

Ficha de Avaliação Final	
Curso:	UFCD 10793
UFCD/Módulo/Temática:	UFCD 10793 - Fundamentos de Python
Ação:	10793_2/AT & 10793_5/N
Formador/a:	Sandra Liliana Meira de Oliveira
Data:	março de 2025
Nome do Formando/a:	

O presente documento é composto por uma parte teórica e uma parte prática. A resolução da parte prática corresponde à realização da Ficha de Avaliação Final.

WEB Scrapping	2
Os princípios básicos do <i>web scraping</i>	3
O <i>crawler</i>	3
O <i>scraper</i>	3
O processo de <i>web scraping</i>	3
Para que é utilizado o <i>web scraping</i> ?	4
Exemplos de Web crawlers	6
Bibliotecas do Python para Web Scraping	6
Atividade 1 – Biblioteca BeautifulSoup (5 valores)	8
Atividade 2 – Biblioteca Scrappy (5 valores)	12
Atividade 3 – Biblioteca Selenium (4 valores)	16
Atividade 4 – Customização (6 valores)	19

Teoria

WEB Scrapping

O *web scraping* é o processo de recolha de dados estruturados da web de uma forma automatizada. É também conhecido como *web data extraction* ou *data scraping*. Alguns dos principais casos de utilização de *web scraping* incluem monitorização de preços, inteligência de preços, monitorização de notícias, geração de *leads* e análise de mercado, entre muitos outros.

Em geral, a extração de dados da Internet é utilizada por pessoas e empresas que querem fazer uso da vasta quantidade de dados disponíveis na Internet para o público em geral, de forma a tomar decisões mais inteligentes.

Se alguma vez copiou e colou informação de um website, desempenhou a mesma função que qualquer *web scraper*, apenas numa escala microscópica e manual. Ao contrário do processo de extração manual de dados, o *web scraping* utiliza a automatização inteligente para recuperar centenas, milhões, ou mesmo milhares de milhões de pontos de dados da fronteira aparentemente interminável da Internet.

O *web scraping* é bastante popular, e mais do que uma utilidade moderna, o verdadeiro poder do *web scraping* reside na sua capacidade de construir e alimentar algumas das aplicações comerciais mais revolucionárias do mundo.



Os princípios básicos do *web scraping*

É extremamente simples, na verdade, e funciona através de duas partes: um *web crawler* e um *web scraper*. O primeiro é o cavalo, e o segundo é a carruagem. O *crawler* conduz o *scraper*, como se fosse pela mão, através da Internet, onde extrai os dados solicitados.

O *crawler*

Um *web crawler*, a que geralmente chamamos "spider", é uma inteligência artificial que navega na Internet para indexar e pesquisar conteúdos, seguindo ligações e explorando, como uma pessoa com demasiado tempo para se dedicar a esse trabalho. Em muitos projetos, primeiro explora/rasteja ("crawl") pela Internet ou um website específico para descobrir URLs que depois passa para o seu *scraper*.

O *scraper*

Um *web scraper* é uma ferramenta especializada concebida para extrair dados de uma página da Internet com precisão e rapidez. Os *web scrapers* variam muito em conceção e complexidade, dependendo do projeto. Uma parte importante de cada *scraper* são os localizadores de dados (ou seccionadores) que são utilizados para encontrar os dados que se pretendem extrair do ficheiro HTML - normalmente aplicam-se *xpath*, *css selectors*, *regex* ou uma combinação de todos.

O processo de *web scraping*

Um processo típico de *web scraping* passa pelas seguintes etapas:

- Identificar o website de destino
- Recolher URLs das páginas de onde se pretende extrair dados
- Fazer um pedido a estes URLs para obter o HTML da página
- Utilizar localizadores para encontrar os dados em HTML
- Guardar os dados num ficheiro JSON, CSV ou outro formato estruturado

Bastante simples, certo? Sim, mas apenas se tiver um projeto pequeno. Infelizmente, existem

bastantes desafios a enfrentar se precisar de dados em grande escala. Por exemplo, manter o *scraper* para o caso do *layout* do website mudar, gerir *proxies*, executar *javascript* ou dar a volta a *antibots*.

