Asimismo, penaliza el acto de eludir un control de acceso, independientemente de que exista o no una infracción real de los derechos de autor.

Por otro lado, la Ley de control de la invasión de pornografía y publicidad no solicitada (CAN-SPAM, por sus siglas en inglés de Controlling the Assault of Non-Solicited Pornography And Marketing) estableció en 2003 los primeros estándares para el envío de correo electrónico comercial.

Aunque la situación en la Unión Europea (UE) se rige por diferentes legislaciones y sistemas jurídicos, muchos de los principios previamente mencionados se aplican de forma similar, por ejemplo, en relación con los términos y condiciones o el contenido protegido por derechos de autor. De hecho, la mayoría de propietarios de páginas web en la UE tienden a confiar en las demandas por infracción de derechos de autor con el objetivo de incriminar a los rastreadores. Otras disposiciones clave en este tipo de juicios se enumeran a continuación:

1) Directiva de la UE sobre bases de datos, de 1996. Esta directiva proporciona protección jurídica a los creadores de bases de datos que no estén cubiertos por derechos de propiedad intelectual, de modo que protege los elementos de una base de datos que no son creación original del autor. En particular, pro-

porciona protección cuando «se ha realizado una inversión sustancial, tanto cualitativa como cuantitativa, para la obtención, verificación o presentación de los resultados».

2) Ley de uso indebido de computadoras. Además de la violación a los derechos de propiedad intelectual, teóricamente, los propietarios de páginas web cuentan con otros argumentos legales para luchar contra el *web scraping*. Es el caso de la Ley de uso indebido de computadoras de 1990, que prohíbe el acceso y modificación no autorizados de material informático.

Así, se puede observar cómo el *web scraping*, especialmente cuando se realiza a gran escala o para uso comercial, puede acompañarse de implicaciones legales complejas. Por ello, es aconsejable consultar a un abogado o a los expertos apropiados antes de embarcarse en este tipo de proyectos, así como tener en cuenta los siguientes principios clave:

- **Obtener permiso por escrito.** La mejor práctica para evitar problemas legales consiste en obtener permiso escrito por parte del propietario de un sitio web, en el que se especifique hasta qué punto se puede extraer información del mismo.

- **Verificar las condiciones de uso.** Estas incluirán a menudo disposiciones explícitas contra la extracción automática de datos. Asimismo, las API de un sitio web suelen acompañarse de sus propios términos de uso, por lo que es aconsejable revisar también estos casos.

- **Rastrear solo información pública.** Generalmente, cuando un sitio web expone información de forma pública, sin ser necesario aceptar una serie de términos y condiciones, se asume que el uso moderado de *web scraping* es adecuado. Aquellos sitios en los que es necesario iniciar sesión para acceder a la información de interés, por el contrario, son más delicados desde el punto de vista legal.

- **No causar daño.** No sobrecargar el servidor con muchas peticiones, mantenerse alejado de los equipos protegidos y no intentar acceder a aquellos servidores a los que no se tiene acceso.

- **Utilizar la información extraída de forma justa.** No utilizar con fines comerciales aquellos datos protegidos por derechos de autor.

8. Mejores prácticas y consejos

La siguiente lista, basada en Vanden Broucke y Baeysens (2018), resume una serie de buenas prácticas y consejos a la hora de realizar *web scraping*:

1) Antes de realizar *web scraping*, **verificar si ya existe una API** que permita recuperar la información de interés, sin limitaciones de descarga.

2) No parsear el HTML manualmente. El uso de librerías como BeautifulSoup facilita considerablemente la tarea.

3) No saturar de peticiones el servidor web, ya que esto aumentará las probabilidades de ser bloqueados. Asimismo, dado que el *webmaster* puede darse cuenta de que se están realizando gran cantidad de peticiones en su página, puede ser interesante contactar con el administrador del sitio para encontrar la forma de trabajar conjuntamente.

