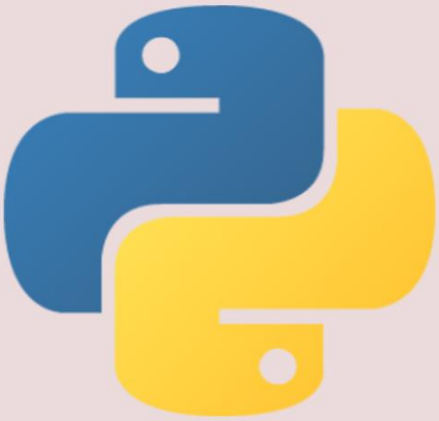
6.1.09

www.opentowork.es

Python_

Web Scraping

El web scraping es una técnica de programación que implica extraer datos de sitios web de manera automatizada. Es como un "raspado" de la información de una página web. Esta técnica se usa para recopilar información de manera eficiente y estructurada para su análisis, almacenamiento u otros fines.



Python

Web Scraping

61TE01_09_Web scraping con Python

Indice

61TE01_09_Web scraping con Python

8. Almacenamiento y Análisis de Datos

8.1 Guardado de datos en formatos comunes (CSV, JSON)

8.2 Almacenamiento en bases de datos

8.3 Análisis preliminar de datos extraídos

9. Ejemplos Prácticos

61TE01_09_Web scraping con Python

1. Introducción al Web Scraping

1.1 ¿Qué es el web scraping?

El web scraping es el proceso de usar bots o programas automatizados para extraer datos de páginas web. Este proceso implica enviar solicitudes HTTP a un sitio web, obtener la respuesta en forma de HTML, analizar el contenido y extraer la información necesaria. Es una alternativa poderosa a la extracción manual de datos, ya que permite recopilar grandes cantidades de información en un período de tiempo relativamente corto.

El web scraping es particularmente útil cuando la información no está disponible en un formato estructurado o cuando no se proporciona una API oficial para el acceso a datos. Los datos obtenidos se pueden transformar en formatos estructurados para su posterior análisis, como archivos CSV, bases de datos SQL, o archivos JSON.

1.2 Aplicaciones comunes del web scraping

- **Monitoreo de precios y productos:** Herramientas de comparación de precios y productos usan web scraping para recopilar datos de múltiples tiendas en línea, permitiendo a los usuarios comparar precios y características.
- **Agregadores de contenido:** Portales que recopilan contenido como noticias, reseñas o eventos de múltiples fuentes para ofrecer a los usuarios una vista consolidada de información.
- **Análisis de mercado:** Empresas de análisis de mercado recopilan datos de diversas fuentes para entender las tendencias de mercado, comportamientos del consumidor y nuevas oportunidades comerciales.
- **Extracción de datos para inteligencia competitiva:** Empresas recopilan información de la competencia, como productos, precios y reseñas, para entender su posición en el mercado y planificar estrategias.
- **Investigación académica y científica:** Los investigadores pueden recopilar grandes conjuntos de datos de forma automatizada para estudios de tendencias, análisis de texto y más.

1.3 Consideraciones éticas y legales

El web scraping debe ser realizado de manera ética y legal, considerando los siguientes puntos:

- **Respeto a los Términos de Servicio:** Muchos sitios web prohíben explícitamente el web scraping en sus Términos de Servicio. Es fundamental revisar estos términos antes de proceder.
- **Privacidad de los datos:** El acceso a datos personales sin consentimiento puede violar leyes de privacidad. Es fundamental asegurarse de que los datos extraídos no infrinjan la privacidad de las personas.
- **Carga en los servidores:** Las solicitudes masivas pueden sobrecargar los servidores y afectar su rendimiento. Implementar limitaciones en la frecuencia de solicitudes es crucial para evitar problemas.
- **Cumplimiento de leyes y regulaciones:** Muchas regiones tienen regulaciones estrictas sobre el uso y almacenamiento de datos. Por ejemplo, el Reglamento General de Protección de Datos (GDPR) en Europa impone fuertes restricciones en el manejo de datos personales.

