

# *Web scraping*

---

## Agentes de extracción de información en Internet

Departamento de Sistemas Informáticos

Escuela Técnica superior de Ingeniería de Sistemas Informáticos - UPM

16 de febrero de 2024

---

License CC BY-NC-SA 4.0

# ¿Qué es el *web scraping*?

---

Técnica para **extraer información de sitios web** mediante programas de software

- Es la única opción si un sitio no ofrece una API (o si lo hace, pero insuficiente)

Es extremadamente sencillo; se basa en dos procesos:

1. Navegar por sitios web (los procesos que navegan denominan *spiders* o *crawlers*)
2. Extraer de datos de sitio web (*scraping*)

Es muy útil, pero tiene ciertos **inconvenientes**

- Es una **técnica más costosa** que usar una API
- Genera **más tráfico** entre cliente y servidor
- Es **sensible a cambios** en la estructura de las páginas web
- Hay que respetar las **condiciones (legales o no)** de uso de los sitios web

# ¿Para qué se utiliza el *web scraping*?

---

Algunos de los dominios de aplicación del *web scraping* son:

- **Automatización del negocio**, evitando tareas manuales tediosas como recopilar información de diversas fuentes
- **Estudios de mercado**, dado que algunos datos de mercado son públicos
- **Generación de *leads***, o lo que es lo mismo, listas de clientes potenciales
- **Seguimiento de precios**, como por ejemplo [CamelCamelCamel](#)
- **Noticias y contenidos**, agregándolos para una consulta más cómoda
- **Monitorización de la marca**, para saber qué se dice de ella en Internet
- **Mercado inmobiliario**, para saber qué se vende y a qué precio

# Protección contra el *web scraping*

---

Es lógico que algunas páginas web protejan sus datos contra el web scraping

- No es infalible, prácticamente **todo** lo que hace un navegador es replicable
- Nos queda el consuelo que al menos dificultan la tarea

Existen varias técnicas para protegerse contra el web scraping, entre ellas:

- **Análisis del comportamiento del usuario:** Técnicas orientadas a detectar si el usuario es un humano o un robot en función de su comportamiento
- **Bloqueo de IP:** Cortar acceso a IP identificadas como *bots*
- **Captcha**<sup>2</sup>: Imagen con texto para demostrar que quien accede es humano<sup>3</sup>
- **Fichero robots.txt:** Archivo que indica a los robots qué rutas no deben visitar
- **HoneyPot:** Sitio falso para la detección de robots

---

<sup>2</sup> Acrónimo para **C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part.

<sup>3</sup> Echad un ojo a <https://www.zenrows.com/>; ¿alguien se anima a hacer un TFG del tema?

# Pero... ¿es legal o no?<sup>4</sup>

---

Anda en un limbo gris entre la legalidad y la no legalidad

- Las condiciones de uso de un sitio específico pueden prohibirlo
- Pero para que la extracción sea ilegal, debería ir en contra de una ley ya existente
- Sí podría haber problemas en caso de que se vulnerase la LGPD o la RGPD

Cuando se trata de ONG defensoras del acceso abierto, la cosa es más divertida

- ¿Cómo de ilegal es <https://archive.org/>?
- ¿Qué pasa con la investigación periodística cómo [Reuters](#), [Reveal](#) o [The Trace](#)?

En España, la Ley de Servicios de la Sociedad de la Información y de Comercio Electrónico (LSSI-CE) **no prohíbe explícitamente el web scraping**

---

<sup>4</sup> *I am not a lawyer* (IANAL), así que todo lo que digamos aquí hay que cogerlo con pinzas.

# Creación de robots software con Python

# La biblioteca `urllib`

Es la encargada de trabajar con URLs y de realizar peticiones HTTP

```
from urllib.request import urlopen, Request

request = Request('https://www.python.org/')
with urlopen(request) as response:
    received_bytes = response.read()
content = received_bytes.decode()
print(content[:72])
```

Este ejemplo realiza una petición `GET` al recurso indicado y devuelve su contenido

- Forma parte de la biblioteca estándar, por lo que no es necesario instalar nada
- Además de recuperar datos, incluye funciones para gestionar cookies y cabeceras

