Web Scraping - Extraindo Dados da Web

Asimov Academy

ASIMOV

Conteúdo

01. Introdução

Web Scraping, ou raspagem de dados, é uma técnica utilizada para extrair informações de sites de forma automatizada. Trata-se do processo de consumir dados diretamente de páginas da internet, sem a necessidade de interação manual constante. Essa prática permite que informações públicas disponíveis online sejam capturadas, processadas e utilizadas para diversos fins analíticos, comerciais ou informativos.

Por que utilizar Web Scraping?

A raspagem de dados pode ser útil em diversas situações. Por exemplo, um profissional da área de finanças pode necessitar de dados em tempo real de bolsas de valores. Caso não exista uma API oficial ou outro meio automatizado, o Web Scraping se torna uma solução viável para capturar essas informações diretamente do site. Em setores que dependem de decisões políticas ou econômicas, é possível construir scripts que monitoram portais de notícias. Dessa forma, notícias de interesse podem ser automaticamente capturadas e utilizadas para análise ou apoio à tomada de decisões.

A grande vantagem do Web Scraping é a possibilidade de obter dados de fontes públicas, estruturando essas informações de maneira útil para o ambiente interno de um negócio ou projeto pessoal.

Diferença entre Web Scraping e API

Ambos os métodos têm o mesmo objetivo: a extração de dados. A principal diferença está na forma como essa coleta é realizada.

- **API (Application Programming Interface)**: permite acessar dados estruturados, geralmente fornecidos de forma oficial por sistemas ou plataformas. A comunicação ocorre diretamente entre programas, por meio de requisições HTTP padronizadas.
- **Web Scraping**: simula a navegação de um usuário comum, acessando páginas web diretamente e coletando dados a partir do conteúdo exibido nelas, como HTML, textos, imagens, etc.

Em resumo, enquanto APIs oferecem dados já organizados e preparados para consumo por máquinas, o Web Scraping atua como um visitante comum, extraindo dados diretamente da camada visual do site.

Considerações sobre Legalidade

Aqui, serão abordadas questões éticas e legais relacionadas à prática de Web Scraping. É importante compreender os limites legais da extração de dados públicos, bem como as boas práticas para evitar

sobrecarga de servidores ou violações de termos de uso.

O que esperar do curso

Nas próximos capítulos, serão apresentadas aplicações práticas da raspagem de dados. O conteúdo abordará:

- Onde e como coletar dados;
- Ferramentas e bibliotecas utilizadas;
- Análise da estrutura de sites;
- Boas práticas e tratamento dos dados coletados.

O objetivo é construir uma base sólida para que o estudante compreenda como acessar, interpretar e extrair dados úteis a partir de qualquer site.

02. Browser e Extração

O primeiro passo no processo de Web Scraping consiste em identificar corretamente as informações que se deseja extrair de uma página da web. Antes de iniciar qualquer codificação, é fundamental compreender a estrutura do site-alvo e localizar com precisão os dados desejados. Essa etapa inicial pode ser realizada diretamente pelo navegador, utilizando ferramentas de desenvolvedor.

Análise da estrutura da página com o navegador

Sites modernos são compostos por diferentes elementos visuais e estruturais. Cabeçalhos, menus, áreas de conteúdo, botões e seções informativas são todos construídos por meio de linguagens como **HTML**, **CSS** e **JavaScript**. Para inspecionar esses elementos, é possível utilizar o **DevTools** do navegador Google Chrome, uma ferramenta que permite visualizar e interagir com o código-fonte da página.

Acessando o DevTools

Existem duas formas principais de abrir o Chrome DevTools:

- Pressionando a tecla **F12** no teclado;
- Clicando com o botão direito sobre qualquer elemento da página e selecionando a opção **"Inspecionar"**.

A interface do DevTools exibe o código-fonte da página em HTML, junto a outras abas como Console, Network e Styles.

Estrutura básica de uma página web

Uma página web é composta, principalmente, pelas seguintes tecnologias:

- **HTML (HyperText Markup Language)**: define a estrutura da página e os elementos visíveis, como textos, imagens, botões, etc.;
- **CSS (Cascading Style Sheets)**: é responsável pela parte estética, como cores, fontes, tamanhos e espaçamentos;
- **JavaScript**: fornece interatividade e dinamismo à página.

> Analogia:
>
> - **HTML** é o esqueleto;

> - **CSS** é a pele (aparência);
> - **JavaScript** são os músculos (ações e comportamentos).

Durante o processo de raspagem de dados, o foco principal será o **HTML**, pois é nessa camada que os dados costumam estar disponíveis.

Utilizando o seletor de elementos

Uma funcionalidade essencial do DevTools é o **seletor de elementos**, localizado no canto superior esquerdo da ferramenta. Ao ativá-lo, é possível passar o cursor do mouse sobre qualquer item da página e visualizar automaticamente o trecho de código HTML correspondente.

Essa funcionalidade permite:

- Identificar **tags HTML** como `<div>`, `<header>`, `<p>`, `<a>`, entre outras;
- Localizar atributos e valores relevantes (como `id`, `class`, `href`, `src`, etc.);
- Verificar se o conteúdo desejado está disponível no HTML de forma direta ou se depende de scripts JavaScript para carregamento.

