1. Web scraping

Python y la web

Alberto Díaz Álvarez
29 de noviembre de 2024
El mejor curso de Python del mundo

Un poco de conocimientos previos

¿Qué es el *Hypertext Transfer Protocol* (HTTP)?

Diseñado en 1990, es la base de la comunicación de datos en la World Wide Web

- Protocolo a nivel de aplicación, sobre TCP/IP, que sigue el modelo cliente-servidor
- Su puerto estándar de comunicación asignado es el 80
- En la actualidad trabajamos con la versión HTTP/2[1] (RFC7540)
- Suficientemente abierto para no impedir el desarrollo de aplicaciones de terceros
  - Por ejemplo, agentes de información

HTTP se utiliza principalmente para transmitir recursos

- Recurso: fragmento de información identificable mediante una URL (de ahí la R)
- El tipo más común de recurso es un archivo
- También puede ser una salida generada dinámicamente (e.g. script CGI)

1 En abril de 2016, el tráfico servido por la compañía KeyCDN superó los dos tercios en HTTP/2 frente a HTTP/1.1, la versión que este reemplazaba https://www.keycdn.com/blog/http2-statistics.

Uniform Resource Locator (URL)

Es la forma de identificar recursos en la Web, y tiene la siguiente estructura:

```
protocolo://servidor:puerto/path/hasta/el/recurso?query=con&algun=parametro
```

Por ejemplo:

```
https://www.google.com/search?q=boniato+al+horno&sclient=gws-wiz-serp
```

Si no se especifica el puerto se usa el de por defecto (http: 80, https: 443)

Modelo cliente-servidor y el protocolo HTTP

Es un modelo de comunicación donde hay dos entidades:

- Cliente: solicita recursos al servidor
- Servidor: proporciona los recursos solicitados

Una sesión HTTP se inicia cuando el cliente envía una petición al servidor

1. El cliente establece una conexion TCP con el servidor
2. El cliente envía una petición (***request***) al servidor, y queda a la espera de la respuesta
    - Esta petición obedece a uno de los **métodos** definidos para el protocolo HTTP
3. El servidor procesa la petición y envía una respuesta (***response***) al cliente
    - Incluyendo, entre otros, un **código de estado**

Mensajes en HTTP

Toda comunicación HTTP entre dispositivos se basa en dos tipos de mensaje

- **Request**: mensaje enviado por el cliente al servidor
- **Response**: mensaje enviado por el servidor al cliente

Ambos mensajes tienen caracterísiticas comunes y diferencias:

- Ambos poseen cabeceras con metainformación (*headers*)
- Ambos pueden poseer un cuerpo adicional donde se envían datos
- La request indica el método a utilizar y el recurso al que se quiere acceder
- La response indica el código de estado de la respuesta

La estructura de los mensajes difiere bastante entre los protocolos 1.1 y 2.0

- Lo bueno es que con las bibliotecas de Python nos abstraemos de ello

Métodos definidos para el protocolo HTTP

- **GET**: solicita un recurso al servidor
- **HEAD**: solicita los metadatos de un recurso al servidor
- **POST**: envía datos al servidor
- **PUT**: envía un recurso al servidor
- **PATCH**: envía una actualización parcial de un recurso al servidor
- **DELETE**: elimina un recurso del servidor
- **CONNECT**: establece un túnel hacia el servidor identificado por el recurso
- **OPTIONS**: solicita los métodos que el servidor soporta para un recurso
- **TRACE**: realiza una prueba de bucle de retorno de mensaje al servidor

Esta es la teoría, porque luego muchas se implementan como se quiere

Códigos de estado (*status codes*)

Los códigos de estado son números de tres dígitos que indican el estado de la respuesta

- `1xx`: información
- `2xx`: éxito
- `3xx`: redirección
- `4xx`: error del cliente
- `5xx`: error del servidor

Al igual que los métodos, los códigos de estado son estándar, pero no obligatorios

Servicios web

Un servicio web es un conjunto de protocolos y estándares que permiten la comunicación entre aplicaciones