# Parámetros por GET

En el ejemplo anterior, la URL no incluía ningún parámetro

- En el caso de una petición GET, estos se añaden a la URL
- Sin embargo, hay que tener cuidado con la codificación de los caracteres especiales

Para ello, podemos usar la función `urlencode` de `urllib.parse`

```
from urllib.parse import urlencode

params = urlencode({'q': 'python'})
url = 'https://www.google.com/search?' + params
print(url)
```

Así nos aseguramos de que todo carácter especial es codificado correctamente



# Datos por POST

Una petición de tipo POST envía los parámetros en el cuerpo de la petición

- POST es más flexible que GET, ya que permite enviar datos binarios (e.g. ficheros)

Para enviar datos por POST, basta con crear la *request* con datos:

```
data = urlencode({'key': 'value'}).encode()  
request = Request('https://httpbin.org/anything', data=data)  
with urlopen(request) as response:  
    print(response.read().decode())
```

Esto no funciona si el cuerpo de datos es vacío

- Se puede solucionar añadiendo el argumento `method='POST'` al crear la *request*
- De hecho con este argumento se especifica el método de la petición

# Formularios y POST

---

Los formularios HTML se envían por POST por defecto

- Cada campo del formulario se envía como un parámetro diferente

Veamos los tipos más comunes:

- text, password, hidden, textarea, select, radio: Se envía el *name/value* del campo
- checkbox: Se envían tantos *name/value* como checkboxes estén marcados

Los formularios se pueden enviar también por GET

- En este caso, los parámetros se añaden a la URL

# ¿Y si queremos enviar ficheros?

De hecho, es una de las características de POST frente a GET

- Es similar al caso anterior, pero enviando el contenido de un fichero
- Es útil añadir ciertas cabeceras para indicar el tamaño y el tipo de contenido

Por ejemplo, para enviar una imagen PNG:

```
import os

image_path = 'images/python.png'
image_size = os.path.getsize(image_path)
image_data = open(image_path, 'rb')
request = Request('https://httpbin.org/anything', data=image_data)
request.add_header('Content-Length', image_size)
request.add_header('Content-Type', 'image/png')
with urlopen(request) as response:
    print(response.read().decode())
```

# ¿Cómo leemos las cabeceras de una respuesta?

---

Toda petición conlleva una respuesta con cabeceras y (a veces) datos

- La biblioteca `urllib` las comprime en un objeto `HTTPMessage`

```
request = Request('https://www.python.org/')  
with urlopen(request) as response:  
    print(response.headers)
```

Este objeto se comporta como un diccionario

- Con el método `get` podemos obtener el valor de la cabecera indicada

# Cabeceras útiles en *web scraping*

# Cabecera *Accept*

---

Indica al servidor web qué formato de datos entiende cliente

- Por ejemplo, si el cliente entiende HTML, JSON, XML, etc.

```
request = Request('https://httpbin.org/anything')  
request.add_header('Accept', 'application/json;q=0.9, */*;q=0.8')
```

El factor de calidad **q** indica la preferencia del cliente

- Se utiliza en las cabeceras que aceptan varios valores

Es importante que la cadena sea similar a la ofrecida por los navegadores

# Cabecera *Accept-Encoding*

---

Notifica al servidor web qué algoritmo de compresión utilizar para gestionar la petición

- Por ejemplo, si el cliente entiende `gzip`, `deflate`, `br`, etc.
- Dicho de otro modo, si el cliente y el servidor pueden comprimir la información

```
request = Request('https://httpbin.org/anything')  
request.add_header('Accept-Encoding', 'gzip, deflate, br')
```

Esta cabecera es especialmente útil para reducir el tamaño de las respuestas

- Menos tiempo de espera y menos ancho de banda, win-win para cliente y servidor

El *encoding* `br` (Brotli) se usa para identificar scrapers, pero `urllib` ya lo soporta

# Cabecera *Accept-Language*

---

Indica al servidor qué idiomas prefiere el cliente

- Se puede especificar un único idioma o una lista de idiomas
- Entra en juego cuando los servidores web no pueden identificar el idioma preferido

```
request = Request('https://httpbin.org/anything')  
request.add_header('Accept-Language', 'en-US, en;q=0.9, es;q=0.8')
```