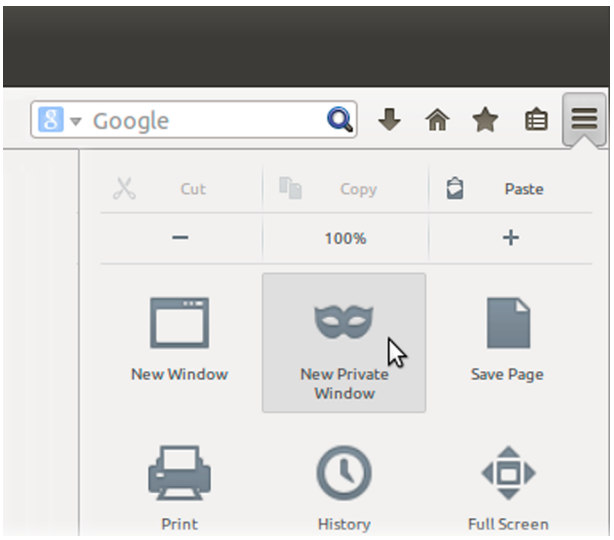
4) Modificar el *user agent*, ya que muchos sitios revisan esta cabecera para prevenir el *web scraping*.

5) Chequear el navegador. Si desconocemos la causa de un problema, puede ser interesante abrir una nueva sesión en el navegador, preferiblemente utilizando los modos «incógnito» o «navegación privada» (*private browsing*, en inglés) para asegurar que el conjunto de cookies se encuentra vacío. Asimismo, se puede usar el comando `curl` para debugar los casos más complejos.

Ejemplo de navegación privada

Para navegar de forma privada, por ejemplo en Mozilla Firefox, se debe acceder al navegador como se muestra en la figura 21. Cabe mencionar que, en estos casos, el navegador muestra una máscara en la parte superior.

Figura 21. Navegación en modo privado



Fuente: pantalla de ayuda de Mozilla

6) Asumir que el *web scraper* dejará de funcionar. Las páginas web son dinámicas, por lo que puede ser de gran utilidad implementar un código que proporcione advertencias tempranas y detalladas cuando algún fragmento deje de funcionar.

7) Tener en cuenta la calidad y robustez de los datos obtenidos. La International Data Management Association del Reino Unido (DAMA UK) define la calidad de los datos a partir de las seis dimensiones siguientes:

Enlace de interés

DAMA UK Working Group. *The six primary dimensions for data quality assessment. Defining Data Quality Dimensions*. https://bit.ly/2qcimnf

a) Completitud (*completeness*, en inglés). Hace referencia a la proporción de datos almacenados frente al potencial de datos completos.
b) Unicidad (*uniqueness*, en inglés). Al comparar los datos con otros conjuntos, estos deben ser únicos.
c) Puntualidad (*timeliness*, en inglés). El grado en que los datos representan la realidad, desde el punto requerido en el tiempo.
d) Validez (*validity*, en inglés). Los datos se consideran válidos si se ajustan a la sintaxis (formato, tipo, rango) de su definición.
e) Exactitud (*accuracy*, en inglés). El grado en que los datos se ajustan a la realidad que se está describiendo.
f) Consistencia (*consistency*, en inglés). Hace referencia a la ausencia de diferencias cuando se comparan dos o más representaciones de algo, con respecto a su definición.

8) Recordar los aspectos legales asociados al *web scraping*, con el objetivo de hacer un buen uso de los datos obtenidos.

9. Ejemplos de *web scraping* y casos de éxito

El *web scraping* puede aplicarse a dominios muy variados, con diferentes fines. A continuación, se citan varios ejemplos de uso potencial en diferentes áreas.

Bibliografía recomendada

S. Vanden Broucke; B. Baesens (2018). *Practical Web Scraping for Data Science*. Springer

- **Analizar la competencia**. El *web scraping* puede ser de utilidad para comparar productos y precios de nuestros principales competidores.

- **Gestionar la reputación de una marca, comercial o personal, en internet**. Esto se puede realizar mediante grafos de relaciones obtenidos de Wikipedia, realizando análisis de sentimientos procedentes de redes sociales, etc.

- **Analizar al cliente**. Conocer los gustos y preferencias del cliente puede ayudar a proveer mejores servicios, así como a desarrollar un plan de marketing más adaptado y, por tanto, más eficiente.