Tratam-se então de problemas profundamente técnicos e que podem consumir muitos recursos. Assim, para resolver esse tipo de desafios, existem [serviços que reúnem as suas necessidades](#), com a implementação dos *scrapers*, criação da infraestrutura, estruturação dos dados e entrega no formato e na periodicidade pretendidos. Esta é parte da razão pela qual muitas empresas optam por externalizar os seus projetos de dados web.

Para que é utilizado o *web scraping*?

São diversas as utilidades que se podem tirar desta tecnologia. Seguem alguns exemplos de casos de uso para ilustrar melhor o potencial desta tecnologia.

1. Inteligência de preços

A inteligência de preços é o caso de utilização mais comum para *web scraping*. Extrair informação sobre produtos e preços de websites de *e-commerce*, transformando-a depois em inteligência. É assim uma parte importante das empresas modernas de *e-commerce* que querem tomar melhores decisões de preços/marketing com base nesses dados.

2. Análise de mercado

Os estudos de mercado são críticos e devem ser conduzidos pela informação mais precisa disponível. A alta qualidade, o elevado volume e a elevada perspicácia dos dados extraídos da Internet de todas as formas e tamanhos alimentam a análise do mercado e a inteligência empresarial em todo o mundo.

3. Dados alternativos para finanças

O processo de tomada de decisão nunca foi tão informado, nem os dados tão ricos. Além disso, as empresas líderes mundiais consomem cada vez mais dados extraídos da Internet, dado o seu incrível valor estratégico para o mercado financeiro.

4. Imobiliário

A transformação digital do setor imobiliário nos últimos vinte anos ameaça perturbar as empresas tradicionais e criar novos participantes poderosos no setor. Ao incorporar dados de produtos extraídos da Internet nos negócios quotidianos, os agentes e corretores podem proteger-se contra todo o tipo de concorrência online e tomar decisões informadas no mercado.

5. Monitorização de conteúdo e de notícias

Os meios de comunicação modernos podem criar um valor excecional ou uma ameaça existencial ao seu negócio, num único ciclo de notícias. Se é uma empresa que depende de análises noticiosas de forma atempada ou uma empresa que aparece frequentemente nas notícias, extrair notícias da Internet é a derradeira solução para monitorizar, agregar e analisar as histórias mais críticas da sua indústria.

6. Geração de leads

A geração de leads é uma atividade de marketing/vendas crucial para todas as empresas.

7. Monitorização de marca

No mercado atual, altamente competitivo, é uma prioridade máxima proteger a sua reputação online. Quer venda os seus produtos online e tenha uma política de preços rigorosa que precisa de aplicar ou apenas queira saber como as pessoas veem os seus produtos online, a monitorização de marca com *web scraping* pode dar-lhe este tipo de informação.

8. Automação de negócios

Em algumas situações pode ser incómodo ter acesso aos seus dados. Talvez tenha alguns dados no seu próprio website ou no website do seu parceiro de que necessita de uma forma estruturada. Mas, em vez de tentar aventurar-se por sistemas internos complicados, faz sentido criar um *scraper* e simplesmente obter esses dados.

9. Monitorização MAP

A monitorização *Minimum advertised price* (MAP) é a prática padrão para assegurar que os preços online de uma marca se encontram alinhados com a sua política de preços. Com uma imensidão de revendedores e distribuidores, é impossível controlar os preços de forma manual. É por isso que o *web scraping* se afigura como uma ferramenta útil, permitindo-lhe vigiar os preços dos seus produtos sem mexer um dedo.