Exemplo prático

Como exemplo, usaremos o site do **G1**, um portal de notícias da Globo. Para localizar o seletor CSS de um título de notícia, abra o navegador e acesse https://g1.globo.com, em seguida pressione `F12` (ou `Ctrl + Shift + I`) para abrir o DevTools. Dentro do DevTools, clique no ícone de seleção de elemento (a seta no canto superior esquerdo), passe o cursor sobre o título de uma notícia na página inicial e clique sobre ele. No painel **Elements** será exibido o código HTML correspondente, por exemplo:

```
<a class="feed-post-link"
 ↪  href="https://g1.globo.com/politica/noticia/2025/05/02/exemplo-de-noticia.html">
  Exemplo de notícia do dia
</a>
```

Para copiar o seletor CSS, clique com o botão direito sobre essa linha no painel **Elements**, escolha **Copy → Copy selector**, o que resultará em algo como:

```
a.feed-post-link
```

Essa identificação passo a passo mostra exatamente **onde** e **como** o dado está estruturado na página, facilitando a construção do script de scraping.

Conclusão

Compreender a estrutura de um site por meio do navegador é uma etapa fundamental para um Web Scraping eficiente. Ao dominar o uso do DevTools, é possível navegar pelas camadas de uma página e identificar precisamente os dados que serão extraídos.

Nas próximas aulas, será abordada a lógica por trás da extração automatizada, bem como a organização e o processamento das informações coletadas.

03. Lógica de Web Scraping

A raspagem de dados (ou *Web Scraping*) consiste em um processo estruturado para acessar e extrair informações disponíveis em páginas da web. Esse processo pode ser dividido em **quatro etapas fundamentais**, que serão a base para o nosso desenvolvimento prático.

Etapa 1 — Escolha da página alvo

O ponto de partida é definir qual página da web contém os dados relevantes para a extração. Sites de notícias, portais institucionais, páginas de produtos, entre outros, são exemplos comuns de fontes. A escolha deve considerar:

- Se o site exibe os dados publicamente;
- A frequência com que os dados mudam;
- A complexidade da estrutura do HTML da página.

Etapa 2 — Identificação dos dados

Após selecionar o site, é necessário identificar os elementos que serão extraídos. Para isso, utiliza-se o navegador (como visto no capítulo anterior), inspecionando o código HTML.

Itens a observar:

- **Tags HTML**: São os contêineres de conteúdo (ex.: `<h1>`, `<p>`, `<div>`, `<span>`).
- **Classes e IDs**: Atributos que ajudam a diferenciar e localizar os elementos no código.
- **Texto ou atributos internos**: Informações específicas (como títulos, links, datas, preços, etc.).

> Exemplo prático:
> Se o objetivo é coletar imagens de uma galeria, pode-se buscar elementos `<img>` com um atributo `src` que contenha palavras-chave como “grêmio”.

Etapa 3 — Extração com Python

Com os elementos identificados, inicia-se o desenvolvimento do script de extração. Nessa etapa será utilizado **Python**, junto a bibliotecas específicas para web scraping (como *Requests*, *BeautifulSoup*, *Selenium*, entre outras — que serão abordadas posteriormente).

A lógica consiste em:

- Fazer uma requisição à página;
- Acessar o conteúdo HTML retornado;
- Navegar pela estrutura do HTML para encontrar os dados alvo;
- Armazenar essas informações de forma organizada.

Etapa 4 — Estruturação dos dados

Após extrair os dados desejados, o passo final é estruturá-los de forma que possam ser utilizados em análises, relatórios ou visualizações. Essa organização é comumente feita utilizando **tabelas**, geralmente por meio da biblioteca `pandas`, que facilita o tratamento de dados em formato tabular.

A estrutura resultante permite manipulações como filtros, agrupamentos, estatísticas e exportação para formatos como CSV ou Excel.

Considerações finais

A estrutura apresentada aqui será recorrente em praticamente todos os projetos de Web Scraping. Dominar cada uma dessas etapas é essencial para garantir a eficiência e a escalabilidade da extração de dados. Nos próximos capítulos, serão introduzidas as principais bibliotecas que viabilizam esse processo.

04. Bibliotecas

Nesta etapa do curso, serão apresentadas as principais bibliotecas utilizadas para a realização de Web Scraping com Python. Apesar de existirem diversas opções no mercado, o foco do curso será em duas bibliotecas bastante consolidadas: **BeautifulSoup** e **Selenium**.

BeautifulSoup

BeautifulSoup é uma biblioteca Python utilizada para o *parsing* (análise e navegação) de arquivos HTML e XML. É ideal para projetos de Web Scraping mais simples, em que as páginas são estáticas (ou seja, o conteúdo já está carregado no HTML no momento do acesso).

Vantagens:

- Sintaxe simples e acessível para iniciantes;
- Ideal para raspar conteúdo de páginas estáticas;
- Boa integração com outras bibliotecas, como `requests` e `pandas`.

Instalação:

Para instalar o BeautifulSoup, basta utilizar o terminal com o seguinte comando:

```
pip install beautifulsoup4
```

Selenium

Selenium é uma biblioteca mais avançada, voltada para automação de navegadores. É especialmente útil quando o conteúdo da página é gerado dinamicamente por JavaScript, o que impede a leitura direta do HTML por bibliotecas como o BeautifulSoup.