- **Realizar tareas específicas**. En ocasiones, es necesario rastrear información de internet de forma puntual, para solucionar problemas concretos.

- **Obtener correos electrónicos**. El *e-mail marketing* es una técnica frecuentemente utilizada por las empresas para contactar con potenciales clientes. El *web scraping* permite encontrar las direcciones de correo electrónico de dichos clientes potenciales con el objetivo de enviarles información comercial.

- **Detectar opiniones fraudulentas**. El *web scraping* permite recuperar información con el objetivo de desarrollar sistemas de detección temprana de comportamientos fraudulentos en internet.

- **Mejorar el posicionamiento: *search engine optimization* (SEO)**. El posicionamiento SEO se basa en diversos parámetros, entre los que se encuentra el contenido de la página web. Así, analizar dicho contenido mediante *web scraping* puede ayudar en la optimización del posicionamiento.

- **Resumir información**. En ocasiones, la información de interés puede estar distribuida en diferentes páginas web, por lo que el *web scraping* puede ayudar a centralizar toda esa información en un mismo lugar.

Pero, en la práctica, ¿quién está realizando en la actualidad *web scraping*? Aunque las posibles aplicaciones de esta herramienta son prácticamente ilimitadas, la siguiente lista, recogida de Broucke y Baeysens (2018), muestra algunos ejemplos interesantes de casos de uso exitosos:

- Muchos productos de Google se benefician de esta técnica. El traductor de Google (Google Translate), por ejemplo, utiliza texto almacenado en la web para entrenarse, aprender y mejorar continuamente. Otros traductores como DeepL se basan en el mismo principio.

Enlace de interés

Podéis acceder al traductor DeepL en este enlace: https://www.deepl.com/translator.

- En RRHH, esta técnica está ganando cada vez más fuerza. La *startup* hiQ, por ejemplo, se ha especializado en la venta de información sobre empleados, o potenciales empleados, mediante el uso de datos recopilados de internet, principalmente extraídos de los perfiles públicos de LinkedIn.

Enlace de interés

Podéis acceder a la página web de hiQ en este enlace: https://www.hiqlabs.com/.

- Las empresas de marketing digital, así como los artistas digitales, utilizan los datos presentes en internet para diseñar todo tipo de proyectos creativos. El proyecto «We Feel Fine», introducido en el apartado 1, en el que Jonathan Harris y Sep Kamvar rastrearon todas aquellas frases presentes en blogs empezando por «I feel/I am feeling», dio lugar a una forma muy creativa de mostrar cómo se sentía el mundo a lo largo del día.

Enlace de interés

Para más información sobre el proyecto, consultad el enlace: wefeelfine.org/.

- En otro estudio, los mensajes descargados de Twitter, blogs y otras redes sociales fueron analizados para construir una base de datos que permitiera construir un modelo predictivo para la identificación de patrones de depresión y pensamientos suicidas. Aunque esta herramienta podría tener un valor incalculable, existe cierta controversia relacionada con la privacidad de las personas cuyos mensajes están siendo evaluados.

Enlace de interés

Acceded al estudio sobre Twitter en este enlace: https://bit.ly/2wIGpsy.

- En un artículo de Cavallo y Rigobon (2016), titulado «The Billion Prices Project: Using Online Prices for Measurement and Research» (el proyecto de los mil millones de precios: el uso de precios en línea para la medición y la investigación), se utilizó *web scraping* para recolectar un conjunto de datos compuesto por gran cantidad de precios presentes en internet, con el objetivo de construir un índice robusto de precios diarios, para diferentes países.

Enlace de interés

Acceded al estudio de Cavallo y Rigobon (2016) en este enlace: http://www.nber.org/papers/w22111.

- Los bancos y otras instituciones financieras utilizan el *web scraping* para analizar la competencia. Por ejemplo, los bancos analizan con frecuencia las páginas web de la competencia para conocer dónde se están abriendo o cerrando sucursales, así como para realizar un seguimiento de las tasas de préstamos ofrecidas. Del mismo modo, las empresas de inversión suelen utilizar *web scraping* para rastrear aquellas noticias relacionadas con los activos de su cartera.