Exemplos de Web crawlers

- **DataparkSearch**
- **Wget**
- **HTTrack**
- **JSpider**
- **Methabot**
- **Pavuk**
- **WebSPHINX**
- **YaCy**
- **Crawljax**
- **Yahoo! Slurp** é o nome do crawler do Yahoo!.
- **Msnbot** é o nome do crawler do Bing - Microsoft.
- **Googlebot** é o nome do crawler do Google.
- **Methabot** é um crawler com suporte a scripting escrito em C.
- **arachnode.net** é um Web crawler open-source usando a plataforma .NET e escrito em C#
- **Goutte** é um Web Scraper para criar um crawler desenvolvido em PHP por Fabien Potencier usando o Symfony.
- **DuckDuckBot** é o web crawler do DuckDuckGo.
- **Patent2net** é um crawler especializado em encontrar, organizar e disponibilizar patentes depositadas na Espacenet.
- **OpenWebSpider** - <http://www.openwebspider.org>

Bibliotecas do Python para Web Scraping

O Python oferece várias bibliotecas que permitem efetuar scraping. As mais utilizadas são:

- **Selenium**: biblioteca de testes web usada para automatizar as atividades do navegador (<https://selenium-python.readthedocs.io/index.html>,
<https://www.browserstack.com/guide/python-selenium-to-run-web-automation-test>);

- **BeautifulSoup**: pacote para análise de documentos HTML e XML. Cria árvores de análise (<https://pypi.org/project/beautifulsoup4/>);
- **Requests**: tem como finalidade tornar as solicitações HTTP mais amigáveis e simples (<https://pypi.org/project/requests/>).
- **Scrapy**: framework open source para extrair informações em websites (<https://docs.scrapy.org/en/latest/>).

Na implementação de web scrapping é importante ter conhecimento acerca da estrutura de uma página web nomeadamente no que concerne a tags html e seletores css (<https://www.jcchouinard.com/css-selectors-for-web-scraping/>

https://www.w3schools.com/cssref/css_selectors.php

<http://api.jquery.com/category/selectors/>

<https://en.wikipedia.org/wiki/CSS#Selector>

<https://docs.scrapy.org/en/latest/topics/selectors.html>

<https://coderslegacy.com/python/scrapy-css-selectors-tutorial/>

<https://www.geeksforgeeks.org/beautifulsoup-find-tags-by-css-class-with-css-selectors/>

<https://www.geeksforgeeks.org/find-the-text-of-the-given-tag-using-beautifulsoup/?ref=lbp>).

Prática

As seguintes atividades têm como objetivo a configuração e a utilização de algumas bibliotecas de Python no processo de Web Scrapping.

A concretização deste projeto implica a realização de todas as atividades práticas abaixo indicadas. Os ficheiros resultantes deverão ser anexados à tarefa de avaliação final.

Atividade 1 – Biblioteca BeautifulSoup (5 valores)

Esta atividade é uma atividade orientada, que implica, apenas, a reprodução do código, de forma a entrar em contato com a biblioteca BeautifulSoup.

Como funciona o web scrapping?

Raspar uma página web significa solicitar dados específicos de uma página web de destino. Quando se raspa uma página, o código que se escreve envia o pedido para o servidor que hospeda a página de destino. O código descarrega depois a página, extraindo apenas os elementos da página definidos, inicialmente na tarefa de rastreamento.

Por exemplo, digamos que estamos à procura de dados alvo em etiquetas de títulos H3. Escreveríamos código para um web scrapping que procurasse especificamente essa informação. O scraper funciona em três passos:

Passo 1: Enviar um pedido para o servidor para descarregar o conteúdo do site.

Passo 2: Filtrar o HTML da página para procurar as etiquetas desejadas (neste caso H3).

Passo 3: Copiar o texto dentro das etiquetas de destino, produzindo os resultados no formato previamente especificado no código.

A biblioteca transforma um documento HTML complexo numa árvore de objetos Python. Também converte automaticamente o documento para Unicode, para que não tenha de pensar em codificações. A biblioteca BeautifulSoup suporta o analisador de HTML incluído na biblioteca padrão de Python, mas também suporta vários analisadores de Python de terceiros como lxml ou html5lib.