Es un factor más a la hora de detectar comportamientos no humanos

- Puede ser útil hacer que los lenguajes se ajusten a la ubicación IP del cliente
- También es útil ajustar los valores de *q* para no destacar frente al resto de tráfico



# Cabecera *User-Agent*

Ayuda al servidor a identificar y adaptar el contenido al cliente

- Por ejemplo, si el cliente es un navegador web, un teléfono móvil, un reproductor multimedia, etc.

```
request = Request('https://httpbin.org/anything')  
request.add_header('User-Agent', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64 ...')
```

Suelen seguir el formato:

```
(<navegador>) (<sistema>) <plataforma> (<detalles plataforma>) <extensiones>
```

Es importante porque ayuda a enmascarar el scraper como un navegador web

- Si está mal formado, lo más normal es que el servidor lo bloquee

# Cabecera *Referer*<sup>5</sup>

---

Indica la URL de la página desde la que se ha hecho la petición

```
request = Request('https://httpbin.org/anything')  
request.add_header('Referer', 'https://www.google.com/')
```

Esta cabecera es importante para identificar patrones de uso en usuarios

- Y los usuarios no suelen entrar en páginas web de forma aleatoria
- Suelen venir de buscadores web, redes sociales, etétera

Especificar esta cabecera para que el agente parezca más «humano»

---

¿Os habéis fijado que está mal escrito? Debería ser *Referrer*. Más información en [https://en.wikipedia.org/wiki/HTTP\\_referer](https://en.wikipedia.org/wiki/HTTP_referer)

# Forma alternativa de establecer cabeceras

---

El posible especificar un diccionario con las cabeceras al construir la *request*:

```
request = Request(url, data=None, headers={  
    'Accept': 'application/json;q=0.9, */*;q=0.8',  
    'Accept-Encoding': 'gzip, deflate',  
    'Accept-Language': 'en-US, en;q=0.9, es;q=0.8'  
    # ...  
})
```

No es ni mejor ni peor, es simplemente otra forma de hacerlo

**Biblioteca requests**

# Sobre esta biblioteca

---

Simplifica mucho el trabajo con HTTP. Entre sus características destacan:

- Facilidad de uso de *cookies* para mantener sesiones
- Codificación automática del contenido de las respuestas
- Codificación automática de URL internacionalizadas y datos POST
- Facilidad en el uso de *proxies* y *certificados SSL*
- Soporte para *streaming* de datos
- Implementación automática de conexiones persistentes (*keep-alive*)

Eso sí, es una biblioteca de terceros, así que habrá que instalarla aparte:

```
pip install requests
```

Documentación oficial en <https://requests.readthedocs.io>

# ¿Cómo realizo una petición HTTP?

Tan sencillo como usar la función `get` de la biblioteca:

```
response = requests.get('https://httpbin.org/anything')
```

De hecho, cada método tiene su propia función:

- `get()`, `post()`, `put()`, `patch()`, `delete()`, `head()`, `options()`

Enviar parámetros también es sencillo, basta con usar el diccionario `params`:

```
response = requests.get('https://httpbin.org/anything', params={'p1':'v1', 'p2':'v2'})
```

Y establecer un tiempo máximo de respuesta con el parámetro `timeout`:

```
response = requests.get('https://httpbin.org/anything', timeout=5)
```

# Envío de datos en peticiones POST/PUT/PATCH

---

En este caso, habría que utilizar el parámetro `data`:

```
response = requests.post('https://httpbin.org/anything', data={  
    'param1': 'val1',  
    'param2': 'val2'  
})  
response.text
```

No es necesario codificar los datos, la biblioteca lo hace por nosotros

# ¿Y si necesitamos especificar el tipo de contenido?

---

En ese caso jugamos con las cabeceras:

```
response = requests.post('https://httpbin.org/anything', data={  
    'param1': 'val1',  
    'param2': 'val2'  
}, headers={'Content-Type': 'text/xml'})  
response.text
```

¡E incluso con los datos!

```
response = requests.post('https://httpbin.org/anything',  
    data='<?xml version="1.0" encoding="UTF-8"?><soap:Envelope...',  
    headers={'Content-Type': 'text/xml'})  
response.text
```



# ¿Y si necesitamos enviar un fichero?

---

Pues nada, para eso tenemos el parámetro `files`:

```
response = requests.post('https://httpbin.org/anything', files={  
    'file1': open('file1.txt', 'rb'),  
    'file2': open('file2.txt', 'rb')  
})
```