- Los científicos sociopolíticos también utilizan el *web scraping* para analizar los sentimientos de la población, así como su orientación política. Un famoso artículo titulado «Dissecting Trump's Most Rabid Online Following» describe los resultados de uno de estos estudios, en el que se analizaron

Enlace de interés

Acceded al artículo en la página web: https://fivethirtyeight.com/features/dissecting-trumps-most-rabid-online-following/.

discusiones entre usuarios de la plataforma Reddit, con el objetivo de caracterizar a los seguidores de Donald Trump.

- La información extraída de imágenes procedentes de Tinder e Instagram permitió crear un modelo predictivo que identificara si una imagen era considerada «atractiva». Un dato interesante es que los fabricantes de teléfonos inteligentes están incorporando este tipo de modelos en sus aplicaciones fotográficas, para mejorar la calidad (o percepción de la calidad) en sus imágenes.

Enlace de interés

Acceded al artículo sobre Tinder e Instagram en este enlace: https://bit.ly/2zAQJGs.

Resumen

En este módulo didáctico se han revisado los aspectos fundamentales relacionados con el *web scraping*. En primer lugar, se ha presentado la utilidad y potencial de esta herramienta, particularmente útil cuando se requiere obtener información de un sitio web que no dispone de una API para tal efecto, o cuya API no satisface por completo las necesidades de nuestro proyecto de ciencia de datos.

Tras revisar algunos pasos previos necesarios para planificar cualquier proceso de *web scraping* de forma óptima, se han revisado las principales herramientas disponibles en las librerías Requests y BeautifulSoup que permiten recuperar textos, imágenes o cualquier otro contenido audiovisual procedentes de internet.

Posteriormente, en el apartado 4, se han presentado los formatos estandarizados más comúnmente utilizados en el almacenamiento de datos rastreados: CSV y JSON. Asimismo, se han introducido las herramientas Flask mediante las cuales es posible implementar una API que contenga y acceda de forma amigable a dichos datos. Finalmente, se han enumerado diferentes repositorios públicos en los que se pueden compartir las bases de datos resultantes.

En el siguiente apartado, se han introducido varias medidas que pueden aplicarse en cualquier sitio web para evitar el *web scraping*. A continuación, en el apartado 6, se han presentado diversos métodos que permiten simular el comportamiento humano con el objetivo de resolver dichas medidas u obstáculos, como el uso de múltiples direcciones IP, la configuración de *timeouts*, la gestión de *logins* y *cookies* de sesión, etc.

A continuación, se han planteado los principales aspectos legales relacionados con la extracción de datos procedentes de internet, para posteriormente enumerar una lista de mejores prácticas y consejos que nos permitan implementar un buen uso del *web scraping*.

Finalmente, con el objetivo de mostrar el potencial de esta herramienta, se han presentado algunos ejemplos de aplicación, así como casos de éxito reales, del *web scraping*.

Ejercicios de autoevaluación

1. Explicad con vuestras propias palabras cuándo es útil realizar *web scraping*. Imaginad que tenéis un negocio, explicad cuándo podría ser útil aplicar *web scraping*.

2. Poned un ejemplo donde publicar datos obtenidos mediante *web scraping* sea legal, y otro en el que no.

3. Enumerad los cinco pasos previos necesarios para planificar cualquier proceso de *web scraping* de forma óptima.

4. ¿Los errores en la descarga de páginas web pueden ser temporales? Explicad por qué.

5. ¿Es necesario establecer un *user agent*? Explicad por qué.

6. Enumerad los métodos utilizados para resolver los obstáculos más habituales en *web scraping*.

7. El objetivo de esta actividad es la creación de un dataset a partir de los datos contenidos en un sitio web. Inicialmente, se deberá analizar si la página web cuenta con un archivo *robots.txt*, así como con un mapa del sitio web, cuál es su tamaño, la tecnología usada y el propietario del mismo. A continuación, se deben indicar las siguientes características del dataset general:

a) Título descriptivo del dataset.

b) Subtítulo del dataset. Descripción ágil del conjunto de datos creado.

c) Imagen. Representación gráfica que identifique el dataset.

d) Contexto. ¿Cuál es la materia del conjunto de datos?

