Clase: Introducción al Web Scraping con Python

Preparación para el ejercicio práctico de LaLiga

1. ¿Qué es el web scraping?

El **web scraping** es el proceso de **extraer información de páginas web mediante un programa**.

En lugar de copiar datos manualmente desde el navegador, escribimos un script que:

- accede a una página web,
- localiza la información que nos interesa,
- la procesa,
- y la guarda en un formato reutilizable (por ejemplo, un CSV).

El scraping se utiliza habitualmente para:

- recopilar datos públicos,
- analizar información de forma periódica,
- automatizar tareas repetitivas.

2. Qué ocurre cuando accedemos a una página web

Cuando abrimos una web en el navegador pasan varias cosas, aunque no las veamos:

1. El navegador hace una **petición HTTP** a una URL.
2. El servidor responde con contenido.
3. El navegador interpreta ese contenido y lo muestra.

Nuestro programa hará exactamente lo mismo, pero **sin interfaz gráfica**.

3. Tipos de contenido que puede devolver una web

3.1 HTML (contenido estático)

En muchas webs, los datos vienen directamente en el HTML:

```
<table>
  <tr>
    <td>Barcelona</td>
    <td>42</td>
  </tr>
</table>
```

En estos casos, basta con:

- descargar el HTML,
- analizarlo,
- extraer los datos.

3.2 Contenido dinámico (JavaScript)

En webs modernas es frecuente que:

- el HTML inicial venga casi vacío,
- y que los datos se pidan después a una **API** usando JavaScript.

El navegador:

1. descarga el HTML,
2. detecta una URL interna,
3. hace una petición adicional,
4. recibe los datos en formato **JSON**,
5. y los muestra en pantalla.

Esto es especialmente común en:

- tablas de resultados,
- clasificaciones deportivas,
- dashboards,
- aplicaciones web modernas.

4. HTML vs JSON

HTML	JSON
Pensado para humanos	Pensado para máquinas
Hay que analizar etiquetas	Estructura clara
Más frágil	Más estable
Depende del diseño	Independiente del diseño

Siempre que una web cargue los datos desde una API, **es preferible acceder directamente al JSON**.

5. Herramientas que vamos a utilizar

5.1 Requests

La librería requests permite hacer peticiones HTTP desde Python.

Se utiliza para:

- descargar páginas web,
- acceder a APIs,
- obtener HTML o JSON.

Es la base de cualquier script de scraping.

5.2 BeautifulSoup

BeautifulSoup sirve para **analizar HTML**.

Permite:

- buscar etiquetas (table, tr, td, th),
- recorrer la estructura de la página,
- extraer texto limpio.

Es importante entender que:

BeautifulSoup **no ejecuta JavaScript**.

Solo trabaja con el HTML que recibe.

5.3 Formato CSV

Un archivo CSV:

- es texto plano,
- se puede abrir con Excel,
- representa datos en filas y columnas.

Guardar los resultados en CSV permite:

- reutilizar los datos,
- analizarlos posteriormente,
- compartirlos fácilmente.

6. Qué es una petición HTTP

Una petición HTTP consta de:

6.1 URL

Es la dirección del recurso que queremos obtener.

6.2 Cabeceras (Headers)

Las cabeceras aportan información adicional sobre la petición.

Una de las más importantes en scraping es **User-Agent**.

6.3 Respuesta

El servidor responde con:

- el contenido solicitado,
- o un código de error (404, 403, etc.).

El programa debe comprobar si la petición ha tenido éxito antes de continuar.

7. El User-Agent

El **User-Agent** identifica quién está haciendo la petición.

Los servidores lo usan para:

- distinguir navegadores reales,
- detectar automatizaciones,
- bloquear accesos sospechosos.

Si una petición no incluye un User-Agent razonable, es frecuente que:

- la web bloquee el acceso,
- o devuelva contenido distinto.

Por eso, en scraping es habitual **definir explícitamente un User-Agent**.

8. Cómo pensar el scraping antes de escribir código

Antes de programar, conviene seguir siempre este razonamiento:

1. ¿Los datos están en el HTML?
2. Si no, ¿existe una API que los devuelva?
3. ¿Qué formato tienen los datos?
4. ¿Cómo se van a guardar?

Este análisis previo evita muchos errores y scripts frágiles.

9. Estrategia general de scraping

Una estrategia habitual y profesional es:

1. Buscar primero una API o endpoint JSON.
2. Si no existe, analizar el HTML.
3. Usar automatización del navegador solo si no hay otra opción.

Esta estrategia es la que se va a aplicar en el ejercicio práctico.

10. Qué vamos a hacer en el ejercicio

En el ejercicio que sigue:

- Se descargará una página de resultados deportivos.
- Se comprobará si los datos se cargan de forma dinámica.
- Si existe una API, se utilizará directamente.
- Si no, se intentará extraer la información desde tablas HTML.
- Los datos se guardarán en un archivo CSV.
- Se realizará una búsqueda concreta dentro de los resultados.