El web scraping es una técnica potente que, cuando se usa correctamente y de manera ética, puede ofrecer grandes beneficios en la recopilación y análisis de datos.

61TE01_09_Web scraping con Python

2. Conceptos Básicos y Herramientas

Para obtener buenos resultados con el web scraping, es crucial comprender los conceptos básicos y las herramientas que se utilizan para extraer datos de los sitios web de manera eficiente.

2.1 Protocolo HTTP y solicitudes web

El Protocolo de Transferencia de Hipertexto (HTTP) es el estándar que permite la comunicación entre navegadores y servidores web. Cada vez que un usuario accede a un sitio web, el navegador realiza una solicitud HTTP al servidor que aloja el sitio y recibe una respuesta en forma de contenido HTML.

- **Métodos HTTP:** Los métodos más comunes son:
  - **GET:** Solicita datos de un servidor. Es el método más utilizado para el web scraping.
  - **POST:** Envía datos al servidor, comúnmente para formularios y carga de archivos.
  - **PUT, DELETE, HEAD:** Usados para acciones específicas, aunque menos comunes en el web scraping.

- **Códigos de estado HTTP:** Los códigos de estado indican el resultado de la solicitud:
  - **2xx:** Éxito. Ej: 200 (OK).
  - **3xx:** Redirección. Ej: 301 (Movido permanentemente), 302 (Movido temporalmente).
  - **4xx:** Error del cliente. Ej: 404 (No encontrado), 403 (Prohibido).
  - **5xx:** Error del servidor. Ej: 500 (Error interno del servidor).

2.2 Herramientas populares para web scraping

Hay muchas herramientas y bibliotecas para realizar web scraping, cada una con sus propias fortalezas:

2.2.1 Requests (Python)

Requests es una biblioteca HTTP popular en Python:
- **Facilidad de uso:** Su sintaxis es sencilla, lo que facilita realizar solicitudes HTTP.
- **Características avanzadas:** Permite manejar autenticación, redirecciones, sesiones y cookies.
- **Compatibilidad:** Funciona bien con otras bibliotecas de análisis de datos, como BeautifulSoup.

2.2.2 BeautifulSoup (Python)

BeautifulSoup es una biblioteca de análisis HTML que facilita la extracción de datos:
- **Navegación del DOM:** Proporciona una API intuitiva para navegar y buscar elementos en el documento HTML.
- **Flexibilidad:** Puede analizar HTML malformado, corrigiéndolo automáticamente.
- **Integración:** Funciona perfectamente con Requests para descargar y analizar el contenido de las páginas.

2.2.3 Selenium

Selenium es una herramienta para automatizar navegadores:

- **Interacción con páginas dinámicas:** Puede manejar sitios que cargan contenido mediante JavaScript.
- **Simulación de usuario:** Automatiza acciones del usuario como hacer clics, escribir texto y desplazarse.
- **Soporte de múltiples navegadores:** Funciona con navegadores populares como Chrome, Firefox y Safari.

2.2.4 Scrapy

Scrapy es un marco completo para web scraping:

- **Eficiencia:** Maneja múltiples solicitudes concurrentes de forma eficiente.
- **Características avanzadas:** Ofrece herramientas para manejo de captchas, redireccionamientos y detección de cambios.
- **Modularidad:** Proporciona una arquitectura basada en arañas (spiders) para un scraping extensible.

2.3 Selección de una herramienta adecuada

La elección de la herramienta depende de la naturaleza del sitio web y los objetivos del proyecto:

- **Requests y BeautifulSoup:** Perfectos para páginas simples con contenido estático.
- **Selenium:** Adecuado para páginas con contenido dinámico y que dependen de JavaScript.
- **Scrapy:** Mejor para proyectos de gran escala que requieren alta velocidad y eficiencia.

Al comprender las características de cada herramienta y evaluar las necesidades del proyecto, es posible seleccionar la más adecuada para la tarea.