- Se basan en estándares abiertos que funcionan sobre el protocolo HTTP
- Su origen proviene de estándares de ejecución remota de código de los 90
- Un conjunto de servicios web con propósito común se denomina API (*Application Programming Interface*)
- Pueden estar documentados también para máquinas usando WSDL (*Web Services Description Language*)

En la actualidad, estos estándares están en desuso por el auge de la web

- La transmisión de datos es a través del puerto 80 o 443 (adios *firewalls*)
- Protocolos que funcionan sobre un estándar ya probado (SOAP)
- O simplemente conjuntos de pautas para hacer la comunicación más ligera (REST)

Acerca de REST

Representational State Transfer (REST) es un estilo de arquitectura de software

- No es un protocolo, es un conjunto de principios para la creación de servicios web

Sus principios arquitectónicos son:

- **Interfaz uniforme**: La información se envía en formato estándar (XML, JSON, etc.)
- **Sin estado**: El servidor completa solicitudes independientes (es inherente a HTTP)
- **Distribuido**: Permite que varios servidores trabajen juntos
- **Almacenamiento en caché**: Ayuda a mejorar el rendimiento
- **Código bajo demanda**: El servidor puede enviar código al cliente para ejecutarlo

Las API que siguen la arquitectura de REST se llaman **API RESTful**

API RESTful

Son aquellos **servicios web que implementan una arquitectura REST**

Requieren que las peticiones contengan los siguientes **elementos principales**:

- **URL**: La ruta hacia el recurso o servicio
- **Método**: La acción que se quiere realizar sobre el servicio, que suelen ser:
  - `GET`: Para obtener listas (e.g. `GET /games`) o elementos (e.g. `GET /game/1`)
  - `POST`: Para enviar información o crear elementos (e.g. `POST /game`)
  - `PUT` o `PATCH`: Para actualizar elementos (e.g. `PUT /game/1`)
  - `DELETE`: Para eliminar elementos (e.g. `DELETE /game/1`)
- **Cuerpo**: La información enviada al servidor (para peticiones `POST`, `PUT` o `PATCH`)
- **Cabeceras**: Información adicional de la petición, (e.g. datos de autenticación)

Es un estándar muy usado en la actualidad, pero no es obligatorio

Vamos a por el *web scraping*

¿Qué es el *web scraping*?

Proceso de **extraer información de sitios web** de forma automatizada

- 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 **más costoso** que usar una API y genera **más tráfico** entre cliente y servidor
- Es **muy 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
- **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***[2]: Imagen con texto para demostrar que quien accede es humano[3]
- **Fichero** `robots.txt`: Archivo que indica a los robots qué rutas no deben visitar
- **HoneyPot**: Sitio falso para la detección de robots

[2] Acrónimo para ***C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part*.
[3] Echad un ojo a https://www.zenrows.com/; ¿alguien se anima a hacer un TFG del tema?

Pero vamos a ver, ¿es legal o no?

Anda en un limbo gris entre la legalidad y la no legalidad[4]

- 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 peridística 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**

[4] *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*[6]

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»

[6] ¿Os habéis fijado que está mal escrito? Debería ser *Referrer*. Más información en 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[7]:

```
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
```

[7] 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

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

- A mano, con los métodos de `string` (i.e. subcadenas, `split`, ... ni os molestéis)
- Con expresiones regulares, con la biblioteca `re`
- Con bibliotecas para scraping estructurado
  - `XPath`: Lenguaje de consulta de documentos XML
  - `BeautifulSoup` para procesamiento de HTML y XML
  - `Scrapy`, que es un framework completo para *web scraping* (incluye *crawling*)
- Con bibliotecas para scraping no estructurado
  - `Selenium`, que emula un navegador web real
  - `Playwright`, alternativa a Selenium, más moderna y rápida
  - `pypeteer`, una versión de Python de Puppeteer

Veamos una pqueña introducción a cada uno de ellos

Expresiones regulares

Extrayendo información del contenido

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 a 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 `ababcababcab` devolverá `ababcababc` en lugar de `ababc`

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

- `.*?c` aplicada a `ababcababcab` 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 = 'ababcababcab'
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 = 'ababcababcab'
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 = 'ababcababcab'
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 comporamiento 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('ababcababcab')
expr.findall('ababcababcab')
```

Algunos consejos

Para extraer varios valores es muy cómodo dividir 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

XPath

Extrayendo información del contenido

¿Qué es XPath?

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

- Desarrollado por el W3C como parte de la recomendación XSLT[8]
- 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
```