1. Instala as seguintes bibliotecas:

pip install beautifulsoup4

pip install html5lib

pip install requests

2. Cria a pasta BeautifulSoupDemo. Abre a mesma no Visual Studio Code. Dentro da pasta adiciona o ficheiro BeautifulSoupDemo.py.
3. **Passo 1:** Tens de enviar um pedido HTTP para o servidor da página que pretendes raspar. O servidor responde enviando o conteúdo HTML da página web. Uma vez que estamos a utilizar Python para os nossos pedidos, precisamos de uma biblioteca HTTP de terceiros, e iremos utilizar a biblioteca **requests**.

Começa por importar a biblioteca de Requests (pedidos) e fazer um simples pedido GET para o URL – escolhi **https://www.accuweather.com** porque tem uma estrutura HTML simples e me permitirá demonstrar facilmente o potencial da BeautifulSoup.

Assim sendo, coloca no ficheiro .py criado anteriormente o seguinte código. O resultado deverá ser o código HTML completo para esta página

```
import requests

url="https://www.accuweather.com"
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36'}

r= requests.get(url, headers=headers)
```

Passo 2: Agora que temos o conteúdo HTML, precisamos de analisar os dados. Para isso, vamos utilizar BeautifulSoup com um analisador html5lib. Precisamos de passar dois valores para BeautifulSoup():

#1: Cadeia HTML do sítio web; 'r.content'

#2: Que analisador de HTML usar; 'html5lib'













Altera o código do ficheiro .py para:

```
import requests
from bs4 import BeautifulSoup

url="https://www.accuweather.com"
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36'}
r= requests.get(url, headers=headers)
```

```
soup = BeautifulSoup(r.content, 'html5lib')
```

Passo 3: Num browser à escolha vai ao url da página e pede para ver o código da página. Vamos procurar a camada superior da tabela. Abaixo temos um print do website

Braga		21°
Bragança		22°
Castelo Branco		18°
Coimbra		19°
Évora		22°
Faro		27°
Funchal		23°
Guarda		24°
Leiria		20°
Lisboa		20°
Marinha Grande		20°
Óbidos		20°

Podemos ver a classe mãe “nearby-locations-list” e as a classes “nearby-location weather-car” quu contêm a informação que pretendemos recolher na forma de tags <a>. Dentro das tags <a> a localização encontra-se na classe “text title no-wrap” numa tag e a temperatura na classe “text temp” numa tag .

Assim para percorrermos e extrairmos a informação que nos interessa basta colocar o seguinte código:

```
for mainrow in soup.findAll('div', attrs = {'class':['nearby-locations-list']}):
    for row in mainrow.findAll('a', attrs = {'class':['nearby-location weather-card']}):

        location = row.find('span', attrs = {'class':'text title no-wrap'}).get_text().strip()
        temp = row.find('span', attrs = {'class':'text temp'}).get_text().strip()
```

Vamos também armazenar a informação obtuda num Dataframe, imprimi-la e exportá-la para ficheiro csv.

Assim sendo, altera a script anterior para:

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

url="https://www.accuweather.com"
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'}

response = requests.get(url, headers=headers)
soup = BeautifulSoup(response.content, 'html5lib')

for mainrow in soup.findAll('div', attrs = {'class':['nearby-locations-list']}):
    for row in mainrow.findAll('a', attrs = {'class':['nearby-location weather-card']}):
        location = row.find('span', attrs = {'class':'text title no-wrap'}).get_text().strip()
        temp = row.find('span', attrs = {'class':'text temp'}).get_text().strip()

        df = pd.DataFrame({'location': location, 'temp': temp})
        df.to_csv('weather_data.csv', index=False)
```

