‹packt›

2ND EDITION

Hands-On
Web Scraping with Python

Extract quality data from the web using
effective Python techniques

ANISH CHAPAGAIN

Hands-On
Web Scraping with Python

Extract quality data from the web using effective
Python techniques

Anish Chapagain

‹packt›

Hands-On Web Scraping with Python

Contributors

About the author

Anish Chapagain is a software engineer with a passion for data science and artificial intelligence processes and Python programming, which began around 2007. He has been working on web scraping, data analysis, visualization, and reporting-related tasks and projects for more than 10 years, and also works as a freelancer. Anish previously worked as a trainer, web/software developer, team leader, and banker, where he was exposed to data and gained insights into topics such as data mining, data analysis, reporting, information processing, and knowledge discovery. He has an MSc in computer systems from Bangor University (United Kingdom), and an executive MBA from Himalayan Whitehouse International College, Kathmandu, Nepal.

About the reviewers

Sanjan Rao is a senior data scientist with over five years of experience in machine learning, deep learning, and cybersecurity solutions. He has a master's degree with a specialty in machine learning and artificial intelligence. His primary interests lie in natural language processing, natural language inference, and reinforcement learning.

Joel Edmond Nguemeta is a full stack web developer from Cameroon. He is highly experienced in software development with Python, Django, JavaScript, and React in both monolithic and microservices architectures. He has worked as a freelance mentor/assessor at OpenClassrooms, supporting students in Python/Django and JavaScript/React application development, as well as assessing them in the Python/Django and JavaScript/React application developer courses. Furthermore, he has worked as a mobile developer at Tamangwa Shipping. He spends his time sharing his knowledge with everyone through his YouTube channel (`proxidev`). He is the co-founder of the start-up EasyLern.

Table of Contents

Part 2: Beginning Web Scraping

3

4

5

Part 3: Advanced Scraping Concepts

6

Working with the Secure Web   137

7

Data Extraction Using Web APIs   157

8

Using Selenium to Scrape the Web   173

9

Part 4: Advanced Data-Related Concepts

10

11

Machine Learning and Web Scraping   249

Part 5: Conclusion

12

After Scraping – Next Steps and Data Analysis   277

Preface

Web scraping is used to scrape and collect data from the web. The data collected from scraping is used to generate and identify patterns in the information.

In today's technical – or, more precisely, data-driven – fields and markets, quick and reliable information is in very high demand. Data collected in CSV or JSON format and from databases is processed to generate error-free and high-quality data, which is then analyzed, trained using machine learning algorithms, and plotted. The resulting information is carried forward for decision-making or supportive business intelligence-related tasks.

The chapters of this book are designed in such a way that each section helps you to understand certain important concepts and practical experiences. If you complete all the chapters of the book, then you will gain practice in scraping data from desired websites and analyzing and reporting data. You will also learn about the career paths and jobs related to web scraping, data analysis, reporting, visualization, and machine learning.

Who this book is for

This book is for Python programmers, data analysts, data reporters, machine learning practitioners, and anyone who wants to begin their professional or learning journey in the field of web scraping and data science. If you have a basic understanding of the Python programming language, you will easily be able to follow along with the book and learn about some advanced concepts related to data.

What this book covers

Chapter 1, *Web Scraping Fundamentals*, provides an introduction to web scraping and also explains the latest core web technologies and data-finding techniques.

Chapter 2, *Python Programming for Data and Web*, provides an overview of choosing and using Python for web scraping. The chapter also explores and explains the **World Wide Web** (**WWW**) and URL-based operations by setting up and using the necessary Python libraries, tools, and virtual environments.

Chapter 3, *Searching and Processing Web Documents*, provides an overview of and introduction to identifying, traversing, and processing web documents using XPath and CSS selectors. The chapter also explains scraping using lxml, collecting data in a file, parsing information from `robots.txt`, and exploring sitemaps.

Chapter 4, *Scraping Using Pyquery, a jQuery-Like Library for Python*, provides an introduction to a jQuery-like Python library: pyquery. This chapter provides information on installing and exploring pyquery's features on web documents. Examples of scraping using pyquery and writing data to JSON and CSV are also covered.

Chapter 5, *Scraping the Web with Scrapy and Beautiful Soup*, provides an overview and examples of using and deploying a popular web-crawling framework: Scrapy. It also introduces parsing and scraping using BeautifulSoup.

Chapter 6, *Working with the Secure Web*, provides an overview of dealing with secure web content, using sessions and cookies. The chapter also guides you through and explores scraping content by processing HTML form- and authentication-related issues, as well as providing a guide with examples of how to use proxies during HTTP communication.

Chapter 7, *Data Extraction Using Web APIs*, provides a detailed overview of the web API, its benefits when used with HTTP content, along with the data formats and patterns available in the API. The chapter also provides a few examples of scraping the web API.

Chapter 8, *Using Selenium to Scrape the Web*, introduces Selenium WebDriver, which helps automate actions in web browsers, and also covers how to use Selenium to scrape data.

Chapter 9, *Using Regular Expressions and PDFs*, provides a detailed overview of regular expressions and their usage and implementation using Python. The chapter also provides examples of data extraction using regular expressions and PDF documents using the `pypdf2` Python library.

Chapter 10, *Data Mining, Analysis, and Visualization*, provides an introduction to and detailed overview of data mining and data analysis using the `pandas` Python library and visualization using Plotly. The chapter also introduces the concept of exploratory data analysis using the `ydata_profiling` Python library.

Chapter 11, *Machine Learning and Web Scraping*, provides a detailed introduction to machine learning, a branch of artificial intelligence. The chapter also provides examples of a few machine learning topics using the `scikit-learn` Python library, along with conducting sentiment analysis from scraped and collected data.

Chapter 12, *After Scraping – Next Steps and Data Analysis*, provides an overview of and introduction to the next steps related to growing technologies, covering topics such as web requests and data processing in more detail. The chapter also provides information on and guides developers in exploring prospective careers and jobs relating to scraping and data.

To get the most out of this book

Having some basic or intermediate knowledge of Python programming will help you get the most out of the book's contents and examples.

Software/hardware covered in the book	Operating system requirements
Python 3.11	Windows, macOS, or Linux
Google Chrome or Mozilla Firefox	Windows, macOS, or Linux
Visual Studio Code or JetBrains PyCharm Community Edition	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Here, we have expressed our interest in only finding the information from the `content` attribute with the `<meta>` tag, which has the `name` attribute with the `keywords` and `description` values, respectively."

A block of code is set as follows:

```
source.find('a:contains("Web")') # [<a.menuitm>, <a>, <a>, <a>,
<a>]
source.find('a:contains("Web"):last').text()
# 'Web Scraper'
source.find('a:contains("Web"):last').attr('href') # '#'
```

Any command-line input or output is written as follows:

```
(secondEd) C:\HOWScraping2E> pip install jupyterlab
```