Características:

- Permite interagir com a página como um usuário real (clicar, rolar, preencher formulários);
- Ideal para páginas dinâmicas;
- Pode ser utilizado com diversos navegadores (Chrome, Firefox, etc.), exigindo instalação de *drivers* específicos.

Instalação:

A instalação e configuração do Selenium será abordada futuramente, durante os módulos voltados para projetos mais avançados.

Conclusão

As bibliotecas de Web Scraping são ferramentas fundamentais para extrair dados da web de forma eficiente. A escolha entre elas dependerá do tipo de página a ser trabalhada.

Na próxima etapa, será realizada uma revisão dos principais conceitos de **HTML, classes, tags e IDs**, que são essenciais para compreender a estrutura das páginas e realizar extrações precisas. Após essa revisão, será iniciada a fase prática da programação com Python.

05. Tags

Neste capítulo, iniciamos o módulo de revisão básica de HTML, essencial para entender como as páginas web são estruturadas. Compreender HTML é fundamental para quem deseja realizar Web Scraping, pois é com base nas **tags**, **classes** e **IDs** que conseguimos localizar e extrair dados relevantes de um site.

O que são tags HTML?

Tags são os elementos básicos da linguagem HTML. Elas indicam ao navegador como deve ser exibido determinado conteúdo em uma página web. Toda tag é envolvida por sinais de menor e maior (< >) e, geralmente, acompanha uma tag de fechamento (</ >).

Exemplos de tags:

- `<html>`: Representa o início de um documento HTML.
- `<body>`: Indica o corpo da página, onde ficam os elementos visíveis.
- `<div>`: Cria um contêiner (ou "caixa") para agrupar conteúdos.
- `<h1>` a `<h6>`: Tags de título, sendo `<h1>` o maior e `<h6>` o menor.
- `<p>`: Define um parágrafo de texto.

Visualizando o HTML de uma página

Para visualizar o HTML de qualquer página no navegador, é possível utilizar o **Chrome DevTools**. Basta apertar a tecla **F12** no teclado.

Ao acessar essa ferramenta, é possível observar várias tags como `<head>`, `<body>`, `<nav>`, `<div>`, entre outras. Dentro dessas tags, também é comum encontrar **atributos**, como `class` e `id`.

Diferença entre tag, class e id

Elemento	Função	Observação
Tag	Define a estrutura do elemento (ex: parágrafo, título, divisão).	Toda tag deve ser aberta e fechada corretamente.
ID	Identificador único para um único elemento da página.	Deve ser exclusivo: um mesmo ID não pode ser usado mais de uma vez.
Class	Grupo de elementos que compartilham o mesmo estilo ou função.	Pode ser usada em diversos elementos diferentes.

Figure 1: Tabela comparativa de tag, classe e id em HTML

Criando um HTML simples

Vamos revisar agora como construir um pequeno exemplo de HTML no **Visual Studio Code (VSCode)**:

Abra o VSCode, ou outro editor de sua preferência, e crie um arquivo chamado `asimov_exemplo.html`, você pode até alterar o nome, mas lembre-se que ele deve finalizar com `.html` para que o interpretador saiba como ler.

Após isso, adicione o código abaixo:

```
<html>
  <body>
    <header>
      <h2>O cabeçalho oficial do site</h2>
    </header>

    <div>
      <h1>Site da Asimov</h1>
      <h3>Nosso primeiro cabeçalho</h3>
      <p>Esse vai ser o nosso primeiro parágrafo.</p>
      <p>Aqui fica o segundo parágrafo das imóveis.</p>
    </div>
  </body>
</html>
```

Explicando o código:

- O `<html>` envolve todo o documento.
- O `<body>` é o corpo da página, onde está o conteúdo visível.
- Dentro do `<body>`, temos:

  - Um <header> com um título <h2>.
  - Uma contendo um título <h1>, um subtítulo <h3>, e dois parágrafos <p>.

Cada tag define a função daquele conteúdo dentro da página. Por exemplo, os parágrafos <p> são usados para textos corridos, enquanto os <h1>, <h2>, <h3> são usados para títulos com diferentes níveis de destaque.

Visualizando o site criado

Após criar o HTML, é possível visualizá-lo no navegador. Para isso, clique com o botão direito em cima do arquivo e selecione a opção "Revelar no explorador de arquivos". Após isso, clique no seu arquivo e ele o código HTML será renderizado no seu navegador padrão.

Com a ferramenta do **DevTools (F12)** aberta, é possível identificar cada uma das tags e elementos que foram definidos no código.

Dessa forma, você pode ver exatamente como o navegador interpreta cada componente da página, reconhecendo na prática o que cada tag representa.

Conclusão

Neste capítulo, aprendemos:

- O que são e para que servem as **tags HTML**;
- Como visualizar o HTML de qualquer site com o **DevTools**;
- A diferença entre **tags**, **classes** e **IDs**;
- Como criar um **HTML básico** estruturado com títulos e parágrafos.

Esse conhecimento é essencial para a próxima etapa do curso, onde será abordada a análise de sites reais. A familiaridade com essas estruturas permitirá identificar com precisão quais informações podem ser extraídas via Web Scraping.