8 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()'))
```

Biblioteca `BeautifulSoup`

Extrayendo información del contenido

Sobre esta biblioteca

Es una biblioteca de Python para extraer datos de archivos HTML y XML

- Crea un árbol de análisis a partir del código fuente de la página
- Proporciona métodos para navegar y buscar en el árbol

Sus ventajas principales son:

- Es muy fácil de usar y muy intuitiva
- Permite manejar documentos que no están del todo bien formados

Para instalarla, simplemente ejecutamos:

```
$ pip install beautifulsoup4
```

Búsqueda de elementos complejos

Supongamos que queremos extraer datos de estructuras HTML complejas utilizando combinaciones de selectores CSS.

```
from bs4 import BeautifulSoup

html = """
<div class="producto">
    <span class="precio">$10</span><span class="precio">$20</span>
</div>
"""

soup = BeautifulSoup(html, "html.parser")
precios = soup.select("div.producto > span.precio")  # Selectores CSS
for precio in precios:
    print(precio.text)
```

- Podemos usar combinaciones como `#id`, `.clase`, `elemento:atributo`

Navegación por la estructura DOM

Podemos navegar por la estructura del documento HTML como si fuera un árbol

```
html = """
<div class="producto">
    <h3>Producto 1</h3>
    <p>Descripción del producto.</p>
</div>
"""


soup = BeautifulSoup(html, "html.parser")
div = soup.find("div", class_="producto")  # Encuentra el primer div de clase 'producto'
titulo = div.find_next("h3").text  # Encuentra el siguiente nodo h3 dentro del div
print(titulo)
```

- `find_next()`: Encuentra el siguiente nodo en el árbol DOM[9]
- También puedes usar `find_previous()` y `find_parent()` para moverte hacia atrás

9 El *Document Object Model* (DOM) define la estructura de un fichero HTML como un árbol. Más información en la Wikipedia.

Estraer información estructurada de tablas

```
html = """
<table>
    <tr><th>Nombre</th><th>Edad</th></tr>
    <tr><td>Andrea</td><td>25</td></tr>
    <tr><td>Carmela</td><td>30</td></tr>
</table>
"""

soup = BeautifulSoup(html, "html.parser")
tabla = soup.find("table")
for fila in tabla.find_all("tr"):
    datos = fila.find_all("td")
    if datos:
        nombre, edad = datos
        print(nombre.text, edad.text)
```

Biblioteca `Scrapy`

Extrayendo información del contenido

Sobre esta biblioteca

Framework de Python diseñado específicamente para *crawling* y_scraping_

- Diseñado para ser extensible y escalable
- Escrito en Python y basado en `Twisted`, un framework de red asíncrono
- Automatiza tareas como seguir enlaces, manejar sesiones y procesar datos
- Integra herramientas para exportar a formatos comunes (e.g. JSON o CSV)

Para instalarlo, simplemente ejecutamos:

```
$ pip install scrapy
```

Estructura de un proyecto Scrapy

Un proyecto Scrapy se compone de varios elementos:

- **Spiders**: Clases que definen cómo se extraen los datos de un sitio web
- **Items**: Clases que definen los datos que se van a extraer
- **Pipelines**: Clases que definen cómo se procesan los datos extraídos
- **Settings**: Configuración del proyecto
- **Middlewares**: Componentes que procesan las peticiones y respuestas

Creación de un proyecto Scrapy

Para crear un proyecto Scrapy, ejecutamos el siguiente comando:

```
$ scrapy startproject my_project
```

Esto creará una estructura de directorios como la siguiente:

```
my_project/
    scrapy.cfg
    my_project/
        __init__.py
        items.py
        middlewares.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
```

Creación de un *spider*

```
import scrapy

class EjemploSpider(scrapy.Spider):
    name = "ejemplo"
    start_urls = ["https://example.com"]

    def parse(self, response):
        for producto in response.css("div.producto"):
            yield {
                "nombre": producto.css("h3::text").get(),
                "precio": producto.css("span.precio::text").get(),
            }
```

Para ejecutarlo:

```
$ scrapy crawl ejemplo -o productos.json
```

Biblioteca `selenium`

Extrayendo información del contenido

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
  - Permite el uso de ***headless browsers***, como PhantomJS
- 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 comporatamiento 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)
```

Biblioteca `pyppeteer`

Extrayendo información del contenido

Sobre esta biblioteca

Biblioteca para controlar un navegador web a través de la API de Puppeteer

- Es un clon de `puppeteer` de Node.js adaptado para Python

Sus usos principales son:

1. Scraping avanzado en páginas web que requieren JavaScript
2. Testing de interfaces web
3. Emulación de dispositivos móviles, condiciones de red, geolocalización, etc.

Para instalarlo, simplemente ejecutamos:

```
$ pip install pyppeteer
```

Ejemplo: Carga de una página y extracción de contenido

```
import asyncio
from pyppeteer import launch

async def main():
    # Lanzamos el navegador
    browser = await launch(headless=True)
    page = await browser.newPage()

    # Vamos a la URL
    await page.goto('https://example.com')

    # Extraemos el título de la página
    title = await page.title()
    print(f"Título de la página: {title}")

    # Extraemos contenido de un elemento
    content = await page.querySelectorEval('h1', 'element => element.textContent')
    print(f"Contenido H1: {content}")

    await browser.close()

asyncio.run(main())
```

Ejemplo: Emulación de dispositivo móvil

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch()
    page = await browser.newPage()

    # Emulamos un dispositivo móvil
    await page.emulate({
        'name': 'iPhone X',
        'viewport': {'width': 375, 'height': 812, 'isMobile': True},
        'userAgent': 'Mozilla/5.0 (iPhone; CPU iPhone OS 13_3 ...',
    })

    # Vamos a una página con el móvil
    await page.goto('https://example.com')
    await page.screenshot({'path': 'mobile_view.png'})

    await browser.close()

asyncio.run(main())
```

Ejemplo: Manejo de contenido dinámico

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch()
    page = await browser.newPage()

    # Vamos a una página que carga contenido dinámico
    await page.goto('https://example.com')

    # Esperamos a que un elemento específico esté cargado
    await page.waitForSelector('.dynamic-content')

    # Extraemos contenido dinámico
    content = await page.evaluate("""() => {
        return document.querySelector('.dynamic-content').innerText;
    }""")
    print(f"Contenido dinámico: {content}")

    await browser.close()

asyncio.run(main())
```

Ejemplo 4 - Rellenar formularios y enviar datos

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch()
    page = await browser.newPage()

    # Vamos a un formulario
    await page.goto('https://example-form.com')

    # Completamos campos del formulario y enviamos
    await page.type('#username', 'mi_usuario')
    await page.click('#submit')

    # Esperamos la carga de la siguiente página
    await page.waitForNavigation()

    print("Formulario enviado y página cargada")

    await browser.close()

asyncio.run(main())
```

Unos pocos goodies

Deshabilitar características que identifican scripts automatizados

```
await page.evaluateOnNewDocument("""() => {
    Object.defineProperty(navigator, 'webdriver', {get: () => undefined});
}""")
```

Tomar capturas de pantalla en puntos críticos para validar el scraping

```
await page.screenshot({'path': 'debug.png'})
```

Desactivar la carga de imágenes o recursos pesados para acelerar el scraping

```
await page.setRequestInterception(True)
page.on('request', lambda req: req.continue_() if req.resourceType != 'image' else req.abort())
```

Técnicas avanzadas

(y buenas prácticas)

De qué va esto

Hasta ahora hemos visto cómo extraer información de una página web

- Pero no todo son flores y alegría

Vremos algunas técnicas para superar los **problemas más comunes** en *scraping*

- Evitar bloqueos de IP
- Manejar *captchas* y contenido dinámico
- Optimizar el rendimiento en *scraping* masivo

También hablaremos de **buenas prácticas** en nuestros proyectos

- Sobre todo para asegurar eficiencia, escalabilidad y sostenibilidad

¿Cuáles son esos problemas más comunes?

En realidad problemas hay muchos, porque esto es como el gato y el ratón

1. Hoy encontramos una técnica para evitar bloqueos
2. Mañana los servidores web se adaptan y nos bloquean de nuevo
3. `GOTO 1`

Pero hay algunos problemas que son comunes, y no solo son de bloqueo:

Bloqueo

- Filtrado de IP sospechosas
- *Rate limiting*/*captchas*

Dinamismo

- AJAX o *WebSockets* (asíncronos)
- JavaScript

Muchos datos

- Necesidad de paralelismo y proceso
- Almacenamiento

Restricciones de *rate limiting* (I)

Demasiadas solicitudes en poco tiempo puede activar bloqueos

- Para ello podemos implementar tiempos de espera entre solicitudes