Aunque también podemos especificar explícitamente el nombre y tipo del fichero:

```
files = {'f1': ('cosa.pdf', open('algo.pdf', 'rb'), 'application/pdf')}  
response = requests.post('https://httpbin.org/anything', files=files)
```

# Obteniendo información de la respuesta

---

El objeto de respuesta tiene muchos miembros interesantes, como lo son:

- `status_code`: Código de estado HTTP
- `headers`: Diccionario con las cabeceras de la respuesta
- `cookies`: Diccionario con las cookies de la respuesta
- `content`: Contenido de la respuesta
- `encoding`: Codificación de la respuesta (se puede establecer)
- `text`: Contenido de la respuesta en formato texto (decodificado)
- `history`: Información de todas las redirecciones que han ocurrido
  - Aunque se puede especificar `allow_redirects = False` para evitar redirecciones
- `json()`: Contenido de la respuesta en formato JSON
- `raise_for_status()`: Lanza una excepción si el código de estado no es 200

# ¿Cómo se pueden especifican las cabeceras?

---

Cualquier método de petición acepta un parámetro `headers`:

```
response = requests.get('https://httpbin.org/anything', headers={
    'Accept': 'application/json;q=0.9, */*;q=0.8',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'en-US, en;q=0.9, es;q=0.8'
})
response.headers
```

# ¿Cómo enviar *cookies* en una petición?

Aunque se puede perfectamente con `urllib`, con `requests` es mucho más sencillo:

```
jar = requests.cookies.RequestsCookieJar()  
jar.set('cookie1', 'v1', domain='httpbin.org', path='/')  
jar.set('cookie2', 'v2', domain='httpbin.org', path='/anything')  
response = requests.get('https://httpbin.org/anything', cookies=jar)  
response.text
```

---

<sup>5</sup> De hecho, trabajar con el objeto `CookieJar` de `urllib` es bastante pesado, por eso ni lo comentamos

# ¿Cómo mantener sesiones?

---

```
session = requests.Session()  
session.get('https://httpbin.org/cookies/set?cookie1=val1')  
session.get('https://httpbin.org/cookies/set?cookie2=val2')  
response = session.get('https://httpbin.org/cookies')  
response.text
```

# Autenticación

---

La biblioteca permite autenticarse con varios métodos:

```
auth = HTTPBasicAuth('user', 'passwd') # HTTPDigestAuth, HTTPProxyAuth, ...  
# auth =  
response = requests.get('https://httpbin.org/hidden-basic-auth/user/passwd', auth=auth)
```

# ¿Cómo podemos usar *proxies*?

```
proxies={'http': 'http://ip:puerto', 'https': 'https://ip:puerto'}  
requests.get('https://httpbin.org/anything', proxies=proxies)
```

Para mantenerlos en una sesión, se puede hacer de la siguiente forma:

```
proxies = {'http': 'http://ip:puerto', 'https': 'https://ip:puerto'}  
session = requests.Session()  
session.proxies.update(proxies)  
session.get('https://httpbin.org/anything')
```

## BONUS TRACK: Peticiones a través de Tor

---

Podemos usar los proxies configurados en la máquina para hacer peticiones a través de Tor:

```
proxies = {  
    'http': 'socks5://127.0.0.1:9050',  
    'https': 'socks5://127.0.0.1:9050'  
}  
requests.get('https://httpbin.org/anything', proxies=proxies)
```



**Extrayendo información del contenido**

# Introducción

---

Cuando tenemos el contenido descargado, tenemos varias formas de trabajar con él:

- Buscando y extrayendo subcadenas con los métodos de `string`
  - E inmediatamente después haremos como que no lo hemos pensado
- Expresiones regulares: biblioteca `re`
- XPath: Lenguaje de consulta de documentos XML
- HTQL: Lenguaje de consulta de documentos HTML

Veamos una pequeña introducción a cada uno de ellos

Extrayendo información del contenido

# Expresiones regulares

# Expresiones regulares

---

Mecanismo muy potente de definir lenguajes (y estructuras) de tipo 3

- Son muy **compactas** y (relativamente) **fáciles** de entender
- Prácticamente **cualquier lenguaje de programación** las soporta

Son cadenas de texto que intentamos que "encajen" con el texto que queremos extraer

- Encajar (*match*) significa que la cadena a extraer cumple una serie de condiciones
  - `a href=` encaja exactamente con ese texto
  - `a.*=` encaja con cualquier texto que empiece por `a`, 0 o más caracteres y un `=`, como `a href=`, `a class=`, `a id=`, etc.