Outputs and comments inside blocks of code look as follows:

```
# [<a.menuitm>, <a>, <a>, <a>, <a>]
# 'Web Scraper'
# '#'
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "In addition, we will create separate files for **author** and **quotes**."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@ packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Hands-On Web Scraping with Python Second Edition*, we'd love to hear your thoughts! Please `https://packt.link/r/1837636214` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781837636211

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1:
Python and Web Scraping

In this part, you will be introduced to and get a detailed overview of web scraping and how it relates to Python programming. You will learn about various technologies related to websites and HTTP. In addition, application tools and the necessary Python libraries are also covered.

This part contains the following chapters:

- *Chapter 1*, Web Scraping Fundamentals
- *Chapter 2*, Python Programming for Data and Web

1

Web Scraping Fundamentals

This book about web scraping covers practical concepts with detailed explanations and example code. We will introduce you to the essential topics in extracting or scraping data (that is, high-quality data) from websites, using effective techniques from the web and the Python programming language.

In this chapter, we are going to understand basic concepts related to web scraping. Whether or not you have any prior experience in this domain, you will easily be able to proceed with this chapter.

The discussion of the web or websites in our context refers to pages or documents including text, images, style sheets, scripts, and video contents, built using a markup language such as HTML. It's almost a container of various content.

The following are a couple of common queries in this context:

- Why web scraping?
- What is it used for?

Most of us will have come across the concept of data and the benefits or usage of data in deriving information, decision-making, gaining insights from facts, or even knowledge discovery. There has been growing demand for data, or high-quality data, in most industries globally (such as *governance, medical sciences, artificial intelligence, agriculture, business, sport, and R&D*).

We will learn what exactly web scraping is, explore the techniques and technologies it is associated with, and find and extract data from the web, with the help of the Python programming language, in the chapters ahead.

In this chapter, we are going to cover the following main topics:

- What is web scraping?
- Understanding the latest web technologies
- Data-finding techniques

Technical requirements

You can use any **Operating System** (**OS**) (such as Windows, Linux, or macOS) along with an up-to-date web browser (such as Google Chrome or Mozilla Firefox) installed on your PC or laptop.

What is web scraping?

Scraping is a process of extracting, copying, screening, or collecting data. Scraping or extracting data from the web (a buildup of **websites**, **web pages**, and **internet-related resources**) for certain requirements is normally called web scraping. Data collection and analysis are crucial in information gathering, decision-making, and research-related activities. However, as data can be easily manipulated, web scraping should be carried out with caution.

The popularity of the internet and web-based resources is causing information domains to evolve every day, which is also leading to growing demand for raw data. Data is a basic requirement in the fields of science and technology, and management. Collected or organized data is processed, analyzed, compared with historical data, and trained using **Machine Learning** (**ML**) with various algorithms and logic to obtain estimations and information and gain further knowledge.

Web scraping provides the tools and techniques to collect data from websites, fit for either personal or business-related needs, but with legal considerations.

As seen in *Figure 1.1*, we obtain data from various websites based on our needs, write/execute crawlers, collect necessary content, and store it. On top of this collected data, we do certain analyses and come up with some information related to decision-making.
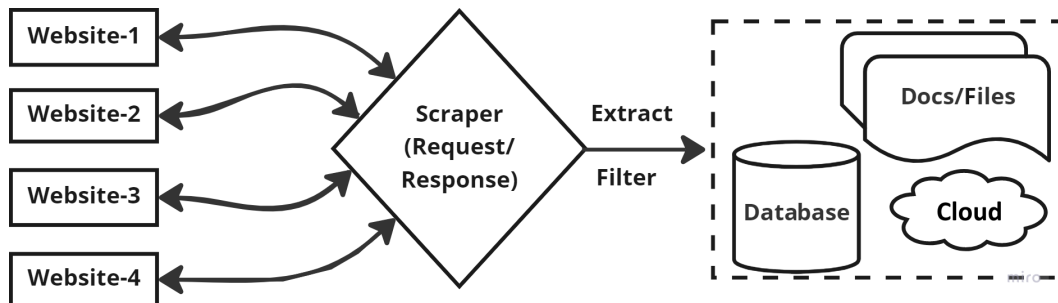
Figure 1.1: Web scraping – storing web content as data

We will explore more about scraping and the analysis of data in later chapters.

There are some legal factors that are also to be considered before performing scraping tasks. Most websites contain pages such as **Privacy Policy**, **About Us**, and **Terms and Conditions**, where information on legal action and prohibited content, as well as general information, is available. It is a developer's ethical duty to comply with these terms and conditions before planning any scraping activities on a website.

> **Important note**
>
> Scraping, web scraping, and crawling are terms that are generally used interchangeably in both the industry and this book. However, they have slightly different meanings. Crawling, also known as spidering, is a process used to browse through the links on websites and is often used by search engines for indexing purposes, whereas scraping is mostly related to content extraction from websites.

You now have a basic understanding of web scraping. We will try to explore and understand the latest web-based technologies that are extremely helpful in web scraping in the upcoming section.

Understanding the latest web technologies

A web page is not only a document or container of content. The rapid development in computing and web-related technologies today has transformed the web, with more security features being implemented and the web becoming a dynamic, real-time source of information. Many scraping communities gather historic data; some analyze hourly data or the latest obtained data.

At our end, we (*users*) use web browsers (*such as Google Chrome, Mozilla Firefox, and Safari*) as an application to access information from the web. Web browsers provide various document-based functionalities to users and contain application-level features that are often useful to web developers.

Web pages that users view or explore through their browsers are not just single documents. Various technologies exist that can be used to develop websites or web pages. A web page is a document that contains blocks of HTML tags. Most of the time, it is built with various sub-blocks linked as dependent or independent components from various interlinked technologies, including JavaScript and **Cascading Style Sheets** (**CSS**).

An understanding of the general concepts of web pages and the techniques of web development, along with the technologies found inside web pages, will provide more flexibility and control in the scraping process. A lot of the time, a developer can also employ reverse-engineering techniques.

Reverse engineering is an activity that involves breaking down and examining the concepts that were required to build certain products. For more information on reverse engineering, please refer to the **GlobalSpec** article *How Does Reverse Engineering Work?*, available at `https://insights.globalspec.com/article/7367/how-does-reverse-engineering-work`.

Here, we will introduce and explore a few of the available web technologies that can help and guide us in the process of data extraction.

HTTP

Hypertext Transfer Protocol (**HTTP**) is an application protocol that transfers resources (web-based), such as HTML documents, between a client and a web server. HTTP is a stateless protocol that follows the client-server model. Clients (web browsers) and web servers communicate or exchange information using **HTTP requests** and **HTTP responses**, as seen in *Figure 1.2*:
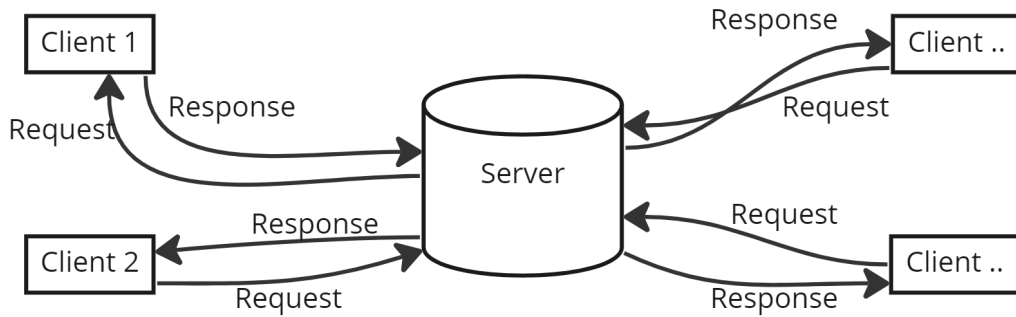
Figure 1.2: HTTP (client and server or request-response communication)

Requests and responses are cyclic in nature – they are like questions and answers from clients to the server, and vice versa.

Another encrypted and more secure version of the HTTP protocol is **Hypertext Transfer Protocol Secure** (**HTTPS**). It uses **Secure Sockets Layer** (**SSL**) (learn more about SSL at `https://developer.mozilla.org/en-US/docs/Glossary/SSL`) and **Transport Layer Security** (**TLS**) (learn more about TLS at `https://developer.mozilla.org/en-US/docs/Glossary/TLS`) to communicate encrypted content between a client and a server. This type of security allows clients to exchange sensitive data with a server in a safe manner. Activities such as banking, online shopping, and e-payment gateways use HTTPS to make sensitive data safe and prevent it from being exposed.

> **Important note**
> An HTTP request URL begins with `http://`, for example, `http://www.packtpub.com`, and an HTTPS request URL begins with `https://`, such as `https://www.packpub.com`.

You have now learned a bit about HTTP. In the next section, you will learn about HTTP requests (or HTTP request methods).

HTTP requests (or HTTP request methods)

Web browsers or clients submit their requests to the server. Requests are forwarded to the server using various methods (commonly known as HTTP request methods), such as **GET** and **POST**:

- GET: This is the most common method for requesting information. It is considered a safe method as the resource state is not altered here. Also, it is used to provide query strings, such as `https://www.google.com/search?q=world%20cup%20football&source=hp`, which is requesting information from Google based on the q (**world cup football**) and `source` (**hp**) parameters sent with the request. Information or queries (q and `source` in this example) with values are displayed in the URL.

- POST: Used to make a secure request to the server. The requested resource state can be altered. Data posted or sent to the requested URL is not visible in the URL but rather transferred with the request body. It is used to submit information to the server in a secure way, such as for logins and user registrations.

We will explore more about HTTP methods in the *Implementing HTTP methods* section of *Chapter 2*.

There are two main parts to HTTP communication, as seen in *Figure 1.2*. With a basic idea about HTTP requests, let's explore HTTP responses in the next section.

HTTP responses

The server processes the requests, and sometimes also the specified HTTP headers. When requests are received and processed, the server returns its response to the browser. Most of the time, responses are found in HTML format, or even, in JavaScript and other document types, in **JavaScript Object Notation** (**JSON**) or other formats.

A response contains status codes, the meaning of which can be revealed using **Developer Tools** (**DevTools**). The following list contains a few status codes along with some brief information about what they mean:

- **200**: OK, request succeeded
- **404**: Not found, requested resource cannot be found
- **500**: Internal server error
- **204**: No content to be sent
- **401**: Unauthorized request was made to the server

There are also some groups of responses that can be identified from a range of HTTP response statuses:

- **100–199**: Informational responses
- **200–299**: Successful responses
- **300–399**: Redirection responses
- **400–499**: Client error
- **500–599**: Server error

> **Important note**
>
> For more information on cookies, HTTP, HTTP responses, and status codes, please consult the official documentation at `https://www.w3.org/Protocols/` and `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`.

Now that we have a basic idea about HTTP responses and requests, let us explore HTTP cookies (one of the most important factors in web scraping).

HTTP cookies

HTTP cookies are data sent by the server to the browser. This data is generated and stored by websites on your system or computer. It helps to identify HTTP requests from the user to the website. Cookies contain information regarding session management, user preferences, and user behavior.

The server identifies and communicates with the browser based on the information stored in the cookies. Data stored in cookies helps a website to access and transfer certain saved values, such as the session ID and expiration date and time, providing a quick interaction between the web request and response.

Figure 1.3 displays the list of request cookies from `https://www.fifa.com/fifaplus/en`, collected using Chrome DevTools:

Name	Value	Domain	Path	Expires / Max-Age
AMCVS_2F2827E253DAF0E10A...	1	.fifa.com	/	Session
AMCV_2F2827E253DAF0E10A4...	1176715910%7CMCMID%7C1777060695...	.fifa.com	/	2024-01-18T09:29:10.8...
OptanonAlertBoxClosed	2022-12-14T09:28:45.371Z	.fifa.com	/	2023-12-14T09:28:45.0...
OptanonConsent	isGpcEnabled=0&datestamp=Wed+Dec...	.fifa.com	/	2023-12-14T09:30:25.0...
TEAL	v:41850ff76b28195076688951430195 46f...	.fifa.com	/	2023-12-14T09:30:24.0...
__gads	ID=6b142883b26258f0:T=1671010175:S...	.fifa.com	/	2024-01-08T09:29:35.0...
__gpi	UID=00000b9024657cd0:T=1671010175:...	.fifa.com	/	2024-01-08T09:29:35.0...
_gcl_au	1.1.1078381313.1671010127	.fifa.com	/	2023-03-14T09:28:46.0...
_schn	_547xoq	.fifa.com	🛈 /fifaplus/en/t...	Session
_scid	981580a8-c0b9-4748-9e66-4ce28a9495c0	.fifa.com	/	2024-01-14T01:47:04.0...
ai_user	EvcaIJCyOcJSJZKnSe+ici|2022-12-14T09:...	www.fifa.com	/	2023-12-14T09:28:35.6...
s_cc	true	.fifa.com	/	Session
s_sq	%5B%5BB%5D%5D	.fifa.com	/	Session

Figure 1.3: Request cookies

We will explore and collect more information about and from browser-based DevTools in the upcoming sections and *Chapter 3*.

> **Important note**
>
> For more information about cookies, please visit *About Cookies* at `http://www.aboutcookies.org/` and *All About Cookies* at `http://www.allaboutcookies.org/`.

Similar to the role of cookies, HTTP proxies are also quite important in scraping. We will explore more about proxies in the next section, and also in some later chapters.

HTTP proxies

A proxy server acts as an intermediate server between a client and the main web server. The web browser sends requests to the server that are actually passed through the proxy, and the proxy returns the response from the server to the client.

Proxies are often used for monitoring/filtering, performance improvement, translation, and security for internet-related resources. Proxies can also be bought as a service, which may also be used to deal with cross-domain resources. There are also various forms of proxy implementation, such as web proxies (which can be used to bypass IP blocking), CGI proxies, and DNS proxies.

You can buy or have a contract with a proxy seller or a similar organization. They will provide you with various types of proxies according to the country in which you are operating. Proxy switching during crawling is done frequently – a proxy allows us to bypass restricted content too. Normally, if a request is routed through a proxy, our IP is somewhat safe and not revealed as the receiver will just see the third-party proxy in their detail or server logs. You can even access sites that aren't available in your location (that is, you see an *access denied in your country* message) by switching to a different proxy.

Cookie-based parameters that are passed in using HTTP GET requests, HTML form-related HTTP POST requests, and modifying or adapting headers will be crucial in managing code (that is, scripts) and accessing content during the web scraping process.

> **Important note**
>
> Details on HTTP, headers, cookies, and so on will be explored more in an upcoming section, *Data-finding techniques used in web pages*. Please visit the HTTP page in the MDN web docs (`https://developer.mozilla.org/en-US/docs/Web/HTTP`) for more detailed information on HTTP and related concepts. Please visit `https://www.softwaretestinghelp.com/best-proxy-server/` for information on the best proxy server.

You now understand general concepts regarding HTTP (including requests, responses, cookies, and proxies). Next, we will understand the technology that is used to create web content or make content available in some predefined formats.

HTML

Websites are made up of pages or documents containing text, images, style sheets, and scripts, among other things. They are often built with markup languages such as **Hypertext Markup Language** (**HTML**) and **Extensible Hypertext Markup Language** (**XHTML**).

HTML is often referred to as the standard markup language used for building a web page. Since the early 1990s, HTML has been used independently as well as in conjunction with server-based scripting languages, such as PHP, ASP, and JSP. XHTML is an advanced and extended version of HTML, which is the primary markup language for web documents. XHTML is also stricter than HTML, and from a coding perspective, is also known as an application built with **Extensible Markup Language** (**XML**).

HTML defines and contains the content of a web page. Data that can be extracted, and any information-revealing data sources, can be found inside HTML pages within a predefined instruction set or markup elements called tags. HTML tags are normally a named placeholder carrying certain predefined attributes, for example, `<a>`, `<b>`, `<table>`, `<img>`, and `<script>`.

HTML is a container or type of markup language. Various factors are involved in building HTML; the next section defines these factors with some examples.

HTML elements and attributes

HTML elements (also referred to as document nodes) are the building blocks of web documents. HTML elements are built with a start tag, `<..>`, and an end tag, `</..>`, with certain content inside them. An HTML element can also contain attributes, usually defined as `attribute-name = attribute-value`, which provide additional information to the element:

```
<p>normal paragraph tags</p>
<h1>heading tags there are also h2, h3, h4, h5, h6</h1>
<a href="https://www.google.com">Click here for Google.com</a>
<img src="myphoto1.jpg" width="300" height="300" alt="Picture" />
<br />
```

The preceding code can be broken down as follows:

- `<p>` and `<h1>` are HTML elements containing general text information (element content).

- `<a>` is defined with an `href` attribute that contains the actual link that will be processed when the text **Click here for Google.com** is clicked. The link refers to `https://www.google.com/`.

- The `<img>` image tag also contains a few attributes, such as `src` and `alt`, along with their respective values. `src` holds the resource, which means the image address or image URL, as a value, whereas `alt` holds the value for alternative text (mostly displayed when there is a slow connection or the image is not able to load) for `<img>`.

- `<br/>` represents a line break in HTML and has no attributes or text content. It is used to insert a new line in the layout of the document.

HTML elements can also be nested in a tree-like structure with a parent-child hierarchy, as follows:

```
<div class="article">
  <p id="mainContent" class="content">
    <b>Paragraph Content</b>
      <img src="mylogo.png" id="pageLogo" alt="Logo"
        class="logo"/>
  </p>
  <p>
```

```
    <h3> Paragraph Title: Web Scraping</h3>
  </p>
</div>
```

As seen in the preceding code, two `<p>` child elements are found inside an HTML `<div>` block. Both child elements carry certain attributes and various child elements as their content. Normally, HTML documents are built with the aforementioned structure.

As seen in the preceding code block in the last example, there are a few extra key-value pairs. The next section explores this.

Global attributes

HTML elements can contain some additional information, such as key-value pairs. These are also known as HTML element attributes. Attributes hold values and provide identification, or contain additional information that can be helpful in many aspects during scraping activities, such as identifying exact web elements and extracting values or text from them and traversing (*moving along*) elements.

There are certain attributes that are common to HTML elements or can be applied to all HTML elements. The following list mentions some of the attributes that are identified as global attributes (`https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes`):

- `id`: This attribute's values should be unique to the element they are applied to

- `class`: This attribute's values are mostly used with CSS, providing equal state formatting options, and can be used with multiple elements

- `style`: This specifies inline CSS styles for an element

- `lang`: This helps to identify the language of the text

> **Important note**
>
> The `id` and `class` attributes are mostly used to identify or format individual elements or groups of them. These attributes can also be managed by CSS and other scripting languages. These attributes can be identified by placing # and ., respectively, in front of the attribute name when used with CSS, or while traversing and applying parsing techniques.

HTML element attributes can also be overwritten or implemented dynamically using scripting languages. As displayed in the following example, `itemprop` attributes are used to add properties to an element, whereas `data-*` is used to store data that is native to the element itself:

```
<div itemscope itemtype="http://schema.org/Place">
    <h1 itemprop="univeristy">University of Helsinki</h1>
    <span>Subject: <span itemprop="subject1">Artificial
        Intelligence</span>
```

```
    </span><span itemprop="subject2">Data Science</span>
</div>
<img class="dept" src="logo.png" data-course-id="324"
datatitle="Predictive  Analysis" data-x="12345" data-y="54321"
data-z="56743" onclick="schedule.load()"/>
```

HTML tags and attributes are very helpful when extracting data.

> **Important note**
> Please visit `https://www.w3.org` or `https://www.w3schools.com/html` for more detailed information on HTML.

In *Chapter 3*, we will explore these attributes using different tools. We will also perform various logical operations and use them for extracting or scraping purposes.

We now have some idea about HTML and a few important attributes related to HTML. In the next section, we will learn the basics of XML, also known as the parent of markup languages.

XML

XML is a markup language used for distributing data over the internet, with a set of rules for encoding documents that are readable and easily exchangeable between machines and documents. XML files are recognized by the `.xml` extension.

XML emphasizes the usability of textual data across various formats and systems. XML is designed to carry portable data or data stored in tags that is not predefined with HTML tags. In XML documents, tags are created by the document developer or an automated program to describe the content.

The following code displays some example XML content:

```
<employees>
  <employee>
    <fullName>Shiba Chapagain</fullName>
    <gender>Female</gender>
  </employee>
  <employee>
    <fullName>Aasira Chapagain</fullName>
    <gender>Female</gender>
  </employee>
</employees>
```

In the preceding code, the `<employees>` parent node has two `<employee>` child nodes, which in turn contain the other child nodes of `<fullName>` and `<gender>`.

XML is an open standard, using the Unicode character set. XML is used to share data across various platforms and has been adopted by various web applications. Many websites use XML data, implementing its contents with the use of scripting languages and presenting it in HTML or other document formats for the end user to view.

Extraction tasks from XML documents can also be performed to obtain the contents in the desired format, or by filtering the requirement with respect to a specific need for data. Plus, behind-the-scenes data may also be obtained from certain websites only.

> **Important note**
> Please visit `https://www.w3.org/XML/` and `https://www.w3schools.com/xml/` for more information on XML.

So far, we have explored content placing and content holding related technologies based on markup languages such as HTML and XML. These technologies are somewhat static in nature. The next section is about JavaScript, which provides dynamism to the web with the help of scripts.

JavaScript

JavaScript (also known as **JS** or **JScript**) is a programming language used to program HTML and web applications that run in the browser. JavaScript is mostly preferred for adding dynamic features and providing user-based interaction inside web pages. JavaScript, HTML, and CSS are among the most-used web technologies, and now they are also used with headless browsers (you can read more about headless browsers at `https://oxylabs.io/blog/what-is-headless-browser`). The client-side availability of the JavaScript engine has also strengthened its usage in application testing and debugging.

`<script>` contains programming logic with JavaScript variables, operators, functions, arrays, loops, conditions, and events, targeting the HTML **Document Object Model** (**DOM**). JavaScript code can be added to HTML using `<script>`, as seen in the following code, or can also be embedded as a file:

```
<!DOCTYPE html>
<html>
<head>
    <script>
        function placeTitle() {
            document.getElementById("innerDiv").innerHTML =
                "Welcome to WebScraping";
        }
    </script>
</head>
<body>
    <div>Press the button: <p id="innerDiv"></p></div>
```