```
import time
import requests

for url in urls:
    response = requests.get(url)
    print(response.status_code)
    time.sleep(1)  # Esperar 1 segundo
```

Si implementamos un tiempo de espera aleatorio, mejor

Restricciones de *rate limiting* (II)

Hay herramientas específicas para ello, como `ratelimit`

```python
from ratelimit import limits, sleep_and_retry
import requests

@sleep_and_retry
@limits(calls=10, period=60)  # Limitar a 10 solicitudes por minuto
def mega_request(url):
    response = requests.get(url)
    return response.text

urls = ["https://example.com"] * 15  # Más solicitudes que el límite
for url in urls:
    try:
        print(mega_request(url))
    except Exception as e:
        print(f"Error: {e}")
```

Resolución de *captchas*

Método para evitar el acceso a según que sitios de sitios web

Flujo básico

1. Descargar el captcha como imagen
2. Enviarlo al servicio externo
3. Recibir el token como respuesta

Porque lo normal es usar servicios

- (e.g. 2Captcha o Anti-Captcha)

Ejemplo con 2Captcha

```python
import requests

APIKEY = "Api key de 2Captcha"
captcha_url = "https://example.com/captcha.jpg"

response = requests.post("http://2captcha.com/in.php", data={
    "key": APIKEY,
    "method": "base64",
    "body": requests.get(captcha_url).content.encode("base64")
})

c_id = response.text.split("|")[1]
url = "http://2captcha.com/res.php?key={APIKEY}&action=get&id={c_id}"

solution = None
while not solution:
    result = requests.get(url)
    if "OK" in result.text:
        solution = result.text.split("|")[1]
```

Rotación de *proxies* para evitar bloqueos por IP (I)

Diferentes *proxies* para que las solicitudes lleguen de diferentes ubicaciones

```
import requests

proxies = [
    {"http": "http://px1.example.com:8080", "https": "https://px1.example.com:8080"},
    {"http": "http://px2.example.com:8080", "https": "https://px2.example.com:8080"}
]
# Rotar proxies en cada petición
for proxy in proxies:
    try:
        response = requests.get("https://example.com", proxies=proxy, timeout=5)
        print(f"Respuesta desde {proxy['http']}: {response.status_code}")
    except Exception as e:
        print(f"Error con proxy {proxy['http']}: {e}")
```

Recomendación: Mantener sesiones, usar selección aleatoria y paralelismo

- Los procesos secuenciales son los más fáciles de detectar

Rotación de *proxies* para evitar bloqueos por IP (II)

Ahora un ejemplo con `Scrapy`:

1. Configuración del fichero `settings.py`:

```
DOWNLOADER_MIDDLEWARES = {
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 110,
 ' my_project.middlewares.ProxyMiddleware': 100,
}
```

2. Implementación del *middleware*:

```
import random

class ProxyMiddleware:
    def process_request(self, request, spider):
        request.meta['proxy'] = random.choice(spider.settings.get('PROXIES'))
```

Rotación de *proxies* para evitar bloqueos por IP (III)

Podemos usar servicios como `ScraperAPI` o `BrightData` para evitar bloqueos

- Ofrecen *proxies* rotativos y *captchas* resueltos
- También ofrecen *headless browsers* para evitar *captchas* y contenido dinámico

ScraperAPI

```python
import requests

# Configuración
SCRAPERAPI_KEY = "API_KEY"
BASE_URL = "http://api.scraperapi.com"
API_ENDPOINT = f"{BASE_URL}?api_key={SCRAPERAPI_KEY}&url="