06. Classes e Ids

Nesta aula, vamos mergulhar em dois conceitos fundamentais do HTML — **IDs** e **classes** — e entender por que eles são tão importantes quando criamos páginas e, especialmente, quando fazemos web scraping. Imagine que você deseja marcar itens na sua página para poder localizá-los depois, seja para estilizar com CSS, manipular com JavaScript ou extrair conteúdo com Python. É aí que entram IDs e classes.

O que é um ID?

Pense no ID como a "impressão digital" de um elemento: ele é único e exclusivo. Não existe dois elementos `<div>`, `<h1>` ou `<p>` com o mesmo ID. Quando você dá a um `<h1>` o atributo `id="titulo-principal"`, por exemplo, está dizendo ao navegador e ao seu scraper: "Este elemento é somente este". Em CSS, você o acessa com `#titulo-principal`; em JavaScript, com `document.getElementById("titulo-principal")`; e em Python, com algo como:

```
titulo = soup.find(id="titulo-principal").get_text()
```

No entanto, na prática de web scraping, raramente você busca por IDs, pois normalmente precisa capturar vários itens semelhantes — título de várias notícias, vários parágrafos, listas, cards etc.

Para que serve uma classe?

Já as **classes** são como "etiquetas" que você cola em quantos elementos quiser. Se você cria `class="texto-destaque"` e aplica em diversos parágrafos, todos eles ficam pertencentes a esse grupo. No CSS, você estiliza com `.texto-destaque { … }`; no scraper, busca com `soup.find_all(class_="texto-destaque")`. Assim, em vez de extrair um único parágrafo (ID), você obtém uma lista de todos que compartilham aquela característica.

Exemplo prático unindo ID e classe

Imagine um pequeno arquivo `index.html`:

```
<h1 id="h1-primeiro" class="classe-h">Bem-vindo ao Web Scraper</h1>
<h3 id="h3-primeiro" class="classe-h">Módulo 1: Fundamentos de HTML</h3>
<p class="classe-p">Nesta aula você vai aprender o básico sobre IDs.</p>
<p class="classe-p">Depois, vamos explorar como usar classes para agrupar elementos.</p>
<h2 class="classe-h">Por que classes são tão úteis?</h2>
```

- Nesse `<h1>`, o **ID** `h1-primeiro` garante um marcador único, e a **classe** `classe-h` o agrupa junto com outras tags de título.

- Nos parágrafos, usamos `classe-p` para identificar todos os trechos de texto.
- O <h2> não tem ID, mas faz parte do mesmo grupo de títulos (`classe-h`).

Quando e por que usar cada um

- **ID**: para um elemento que você precisa referenciar individualmente — um rodapé, um banner especial, o título da página.
- **Classe**: quando há vários elementos "iguais" ou semelhantes e você quer tratá-los em conjunto, seja para estilizar ou para coletar dados em massa.

Em web scraping, quase sempre você vai navegar pela árvore de elementos usando classes. Por exemplo, para pegar todos os títulos de serviço em um site real (como o da Anvisa), você faria algo assim:

```
header = soup.find("div", class_="header-wrapper")
site_name = header.find("span", class_="site-name").get_text()
noticias = soup.find_all("span", class_="text")
for noticia in noticias:
    print(noticia.get_text())
```

Conclusão

Entender IDs e classes é essencial para estruturar bem seu HTML e, mais adiante, para extrair informações com precisão. Na próxima aula, vamos combinar esses conhecimentos de HTML com Python, montando scripts que percorrem sites reais e coletam exatamente os dados que você deseja.

07. Exercitando a lógica de Web Scraping

Neste capítulo, vamos colocar a mão na massa e exercitar o processo de web scraping **em modo local**, usando um arquivo HTML de exemplo. O objetivo é entender como navegamos na estrutura de um documento, localizamos elementos e extraímos exatamente o que precisamos. Não se preocupe com termos avançados ou detalhes de importação — foque apenas no fluxo de leitura, busca e extração.

Preparando o ambiente

Antes de mais nada, certifique-se de ter a biblioteca **BeautifulSoup** instalada no seu Python. No terminal, basta executar:

```
pip install beautifulsoup4
```

HTML de exemplo

Lembram que criamos um arquivo HTML para testes no capítulo 5? Pois bem, usaremos ele para a nossa extração local.

```
<html>
  <body>
    <header>
      <h2>O cabeçalho oficial do site</h2>
    </header>

    <div>
      <h1>Site da Asimov</h1>
      <h3>Nosso primeiro cabeçalho</h3>
      <p>Esse vai ser o nosso primeiro parágrafo.</p>
      <p>Aqui fica o segundo parágrafo das imóveis.</p>
    </div>
  </body>
</html>
```

Nosso objetivo será abrir esse arquivo, transformar em objeto BeautifulSoup e buscar tags específicas ou grupos de tags.

Lendo o HTML no Python

Com o arquivo pronto, passamos ao script. Abra um novo arquivo `scraper.py` e importe o BeautifulSoup. É importante que esse arquivo esteja localizado na mesma pasta que o nosso arquivo `html`. Usaremos o comando `with open` para ler o conteúdo sem se preocupar em fechar o arquivo manualmente:

```
from bs4 import BeautifulSoup

with open("exemplo.html", "r", encoding="utf-8") as arquivo:
    texto_html = arquivo.read()
```

A variável `texto_html` agora contém tudo que foi escrito no arquivo, incluindo as tags. No próximo passo, transformaremos essa string em um objeto navegável.