e) Contenido. ¿Qué campos incluye? ¿Cuál es el periodo de tiempo de los datos y cómo se han recogido?

f) Agradecimientos. ¿Quién es el propietario del conjunto de datos?

g) Inspiración. ¿Por qué es interesante este conjunto de datos? ¿Qué preguntas le gustaría responder a la comunidad?

h) Licencia. Se debe seleccionar una de las siguientes licencias y justificar la elección: Released Under CC0: Public Domain License, Released Under CC BY-NC-SA 4.0 License, Released Under CC BY-SA 4.0 License, Database Released Under Open Database License, Individual Contents Under Database Contents License, Other (specified above) o Unknown License.

i) Código. Se debe especificar el código utilizado para generar el dataset.

j) Dataset resultante, en formato CSV.

8. Imaginad que queréis crear una empresa con el objetivo de vender un producto o servicio. Después de explicar en tres líneas la misión de dicha empresa, detallad diferentes conjuntos de datos, ya existentes o creados mediante *web scraping*, que utilizaríais para mejorar el producto o servicio ofrecido.

Solucionario

Ejercicios de autoevaluación

1. Es útil realizar *web scraping* cuando no disponemos de API para acceder a los datos web, o cuando las API disponibles no aportan información suficiente para nuestro proyecto de datos. Como ejemplo de negocio, podemos suponer una zapatería que pretende hacer un seguimiento de los precios de la competencia. Podríamos analizar el sitio web del principal competidor todos los días, con el objetivo de comparar los precios de los diferentes zapatos a la venta; sin embargo, esto tomaría mucho tiempo y no permitiría controlar cambios frecuentes en la oferta. Por lo tanto, una alternativa consistiría en reemplazar este proceso manual y repetitivo por una solución automatizada, basada en técnicas de *web scraping*.

2. Ejemplo legal: listas de teléfonos. Ejemplo ilegal: opiniones (asociadas a derechos de autor).

3. Revisar el archivo *robots.txt*, examinar el mapa del sitio web, estimar su tamaño, identificar la tecnología utilizada y conocer el webmaster o propietario.

4. Sí, los errores de descarga pueden ser temporales. Un ejemplo de error temporal es el código de estado `503 Service Unavailable`, tras el que se puede intentar la descarga más tarde.

5. No es necesario, pero es recomendable ya que algunos sitios web bloquean al usuario por defecto, para evitar el uso de rastreadores automáticos.

6. Modificar el *user agent*, gestionar los *logins* y *cookies* de sesión, analizar el archivo *robots.txt*, espaciar las peticiones HTTP, usar múltiples direcciones IP mediante servidores proxy, configurar *timeouts* y evitar las *spider traps* o trampas de araña.

7. La solución de esta actividad se encuentra en el repositorio Github (https://github.com/datalifecicleuoc/web-scraping).

8. La misión de la empresa es ofrecer calzado fabricado con materias primas 100 % reciclables y ecológicas. Tanto la materia prima como la elaboración del producto serán de proximidad, fomentando el empleo local y el comercio justo. Algunos conjuntos de datos ya existentes de utilidad podrían ser los siguientes:

a) Women's Shoe Prices obtenido de Kaggle.

b) Men's Shoe Prices obtenido de Data World.

c) Otro ejemplo de conjunto de datos es UT Zappos50K.

Asimismo, con el objetivo de realizar un estudio de mercado, se podría aplicar *web scraping* para recuperar información sobre los productos y precios que ofrece la competencia.

Además, la información disponible en páginas como Wikipedia o DBpedia podría proporcionar datos relevantes acerca de la reputación de la empresa con los que desarrollar un grafo de relaciones que podríamos analizar con programas como Gephi.

Un análisis de sentimientos permitiría evaluar la polaridad y subjetividad de dicha información, permitiendo detectar opiniones fraudulentas.

Por otro lado, sería interesante analizar las opiniones de nuestros clientes en el sitio web o en redes sociales, con el objetivo de adaptar tanto el producto/servicio ofrecido como el plan de marketing.