61TE01_09_Web scraping con Python

3. Primeros Pasos con Requests y BeautifulSoup

Requests y BeautifulSoup son dos herramientas populares para el web scraping en Python. Requests permite hacer solicitudes HTTP, mientras que BeautifulSoup facilita el análisis y la extracción de datos del contenido HTML.

3.1 Instalación de herramientas

Para comenzar, primero debes instalar las herramientas necesarias:

Requests: Se instala con

```
pip install requests
```

BeautifulSoup: Se instala con

```
pip install beautifulsoup4
```

También es útil instalar **lxml** para un análisis HTML más rápido y preciso, utilizando

```
pip install lxml
```

3.2 Haciendo solicitudes con Requests

Requests facilita el envío de solicitudes HTTP para obtener el contenido de una página web. Aquí tienes un ejemplo básico:

```
import requests

# Hacer una solicitud GET a la URL
response = requests.get('http://books.toscrape.com/')

# Verificar el estado de la respuesta
if response.status_code == 200:
    print('Solicitud exitosa')
    print(response.content)  # Contenido de la página en formato binario
else:
    print('Error al hacer la solicitud')
```

La respuesta contiene la página HTML en bruto, lista para ser analizada.

3.3 Análisis de HTML con BeautifulSoup

BeautifulSoup convierte el contenido HTML en un objeto que permite buscar y extraer datos fácilmente:

```
from bs4 import BeautifulSoup

# Analizar el HTML con BeautifulSoup
soup = BeautifulSoup(response.content, 'lxml')

# Ejemplo: encontrar el primer título de libro
book_title = soup.find('h3').text
print(f'Título del libro: {book_title}')
```


3.4 Extracción de datos básicos

Una vez analizado el contenido, puedes usar BeautifulSoup para extraer datos específicos. Por ejemplo, para obtener una lista de títulos de libros y precios:

```
# Encontrar todos los libros
books = soup.find_all('article', class_='product_pod')

# Extraer datos de cada libro
for book in books:
    # Título del libro
    title = book.h3.a['title']

    # Precio del libro
    price = book.find('p', class_='price_color').text

    print(f'Título: {title}, Precio: {price}')
```

Este ejemplo muestra cómo usar Requests y BeautifulSoup para extraer datos de una página web. Requests se encarga de descargar el contenido, mientras que BeautifulSoup lo analiza y permite extraer información.

4. Exploración Avanzada con BeautifulSoup

BeautifulSoup es una herramienta poderosa para analizar HTML y XML, permitiendo extraer datos de manera avanzada y precisa. Aquí, vamos a explorar algunas técnicas avanzadas para aprovechar todo su potencial.

4.1 Búsqueda de elementos complejos

A veces, los datos que deseas extraer están anidados en estructuras HTML complejas. BeautifulSoup ofrece varias formas de encontrar elementos:

- **Uso de selectores CSS**: soup.select('div.book a') permite encontrar un enlace dentro de un div con clase book.
- **Uso de atributos personalizados**: soup.find_all(attrs={"data-category": "science"}) encuentra elementos con el atributo data-category igual a science.
- **Combinando selectores**: Puedes combinar filtros para encontrar elementos que cumplan múltiples criterios, como soup.find('a', class_='author', href='/authors').

4.2 Navegación y manejo de estructura HTML

BeautifulSoup ofrece métodos para navegar por la estructura del árbol HTML:

- **Recorrer el árbol**: Utiliza parent, find_parent, find_next_sibling, find_previous_sibling para navegar por elementos relacionados.
- **Navegar descendientes**: find_all y children permiten iterar sobre todos los descendientes directos o encontrar elementos específicos.
- **Búsqueda por texto**: soup.find_all(text='Book Title') devuelve los elementos que contienen el texto exacto.

61TE01_09_Web scraping con Python

4.3 Manejo de datos con múltiples páginas

En muchos casos, los datos están distribuidos en varias páginas. Para recopilar datos de todas las páginas:

- **Identificar los enlaces de paginación**: Encuentra los enlaces que llevan a las páginas siguientes.
- **Iterar a través de las páginas**: Usa un bucle para cambiar el número de página en la URL y realiza solicitudes para cada página.
- **Concatenar los datos**: Recopila datos de cada página y combínalos en una lista o archivo.