Criando o objeto BeautifulSoup

Para o Python entender e percorrer cada elemento, precisamos de um parser. A linha a seguir cria o `soup`, que funciona como um mapa hierárquico do nosso HTML:

```
soup = BeautifulSoup(texto_html, "lxml")
```

Aqui, o parser `lxml` organiza o HTML em uma árvore de objetos, permitindo que usemos métodos como `.find()` e `.find_all()` para localizar o que quisermos.

Encontrando o primeiro elemento com `.find()`

Em muitas situações, queremos apenas a primeira ocorrência de um elemento — por exemplo, o primeiro parágrafo do documento. O método `.find()` resolve isso:

```
primeiro_p = soup.find("p")
print(primeiro_p)           # Exibe a tag completa, ex: <p>Parágrafo inicial sem classe.</p>
print(primeiro_p.get_text()) # Exibe apenas o texto: Parágrafo inicial sem classe.
```

Esse approach é útil quando sabemos que há apenas um elemento de interesse ou quando precisamos do primeiro de uma lista.

Coletando múltiplos elementos com `.find_all()`

Quando queremos todos os parágrafos, títulos ou itens de lista, recorremos ao `.find_all()`. Ele retorna uma lista com todos os elementos que correspondem ao critério:

```
paragrafos = soup.find_all("p")
for parag in paragrafos:
    print(parag.get_text())
```

Dessa forma, extraímos o conteúdo de cada parágrafo, independente de quantos existam.

Filtrando por atributos: classes e IDs

Para casos mais sofisticados, podemos buscar apenas elementos que carreguem um atributo específico. Se vários elementos compartilham `class="class1"`, escrevemos:

```
itens_class1 = soup.find_all(class_="class1")
for el in itens_class1:
    print(f"<{el.name}>: {el.get_text()}")
```

No Python, `class` é palavra reservada, então usamos `class_` com underscore. É possível combinar filtros, por exemplo, buscando itens de lista que tenham mais de uma classe:

```
destaques = soup.find_all("li", class_["item", "destaque"])
```

Isso retorna `<li>` que combinam ambos os atributos.

Lidando com ausência de resultados

Nem sempre o elemento existe. O retorno de `.find()` será `None` se nada for encontrado. É importante verificar isso para evitar erros ao chamar métodos em um objeto `None`:

```
titulo_h3 = soup.find("h3")
if titulo_h3:
    print(titulo_h3.get_text())
else:
    print("Elemento <h3> não encontrado.")
```

Conclusão

Com este exemplo local bem compreendido, estaremos prontos para avançar às páginas reais na web. Nos próximos capítulos, abordaremos como fazer requisições HTTP, lidar com sites que usam JavaScript e implementar estratégias de retry e paralelismo.

08. Analisando a documentação do BeautifulSoup

A documentação oficial da biblioteca **BeautifulSoup** é um recurso fundamental para o desenvolvimento de projetos de web scraping mais robustos e eficientes. Neste capítulo, faremos uma análise geral de como consultar e utilizar essa documentação para extrair o máximo proveito das funcionalidades oferecidas pela biblioteca.

Acesso à documentação

Para acessar a documentação do BeautifulSoup, basta realizar uma busca simples no Google pelo termo "BeautifulSoup" e selecionar o site oficial correspondente, geralmente identificado como **BeautifulSoup4**.

Dentro do site, é possível navegar pela documentação utilizando a sidebar ou rolando a página com o mouse. A documentação é bastante extensa, cobrindo desde conceitos básicos de uso até funcionalidades avançadas para a manipulação de documentos HTML e XML.

Principais funções apresentadas

A documentação apresenta diversas funções para:

- **Extração de dados**
- **Organização de informações**
- **Identificação de elementos dentro de um documento HTML**

Entre as primeiras funções introduzidas, destacam-se as formas básicas de seleção de elementos, como:

```
soup.title    # Acessa o título da página
soup.p        # Acessa o primeiro parágrafo encontrado
soup.a        # Acessa o primeiro link encontrado
```

Essas chamadas representam maneiras simples de se navegar na estrutura HTML após instanciar o conteúdo com o BeautifulSoup.

Método `.find_all()`

Uma função de destaque na documentação é a **`.find_all()`**, que permite localizar **todos** os elementos correspondentes a um critério específico.

Funcionamento básico:

```
soup.find_all('p')  # Retorna todos os elementos <p> da página
soup.find_all('h2') # Retorna todos os elementos <h2> da página
```

- A função `.find_all()` recebe como parâmetro a **tag** que desejamos localizar.
- O retorno é uma lista contendo todos os elementos encontrados que correspondem ao parâmetro passado.

Essa função é extremamente útil para casos em que queremos extrair **vários elementos** similares em uma única operação.

Método `.select()`

Outra funcionalidade importante é o método **`.select()`**, que permite realizar buscas mais específicas com base em **classes**, **ids** e outros seletores CSS.