El *web scraping* también podría servir para encontrar direcciones de correo electrónico de potenciales clientes, interesados en el comercio justo, en proteger el medio ambiente y en el calzado de diseño.

Por último, contar con un resumen de toda la información disponible sobre calzado ecológico de comercio justo en un mismo sitio podría ser interesante a la hora de tomar decisiones sobre nuestro modelo de negocio.

Glosario

API *f* véase **interfaz de programación de aplicaciones**.

asociación de gestión de datos *f* Asociación sin ánimo de lucro e independiente, dedicada al desarrollo de la gestión de recursos de datos (*data resource management* o DRM) y de la gestión de recursos de información (*information resources management* o IRM).
sigla **DAMA**
en data management association

atom feeds *m* Formato de redifusión web basado en un fichero XML, desarrollado como alternativa al formato RSS.

bot *m* Aféresis de robot. Programa informático autónomo, capaz de llevar a cabo tareas concretas y repetitivas a través de internet, cuya realización por parte de un humano sería imposible o muy tediosa.

CAN-SPAM *m* véase **control de la invasión de pornografía y publicidad no solicitada**.

CAPTCHA Siglas de *completely automated public turing test to tell computers and humans apart* (prueba de Turing completamente automática y pública para diferenciar ordenadores de humanos). Prueba de tipo desafío-respuesta, utilizada en computación para determinar cuándo un usuario es o no humano.

control de la invasión de pornografía y publicidad no solicitada *m* Ley federal estadounidense que establece las reglas y sanciones aplicables al correo electrónico comercial.
sigla **CAN-SPAM**
en Controlling the Assault of Non-Solicited Pornography And Marketing

galleta *f* Pequeño archivo con datos procedentes de un sitio web, almacenados en el navegador del usuario. Dado que proporciona información sobre la actividad previa del usuario en dicho sitio web, permite agilizar la navegación.
en cookie

CSS *f* Véase **hoja de estilo en cascada**.

CSS *sprites* *f* En una página web, conjunto de imágenes o iconos agrupados en una misma imagen.

CSV *m* Véase **valor separado por comas**.

DAMA *f* Véase **asociación de gestión de datos**.

DOM *m* Véase **modelo de objetos del documento**.

Github *m* Plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git.

hoja de estilo en cascada *f* Lenguaje de diseño gráfico que permite presentar, de manera estructurada, un documento escrito en lenguaje de marcado. Se usa principalmente en el diseño visual de documentos web e interfaces de usuario escritas en XML o HTML.
sigla **CSS**
en cascading style sheets

HTML *m* Véase **lenguaje de marcas de hipertexto**.

***honeypot* (tarro de miel, en castellano)** Herramienta de seguridad informática diseñada para ser el objetivo de un posible ataque, con el fin de detectarlo y obtener información del mismo, así como del atacante.

HTTP Siglas de Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto). Protocolo de comunicación que permite las transferencias de información en la World Wide Web.

interfaz de programación de aplicaciones *f* Conjunto de rutinas que permiten acceder a funciones de un determinado software; en internet, las API permiten acceder al contenido de un sitio web.
sigla **API**
en application programming interface

JSON Acrónimo de JavaScript Object Notation. Formato de texto ligero utilizado para el intercambio de datos.

lenguaje de marcado extensible *m* Metalenguaje extensible de etiquetas, desarrollado por el World Wide Web Consortium (W3C) y adaptado del SGML (Standard Generalized Markup Language).
sigla **XML**
en eXtensible Markup Language

lenguaje de marcas de hipertexto *m* Lenguaje de marcado utilizado para la elaboración de páginas web.
sigla **HTML**
en hypertext markup language

mapa de sitio web *m* Conjunto de páginas de un sitio web, accesibles por parte de buscadores y usuarios.
en sitemap

marco de descripción de recursos *m* Familia de especificaciones de la World Wide Web Consortium (W3C), originalmente diseñado como un modelo de datos para metadatos.
sigla **RDF**
en resource description framework

modelo de objetos del documento *m* Interfaz de plataforma que proporciona un conjunto estándar de objetos para representar, de forma jerárquica, documentos HTML, XHTML y XML. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de dichos documentos.
sigla **DOM**
en document object model