Hay mucho escrito sobre expresiones regulares, aquí veremos lo **muy** básico

- Disponible en <http://docs.python.org/library/re.html>

# Metacaracteres

---

- `.`: Cualquier carácter
- `*`: La expresión precedente se repite 0 o más veces
- `+`: La expresión precedente se repite 1 o más veces
- `?`: La expresión precedente es opcional
- `{n}`: La expresión precedente se repite exactamente `n` veces
- `{n, }`: La expresión precedente se repite al menos `n` veces
- `{n, m}`: La expresión precedente se repite entre `n` y `m` veces
- `\n`: Salto de línea
- `\t`: Tabulador
- `\s`: Cualquier carácter de espaciado (blanco, tabulador, salto de línea, etc.)
- `^`: Distinto de
  - `^<*`: Cualquier carácter distinto de `<` 0 o más veces
- `\`: Carácter de escape para usar metacaracteres como caracteres normales
- `[]`: Conjunto de caracteres
  - `[abc]`, `[a-zA-Z]`, `[a-z0-9]`, etc.

# Encaje "voraz"

---

Por defecto, las expresiones regulares intentan encajar con el texto de la forma más larga posible

- `. *c` aplicada a `ababcababcb` devolverá `ababcababc` en lugar de `ababc`

Para hacer que no sea así, podemos usar el carácter `?` después del `*`, del `+` o del `?`

- `. *?c` aplicada a `ababcababcb` devolverá `ababc`

En *scraping* se suele usar siempre el encaje no voraz

# Comprobar existencia de un texto

Podemos determinar la existencia de cierto texto o patrón mediante `search`

```
import re

texto = 'ababcababcbab'
patron = 'a.*?c'
print('Encontrado' if re.search(patron, texto) else 'No encontrado')
```

Si queremos recuperar el texto, tenemos que usar "grupos" (con paréntesis)

```
texto = 'ababcababcbab'
patron = 'a(.*?)c'
if m := re.search(patron, texto):
    print(f'Encontrado: {m.group(1)}')
else:
    print('No encontrado')
```

# ¿Qué pasa si buscamos muchos patrones?

Pues que usaremos `findall` en lugar de `search`

```
texto = 'ababcbababcbab'  
patron = 'a(.*?)c'  
print(re.findall(patron, texto))
```

En lugar de un objeto *match* nos devuelve una lista de cadenas

- Hay *flags* que vienen bien porque modifican el comportamiento de los metacaracteres:
  - `re.DOTALL`: `.` coincide con cualquier carácter, incluido el salto de línea `\n`
  - `re.MULTILINE`: `^` coincidirá con el comienzo de línea y `$` con el final de línea
  - `re.IGNORECASE`: Hace que las expresiones regulares sean insensibles a mayúsculas y minúsculas



# Ganando eficiencia con expresiones regulares

---

Las expresiones regulares se compilan para ejecutarse sobre un texto

- Cada vez que usamos uno de sus métodos (e.g. `search`, `findall`, ...) compilamos la expresión

Es recomendable compilar una expresión previamente si la vamos a usar mucho

```
expr = re.compile('a(.*)c')  
expr.search('ababcababcb')  
expr.findall('ababcababcb')
```

# Algunos consejos

---

Para extraer varios valores es muy cómodo dividir la cadena de la expresión en trozos

```
xtr = '.*?<a href="(.*?)"' # url  
xtr += '.*?<td>(.*?)</td>' # incidencia
```

Hay herramientas online que ayudan a explicar los patrones que necesitamos

- Por ejemplo, <https://www.regex101.com>

**Extrayendo información del contenido**

**XPath**

# ¿Qué es XPath?

Lenguaje de consola para navegar a través de documentos XML y HTML

- Desarrollado por el W3C como parte de la recomendación XSLT<sup>6</sup>
- Permite seleccionar nodos y elementos específicos dentro de un documento

Ofrece un mecanismo muy preciso para la extracción de datos de páginas web

- Permite seleccionar elementos basados en su nombre, atributos y posición.
- Admite operadores lógicos y aritméticos para filtrar y manipular los datos.
- Permite acceder a elementos muy anidados y a estructuras complejas de datos.