```
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36'}
r= requests.get(url, headers=headers)

soup = BeautifulSoup(r.content, 'html5lib')

data=[]

for mainrow in soup.findAll('div', attrs = {'class':['nearby-locations-list']}):
    for row in mainrow.findAll('a', attrs = {'class':['nearby-location weather-card']}):





















        location = row.find('span', attrs = {'class':'text title no-wrap'}).get_text().strip()
        temp = row.find('span', attrs = {'class':'text temp'}).get_text().strip()

        one = {}
        one['localização']=location
        one['temperatura']=temp

        data.append(one)

df=pd.DataFrame(data)
print(df)
df.to_csv("temperaturas.csv",index="False",encoding="utf-8-sig")
```

Conseguimos desta forma ler os dados da tabela:

Amadora		20°	Aveiro		19°
Braga		21°	Bragança		22°
Castelo Branco		18°	Coimbra		20°
Évora		22°	Faro		26°
Funchal		24°	Guarda		24°
Leiria		20°	Lisboa		20°
Marinha Grande		20°	Óbidos		21°
Ponta Delgada		24°	Porto		18°
Santarém		20°	Setúbal		18°
Vila Nova da Cerveira		20°	Viseu		19°

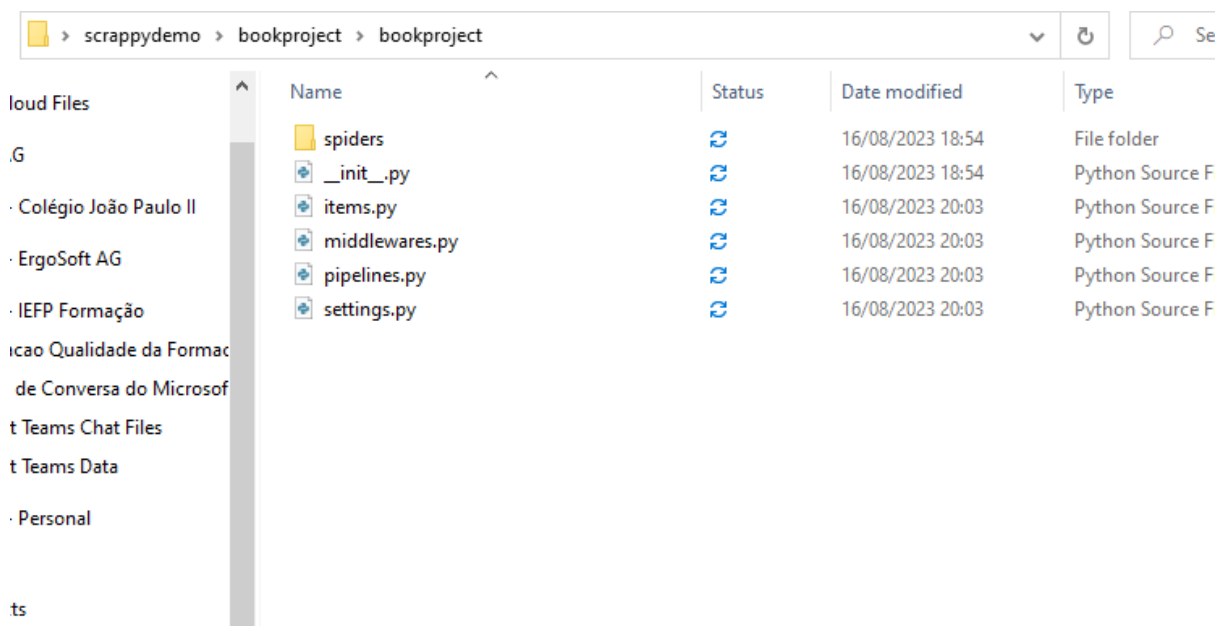
Atividade 2 – Biblioteca Scrappy (5 valores)

Esta atividade é uma atividade orientada, que implica, apenas, a reprodução do código, de forma a entrar em contato com a biblioteca Scrappy.

1. Instala a biblioteca scrappy: **pip install Scrappy**;
2. Cria uma pasta com o nome **scrappydemo**;
3. Na linha de comandos do Windows coloca-te na pasta anterior e executa o seguinte comando: **scrapy startproject bookproject**.

Nota: O scrapy não é apenas uma biblioteca que podemos apenas importar. Essa ferramenta necessita que seja criada uma estrutura de pastas e ficheiros nos quais precisamos configurar as nossas **spiders** para visitar as páginas e extrair os dados.

Obterás dentro da pasta a seguinte estrutura:

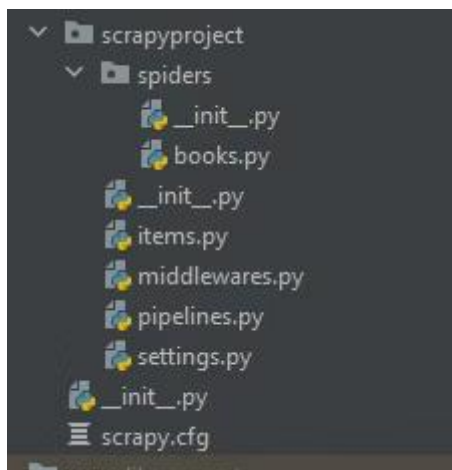


Name	Status	Date modified	Type
spiders		16/08/2023 18:54	File folder
__init__.py		16/08/2023 18:54	Python Source F
items.py		16/08/2023 20:03	Python Source F
middlewares.py		16/08/2023 20:03	Python Source F
pipelines.py		16/08/2023 20:03	Python Source F
settings.py		16/08/2023 20:03	Python Source F

4. Ainda no terminal do Windows, localiza-te agora na pasta spiders criada na estrutura hierárquica do ponto anterior e execute o seguinte comando, que irá gerar uma aranha (books) que realizará o crawling do site.