Exemplos de uso:

```
soup.select('.nome-da-classe')  # Seleciona todos os elementos com uma determinada classe
soup.select('#id-do-elemento')  # Seleciona o elemento com um determinado id
```

- Essa função é útil principalmente quando precisamos localizar elementos que compartilham um mesmo rótulo, como classes ou identificadores únicos.
- Permite combinações mais complexas de seleção, utilizando a sintaxe familiar do CSS.

O `.select()` amplia o leque de possibilidades na hora de extrair dados, tornando o processo mais preciso e eficiente.

A importância da prática e da análise

Embora a documentação forneça inúmeras funções, o sucesso do scraping depende da **análise criteriosa** da estrutura HTML das páginas e da **prática contínua** com as ferramentas disponíveis.

- É fundamental **inspecionar o HTML** (utilizando a tecla `F12` no navegador) para identificar as **tags**, **classes** e **ids** relevantes.
- É preciso combinar essas informações com as funções do BeautifulSoup para construir seletores eficazes.
- A responsabilidade de direcionar corretamente as funções para a extração dos dados desejados é do desenvolvedor.

A biblioteca disponibiliza poderosas ferramentas de extração, mas é necessário utilizá-las com inteligência para obter resultados de qualidade.

Considerações finais

Neste capítulo, foi realizada uma introdução prática à consulta da documentação oficial do **BeautifulSoup**, destacando funções essenciais como `.find_all()` e `.select()`.
Na próxima etapa, partiremos para a prática: aplicaremos essas funções na extração real de dados de páginas HTML, reforçando o aprendizado com exemplos concretos.

09. Aplicação dos conceitos

Neste capítulo, iniciaremos a aplicação prática dos conceitos apresentados anteriormente sobre a biblioteca **BeautifulSoup**, utilizando o ambiente **Jupyter Notebook** para realizar uma operação básica de web scraping.

Ambiente de Trabalho

Nosso ambiente será o **Jupyter Notebook**, onde criaremos um novo arquivo para o projeto, nomeado como `WebScraping`. Antes de iniciar a extração de dados, é necessário definir a página alvo, pois o **BeautifulSoup** atua navegando e extraindo dados a partir de páginas HTML.

Neste exemplo inicial, optamos por uma abordagem prática com uma página de notícias, escolhendo o site **globo.com** para extrair as principais manchetes da sua página inicial.

Definindo a Página Alvo

A página escolhida foi o portal de notícias **Globo**. Após navegar até o site e analisar seu conteúdo, identificamos diversas manchetes na primeira página, relativas a esportes, política, entretenimento, entre outros.

Nosso objetivo será importar essas manchetes para dentro do Jupyter Notebook.

Importando Bibliotecas Necessárias

Antes de iniciar a extração, precisamos instalar e importar as bibliotecas fundamentais para a execução da tarefa.

Usaremos, nesse exemplo, além da biblioteca BeautifulSoup, a biblioteca request:

```
pip install BeautifulSoup requests
```

Agora a importação:

```
import requests
from bs4 import BeautifulSoup
```

- **`requests`**: Responsável por realizar a conexão HTTP com a página web.
- **`BeautifulSoup`**: Utilizada para analisar e navegar pela estrutura HTML da página. ## Estabelecendo a Conexão com o Site

Estabelecemos uma única conexão com o servidor para evitar múltiplas requisições que possam sobrecarregar o site e causar bloqueios.

```
# URL do site
globo_url = "https://www.globo.com"
# Fazendo a requisição HTTP
page = requests.get(globo_url)
# Verificando o status da requisição
print(page)
```

Se o status retornado for `200`, significa que a conexão foi realizada com sucesso.

Tratando o Conteúdo HTML

Após a conexão bem-sucedida, obtemos todo o conteúdo HTML da página. Contudo, para manipular esse conteúdo de forma estruturada, utilizamos o BeautifulSoup:

```
# Criando o objeto BeautifulSoup
soup = BeautifulSoup(page.text, 'html.parser')
```

Este processo organiza o código HTML, permitindo navegação e seleção de elementos de forma mais eficiente.

Inspecionando a Estrutura da Página

Para identificar os elementos que contêm as manchetes, utilizamos a ferramenta de inspeção do navegador (`F12`).

Durante a inspeção, verificamos que:

- As manchetes estão contidas dentro de tags `<h2>`.
- Essas tags `<h2>` possuem a classe `post__title`.

Essa análise estrutural é essencial para que o BeautifulSoup localize corretamente os dados desejados.

Extraindo as Manchetes

Utilizando o padrão identificado (`<h2>` com classe `post__title`), realizamos a extração:

```
# Buscando todos os elementos h2 com a classe post__title
noticias = soup.find_all('h2', class_='post__title')
# Exibindo as manchetes extraídas
for noticia in noticias:
    print(noticia.text.strip())
```

O método `.find_all()` busca todos os elementos que correspondem aos critérios definidos, enquanto `.text.strip()` limpa o texto retornado, removendo espaços em branco desnecessários.