# URL del sitio a scrapear
target_url = "https://example.com"

# Hacemos una solicitud
response = requests.get(API_ENDPOINT + target_url)
if response.status_code == 200:
    print(response.text)
else:
```

BrightData

```python
import requests

# Configuración
BD_USER = "USERNAME"
BD_PASSWORD = "PASSWORD"
proxies = {
    "http": f"http://{BD_USER}:{BD_PASSWORD}@zproxy.lum-superproxy.io:22225",
    "https": f"http://{BD_USER}:{BD_PASSWORD}@zproxy.lum-superproxy.io:22225"
}

# URL del sitio a scrapear
target_url = "https://example.com"

# Hacemos una solicitud
response = requests.get(target_url, proxies=proxies)
if response.status_code == 200:
    print(response.text)
else:
    print(f"Error: {response.status_code}")
```

Rotación de `User-Agent` para evitar bloqueos

Es muy común bloquear peticiones de clientes que no tienen un *User-Agent* válido

- Tan fácil como cambiarlo en cada petición, pero estos cambian con el tiempo
  - Y los obsoletos son carne de cañón para los bloqueos
- Podemos usar la biblioteca `fake_useragent` para obtener *User-Agents* aleatorios

```
from fake_useragent import UserAgent
import requests

ua = UserAgent()
headers = {"User-Agent": ua.random}

response = requests.get("https://example.com", headers=headers)
print(response.status_code)
```

Recomendación: Mantener el mismo *User-Agent* durante una sesión

Sobre el contenido dinámico

El contenido dinámico es aquel que aparece tras la carga de la página

- Diferentes formas: *AJAX*, *WebSockets*, *JavaScript*, etc.
- Ejemplo: Resultados de búsqueda que aparecen al desplazarse (i.e. *infinite scroll*)

Las soluciones son variadas, pero las más comunes son:

- Usar herramientas que interactúen con JavaScript:
  - Navegadores automatizados: Selenium, Playwright
  - `Scrapy` con middlewares para manejar contenido dinámico
- Análisis de peticiones de red para **acceder directamente a las API subyacentes**

Automatización de navegadores con Selenium (I)

Esperaremos unos segundos para que se cargue la página

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

# Configuración del navegador
driver = webdriver.Chrome()
driver.get("https://example.com/dinamico")

# Esperar a que el contenido dinámico cargue
driver.implicitly_wait(10)
elementos = driver.find_elements(By.CLASS_NAME, "resultado")

for elemento in elementos:
    print(elemento.text)

driver.quit()
```

Automatización de navegadores con Selenium (II)

Algunas páginas tardan más en cargar el contenido dinámico

- Podemos esperar a que un elemento específico aparezca en la página

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

driver = webdriver.Chrome()
driver.get("https://example.com/dinamico")

WebDriverWait(driver, 20).until(EC.presence_of_element_located((By.ID, "res")))
resultados = driver.find_elements((By.ID, "res"))
for resultado in resultados:
    print(resultado.text)

driver.quit()
```

Automatización de navegadores con Selenium (III)

Muchas páginas cargan contenido dinámico al ir hacia abajo (e.g. *infinite scroll*)

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

driver = webdriver.Chrome()
driver.get("https://example.com/infinite-scroll")

for _ in range(5):  # Nos desplazamos hacia abajo 5 veces
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    time.sleep(2)  # Esperamos a que se cargue el nuevo contenido

resultados = driver.find_elements(By.ID, "res")
for resultado in resultados:
    print(resultado.text)

driver.quit()
```

Automatización de navegadores con Selenium (IV)

Las web dinámicas suelen hacer uso de API para cargar contenido

- Podemos cogerlo de ahí directamente, suele ser más rápido y estable

```
import requests

# Solicitud directa a la API interna
url = "https://example.com/api/resultados"
params = {"page": 1}
response = requests.get(url, params=params)
data = response.json()

for item in data["resultados"]:
    print(item["nombre"])
```

Sí, se puede hacer con selenium, pero no hace falta matar moscas a cañonazos

Problemas con el *scraping* secuencial

El *scraping* secuencial es lento y poco eficiente:

- Prácticamente todos los proyectos requieren muchas solicitudes
- La memoria se llena si no se procesan los datos de inmediato
- Incurrimos en riesgo de bloqueo si hacemos muchas solicitudes desde una sola IP

¿Cómo podemos solucionar (o al menos aliviar un poco) esto?