```
scrapy genspider books books.toscrape.com
```

Obterás a seguinte estrutura:



5. No ponto anterior foi criado um ficheiro books.py. Vamos agora customizar a nossa aranha. Altera o código nesse ficheiro para:

```
import scrapy

class BooksSpider(scrapy.Spider):
    name = 'books'

    def start_requests(self):
        URL = 'https://books.toscrape.com/'
        yield scrapy.Request(url=URL, callback=self.response_parser)

    def response_parser(self, response):
        for selector in response.css('article.product_pod'):
            yield {
                'title': selector.css('h3 > a::attr(title)').extract_first(),
                'price': selector.css('.price_color::text').extract_first()
            }

        next_page_link = response.css('li.next a::attr(href)').extract_first()
        if next_page_link:
            yield response.follow(next_page_link, callback=self.response_parser)
```

O script acima contém dois geradores: o `start_requests()` e o `response_parser()`. O gerador `start_requests()` é executado automaticamente sempre que um comando crawl é emitido. Aqui, ele recupera o conteúdo da URL e emite uma chamada de volta para `response_parser()`.

O gerador `response_parser()`, quando executado, extrai, da `resposta` iterável, as informações do produto desejado. Quando as chamadas de todos os 20 produtos na resposta atual forem concluídos, será utilizado o método `response.follow()` para recuperar o conteúdo da próxima página. O método `follow()` chama novamente o `response_parser()` para extrair e produzir produtos da nova página. O ciclo continua até que `next_page_link` se torne NULL ou vazio.

6. Para executar o crawler e guardar os dados num ficheiro .csv. Na linha de comandos do Windows, localiza-te na pasta **spiders** e executa o seguinte comando.

scrapy crawl -o out.csv books

7. Podemos também executar o projeto scrapy dentro de uma script .py. Para tal, altera o código da “spider” books.py para:

```
import csv
import scrapy
from scrapy import signals
from scrapy.crawler import CrawlerProcess
from scrapy.signalmanager import dispatcher

class BooksSpider(scrapy.Spider):
    name = 'books'

    def start_requests(self):
        URL = 'https://books.toscrape.com/'
        yield scrapy.Request(url=URL, callback=self.response_parser)

    def response_parser(self, response):
        for selector in response.css('article.product_pod'):
            yield {
                'title': selector.css('h3 > a::attr(title)').extract_first(),
                'price': selector.css('.price_color::text').extract_first()
            }

        next_page_link = response.css('li.next a::attr(href)').extract_first()
        if next_page_link:
            yield response.follow(next_page_link, callback=self.response_parser)

def book_spider_result():
    books_results = []
```

```
def crawler_results(item):
    books_results.append(item)

dispatcher.connect(crawler_results, signal=signals.item_scraped)
crawler_process = CrawlerProcess()
crawler_process.crawl(BooksSpider)
crawler_process.start()
return books_results

if __name__ == '__main__':
    books_data=book_spider_result()

    keys = books_data[0].keys()
    with open('books_data.csv', 'w', newline='') as output_file_name:
        writer = csv.DictWriter(output_file_name, keys)
        writer.writeheader()
        writer.writerows(books_data)
```

O `start_requests()` e `response_parser()` são iguais ao código anterior. O `__main__` serve como ponto de partida para execução direta. Chama a função `book_spider_result()` e espera que esta retorne um valor.

O `book_spider_result()` funciona da seguinte forma:

- Define o *controller dispatcher* para executar a função `crawler_results()` ao até encontrar um sinal `item_scraped`. O `item_scraped` é gerado sempre que o spider “raspa” um item do alvo.
- Cria um processo rastreador para o `BooksSpider` inicia-o.
- Sempre que o `BookSpider` conclui a raspagem de um item, ele emite o sinal `item_scraped`. Isso faz com que a função `crawler_results()` seja executada e anexe esse item copiado à lista `books_results`.
- Depois que o processo rastreador terminar de recolher os itens, o `book_spider_result()` retornará a lista `books_results`.

A função `__main__`, no retorno, grava os `books_data` retornados no ficheiro “books_data.csv”.

Basta navegar na pasta “spiders” no projeto Scrapy e clicar duas vezes no ficheiro “books.py”, ou abrir este ficheiro no Visual Studio Code e executá-lo.

8. Abre a script books.py no Visual Studio Code e executa-a. É gerado um ficheiro csv na pasta raíz do projeto com a seguinte estrutura:

	A	B
1	title	price
2	A Light in the Attic	£51.77
3	Tipping the Velvet	£53.74
4	Soumission	£50.10
5	Sharp Objects	£47.82
6	Sapiens: A Brief History of Humankind	£54.23
7	The Requiem Red	£22.65
8	The Dirty Little Secrets of Getting Your Dream Job	£33.34
9	The Coming Woman: A Novel Based on the Life of the Infamous Feminist, Victoria Woodhull	£17.93
10	The Boys in the Boat: Nine Americans and Their Epic Quest for Gold at the 1936 Berlin Olympics	£22.60
11	The Black Maria	£52.15
12	Starving Hearts (Triangular Trade Trilogy, #1)	£13.99
13	Shakespeare's Sonnets	£20.66
14	Set Me Free	£17.46
15	Scott Pilgrim's Precious Little Life (Scott Pilgrim #1)	£52.29
16	Rip it Up and Start Again	£35.02
17	Our Band Could Be Your Life: Scenes from the American Indie Underground, 1981-1991	£57.25
18	Olio	£23.88
19	Mesaerion: The Best Science Fiction Stories 1800-1849	£37.59
20	Libertarianism for Beginners	£51.33
21	It's Only the Himalayas	£45.17
22	In Her Wake	£12.84
23	How Music Works	£37.32
24	Foolproof Preserving: A Guide to Small Batch Jams, Jellies, Pickles, Condiments, and More: A Foolproof	£30.52
25	Chase Me (Paris Nights #2)	£25.27
26	Black Dust	£34.53
27	Birdsong: A Story in Pictures	£54.64
28	America's Cradle of Quarterbacks: Western Pennsylvania's Football Factory from Johnny Unitas to Joe	£22.50

scrappydemo > bookproject

Name	Status	Date modified	Type	Size
bookproject	✓	16/08/2023 20:07	File folder	
books_data.csv	🔄	16/08/2023 20:22	Microsoft Excel C...	48 KB
scrappy.cfg	✓	16/08/2023 20:03	Configuration Sou...	1 KB

Atividade 3 – Biblioteca Selenium (4 valores)

Para a realização desta tarefa cria a pasta seleniumDemo e dentro da mesma o ficheiro seleniumdemo.py.

Passo 1: Instala a biblioteca

Pip install selenium

Esta biblioteca permite-nos interagir com a página para além da recolha de informação na página.

Passo 2: Definir o driver que iremos utilizar (definição do browser). A biblioteca tem que executar o browser para que seja possível a leitura da página.