Resultado

O script retorna uma lista com todas as manchetes principais do site **globo.com**, como por exemplo:

```
[4]: # Buscando todos os elementos h2 com a classe post__title
noticias = soup.find_all('h2', class_='post__title')

# Exibindo as manchetes extraídas
for noticia in noticias:
    print(noticia.text.strip())
```

```
Apagão massivo atinge Portugal, Espanha e outros países da Europa
Energia começa a ser reestabelecida em Portugal e na Espanha
'Fenômeno atmosférico raro' causou apagão, diz Portugal; SIGA
Falha de energia causa acidentes, falhas no metrô e cancela voos
Controle mostra aviões 'sumindo' após apagão; vídeo
Mapas mostram confusão no trânsito das cidades
Celibato, bênção a gays: como pensam os cotados a papa?
Vaticano: conclave começará no dia 7
Blog: integrantes do governo defendem que Lupi se demita
Aliado, Ciro Gomes silencia sobre fraudes
Ucrânia terá 3 dias de cessar-fogo em maio, anuncia Putin
Trégua celebra o Dia da Vitória contra Hitler
Cela com TV e sem visitas: o fim de semana de Collor
STF retoma hoje o julgamento sobre prisão
Concurso da PF: veja as funções do cargo com o maior salário
1º de maio é feriado; veja  próximas folgas de 2025
Jardim explica por que Dudu foi afastado do Cruzeiro e rebate crise
Atitude de Dudu após derrota motivou afastamento; bastidores
Vasco quer Diniz, mas faz contato por Thiago Carpini, técnico do Vitória
Pedrinho demite o 3º técnico
```

Figure 2: resultado

Esses dados agora podem ser utilizados em análises, armazenados em bases de dados ou apresentados em relatórios.

Considerações Finais

Neste capítulo, vimos a aplicação prática dos conceitos de **web scraping** utilizando **requests** e **BeautifulSoup**:

- Aprendemos a importar as bibliotecas necessárias;
- Estabelecemos conexão com uma página web;
- Analisamos a estrutura do HTML para identificar os dados desejados;
- Extraímos e exibimos as informações com sucesso.

Este é apenas o início da utilização do BeautifulSoup. A prática de inspecionar diferentes páginas e adaptar o código conforme as variações estruturais é essencial para o domínio completo da técnica de scraping.

10. Extraindo dados do IBGE

Neste capítulo, avançaremos em nossos conhecimentos sobre **Web Scraping** criando um projeto completo: a extração de dados de **estados e cidades** do site do **IBGE**.
Diferente das aulas anteriores, aqui enfrentaremos as particularidades reais de um site — o que é essencial para desenvolver a habilidade de análise e adaptação a diferentes páginas da web.

Nosso objetivo será criar uma **função** que, a partir da **sigla de um estado (UF)**, retorne um conjunto de informações estruturadas em um **DataFrame** organizado.

Bibliotecas Necessárias

Utilizaremos as seguintes bibliotecas:

- **requests**: para realizar a conexão com o site;
- **BeautifulSoup**: para interpretar e navegar pelo código HTML da página;
- **pandas**: para estruturar os dados em formato de tabela (DataFrame).

A instalação, será semelhante ao que vimos no capítulo anterior, com a adição da Biblioteca Pandas:

```
Pip install pandas
```

O mesmo vale para as importações. Seguem o mesmo modelo com a adição do Pandas:

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
```

Entendendo a Lógica da Página

A página inicial de cidades e estados do IBGE é: https://www.ibge.gov.br/cidades-e-estados/ . Ao pesquisarmos um estado (ex: “São Paulo”), o site altera a URL para: https://www.ibge.gov.br/cidades-e-estados/sp.html . Ou seja, **a URL muda apenas na sigla do estado**!
Isso nos permite montar a URL dinamicamente, bastando informar a sigla correta.

Exemplos:

Estado	Sigla	URL resultante
São Paulo	SP	https://www.ibge.gov.br/cidades-e-estados/sp.html
Rio Grande do Sul	RS	https://www.ibge.gov.br/cidades-e-estados/rs.html

Construindo a Função

A função que criaremos será chamada `scraping_UF`. Ela receberá como parâmetro a sigla do estado.

1. Montando a URL Dinâmica

```
def scraping_UF(UF: str):
    uf_url = f"https://www.ibge.gov.br/cidades-e-estados/{UF.lower()}.html"
```

Observação: Usamos `lower()` para garantir que a sigla esteja em letras minúsculas, como o site exige.

2. Simulando um Navegador

Alguns sites podem bloquear requisições feitas diretamente por scripts. Para evitar isso, usamos um cabeçalho (`headers`) que simula uma requisição feita por um navegador real:

```
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'
}
```

3. Realizando a Requisição

Agora realizamos a conexão com o site:

```
response = requests.get(uf_url, headers=headers)
```

Se a resposta for **200**, significa que a conexão foi estabelecida com sucesso.

4. Interpretando o Código HTML

```
soup = BeautifulSoup(response.content, 'html.parser')
```

Utilizamos o `BeautifulSoup` para interpretar o conteúdo da página.

5. Localizando os Dados

Analisando o HTML pelo navegador (tecla `F12` > `Inspecionar`), identificamos que os dados de interesse (área, população, governador etc.) estão dentro de elementos:

- `div` com classe **indicador**;
- Dentro desta `div`, o rótulo (ex: "Área territorial") está na classe **ind-label**;
- O valor (ex: "248.222,362 km²") está na classe **ind-value**.

Vamos buscar essas informações:

```
indicadores = soup.select('div.indicador')
```

6. Organizando os Dados em um Dicionário

Criamos um dicionário com as informações extraídas:

```
uf_dict = {}

for dado in indicadores:
    chave = dado.select_one('.ind-label').text.strip()
    valor = dado.select_one('.ind-value').text.strip()
    uf_dict[chave] = valor
```

7. Limpando os Dados

Alguns valores vêm acompanhados de informações adicionais como o ano entre colchetes `[2019]` e caracteres estranhos.
Queremos limpar esses valores para manter apenas o dado principal. Implementamos a limpeza:

```
for chave in uf_dict:
    if "[" in uf_dict[chave]:
        uf_dict[chave] = uf_dict[chave].split('[')[0].strip()
```

Assim, eliminamos qualquer texto que venha depois dos colchetes.

8. Retornando um DataFrame

Para facilitar a visualização e manipulação posterior, transformamos o dicionário em um DataFrame:

```
return pd.DataFrame(uf_dict.items(), columns=['Descrição', 'Valor'])
```

Função Final

Aqui está o exemplo completo de código, utilizando todos os conceitos abordados neste capítulo. Além disso, incluímos uma interação com o usuário, solicitando que ele informe a sigla do estado para o qual deseja consultar as informações.

```
# Importamos as bibliotecas necessárias
import requests
from bs4 import BeautifulSoup
import pandas as pd

# Definimos a função principal que irá realizar o scraping para o estado informado
def scraping_UF(UF: str):
    # Monta a URL específica para o estado desejado
    uf_url = f"https://www.ibge.gov.br/cidades-e-estados/{UF.lower()}.html"

    # Definimos os cabeçalhos da requisição para simular um navegador real
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'
    }

    # Fazemos a requisição para o site do IBGE
    response = requests.get(uf_url, headers=headers)

    # Verificamos se a requisição foi bem-sucedida (status 200)
    if response.status_code != 200:
        raise Exception('Erro na conexão com o site do IBGE.')

    # Interpretamos o conteúdo HTML retornado usando BeautifulSoup
    soup = BeautifulSoup(response.content, 'html.parser')

    # Selecionamos todas as divs que possuem os indicadores de interesse
    indicadores = soup.select('div.indicador')

    # Criamos um dicionário vazio para armazenar os dados extraídos
    uf_dict = {}

    # Percorremos cada indicador encontrado
    for dado in indicadores:
        # Extraímos o texto da descrição do indicador (ex: "Área territorial")
        chave = dado.select_one('.ind-label').text.strip()
        # Extraímos o texto do valor do indicador (ex: "248.222,362 km²")
        valor = dado.select_one('.ind-value').text.strip()
        # Adicionamos o par descrição-valor no dicionário
        uf_dict[chave] = valor

    # Limpamos os valores que possam conter referências bibliográficas (entre colchetes)
    for chave in uf_dict:
        if "[" in uf_dict[chave]:
            uf_dict[chave] = uf_dict[chave].split('[')[0].strip()

    # Convertendo o dicionário em um DataFrame organizado
    return pd.DataFrame(uf_dict.items(), columns=['Descrição', 'Valor'])
```

```
# ------------------------------
# Agora, parte adicional para interação com o usuário:

# Solicitamos que o usuário digite a sigla do estado desejado
estado = input("Digite a sigla do estado que deseja consultar (ex: SP, RJ, MG):
↪  ").strip().upper()

# Chamamos a função passando a sigla informada e armazenamos o resultado
dados_estado = scraping_UF(estado)

# Imprimimos o DataFrame completo com os dados do estado
print(dados_estado)
```

Ao executarmos este código, seremos solicitados a informar a sigla do estado que desejamos consultar.
Por exemplo, ao informarmos a sigla **PE** (que corresponde a Pernambuco), receberemos um resultado semelhante ao apresentado abaixo:

```
Digite a sigla do estado que deseja consultar (ex: SP, RJ, MG):  PE
                                  Descrição                          Valor
0                                 Governador  RAQUEL TEIXEIRA LYRA LUCENA
1                                    Capital                        Recife
2                                   Gentílico                    pernambucano
3                            Área Territorial                98.067,877 km²
4                  População no último censo              9.058.931 pessoas
5                      Densidade demográfica                  92,37 hab/km²
6                         População estimada              9.539.029 pessoas
7          Matrículas no ensino fundamental           1.229.419 matrículas
8      IDH Índice de desenvolvimento humano                          0,719
9        Total de receitas brutas realizadas           53.294.049.711,50 R$
10       Total de despesas brutas empenhadas           48.872.197.159,80 R$
11  Rendimento mensal domiciliar per capita                       1.453 R$
12                          Total de veículos             3.730.069 veículos
```

Figure 3: resultado

Conclusão

Neste capítulo, aprendemos como realizar o processo de **web scraping** para coletar dados diretamente do site do IBGE, de forma automática.
Exploramos a construção de uma função que:

- Monta uma URL baseada na sigla do estado informada pelo usuário.
- Realiza uma requisição simulando o acesso de um navegador.
- Extrai informações relevantes usando a biblioteca **BeautifulSoup**.
- Organiza os dados extraídos em um **DataFrame** do **pandas**.

Com essa técnica, é possível automatizar a coleta de dados de maneira eficiente e prática, facilitando análises posteriores.
Agora, você já tem uma base sólida