lenguaje extensible de hoja de estilo *m* Familia de lenguajes basados en el estándar XML que permite describir cómo la información contenida en un documento XML cualquiera debe ser transformada o formateada para su presentación en un medio.
sigla **XSL**
en extensible stylesheet language

optimización para motores de búsqueda *f* Técnica que consiste en optimizar la estructura e información de un sitio web con el objetivo de mejorar su visibilidad en los resultados orgánicos de los diferentes buscadores de internet.
sigla **SEO**
en search engine optimization

parsear Recorrer todos los registros de una base de datos. Un *parser*, en computación, es un analizador sintáctico, es decir, un programa informático que analiza la sintaxis de un documento escrito en un lenguaje en particular.

profundidad (de una página web) *f* Número promedio de clics necesarios para llegar a una determinada página del sitio.

RDF *m* Véase **marco de descripción de recursos**.

RSS *feed* Siglas de Really Simple Syndication (Sindicación Realmente Simple). Formato XML utilizado para la difusión de contenido web.

sindicación realmente simple *f* Formato XML utilizado para la difusión de contenido web.
sigla **RSS**
en really simple syndication

SEO *m* Véase **optimización para motores de búsqueda**.

servidor proxy *m* Equipo dedicado o sistema de software que actúa como intermediario en las peticiones de recursos que realiza un cliente a otro servidor.

trampa de araña *f* Conjunto de páginas web que, intencionadamente o no, pueden causar que un rastreador web o bot de búsqueda se bloquee entre un número infinito de peticiones.
en spider trap

valor separado por comas *m* Archivo de texto que almacena los datos en forma de tabla, donde las columnas se separan por comas (o punto y coma en aquellos idiomas en los que la coma es el separador decimal) y las filas por saltos de línea.
sigla **CSV**

en comma-separated values

XLSX *m* Formato y extensión de archivo empleado en Microsoft Excel a partir de su versión 2007 (anteriormente XLS).
en Excel Microsoft Office Open XML Format Spreadsheet

XML *m* Véase **lenguaje de marcado extensible**.

XSL *m* Véase **lenguaje extensible de hoja de estilo**.

webmaster *m* Persona responsable del desarrollo, coordinación y mantenimiento de un sitio web.

Bibliografía

Acodemy (2015). *Learn Web Scraping With Python In A Day: The Ultimate Crash Course to Learning the Basics of Web Scraping With Python In No Time*. CreateSpace Independent Publishing Platform.

Bosch, O. (2017). *An introduction to web scraping, IT and Legal aspects*. <https://bit.ly/2pMUYKC>

Brody, H. (2017). *The ultimate guide to web scraping*. LeanPub.

Broucke, S. Vanden; Baesens, B. (2018). *Practical Web Scraping for Data Science*. Springer.

Casas, J.; Conesa, J. (2016). *Datos abiertos y enlazados*. Editorial UOC.

Cavallo, A.; Rigobon, R. (2016). *The Billion Prices Project: Using Online Prices for Measurement and Research. Journal of Economic Perspectives* (vol. 30, núm. 2, págs. 151-178).

Dale, K. (2016). *Data Visualization with Python and JavaScript*. O'Reilly.

Heydt, M. (2018). *Python Web Scraping Cookbook: Over 90 proven recipes to get you scraping with Python, microservices, Docker, and AWS*. Packt Publishing.

Kouzis-Loukas, D. (2016). *Learning Scrapy*. Packt Publishing.

Lawson, R. (2015). *Web Scraping with Python*. Packt Publishing Ltd.

Minguillón, J. (2016). *Fundamentos de Data Science*. Editorial UOC.

Mitchell, R. (2015). *Web Scraping with Python: Collecting Data from the Modern Web*. O'Reilly.

Munzert, S.; Rubba, C.; Meißner, P.; Nyhuis, D. (2014). *Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining*. Hoboken, NJ; Chichester; West Sussex: John Wiley & Sons.

Nair, V.G. (2014). *Getting started with BeautifulSoup*. Packt Publishing Ltd. Open Source Collaborative framework in Python. [Fecha de consulta: 15 de marzo de 2018]. <https://scrapy.org>