```
driver = webdriver.Chrome()
```

Passo 3: Navegar na página

(<https://www.selenium.dev/documentation/webdriver/interactions/navigation/>).

```
driver.get("https://www.selenium.dev/selenium/web/web-form.html")
```

Passo 4: Solicitar Informações ao browser

Existem vários tipos de informações sobre o navegador que você pode solicitar, incluindo alças de janela, tamanho/posição do navegador, cookies, alertas, etc.(

<https://www.selenium.dev/documentation/webdriver/interactions/>)

```
title = driver.title
```

Passo 4: Estabelecer uma estratégia de espera

Sincronizar o código com o estado atual do navegador é um dos maiores desafios do Selenium, e fazê-lo bem é um tópico avançado.

Essencialmente, deseja-se certificar de que o elemento esteja na página antes de tentar localizá-lo e de que o elemento esteja em um estado interativo antes de tentar interagir com ele.

Uma espera implícita raramente é a melhor solução, mas é a mais fácil de demonstrar aqui, então vamos usá-la como um espaço reservado.

```
driver.implicitly_wait(0.5)
```

<https://www.selenium.dev/documentation/webdriver/waits/>

Passo 5: Encontrar o elemento

A maioria dos comandos na maioria das sessões do Selenium são relacionados a elementos, e não se pode interagir com um sem primeiro encontrar um elemento

<https://www.selenium.dev/documentation/webdriver/elements/>

```
text_box = driver.find_element(by=By.NAME, value="my-text")  
submit_button = driver.find_element(by=By.CSS_SELECTOR, value="button")
```

Passo 6: Agir no elemento

Existem apenas algumas ações a serem executadas em um elemento que são utilizadas frequentemente.

<https://www.selenium.dev/documentation/webdriver/elements/interactions/>

```
text_box.send_keys("Selenium")  
submit_button.click()
```

Passo 7: Solicitar informações do elemento

Os elementos armazenam muitas informações que podem ser solicitadas .

<https://www.selenium.dev/documentation/webdriver/elements/information/>

```
value = message.text
```

Passo 8: Terminar a sessão

```
driver.quit()
```

Encerra o processo do driver, que por padrão também fecha o navegador. Nenhum outro comando pode ser enviado para esta instância do driver

Juntando Tudo

Coloca o seguinte código na script anteriormente criada.

```
from selenium import webdriver  
from selenium.webdriver.chrome.options import Options  
from selenium.webdriver.common.by import By  
  
def test_eight_components():  
  
    driver = webdriver.Firefox()  
  
    driver.get("https://www.selenium.dev/selenium/web/web-form.html")  
  
    title = driver.title  
    assert title == "Web form"  
  
    driver.implicitly_wait(0.5)  
  
    text_box = driver.find_element(by=By.NAME, value="my-text")  
    submit_button = driver.find_element(by=By.CSS_SELECTOR, value="button")  
  
    text_box.send_keys("Selenium")  
    submit_button.click()  
  
    message = driver.find_element(by=By.ID, value="message")  
    value = message.text
```

```
assert value == "Received!"  
  
#driver.quit()  
  
test_eight_components()
```

Com este script conseguimos efetuar a submissão do formulário na página.

Atividade 4 – Customização (6 valores)

Escolhe uma página e, utilizando a informação adquirida até ao momento, efetua o web scrapping de informação contida na mesma.

Bom Trabalho