Ejemplo avanzado

Supongamos que deseas extraer datos de un sitio web con múltiples páginas. Aquí tienes un ejemplo:

```
import requests
from bs4 import BeautifulSoup

# Función para obtener los datos de cada página
def get_books_from_page(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'lxml')
    books = soup.find_all('article', class_='product_pod')

    for book in books:
        title = book.h3.a['title']
        price = book.find('p', class_='price_color').text
        print(f'Título: {title}, Precio: {price}')

# Iterar sobre las páginas
base_url = 'http://books.toscrape.com/catalogue/page-{}.html'
page_num = 1

while True:
    url = base_url.format(page_num)
    response = requests.get(url)
    if response.status_code != 200:
        break
    get_books_from_page(url)
    page_num += 1
```

Este ejemplo:

- Hace solicitudes a cada página.
- Utiliza BeautifulSoup para analizar el HTML y encontrar datos.
- Rompe el bucle cuando no hay más páginas.

Estas técnicas avanzadas permiten extraer datos complejos y estructurados de sitios web, y combinarlas te ayudará a obtener datos precisos en proyectos más sofisticados.

5. Navegación Dinámica con Selenium

Selenium es una herramienta que permite automatizar navegadores web. Es especialmente útil para interactuar con páginas web que dependen en gran medida de JavaScript para cargar contenido dinámico.

5.1 Instalación y configuración de Selenium

Para comenzar a usar Selenium, primero debes instalarlo:

- **Instalación: pip install selenium**
- **Instalación del WebDriver:** Necesitas un WebDriver compatible con tu navegador. Por ejemplo, **chromedriver** para Google Chrome.
-

5.2 Uso de controladores web (WebDrivers)

El WebDriver actúa como un navegador controlado por Selenium para interactuar con páginas web:

```
from selenium import webdriver

# Inicializar el WebDriver para Chrome
driver = webdriver.Chrome(executable_path='/path/to/chromedriver')

# Navegar a una página web
driver.get('http://example.com')

# Realizar acciones en la página
element = driver.find_element('name', 'username')
element.send_keys('mi_usuario')

# Cerrar el navegador
driver.quit()
```


61TE01_09_Web scraping con Python

5.3 Interacción con elementos de la página

Selenium permite interactuar con los elementos de la página, como si fueras un usuario real:

- **Hacer clic en un botón:**

```
button = driver.find_element('id', 'submit_button')
button.click()
```

- **Completar un formulario:**

```
input_field = driver.find_element('name', 'search')
input_field.send_keys('Python')
input_field.submit()
```

- **Esperar elementos dinámicos:** Selenium puede esperar hasta que un elemento esté disponible:

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

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

5.4 Extracción de datos de sitios dinámicos

El web scraping con Selenium permite acceder a datos cargados dinámicamente con JavaScript:

- **Esperar el contenido:** Es importante esperar a que la página cargue el contenido antes de extraer datos.
- **Analizar HTML:** Puedes usar BeautifulSoup para analizar el HTML de Selenium:

```
from bs4 import BeautifulSoup

# Obtener el HTML de la página cargada
soup = BeautifulSoup(driver.page_source, 'lxml')

# Extraer datos como lo harías con BeautifulSoup
items = soup.find_all('div', class_='item')
for item in items:
    title = item.find('h3').text
    print(title)
```

Selenium permite manejar sitios con contenido dinámico que sería difícil de extraer con solo Requests y BeautifulSoup.

61TE01_09_Web scraping con Python

6. Scrapy: El Enfoque del Marco Completo

Scrapy es un marco completo para web scraping y web crawling, diseñado para la extracción eficiente de grandes cantidades de datos. Es ampliamente utilizado para proyectos de web scraping que requieren alta velocidad y complejidad.

6.1 Instalación y configuración de Scrapy

Para comenzar, debes instalar Scrapy:

Instalación:

```
pip install scrapy
```

Configuración inicial: Puedes crear un nuevo proyecto de Scrapy con scrapy startproject project_name.