Para su uso, tenemos que instalar la biblioteca `lxml`

```
$ pip install lxml
```

---

<sup>6</sup> Recomendación oficial de la W3C en <https://www.w3.org/TR/1999/REC-xpath-19991116/>

# Un ejemplo de código

```
import request
from lxml import html

# Obtenemos el html de una página
page = requests.get('https://httpbin.org')
# La transformamos en un árbol de elementos
tree = html.fromstring(page.content)
# Seleccionamos un elemento específico usando XPath
titulo = tree.xpath('//title/text()')
```

Por ejemplo, para extraer los títulos de las noticias de la BBC:

```
page = requests.get('https://www.bbc.com/news')
tree = html.fromstring(page.content)
print(tree.xpath('//a[@class="gs-c-block-link__overlay-link"]/text()'))
```

Extrayendo información del contenido

**Biblioteca** `selenium`

# Sobre esta biblioteca

---

Nació como entorno de pruebas para aplicaciones web

- Su principal uso es el de automatizar comportamientos en navegadores web
- Automatiza comportamientos en navegadores...

En realidad, en la actualidad se usa como herramienta de scraping

- Es muy potente ya que trabaja directamente con navegadores
- Es de las pocas formas que tenemos de interpretar código del lado del cliente en nuestro agente

Más información en la [página oficial](#)

- Y en la documentación de [Python](#)

# ¿Cómo buscar elementos en una página?

Se apoya en dos métodos principales:

- `find_element`: Primer elemento que cumpla con el criterio de búsqueda
- `find_elements`: Lista de elementos que cumplen con el criterio de búsqueda

Se pueden buscar por varios criterios:

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

driver = webdriver.Firefox()
driver.get('https://httpbin.org/anything')

element = driver.find_element(By.ID, 'id')
element = driver.find_element(By.NAME, 'name')
element = driver.find_elements(By.CLASS_NAME, 'class')
element = driver.find_element(By.TAG_NAME, 'tag')
element = driver.find_element(By.CSS_SELECTOR, 'selector')
element = driver.find_element(By.XPATH, 'xpath://etiqueta[@atributo=valor]')
```



# ¿Cómo interactuar con los elementos?

---

Al emular el comportamiento en un navegador, podemos interactuar con los elementos de la siguiente forma:

- Hacer click sobre un elemento

```
button.click()
```

- Escribir sobre un elemento

```
text.send_keys('texto')
```

- Enviar un formulario

```
form.submit()
```

# Esperando a que pasen cosas...

---

Podemos esperar a que ocurran cosas en la página con el método `WebDriverWait`:

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

element = WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.ID, 'id'))
)
```

# Ejemplo: Extraer enlaces de una página

---

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

driver = webdriver.Firefox()
driver.get('https://httpbin.org/anything')

links = [
    a.get_attribute('href')
    for a in driver.find_elements(By.TAG_NAME, 'a')
]

print(links)
```

**Evitando bloqueos**

# Técnicas para evitar los bloqueos

---

Aunque algo hemos comentado, algunas técnicas para evitar bloqueos son:

- **Rotación de cadenas de *user-agent*** entre peticiones
  - Suele ser conveniente mantener el *user-agent* durante una misma sesión
- **Intervalos aleatorios entre solicitudes** para asemejarse a un usuario real
  - El intervalo puede ser directamente proporcional a la cantidad de contenido
- **Agentes de usuario actualizados**, porque los obsoletos son factores de bloqueo
- **Proxies** para evitar el bloqueo de la IP
- Uso de ***headless browsers***, como [PhantomJS](#), junto con bibliotecas como [Selenium](#)
- **Programar contra trampas de *HoneyPots***
  - E.g. Enlaces ocultos o con el mismo color de fondo que la página
- **Hacer caso de** la información del fichero **`robots.txt`**
- **Usar las API** de los servicios web para la extracción de datos
- Ordenar las cabeceras de la misma manera que los navegadores web

**Gracias**