- Procesamiento paralelo
  - Básicamente, realizar múltiples peticiones simultáneamente
- Procesamiento en tiempo real
  - Procesar los datos a medida que están disponibles, evitando la acumulación
- Almacenamiento escalable
  - Guardar los datos en un sistema de almacenamiento adecuado (e.g. mongoDB)

Procesamiento paralelo

Podemos usar `concurrent.futures` para realizar peticiones en paralelo

```
import requests
from concurrent.futures import ThreadPoolExecutor

def get_url(url):
    response = requests.get(url)
    return {"url": url, "status_code": response.status_code}

urls = ["https://example.com"] * 10
with ThreadPoolExecutor(max_workers=5) as ex:  # Ajustar para no saturar
    results = ex.map(get_url, urls)

for result in results:
    print(result)
```

Scraping masivo con `Scrapy` (I)

Configuración para paralelismo eficiente (`settings.py`):

1. Aumentar concurrencia en el agente

```
CONCURRENT_REQUESTS = 100
DOWNLOAD_DELAY = 0.25  # Reducir tasa de peticiones al servidor
```

2. Añadir *middlewares* para manejar contenido dinámico

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 110,
    'my_project.middlewares.ProxyMiddleware': 100,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
    'scrapy_user_agents.middlewares.RandomUserAgentMiddleware': 400,
}
```

Scraping masivo con `Scrapy` (II)

Ahora el agente `Scrapy` puede manejar muchas solicitudes simultáneas

```
import scrapy

class MasivoSpider(scrapy.Spider):
    name = "masivo"
    start_urls = ["https://example.com"]

    def parse(self, response):
        for producto in response.css("div.producto"):
            yield {
                "nombre": producto.css("h3::text").get(),
                "precio": producto.css("span.precio::text").get(),
            }
        next_page = response.css("a.next::attr(href)").get()
        if next_page:
            yield response.follow(next_page, self.parse)
```

Procesamiento en tiempo real

Podemos minimizar le uso de memoria procesando los datos a medida que llegan

```
def scrape_data():
    urls = ["https://example.com/page1", "https://example.com/page2"]
    for url in urls:
        response = requests.get(url)
        yield {"url": url, "status_code": response.status_code}

for data in scrape_data():
    print(data)
```

El uso de generadores (`yield`) permite procesar los datos a medida que llegan

Almacenamiento escalable (I)

Podemos conectar con una base de datos moderna como `MongoDB` para almacenar los datos

```
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["scraping_db"]
collection = db["productos"]

# Insertar datos
producto = {"nombre": "Producto 1", "precio": 10}
collection.insert_one(producto)

# Consultar datos
for doc in collection.find():
    print(doc)
```

Almacenamiento escalable (II)

MongoDB no es la única opción, hay muchas otras como por ejemplo Redis

- BBDD en memoria $\rightarrow$ Útil para almacenamiento temporal, colas de tarea y caché

```python
import redis

# Conectar a Redis
r = redis.Redis(host='localhost', port=6379, db=0)

# Guardar datos
r.set("producto_1", "Producto 1: $10")
print(r.get("producto_1").decode("utf-8"))
```

Ejemplo: Scraper en paralelo con MongoDB

Un ejemplo de extracción y almacenamiento de datos de múltiples páginas

```
from concurrent.futures import ThreadPoolExecutor
import requests
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")  # Configurar MongoDB
db = client["scraping_db"]
collection = db["productos"]

def scrape_and_store(url):  # Función de scraping
    response = requests.get(url)
    if response.status_code == 200:
        data = {"url": url, "content": response.text}
        collection.insert_one(data)

urls = ["https://example.com/page1", "https://example.com/page2", "https://example.com/page2"]

with ThreadPoolExecutor(max_workers=5) as executor:  # Procesamiento en paralelo
    executor.map(scrape_and_store, urls)
```

Gracias