6.2 Creación de un proyecto de Scrapy

Un proyecto de Scrapy consta de varias arañas (spiders) que definen cómo extraer datos de diferentes sitios web:

1. Crear el proyecto:

   ```
   scrapy startproject myproject
   ```

2. Crear una araña: Navega a la carpeta del proyecto y usa

   ```
   scrapy genspider spider_name domain.com.
   ```

3. Estructura del proyecto: Scrapy genera un directorio con archivos para configuraciones, arañas, pipelines, etc.

6.3 Definición de arañas y reglas de scraping

Las arañas (spiders) son clases de Python que definen cómo extraer datos:

Estructura básica de una araña:

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = ['http://example.com']

    def parse(self, response):
        # Extraer datos
        title = response.xpath('//title/text()').get()
        yield {'title': title}
```


Manejo de múltiples páginas: Usa response.follow para seguir enlaces y recorrer múltiples páginas:

```
def parse(self, response):
    for href in response.css('a.next::attr(href)'):
        yield response.follow(href, self.parse)
```

6.4 Exportación y almacenamiento de datos

Scrapy facilita la exportación de datos en diferentes formatos:

- Exportar a JSON: Ejecuta la araña con scrapy crawl myspider -o output.json.
- Exportar a CSV: Cambia la extensión de salida: scrapy crawl myspider -o output.csv.
- Uso de pipelines: Los pipelines permiten manipular los datos antes de guardarlos. Define un pipeline y actívalo en el archivo settings.py.

Scrapy es una herramienta poderosa para proyectos complejos de web scraping, permitiendo definir reglas, manejar múltiples páginas y exportar datos de manera eficiente.

7. Manejo de Obstáculos y Buenas Prácticas

El web scraping puede encontrarse con obstáculos debido a diversas restricciones impuestas por los sitios web. Es fundamental conocer y abordar estos desafíos para asegurar una extracción de datos eficaz y ética.

7.1 Manejo de CAPTCHA y bloqueos

- **CAPTCHA:** Los CAPTCHAs están diseñados para evitar que los bots accedan a sitios web. Superarlos puede requerir técnicas complejas como reconocimiento óptico de caracteres (OCR) o servicios de resolución de CAPTCHAs.

- **Bloqueos basados en IP:** Los sitios web pueden bloquear direcciones IP después de detectar un comportamiento sospechoso. Para evitarlo:
    - **Rotación de IP:** Utiliza servidores proxy o redes VPN para cambiar tu dirección IP.
    - **Distribución de solicitudes:** Distribuye las solicitudes a lo largo del tiempo para evitar patrones de comportamiento anómalos.

7.2 Pausas y retrasos para evitar bloqueos

Para evitar sobrecargar los servidores y evitar bloqueos:

- **Introducir retrasos entre solicitudes:** Utiliza **time.sleep()** en Python para pausar entre solicitudes.
- **Aleatorizar los retrasos:** Implementa retrasos aleatorios para simular el comportamiento humano.
- **Respetar el archivo robots.txt:** Verifica el archivo robots.txt del sitio para respetar las restricciones.

7.3 Extracción eficiente y respetuosa

- **Límites de frecuencia:** No hagas más solicitudes de las que el servidor pueda manejar.
- **Manejo de errores:** Implementa manejadores para errores comunes como el tiempo de espera o los errores 403/429.
- **Manejo de redirecciones:** Asegúrate de seguir las redirecciones automáticamente si es necesario.
- **Descargas incrementales:** Descarga solo las páginas nuevas o modificadas para minimizar la carga del servidor.

Adoptar estas buenas prácticas asegura una extracción de datos respetuosa y eficiente, evitando la mayoría de los problemas comunes asociados con el web scraping.

8. Almacenamiento y Análisis de Datos

Una vez que se han recopilado datos mediante web scraping, es importante almacenarlos y analizarlos adecuadamente para extraer información útil.

8.1 Guardado de datos en formatos comunes (CSV, JSON)

CSV (Valores separados por comas): Es un formato ampliamente utilizado y fácilmente legible por humanos y programas. Cada línea representa una fila y las columnas están separadas por comas.