El objetivo no es solo obtener los datos, sino **entender el proceso completo**.

Parte 2: Lectura guiada del script y flujo del programa

1. Enfoque general del script

Antes de entrar en el código, es importante entender **qué tipo de programa es**.

Este script:

- no asume cómo está construida la página,
- intenta varias estrategias,
- y se adapta a la forma en que la web entrega los datos.

Es un enfoque **robusto**, pensado para funcionar aunque el HTML cambie ligeramente.

2. Estructura general del programa

El script está dividido en cuatro bloques principales:

1. Configuración e importaciones
2. Funciones auxiliares
3. Función principal (main)
4. Punto de entrada del programa

Separar el código en funciones permite:

- leerlo mejor,
- probar partes concretas,
- reutilizar lógica.

3. Importaciones y configuración inicial

```
import requests
from bs4 import BeautifulSoup
import sys
import csv
from datetime import datetime
```

Cada módulo tiene un propósito claro:

- requests: realizar peticiones HTTP
- BeautifulSoup: analizar HTML
- sys: finalizar el programa en caso de error
- csv: guardar resultados en archivo
- datetime: registrar la fecha de extracción

4. Descarga de la página web

```
def fetch_html(url, timeout=15):
```

Esta función:

- realiza la petición HTTP,

- define un User-Agent,
- comprueba que la respuesta sea correcta,
- y devuelve el HTML como texto.

Si la descarga falla, el programa no continúa.

Este control de errores es esencial en scraping real.

5. Análisis inicial del HTML

Una vez descargado el HTML, se crea el objeto soup:

soup = BeautifulSoup(html, "lxml")

Esto transforma el HTML en una estructura navegable:

- se pueden buscar etiquetas,
- recorrer nodos,
- extraer texto y atributos.

El parser lxml se usa por ser más rápido y robusto.

6. Detección de contenido dinámico

ranking_url = find_table_ranking_url(soup)

Aquí ocurre algo clave:

- se busca si la página contiene un componente especial
- que apunte a una URL interna con los datos reales

Si se encuentra esa URL, significa que:

- los datos no están en el HTML,
- sino en una API que devuelve JSON.

Esta es la forma más limpia de obtener la información.

7. Obtención de datos desde la API

Si se detecta la URL del ranking:

data = fetch_json(ranking_url)

Esta función:

- hace una petición HTTP,
- espera una respuesta en formato JSON,
- y la convierte directamente en estructuras de Python.

Trabajar con JSON evita tener que analizar HTML complejo.

8. Navegación por la estructura JSON

El script no asume una estructura fija a ciegas:

- comprueba que existan las claves esperadas,
- accede progresivamente a los datos relevantes,
- controla errores si el formato no es el esperado.

Esto es importante porque las APIs pueden cambiar.

9. Construcción de los datos finales

Para cada equipo se extraen campos como:

- posición,
- nombre,
- puntos,
- partidos jugados,
- goles a favor y en contra.

Los datos se almacenan en una lista de diccionarios, que luego se usará para:

- imprimir resultados,
- guardar el CSV.

10. Guardado en archivo CSV

save_to_csv(csv_data, "clasificacion_laliga.csv")

Esta función:

- crea el archivo CSV,
- escribe la cabecera,
- guarda todas las filas,
- muestra información resumen.

Separar esta lógica en una función evita repetir código.

11. Búsqueda de un equipo concreto

El script incluye un ejemplo práctico:

- buscar un equipo específico (FC Barcelona),
- mostrar sus datos de forma directa.

Esto demuestra cómo:

- filtrar datos,
- recorrer estructuras,
- localizar información concreta.

12. Plan alternativo: scraping de HTML

Si no se encuentra ninguna API:

tables = find_standings_tables(soup)

Aquí el script:

- busca tablas que parezcan una clasificación,
- analiza encabezados,

- intenta identificar columnas relevantes.

Es un enfoque heurístico, útil cuando no hay API disponible.

13. Extracción de filas y columnas

Se recorre la tabla:

- fila por fila (tr),
- celda por celda (td, th),
- limpiando el texto.

El objetivo es convertir una tabla visual en datos estructurados.

14. Identificación de equipos y puntos

No todas las tablas tienen el mismo formato.

Por eso el script:

- distingue columnas numéricas,
- identifica el nombre del equipo,
- detecta los puntos de forma flexible.

Esto evita depender de posiciones fijas.

15. Flujo completo del programa

Resumiendo el recorrido:

1. Descargar la página
2. Analizar el HTML
3. Buscar una API oculta
4. Si existe, usar JSON
5. Si no, analizar tablas HTML
6. Procesar los datos
7. Guardar resultados
8. Mostrar información relevante

16. Qué se espera que aprendáis con este ejercicio

Al finalizar el ejercicio, el alumno debería:

- entender cómo se obtienen los datos de una web,
- distinguir entre HTML y JSON,
- saber cuándo usar cada técnica,
- manejar errores básicos,
- generar archivos de salida reutilizables.