```
import csv

data = [{'title': 'Book1', 'price': '$20'}, {'title': 'Book2', 'price': '$30'}]

with open('books.csv', mode='w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=['title', 'price'])
    writer.writeheader()
    writer.writerows(data)
```

JSON (JavaScript Object Notation): Es un formato popular para el intercambio de datos por su legibilidad y facilidad de uso.

```
import json

data = [{'title': 'Book1', 'price': '$20'}, {'title': 'Book2', 'price': '$30'}]

with open('books.json', 'w') as file:
    json.dump(data, file)
```


61TE01_09_Web scraping con Python

8.2 Almacenamiento en bases de datos

Para grandes volúmenes de datos o cuando se requiere un acceso rápido y eficiente, las bases de datos son la opción ideal:

```
SQL (MySQL, PostgreSQL, SQLite): Ideal para datos estructurados con relaciones
complejas.

import sqlite3

connection = sqlite3.connect('books.db')
cursor = connection.cursor()
cursor.execute('''CREATE TABLE books (title TEXT, price TEXT)''')

data = [('Book1', '$20'), ('Book2', '$30')]
cursor.executemany('INSERT INTO books VALUES (?, ?)', data)
connection.commit()
connection.close()
```

8.3 Análisis preliminar de datos extraídos

Después de almacenar los datos, el análisis preliminar puede incluir:

- Estadísticas básicas: Cálculo de conteos, promedios y distribuciones.
- Visualizaciones: Gráficas y tablas para obtener información rápidamente.
- Filtrado y clasificación: Ordenar y filtrar los datos para encontrar patrones y correlaciones.

Un ejemplo básico de análisis usando pandas:

```
import pandas as pd

# Leer datos CSV
df = pd.read_csv('books.csv')

# Estadísticas básicas
print(df.describe())

# Visualización de datos
df.plot(kind='bar', x='title', y='price')
```

Estas técnicas de almacenamiento y análisis te permiten transformar datos crudos en información valiosa.

9. Ejemplos Prácticos

9.1 Web para practicar

Para practicar web scraping, es importante asegurarse de hacerlo de manera ética y legal. Algunas páginas están diseñadas específicamente para practicar scraping:

1. **Books to Scrape**:
   - URL: http://books.toscrape.com/
   - Una página sencilla que proporciona datos de libros, diseñada específicamente para practicar web scraping. Puedes extraer títulos, precios, disponibilidad, etc.
2. **Quotes to Scrape**:
   - URL: http://quotes.toscrape.com/
   - Un sitio con citas y datos sobre los autores. Está diseñado para practicar la extracción de datos, paginación y más.
3. **Fake Store API**:
   - URL: https://fakestoreapi.com/
   - Es una API, no una página web, pero proporciona una interfaz sencilla basada en JSON para practicar solicitudes de API.
4. **ScrapingBee's Sandbox**:
   - URL: https://scrapingbee.com/sandbox/
   - Un entorno interactivo donde puedes practicar scraping en un ambiente controlado.

Recuerda siempre:

- Respetar los términos de servicio de los sitios web.
- Evitar la extracción de datos sensibles o personales.
- Respetar los límites de velocidad del sitio web para no sobrecargar sus servidores.

61TE01_09_Web scraping con Python

9.2 Web scraping de web de libros

9.2.1 Ejemplo básico

Para practicar web scraping con el sitio "Books to Scrape", es recomendable usar bibliotecas como requests para obtener el contenido HTML y BeautifulSoup para analizarlo. Aquí te muestro un ejemplo básico en Python para extraer títulos y precios de los libros:

```
import requests
from bs4 import BeautifulSoup

# URL de la página
url = "http://books.toscrape.com/"

# Hacer una solicitud GET a la página
response = requests.get(url)

# Analizar el contenido de la página con BeautifulSoup
soup = BeautifulSoup(response.content, 'html.parser')

# Buscar todos los contenedores de libros
books = soup.find_all('article', class_='product_pod')

# Extraer información de cada libro
for book in books:
    # Título del libro
    title = book.h3.a['title']

    # Precio del libro
    price = book.find('p', class_='price_color').text

    print(f"Titulo: {title}, Precio: {price}")
```

Desglose del código:

- requests.get(url): Realiza una solicitud HTTP GET para obtener el contenido de la página.
- BeautifulSoup(response.content, 'html.parser'): Analiza el contenido HTML con BeautifulSoup.
- soup.find_all('article', class_='product_pod'): Encuentra todos los elementos HTML que contienen información sobre los libros.
- book.h3.a['title']: Extrae el título del libro.
- book.find('p', class_='price_color').text: Extrae el precio del libro.

61TE01_09_Web scraping con Python

9.1.2. Ejemplo de conseguir todos los libros

Para obtener más libros, puedes ajustar el script para recorrer las páginas del sitio "Books to Scrape". Aquí te muestro un ejemplo que te permite extraer información de los primeros 20 libros:

```
import requests
from bs4 import BeautifulSoup

# Función para extraer libros de una página específica
def get_books_from_page(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    return soup.find_all('article', class_='product_pod')

# URL base
base_url = "http://books.toscrape.com/catalogue/page-{}.html"

# Inicializar lista para almacenar la información de los libros
books_info = []

# Recorrer las páginas necesarias para obtener al menos 20 libros
page_num = 1
while len(books_info) < 20:
    # Obtener libros de la página actual
    books = get_books_from_page(base_url.format(page_num))
    for book in books:
        if len(books_info) >= 20:
            break
        title = book.h3.a['title']
        price = book.find('p', class_='price_color').text
        books_info.append({'title': title, 'price': price})
    page_num += 1

# Imprimir información de los primeros 20 libros
for index, book in enumerate(books_info, start=1):
    print(f"{index}. Título: {book['title']}, Precio: {book['price']}")
```

Desglose del código:

- get_books_from_page(url): Esta función toma una URL como entrada y devuelve una lista de elementos de libros de esa página.
- base_url: Es la URL base con un marcador de posición para el número de página.
- books_info: Lista para almacenar información de los libros.
- while len(books_info) < 20: Recorre las páginas hasta que se hayan extraído al menos 20 libros.
- base_url.format(page_num): Ajusta la URL con el número de página actual.
- if len(books_info) >= 20: Verifica si se han obtenido los 20 libros necesarios.
- books_info.append: Agrega la información de los libros a la lista.
- Impresión: Se imprimen los títulos y precios de los primeros 20 libros.

61TE01_09_Web scraping con Python

9.1.3. Conseguir todas las categorias

Para obtener todas las categorías de libros de "Books to Scrape", puedes analizar la página principal del sitio web. Aquí te muestro cómo obtener la lista de categorías con Python:

```
import requests
from bs4 import BeautifulSoup

# URL de la página principal
url = "http://books.toscrape.com/index.html"

# Hacer una solicitud GET a la página
response = requests.get(url)

# Analizar el contenido de la página con BeautifulSoup
soup = BeautifulSoup(response.content, 'html.parser')

# Encontrar el elemento que contiene las categorías
categories_section = soup.find('ul', class_='nav-list')

# Extraer las categorías
categories = categories_section.find_all('a')

# Imprimir la lista de categorías
for category in categories[1:]:  # Omitimos la primera porque es la categoría 'Books'
    category_name = category.text.strip()
    category_link = "http://books.toscrape.com/" + category['href']
    print(f"Categoría: {category_name}, Enlace: {category_link}")
```

Desglose del código:

- url: La URL de la página principal del sitio.
- soup.find('ul', class_='nav-list'): Encuentra el elemento <ul> que contiene la lista de categorías.
- categories_section.find_all('a'): Encuentra todos los enlaces <a> dentro del contenedor de la lista.
- categories[1:]: Se omite el primer enlace que suele ser un enlace "All Books".
- category.text.strip(): Elimina espacios innecesarios del nombre de la categoría.
- "http://books.toscrape.com/" + category['href']: Construye la URL completa para la categoría.