```
    <button id="btnTitle" name="btnTitle" type="submit"
        onclick="placeTitle()">
        Load Page Title!
    </button>
</body>
</html>
```

As seen in the preceding code, the HTML `<head>` tag contains `<script>` with the `placeTitle()` JavaScript function. The function defined fires up the event as soon as `<button>` is clicked and changes the content for `<p>` with `id=innerDIV` (this particular element is defined as empty) to display the text `Welcome to WebScraping`.

> **Important note**
>
> The HTML DOM is a standard for how to get, change, add, or delete HTML elements. Please visit the page on JavaScript HTML DOM on W3Schools (`https//www.w3schools.com/js/js_htmldom.asp`) for more detailed information.

The dynamic manipulation of HTML content, elements, attribute values, CSS, and HTML events with accessible internal functions and programming features makes JavaScript very popular in web development. There are many web-based technologies related to JavaScript, including JSON, **JavaScript Query** (**jQuery**), AngularJS, and **Asynchronous JavaScript and XML** (**AJAX**), among many more. Some of these will be discussed in the following subsections.

jQuery

jQuery, or more specifically **JavaScript-based DOM-related query**, is a JavaScript library that addresses incompatibilities across browsers, providing API features to handle the HTML DOM, events, and animations. jQuery has been acclaimed globally for providing interactivity to the web and the way JavaScript is used to code. jQuery is lightweight in comparison to the JavaScript framework. It is also easy to implement and takes a short and readable coding approach.

jQuery is a huge topic and will require adequate knowledge of JavaScript before embarking on it. A jQuery-like Python-based library will be used by us in *Chapter 4*.

> **Important note**
>
> For more information on jQuery, please visit `https://www.w3schools.com/jquery/` and `http://jquery.com/`.

jQuery is mostly used for DOM-based activities, as discussed in this section, whereas AJAX is a collection of technologies, which we are going to learn about in the next section.

AJAX

AJAX is a web development technique that uses a group of web technologies on the client side to create asynchronous web applications.

JavaScript **XMLHttpRequest** (**XHR**) objects are used to execute AJAX on web pages and load page content without refreshing or reloading the page. Please visit the AJAX page on W3Schools (`https://www.w3schools.com/js/js_ajax_intro.asp`) for more information on AJAX. From a scraping point of view, a basic overview of JavaScript functionality will be valuable to understand how a page is built or manipulated, as well as to identify the dynamic components used.

> **Important note**
>
> Please visit `https://developer.mozilla.org/en-US/docs/Web/JavaScript`, `https://www.javascript.com/`, `https://www.w3schools.com/js/js_intro.asp`, and `https://www.w3schools.com/js/js_ajax_intro.asp` for more information on JavaScript and AJAX.

We have learned about a few JavaScript-based techniques and technologies that are commonly deployed in web development today. In the next section, we will learn about data-storing objects.

JSON

JSON is a format used for storing and transporting data from a server to a web page. It is language-independent and preferred in web-based data interchange actions due to its size and readability. JSON files are files that have the `.json` extension.

JSON data is normally formatted as a name:value pair, which is evaluated as a JavaScript object and follows JavaScript operations. JSON and XML are often compared, as they both carry and exchange data between various web resources. JSON is usually ranked higher than XML for its structure, which is simple, readable, self-descriptive, understandable, and easy to process.

For web applications using JavaScript, AJAX, or RESTful services, JSON is preferred over XML due to its fast and easy operation. JSON and JavaScript objects are interchangeable. JSON is not a markup language, and it doesn't contain any tags or attributes. Instead, it is a text-only format that can be accessed through a server, as well as being able to be managed by any programming language.

JSON objects can also be expressed as arrays, dictionaries, and lists:

```
{"mymembers":[
{ "firstName":"Aasira",
"lastName":"Chapagain","cityName":"Kathmandu"},
{ "firstName":"Rakshya", "lastName":"Dhungel","cityName":"New Delhi"},
{ "firstName":"Shiba", "lastName":"Paudel","cityName":"Biratnagar"},
]}
```

You have learned about JSON, which is a content holder. In the following section, we will discuss HTML styling using CSS and providing HTML tags with extra identification.

> **Important note**
>
> JSON is also known for the mixture of dictionary and list objects it provides in Python. JSON is written as a string, and we can find plenty of websites that convert JSON strings into JSON objects, for example, `https://jsonformatter.org/`, `https://jsonlint.com/`, and `https://www.freeformatter.com/json-formatter.html`.
>
> Please visit `http://www.json.org/`, `https://jsonlines.org/`, and `https://www.w3schools.com/js/js_json_intro.asp` for more information regarding JSON and JSON Lines.

CSS

The web-based technologies we have introduced so far deal with content, including binding, development, and processing. CSS describes the display properties of HTML elements and the appearance of web pages. CSS is used for styling and providing the desired appearance and presentation of HTML elements.

By using CSS, developers/designers can control the layout and presentation of a web document. CSS can be applied to a distinct element in a page, or it can be embedded through a separate document. Styling details can be described using the `<style>` tag.

The `<style>` tag can contain details targeting repeated and various elements in a block. As seen in the following code, multiple `<a>` elements exist, and it also possesses the `class` and `id` global attributes:

```html
<html>
<head>
<style>
a{color:blue;}
h1{color:black; text-decoration:underline;}
#idOne{color:red;}
.classOne{color:orange;}
</style>
</head>
<body>
<h1> Welcome to Web Scraping </h1>Links:<a href="https://www.google.
com"> Google </a>  
<a class='classOne' href="https://www.yahoo.com"> Yahoo </a>
<a id='idOne' href="https://www.wikipedia.org"> Wikipedia </a>
</body>
</html>
```

Attributes that are provided with CSS properties or have been styled inside `<style>` tags in the preceding code block will result in the output shown in *Figure 1.4*:

Welcome to Web Scraping

Links: Google Yahoo Wikipedia

Figure 1.4: Output of the HTML code using CSS

Although CSS is used to manage the appearance of HTML elements, CSS selectors (*patterns used to select elements or the position of elements*) often play a major role in the scraping process. We will be exploring CSS selectors in detail in *Chapter 3*.

> **Important note**
> Please visit `https://www.w3.org/Style/CSS/` and `https://www.w3schools.com/css/` for more detailed information on CSS.

In this section, you were introduced to some of the technologies that can be used for web scraping. In the upcoming section, you will learn about data-finding techniques. Most of them are built with web technologies you have already been introduced to.

Data-finding techniques used in web pages

To extract data from websites or web pages, we must identify where exactly the data is located. This is the most important step in the case of automating data collection from the web.

When we browse or request any URL in a web browser, we can see the contents as responses to us. These contents can be some dynamically added values or dynamically generated or rendered to the HTML templates by processing some API or JavaScript code. Knowing the URL of response content or finding the availability of content in some files is the first action toward scraping. Content can also be retrieved using third-party sources or sometimes even embedded in a view to end users.

In this section, we will explore a few key techniques that will help us identify, search for, and locate contents we have received via a web browser.

HTML source page

Web browsers are used for client-server-based GUI interaction to explore web content. The browser address bar is supplied with the web address or URL, the requested URL is communicated to the server (host), and a response is received, which means it is loaded by the browser. This obtained response or page source can be further explored and searched for the desired content in raw format.

> **Important note**
>
> You are free to choose which web browser you wish to use. Most web browsers will display the same or similar content. We will be using Google Chrome for most of the book's examples, installed on the Windows OS.

To access the HTML source page, follow these steps:

1. Open `https://www.google.com` in your web browser (you can try the same scenario with any other URL).

2. After the page is loaded completely, *right-click* on any section of the page. The menu shown in *Figure 1.5* should be visible, with the **View page source** option:

Figure 1.5: View page source (right-click on any page and find this option)

3. If you click the **View page source** option, it will load a new tab in the browser, as seen in *Figure 1.6*:

Figure 1.6: Page source (new tab loaded in the web browser, with raw HTML)

You can see that a new tab will be added to the browser with the text **view-source:** prepended to the original URL, `https://www.google.com`. Also, if we add the text *view-source:* to our URL, once the URL loads, it displays the page source or raw HTML.

> **Important note**
>
> You can try to find any text or DOM element in the web browser by searching inside the page source. Load the URL `https://www.google.com` and search *web scraping*. Find some of the content displayed by Google using the page source.

We now possess a basic idea of data-finding techniques. The technique we used in this section is a primary or base concept. There are a few more techniques that are more sophisticated and come with a large set of functionality and tools, which help or guide us in the data-finding context – we will cover them in the next section.

Developer tools

DevTools are found embedded within most browsers on the market today. Developers and end users alike can identify and locate resources and search for web content that is used during client-server communication, or while engaged in an HTTP request and response.

DevTools allow a user to examine, create, edit, and debug HTML, CSS, and JavaScript. They also allow us to handle and figure out performance problems. They facilitate the extraction of data that is dynamically or securely presented by the browser.

DevTools will be used for most data extraction cases. For more detailed information on DevTools, here are some links:

- **Google Chrome**: `https://developer.chrome.com/docs/devtools/`

- **Firefox**: `https://firefox-source-docs.mozilla.org/devtools-user/`

Similar to the **View page source** option, as discussed in the *HTML source page* section, we can find the **Inspect** menu option, which is another option for viewing the page source, when we right-click on a web page.

Alternatively, you can access DevTools via the main menu in the browser. Click **More tools | Developer tools**, or press *Ctrl + Shift + I*, as seen in *Figure 1.7*:

Figure 1.7: Accessing DevTools (web browser menu bar)

Let's try loading the URL `https://en.wikipedia.org/wiki/FIFA` in the web browser. After the page gets loaded, follow these steps:

1. *Right-click* the page and click the **Inspect** menu option.

   We'll notice a new menu section with tabs (**Elements**, **Console**, **Sources**, **Network**, **Memory**, and more) appearing in the browser, as seen in *Figure 1.8*:

Figure 1.8: Inspecting the DevTools panels

2.   Press *Ctrl + Shift + I* to access the DevTools or click the **Network** tab from the **Inspect** menu option, as shown in *Figure 1.9*:

Figure 1.9: DevTools Network panel

Important note

The **Search** and **Filter** fields, as seen in *Figure 1.9*, are often used to find content in the HTML page source or other available resources that are available in the **Network** panel. The **Search** box can be supplied with a regex pattern – case-sensitive information to find or locate content statically or dynamically.

All panels and tools found inside DevTools have a designated role. Let's get a basic overview of a few important ones next.

Exploring DevTools

Here is a list of all the panels and tools found in DevTools:

- **Elements**: Displays the HTML content of the page viewed. This is used for viewing and editing the DOM and CSS, and for finding CSS selectors and XPath content. *Figure 1.10* shows the icon as found in the **Inspect** menu option, which can be *clicked and moved* to the HTML content in the page or code inside the **Elements** panel, to locate HTML tags or XPath and DOM element positions:

Figure 1.10: Element inspector or selector

  This icon acts similarly to the mouse cursor moving across the screen. We will explore CSS selectors and XPath further in *Chapter 3*.

> **Important note**
> HTML elements displayed or located in the **Elements** or **Network | Doc** panel may not be available in the page source.

- **Console**: Used to run and interact with JavaScript code, and to view log messages.

- **Sources**: Used to navigate pages and view available scripts and document sources. Script-based tools are available for tasks such as script execution (that is, resuming and pausing), stepping over function calls, activating and deactivating breakpoints, and handling exceptions.

- **Network**: Provides us with HTTP request and response-related resources. Resources found here feature options such as recording data to network logs, capturing screenshots, filtering web resources (JavaScript, images, documents, and CSS), searching web resources, and grouping web resources, and can also be used for debugging tasks. *Figure 1.11* displays the HTTP request URL, request method, status code, and more, by accessing the **Headers** tab from the **Doc** option available inside the **Network** panel.

Figure 1.11: DevTools – Network | Doc | Headers option (HTTP method and status code)

Network-based requests can also be filtered by the following types:

- **All**: Lists all requests related to the network, including document requests, images, fonts, and CSS. Resources are placed in the order of them being loaded.

- **Fetch/XHR**: Lists XHR objects. This option lists dynamically loaded resources, such as API and AJAX content.

- **JS**: Lists JavaScript files involved in the request and response cycle.

- **CSS**: Lists all style files.

- **Img**: Lists image files and their details.

- **Doc**: Lists requested HTML or web-related documents.

- **WS**: Lists WebSocket-related entries and their details.

- **Other**: Lists any unfiltered type of request-related resources.

For each of the filter options just listed, there are some child tabs for selected resources in the **Name** panel, which are as follows:

- **Headers**: Loads HTTP/HTTPS header data for a particular request. A few important and automation-based types of data are also found, for example, request URL, method, status code, request/response headers, query string, payload, or POST information.

- **Preview**: Provides a preview of the response found, similar to the entities viewed in the web browser.

- **Response**: Loads the response from particular entities. This tab shows the HTML source for HTML pages, JavaScript code for JavaScript files, and JSON or CSV data for similar documents. It actually shows the raw source of the content.

- **Initiator**: Provides the initiator links or chains of initiator URLs. It is similar to the referer in the request headers.

- **Timing**: Shows a breakdown of the time between resource scheduling, when the connection starts, and the request/response.

- **Cookies**: Provides cookie-related information, its keys and values, and expiration dates.

Important note

The **Network** panel is one of the most important resource hubs. We can find/trace plenty of information and supporting details for each request/response cycle in this panel. For more detailed information on the **Network** panel, please visit `https://developer.chrome.com/docs/devtools/network/` and `https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/`.

- **Performance**: Screenshots and a memory timeline can be recorded. The visual information obtained is used to optimize the website speed, improve load times, and analyze the runtime or overall performance.

- **Memory**: Information obtained from this panel is used to fix memory issues and track down memory leaks. Overall, the details from the **Performance** and **Memory** panels allow developers to analyze website performance and embark on further planning related to optimization.

- **Application**: The end user can inspect and manage storage for all loaded resources during page loading. Information related to cookies, sessions, application cache, images, databases on the fly, and more can be viewed and even deleted to create a fresh session.

- **Security**: This panel might not be available in all web browsers. It normally shows security-related information, such as resources, certificates, and connections. We can even browse more about certificate details, from a few detail links or buttons available in this panel, as shown here in *Figure 1.12*:

Figure 1.12: Security panel (details about certificate, connection, and resources)

After exploring the HTML page source and DevTools, we now have an idea about where data and request/response-related information is stored, and how we can access it. Overall, the scraping process involves extracting data from web pages, and we need to identify or locate the resources with data or those that can carry data. Before proceeding with data exploration and content identification, it is beneficial to identify the page URL, DevTools resources, XHR, JavaScript, and a general overview of browser-based activities.

Finally, there are more topics related to links, child pages, and more. We will be using techniques such as `Sitemaps.xml` and `robots.txt` in depth in *Chapter 3*.

Important note

For basic concepts related to `sitemaps.xml` and `robots.txt`, please visit the Sitemaps site (`https://www.sitemaps.org`) and the Robots Exclusion Protocol site (`http://www.robotstxt.org`).

In this chapter, you have learned about web scraping, selected web technologies that are involved, and how data-finding techniques are used.

Summary

Websites are dynamic in nature, so the fundamental activities introduced in this chapter will be applicable in most cases. We also explained and explored some of the core technologies related to the **World Wide Web** (**WWW**) and web scraping. Identifying or finding content with the use of DevTools and page sources for targeted content was the focus of this chapter. This information will guide you through various aspects of taking primary and professional steps in web scraping.

In the next chapter, we will be using the Python programming language to interact with the web or data sources and explore a few main libraries that we have chosen for data extraction.

Further reading

- *HTML*: `https://developer.mozilla.org/en-US/docs/Web/HTML`
- *CSV*: `https://docs.python.org/3/library/csv.html`
- *JSON*:
  - `https://www.json.org/json-en.html`
  - `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON`
- *XML*:
  - `https://aws.amazon.com/what-is/xml/`
  - `https://www.w3.org/XML/`
- *HTTP*:
  - `https://developer.mozilla.org/en-US/docs/Web/HTTP`
  - `https://http.dev/methods`
  - `https://www.rfc-editor.org/rfc/rfc9110.html`
- *JavaScript*:
  - `https://www.javascript.com/`
  - `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
- *jQuery*:
  - `https://jquery.com/`
  - `https://www.w3schools.com/jquery/default.asp`
- *Browser developer tools*:
  - `https://developer.chrome.com/docs/devtools/`
  - `https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Tools_and_setup/What_are_browser_developer_tools`
- *Web technology*: `https://developer.mozilla.org/en-US/docs/Web`
- *Reverse engineering*: `https://insights.globalspec.com/article/7367/how-does-reverse-engineering-work`

2

Python Programming for Data and Web

In *Chapter 1*, you got an idea of what web scraping is, what core technologies exist, and how and where you can plan to find the resources or data you're looking for.

Web scraping requires tools and techniques to be implemented and deployed using scripts or programs. We have chosen Python (`https://www.python.org/`) for this purpose, as it is very easy to learn and has a huge set of libraries for communicating with the **World Wide Web** (**WWW**), data-related processes, and finally, web scraping. In internet search results, we often find Python mentioned in conjunction with data science and **Machine Learning** (**ML**). This is because of the wide use of Python in such projects.

In this chapter, we will explore the key benefits of Python and communicate with web resources using Python libraries. This chapter will also provide a detailed overview of installing and using Python libraries such as `requests` and `urllib`.

In particular, we will learn about the following topics:

- Why Python (for web scraping)?
- Accessing the WWW with Python
- URL handling and operations
- Implementing HTTP methods

> **Important note**
>
> We assume that you have some prior basic understanding and experience in using Python. If not, then please refer to Python tutorials or training materials from popular sites such as *Packt Publishing* (`https://www.packtpub.com`), *W3Schools* (`https://www.w3schools.com/python/default.asp`), or *Python Course* (`https://www.python-course.eu`), or search the internet for `learn python programming` to get more such material.

Technical requirements

We will be using Python 3.11.1 installed on Windows. There are plenty of choices for code editors; choose one that is convenient for you to use and can deal with the libraries used in this chapter's code examples. We will be using **Notebooks** and **JupyterLab** from **Jupyter** (`https://jupyter.org/`), **Integrated Development and Learning Environment** (**IDLE**) (the default editor from Python), and Windows' **Command Prompt** (`cmd`) side by side.

To follow along with this chapter, you will need to install the following applications:

- Python 3.11.* or the latest version appropriate for your OS from `https://www.python.org/downloads/`

- The `pip` Python package management tool (`https://pypi.org/project/pip/`)

- JupyterLab (`https://jupyter.org/install`) for accessing notebooks or files with the `.ipynb` extension

- Google Chrome, Mozilla Firefox, or any other web browser equipped with **Developer Tools** (**DevTools**)

- **JetBrains PyCharm** (`https://www.jetbrains.com/pycharm/`), **Visual Studio Code** (`https://code.visualstudio.com/`), or any editor that you are used to

The Python libraries that are required for this chapter are as follows:

- `requests`

- `urllib`

The code files for this chapter are available online on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter02`.

Why Python (for web scraping)?

Python is a popular programming language that is used to code various types of applications, from simple scripts to software development, ML/AI algorithms, and CLI/GUI apps, and also to create web applications. Python's simple **Object-Oriented Programming** (**OOP**) syntax is very readable, which allows developers to write code in not many lines and work on different platforms (which means Python is platform independent), which makes it the number-one choice among programming languages.

Python is a buzzword in today's programming domain. Quite a lot of tools and web applications are produced and managed using Python, which is compatible with small and large apps and is supplied with adequate and up-to-date libraries by its global developer audience.

Figure 2.1 (taken from `https://www.python.org`) shows **Success Stories** and **Use Python for…** information for various types of applications:

Figure 2.1: Python success stories and usage

> **Important note**
>
> Please visit the official Python website at `https://www.python.org` for updates, downloads, and more information. Regarding **applications for Python**, visit `https://www.python.org/about/apps/`, and for **success stories**, visit `https://www.python.org/success-stories/`.

In terms of Python usage in the scientific, mathematical, and data domains (data analysis, data science, data wrangling, big data, and many more), Python is the number-one choice and, as shown in *Figure 2.2* (collected from `https://www.ideamotive.co/python/guide#what-is-python`), it is also being used by some top global organizations to build products:

Figure 2.2: Python is used for product development on popular websites

Python's flexibility, from file handling and web applications to database management and dealing with numerous ML and **Natural Language Processing** (**NLP**) concepts in easy steps, has empowered Python's development. This growth has also been seen in Python learners, developer groups, and even global companies using and asking for Python experience and knowledge.

Web scraping requires various tools and techniques to collect data from websites that are fit for the task for either personal or business needs, but keeping in mind any legal restrictions. With Python, we can access and manage our projects using a number of effective libraries, such as `requests`, `http`, `urllib`, `json`, `csv`, and `cookielib`.

Python development, and the Python language itself, is growing and is updated from time to time. It addresses most market-based demands and helps users to be productive, along with providing developers with effective and efficient libraries. At the time of writing this book, Python's version 3.11.1 is available at `https://www.python.org`, as shown in *Figure 2.3*:

Figure 2.3: Python website (latest versions, jobs, and documentation)

Python also possesses globally admired libraries for various tasks and domains. These libraries are famous in the Python community:

- `requests`, `http`, `urllib`, `cookielib`, and `Starlette`: For **HTTP** communication
- `lxml`, `PyQuery`, and `BeautifulSoup`: For **HTML**, **XML**, and **DOM** parsing
- `json` and `csv`: For file management and content management
- `numpy` and `pandas`: For processing and analyzing numerical and tabular data
- `scipy`: For scientific computation
- `nltk`, `textblob`, and `vader`: For NLP
- `PyTorch`: An open source ML framework
- `scikit-learn`: Efficient tools for predictive data analysis
- `matplotlib`, `seaborn`, and `plotly`: Visualization libraries
- `PySpark`: For handling big data, streaming, and chunk management
- `FastApi`, Flask, and Django: Web-based frameworks
- `Pydantic` and `Pandera`: Data validation and data testing toolkits
- `PyArrow`: For interoperability and integrating Python objects

For more detailed information on these libraries, please visit `https://www.python.org`, `https://pypi.org`, and `https://pymotw.com/3/`.

In this section, you have learned about Python, its popularity, some of its libraries, and the usage of Python in top companies. In the next section, we will set up Python, install the required libraries, and use Python for communicating with the web.

Accessing the WWW with Python

As we saw in the *Why Python (for web scraping)?* section, there are plenty of Python libraries for interacting with HTTP. `requests` and `urllib` are the two libraries that we are interested in using because of their in-depth features, various functions for dealing with HTTP communication, easy-to-read documentation, and popularity.

In order to start accessing the WWW with Python using these libraries, let's verify that we have installed all of the required resources. In the following subsections, we will start setting things up, such as installing Python, creating a **virtual environment**, installing libraries in the created environment, and accessing the web using Python libraries.

Setting things up

It is assumed that the latest version of Python has been installed on your system. If not, please visit `https://www.python.org/downloads/` for the latest version of Python for your OS. Regarding the general setup and installation procedure, please visit `https://docs.python.org/3/using/windows.html` and follow the detailed steps.

> **Important note**
>
> During installation, Python users can choose **Customize Installation** and select features such as **Add Python… to PATH**. Also, for beginners and intermediate users, it is recommended to use an easy location or the root folder to set up Python, for example, `C:\Python311` or `D:\Python\Py311`.

We will be using Python 3.11.1 on Windows. After the installation is complete, we can confirm Python's availability in the OS using these steps:

1. Press *Windows* + *R* to open the **Run** window and type `cmd` to load the **command-line interface** (**CLI**).

2. Move to your root directory or desired path.

3. Type `python -h`. This will load the Python options, as shown in *Figure 2.4*:

```
C:\Python311>python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : turn on parser debugging output (for experts only, only works on
         debug builds); also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also -? or --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : remove assert and __debug__-dependent statements; add .opt-1 before
         .pyc extension; also PYTHONOPTIMIZE=x
-OO    : do -O changes and also discard docstrings; add .opt-2 before
         .pyc extension
-P     : don't prepend a potentially unsafe path to sys.path
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : force the stdout and stderr streams to be unbuffered;
         this option has no effect on stdin; also PYTHONUNBUFFERED=x
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
         when given twice, print more information about the build
```

Figure 2.4: CMD window (listing Python command-line options with python -h)

4.   As shown in *Figure 2.5*, the -V or —version option provides us with the available Python version:

```
C:\Python311>python --version
Python 3.11.1

C:\Python311>python -V
Python 3.11.1
```

Figure 2.5: Terminal window (showing Python version)

In addition to these steps, we can access the Python version using the Start menu or by pressing the *Windows* button, navigating to the menu option, and opening or loading Python's IDLE. Upon loading IDLE, it will show us the Python version in its title bar and also in the editor window (in the first line), or we can access version information using code, as shown in *Figure 2.6*:

```
IDLE Shell 3.11.1                                                    —    □    ×
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>> import sys
>>> sys.version
'3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)]'
>>>
```

Figure 2.6: Python IDLE showing the Python version and some extra information

In *Figure 2.6*, we have imported the sys library (also known as system), and sys has an attribute named version that displays the current Python version along with some system information. For more information on sys and IDLE, please visit https://docs.python.org/3/library/sys.html and https://docs.python.org/3/library/idle.html, respectively.

The default Python setup installs most of the internal or default libraries (sys, urllib, os, math, and more), but if we want libraries that are not available, the system throws a ModuleNotFoundError error, as shown in *Figure 2.7*:

```
>>> import urllib
>>> import requests
    Traceback (most recent call last):
      File "<pyshell#3>", line 1, in <module>
        import requests
    ModuleNotFoundError: No module named 'requests'
>>> import pycurl
    Traceback (most recent call last):
      File "<pyshell#4>", line 1, in <module>
        import pycurl
    ModuleNotFoundError: No module named 'pycurl'
```

Figure 2.7: ModuleNotFoundError encountered when importing the requests library

Cases similar to *Figure 2.7* will require additional installation of such libraries, which can be handled using pip.

So far, Python has been installed, and we have verified the installation using the CLI and Python's **Graphical User Interface** (**GUI**), IDLE. To set up additional libraries, such as requests, we first have to take some additional steps, which we are going to cover in the next section.

Creating a virtual environment

Before proceeding with the installation of the required libraries or additional libraries that are not available by default with Python, it is recommended to create a virtual environment for your project.

A virtual environment maintains a directory for a project and its required libraries so that no conflict between libraries and the system can happen. It normally provides a complete environment for targeted projects with their particular requirements and specifications. For more information on virtual environments, please visit https://docs.python.org/3/tutorial/venv.html or https://docs.python-guide.org/dev/virtualenvs/.

We will install all the required libraries and tools for our book in a virtual environment. So, let's begin by creating the environment, and we will load it with the required libraries using `pip` by following these steps:

1. Create a directory first, for example, `HOWScraping2E` (this is where we want all of our code in the book to be placed) in the root drive (`C:\HOWScraping2E>`).

2. In `HOWScraping2E`, run the following command:

   ```
   C:\HOWScraping2E>python -m venv secondEd
   ```

   Here we have created a virtual environment named `secondEd` using `venv`.

   This step also creates a directory named `secondEd` inside the `HOWScraping2E` directory, with a few more folders.

3. To activate the environment (`secondEd`) we have created, run the `activate` command using `secondEd\Scripts`, as shown in *Figure 2.8*:

   ```
   C:\HOWScraping2E>secondEd\Scripts\activate

   (secondEd) C:\HOWScraping2E>_
   ```

   Figure 2.8: Activating the virtual environment

   The virtual environment has been successfully activated, as the `secondEd` environment is visible at the start of the command prompt, as shown in *Figure 2.8*.

4. To deactivate or close the environment, we can just issue the `deactivate` command, for example, `(secondEd) C:\HOWScraping2E> deactivate`. It is advisable to deactivate the environment at the end or before closing the system.

Now that the location and environment are ready, we need to install the required libraries and tools using `pip`. The `pip` package management system is used to install and manage software packages written in Python. For more information, please visit `https://pypi.org/project/pip/` and `https://packaging.python.org/en/latest/tutorials/installing-packages/`.

Let's get some information on the version of `pip` that we are going to use, along with the path of `pip` in our virtual environment, as shown in *Figure 2.9*:

```
(secondEd) C:\HOWScraping2E>pip --version
pip 22.3.1 from C:\HOWScraping2E\secondEd\Lib\site-packages\pip (python 3.11)

(secondEd) C:\HOWScraping2E>_
```

Figure 2.9: pip location and version, and Python version

> **Important note**
>
> Normally, the `lib` folder contains libraries from the default Python installation in the system, and external libraries are found inside `site-packages`. As shown in *Figure 2.9*, we now have `lib` and `site-packages` inside our virtual environment, and in a project-specific or dedicated folder. Therefore, any library installations or updates will occur inside our targeted environment.
>
> If you prefer using *Anaconda*, refer to `https://docs.anaconda.com/free/navigator/tutorials/manage-environments/` and `https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html`.

So far, our environment is ready and working. We have verified that `pip` exists in our system. In the next section, we will install the libraries we need.

Installing libraries

In this section, we will install some libraries in our environment, such as `requests`, `jupyterlab`, and a few others as required.

Python notebooks are now in high demand – they provide easy visibility of code and a lot of control over input and output. In addition, the export feature is very useful to export code for presentations or explanations. Let's install JupyterLab (`https://jupyter.org`) so we can use Python notebooks (`.ipynb` files). We can use `pip install` as shown in the following command to install JupyterLab:

```
(secondEd) C:\HOWScraping2E> pip install jupyterlab
```

After successfully installing JupyterLab, let's run it and check the availability of other libraries (many dependent libraries get installed during the setup of JupyterLab). To load `jupyterlab`, use the following command:

```
(secondEd) C:\HOWScraping2E>jupyter-lab
```

This will open a new window pointing to `http://localhost:8888/lab` in your default browser, as shown in *Figure 2.10*:

Figure 2.10: JupyterLab (the default window in Google Chrome)

> **Important note**
>
> For more information on using Python notebooks and JupyterLab, please visit `https://docs.jupyter.org/en/latest/`. In addition, you can create a notebook and check whether you need to install other libraries too – if they are missing, we can install them from the notebook itself (using `!pip install requests`).

Our decision to install and use JupyterLab is quite convenient because major libraries such as `requests` are automatically installed and available to us. As shown in *Figure 2.7*, the `requests` library is not available with the default Python installation. Let's verify that our other required libraries are working fine; use them or import them in a notebook, as shown in *Figure 2.11* (the code is available in the `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter02/test_install_libraries.ipynb` file):

```
[2]: import requests

[3]: requests.__version__

[3]: '2.28.1'

[4]: import urllib

[5]: list(dir(urllib))

[5]: ['__builtins__',
      '__cached__',
      '__doc__',
      '__file__',
      '__loader__',
      '__name__',
      '__package__',
      '__path__',
      '__spec__',
      'error',
      'parse',
      'request',
      'response']
```

Figure 2.11: Verifying the requests and urllib libraries

If you do not wish to use JupyterLab, you can set up the libraries from the terminal using the following code:

```
(secondEd) C:\HOWScraping2E> pip install requests
```

Now, our environment is set up and the required libraries are in our system. We will use these libraries to communicate with URLs (send requests to the web and collect responses) in the next section.

Loading URLs

Loading URLs generally involves **HTTP** communication, such as sending requests to the server and receiving responses from the server. The upcoming *URL handling and operations* and *Implementing HTTP methods* sections in this chapter will explore these topics in more detail. Data extraction activities will be covered in *Chapter 3*.

> **Important note**
>
> Before loading URLs using Python, it's also advisable to verify that the URLs are working properly and contain the data that we are looking for. This basic feasibility lookup will save many resources while running and managing our projects.

Let's load two different URLs, one using `urllib` and another using `requests`:

- *Task 1*: Load `https://www.python.org` using `urllib`. We are using code in a notebook, as shown in *Figure 2.12*:

```
[1]:  from urllib import request

[2]:  url_a='https://www.python.org/'
```

Figure 2.12: Import request from urllib

`urllib` has many submodules, such as `request`, `response`, `parse`, and `error`. To send requests to `https://www.python.org` or load a URL, we need to import `requests` and execute the `urlopen()` method with the URL as a parameter, which returns the `HTTPResponse` object, as shown in *Figure 2.13*:

```
[3]:  response_a = request.urlopen(url_a)

[4]:  response_a

[4]:  <http.client.HTTPResponse at 0x191b0387880>
```

Figure 2.13: Reading the URL and the HTTPResponse object

We need to read the object's content using the `read()` method and display the page source or HTML content received from the website, as shown in *Figure 2.14*:

```
[5]: response_a.read()
recomposed.png">\n     <link rel="apple-touch-icon" href="/static/apple-touch-icon-precomposed.png">\n\n     \n     <meta name="msap
plication-TileImage" content="/static/metro-icon-144x144-precomposed.png"><!-- white shape -->\n     <meta name="msapplication-Til
eColor" content="#3673a5"><!-- python blue -->\n     <meta name="msapplication-navbutton-color" content="#3673a5">\n\n    <title>W
elcome to Python.org</title>\n\n     <meta name="description" content="The official home of the Python Programming Language">\n
<meta name="keywords" content="Python programming language object oriented web free open source software license documentation do
wnload community">\n\n     \n     <meta property="og:type" content="website">\n     <meta property="og:site_name" content="Python.or
g">\n     <meta property="og:title" content="Welcome to Python.org">\n     <meta property="og:description" content="The official ho
```

Figure 2.14: Displaying content by executing read() on the HTTPResponse object

- *Task 2*: Load `https://pypi.org/project/pycurl/` using `requests`. As shown in *Figure 2.15*, we have defined the URL to be processed and imported the `requests` library. The `get()` method from `requests` loads the URL as a parameter, along with the `content` attribute, and returns the response:

```
[6]: import requests

[7]: url_b="https://pypi.org/project/pycurl/"

[8]: response_b=requests.get(url_b).content

[9]: response_b
```

```
[9]: b'\n\n\n\n\n\n\n<!DOCTYPE html>\n<html lang="en" dir="ltr">\n  <head>\n    <meta charset="utf-8">\n    <meta http-equiv="X-UA-Comp
     atible" content="IE=edge">\n    <meta name="viewport" content="width=device-width, initial-scale=1">\n\n    <meta name="defaultLangu
     age" content="en">\n    <meta name="availableLanguages" content="en, es, fr, ja, pt_BR, uk, el, de, zh_Hans, zh_Hant, ru, he, eo">\n
     \n    \n\n    <title>pycurl \xc2\xb7 PyPI</title>\n    <meta name="description" content="PyCURL -- A Python Interface To The cURL li
     brary">\n\n    <link rel="stylesheet" href="/static/css/warehouse-ltr.75f501f1.css">\n    <link rel="stylesheet" href="/static/css/f
```

Figure 2.15: Displaying content using requests

After both of these tasks, it looks like `requests` has some simple and short methods for dealing with URLs. The code for these tasks is available at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter02/loading_url.ipynb`.

> **Important note**
>
> There are plenty of WWW or HTTP libraries in Python. Among them, `requests` (`https://requests.readthedocs.io/en/latest/`) (with the tagline HTTP for Humans) has attained a certain prestige. Its performance has been recognized by developers and organizations globally. It has simple, elegant, and precise methods, thanks to `urllib3` (`https://github.com/urllib3/urllib3`).

We will discuss and use both libraries in the next section and the upcoming chapters. We have now installed and verified the required libraries and a few system-related requirements, such as appropriate versions of libraries. In the next section, we will explore various URL- and HTTP-based activities and operations that are essential before we dig deeper into scraping using the `requests` library.

URL handling and operations

In this section, we will explore the operations that are required when handling URLs. In the browser, we input a URL as a request and receive an output or a response, but plenty of operations take place behind the scenes, and we can view these operations using browser-based DevTools.

We mentioned DevTools in the *Developer tools* section in *Chapter 1*, when we discussed the role of certain panels, such as the **Network** panel, found in DevTools. As we are diving deep into using libraries for HTTP-based communication and creating or dealing with code, it is quite important to deal with or monitor the HTTP information found in the **Network** panel while accessing the URL in the browser.

The information found in the different sections of the **Network** panel, such as **Request URL, Request Headers, Request Method**, **Response Headers**, **Status Code**, and **Cookies**, are important in the sense that we as developers are trying to automate, verify, inject, or use that information in our code.

Here is the information available in DevTools that can be used in our code:

- **Request URL**: We might find hidden or redirected URLs, APIs with the desired contents, or other URLs that are easier to process.

- **Request Headers**: Information sent by the browser to the server during request processing, for example, cookies, user-agent, referer, and pragma.

- **Response Headers**: Information that the server exchanges with the browser during the response, for example, server information, content/types, and etags.

- **Cookies**: Security, permissions, or session content in key-value pairs with expiration dates and times. The priority of cookies determines the complexity of the HTTP-based communication cycles.

- **Status Code**: Information about HTTP responses, whether they are client, server, or resource issues.

- **Request Method**: HTTP methods (`GET`/`POST`) being used when sending requests or receiving responses.

Developers might also encounter situations where URL manipulation is required (altered or cleaning, for example). There might be some hidden URLs, APIs, or even static content with data that we require that is not visible directly through web browsers.

After all, we are developing crawlers or Python scripts to automate our browser-based activities. Hence, we need to fully understand the DevTools and develop crawlers that can perform similarly to the browser with some additional tasks, such as extracting data. For this purpose, we need to manage and handle the URLs and HTTP communication, which we will cover in the upcoming sections.

> **Important note**
>
> DevTools shows us the complete communication logs during the requests and responses from the server or other third parties. Please check out the *Developer tools* section of *Chapter 1* for more information, or check out these URLs: `https://developer.chrome.com/docs/devtools/` and `https://firefox-source-docs.mozilla.org/devtools-user/`.

requests – Python library

`requests`, an HTTP-based Python library released in 2011, is still one of the most renowned libraries among developers. It is an elegant and simple HTTP library for Python, written in natural language (`https://requests.readthedocs.io/en/latest/`), as shown in *Figure 2.16*:

Figure 2.16: requests library – HTTP for Humans

We took a look at `requests` in the *Installing libraries* section. For our purposes, we are using `requests` version 2.28.1.

Compared to other HTTP libraries in Python, `requests` is highly rated because of its readable, developer-friendly, and simple-to-use attributes. Some of its main qualities are as follows:

- Short, simple, and readable functions and attributes
- Access to various HTTP methods
- Automatic content decoding and decompression
- Easy processing of query strings
- Customization of HTTP headers
- Session and cookie processing
- Ability to deal with JSON requests and content
- Proxy support
- Multipart file uploads (form handling)
- Connection pooling and timeouts

We will be using the `requests` library, and we'll explore some of its main properties. In the *Loading URLs* section, we used the `get()` method to load a URL and view its source using `content`. We were actually using the **HTTP GET method** to communicate with the URL using `get()`.

The `requests` library also supports other HTTP methods, such as **PUT**, **POST**, **DELETE**, **HEAD**, and **OPTIONS**, using the `put()`, `post()`, `delete()`, `head()`, and `options()` methods, respectively.

With this brief introduction and overview of `requests`, we will cover some more details with code examples in the following sections.

General usage

Let's explore some of the key features of the `requests` library with some examples, as shown in *Figure 2.17*. The code is available at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter02/requests.ipynb`:

```python
import requests

link="https://requests.readthedocs.io/"

response = requests.get(link)     #making HTTP Request to Link

type(response)

requests.models.Response

response

<Response [200]>
```

Figure 2.17: Using get() and showing the response type and status code

> **Important note**
>
> We recommend using `dir(requests)` and `dir(response)`, or using `dir()` whenever a new object is found in the code. This will show detailed information about that object and will even help you to decide which features to use.

As shown in *Figure 2.17*, we have imported `requests` and loaded the URL defined in `link`. `response` is typically an object of `requests.models.Response`, and it shows the HTTP status code too, which is `200`.

As shown in *Figure 2.18*, to get the actual status code, we can use the `status_code` attribute:

```
response.status_code  #status_code
```

```
200
```

```
response.status_code == requests.codes.ok
```

```
True
```

```
response.url
```

```
'https://requests.readthedocs.io/en/latest/'
```

```
response.history
```

```
[<Response [302]>]
```

Figure 2.18: Using requests with status_code, url, and history

In addition, `response.url` shows you the final URL after the HTTP communication is complete. We can see that the value of `response.url` is different from that of `link` (as shown in *Figure 2.17*) – if this is true, then there must have been some redirection, which is confirmed by the status code received from `history` (`302` indicates a temporary redirect).

Encoding is also an important factor; we can get some information on encoding, as shown in *Figure 2.19*:

```
response.encoding
```

```
'ISO-8859-1'
```

Figure 2.19: The encoding attribute of a response returning the encoding value

Here, we have explored some basic usage of `requests` with the help of a few code examples. In the next section, we will explore HTTP headers.

Request/response headers

As mentioned in the introduction of the *URL handling and operations* section, HTTP headers are quite significant to developers; HTTP headers shared by servers and clients can differ for requests and responses. Some examples are shown in *Figure 2.20*:

```
response.request.headers                                           ⎘ ↑ ↓ ⎙ ⎙ 🗑

{'User-Agent': 'python-requests/2.28.1', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}

response.headers #response headers

{'Date': 'Fri, 30 Dec 2022 07:07:57 GMT', 'Content-Type': 'text/html', 'Transfer-Encoding': 'chunked', 'Connection': 'keep-alive', 'Vary':
'Accept-Encoding', 'x-amz-id-2': '9Q0pv+loGFax07QRWPdkwDvhgNJgdFt6y5A+7SFKuZgi2VOahAhJi8AXAeUnC22cPbOu6bw6U/Y=', 'x-amz-request-id': 'ZMJPXF
BNVEH87ZJR', 'Last-Modified': 'Mon, 26 Dec 2022 04:25:28 GMT', 'ETag': 'W/"9ed64db0a133e3c979186c38a229291d"', 'X-Served': 'Nginx-Proxito-Se
ndfile', 'X-Backend': 'web-i-08f3b4c96e7dacf54', 'X-RTD-Project': 'requests', 'X-RTD-Version': 'latest', 'X-RTD-Path': '/proxito/html/reques
ts/latest/index.html', 'X-RTD-Domain': 'requests.readthedocs.io', 'X-RTD-Version-Method': 'path', 'X-RTD-Project-Method': 'subdomain', 'Refe
rrer-Policy': 'no-referrer-when-downgrade', 'Strict-Transport-Security': 'max-age=31536000; includeSubDomains; preload', 'Content-Encoding':
'gzip', 'CF-Cache-Status': 'HIT', 'Age': '670', 'Expires': 'Sat, 31 Dec 2022 07:07:57 GMT', 'Cache-Control': 'public, max-age=86400', 'Serve
r': 'cloudflare', 'CF-RAY': '7818f104384c8e54-KTM', 'alt-svc': 'h3=":443"; ma=86400, h3-29=":443"; ma=86400'}

response.headers['ETag']

'W/"9ed64db0a133e3c979186c38a229291d"'
```

Figure 2.20: Headers (from both responses and requests)

`response.request.headers` returns the headers during HTTP requests, while `response.headers` returns headers during HTTP responses. Both headers are available as dictionaries, and we can use the values to set and update the required key-value pair of the dictionaries while processing scraping scripts.

It's sometimes compulsory with crawler scripts for headers such as `User-Agent`, `Accept`, `Referer`, and `Cookies` to be set and passed while making HTTP requests. In the next section, we will explore cookies.

Cookies/session

HTTP cookies are data sent by the server to the browser. Cookies are data that is generated and stored by websites on the user's system or computer. Data stored in cookies helps to identify HTTP requests from the user to the website. Cookies contain information regarding session management, user preferences, and user behavior. Many websites ask for permission to set cookies while you're browsing the web.

As shown in *Figure 2.21*, `cookies` is returned as an object of `RequestCookieJar`:

```
response.cookies

<RequestsCookieJar[]>

response_github = requests.get('https://www.github.com/anishchapagain')

response_github.cookies

<RequestsCookieJar[Cookie(version=0, name='_gh_sess', value='GPeokFKzjBt19jxJKvA7hrB01L71TgBvLNOUERBnBvZOr1Oazt1EpMzDhLNJmIrUFQZKxOVSmAO%2BS
7w1gO2VWWq0OmkvX%2F%2BPW%2BVgzN8BVo0IZCnsviJB2aH8tPaA%2BePK85rCsOStdPIBxbkOl4CIHPhhORq2PMjixTCgbr8WwgxxHbhOOUxKeTlflykQsJAsRpqq71zq4DgN3OZik
ED3dC%2B44cYSFWtpY37usr85IIkPmOEScu5Mc%2F7R2ItGwDZV1mppdRnJB2UqQLP6OPPllA%3D%3D--FXvvagsYkwA%2Fvh5m--iJ8FjbbELXJH8YsYj2Bkkw%3D%3D', port=Non
e, port_specified=False, domain='github.com', domain_specified=False, domain_initial_dot=False, path='/', path_specified=True, secure=True,
expires=None, discard=True, comment=None, comment_url=None, rest={'HttpOnly': None, 'SameSite': 'Lax'}, rfc2109=False), Cookie(version=0, na
me='_octo', value='GH1.1.504589393.1672387987', port=None, port_specified=False, domain='.github.com', domain_specified=True, domain_initial
_dot=False, path='/', path_specified=True, secure=True, expires=1703923987, discard=False, comment=None, comment_url=None, rest={'SameSite':
'Lax'}, rfc2109=False), Cookie(version=0, name='logged_in', value='no', port=None, port_specified=False, domain='.github.com', domain_specif
ied=True, domain_initial_dot=False, path='/', path_specified=True, secure=True, expires=1703923987, discard=False, comment=None, comment_url
=None, rest={'HttpOnly': None, 'SameSite': 'Lax'}, rfc2109=False)]>
```

Figure 2.21: Cookies obtained as an object of RequestCookieJar

Our earlier response from `https://requests.readthedocs.io`, as shown in *Figure 2.17*, did not have any cookies as it returns an empty object, so we loaded new URL requests with `https://github.com/anishchapagain` and obtained cookies from the `response_github` variable.

We can create a loop on `response_github.cookies` and collect or inspect details or values from the cookies, such as `domain`, `key`, `value`, `expires`, `secure`, `path`, and `port`, as shown in *Figure 2.22*:

```python
for cookie in response_github.cookies:
    if cookie.domain=='github.com':
        print(cookie.name,' : ',cookie.value)
        break
```

```
_gh_sess  :  GPeokFKzjBt19jxJKvA7hrB01L71TgBvLNOUERBnBvZOr1Oaz
```

Figure 2.22: Looping on response_github.cookies (domain, name, and value)

In addition, we can *pass cookies* while making HTTP requests, and even *set cookies* in the `RequestCookieJar` object and forward them to the server or website while making HTTP requests. These steps are to be handled carefully and during required cases only (for example, when the pages are loaded with some cookie values, such as token and ID. If there are tokens when visiting a page, then those values are also required for the scraping script. This is why we collect cookies from the targeted site). Some examples of cookie-related code are shown in *Figure 2.23*:

```python
cookies = dict(name='somedomain', path='/') #dictionary
cookies
#responseA = requests.get(someurl, cookies=cookies)
```

```
{'name': 'somedomain', 'path': '/'}
```

```python
cookiesJar = requests.cookies.RequestsCookieJar() #Jar: key,value,domain,path
cookiesJar.set('some_name', 'some_value', domain='some_domain.com', path='/secureHttps')
cookiesJar
#responseB = requests.get(someurl, cookies=cookiesJar)
```

```
<RequestsCookieJar[Cookie(version=0, name='some_name', value='some_value', port=None, port_
specified=False, domain='some_domain.com', domain_specified=True, domain_initial_dot=False,
path='/secureHttps', path_specified=True, secure=False, expires=None, discard=True, comment
=None, comment_url=None, rest={'HttpOnly': None}, rfc2109=False)]>
```

Figure 2.23: Setting up cookies and passing them to the HTTP request

> **Important note**
>
> For more information on cookies, please visit `https://www.aboutcookies.org/` and `https://www.allaboutcookies.org`.

The server identifies the cookies' values and communicates with the browser based on the information that is stored in the cookies. Data in cookies helps a website access and transfer values such as session ID, expiration date, and the time the cookie information was stored.

A session or web session is normally identified as information stored in the server (temporarily), and it persists throughout the user's interaction with the site.

As shown in *Figure 2.24*, a session is often related to a user's credentials on a website:

```python
session_obj = requests.Session()
```

```python
response_obj = session_obj.get('https://httpbin.org/cookies', cookies={'name': 'anishchapagain','date':'30-dec-2022'})
response_obj.text
```

```
'{\n  "cookies": {\n    "date": "30-dec-2022", \n    "name": "anishchapagain"\n  }\n}\n'
```

```python
response_obj.headers
```

```
{'Date': 'Fri, 30 Dec 2022 09:24:12 GMT', 'Content-Type': 'application/json', 'Content-Length': '80', 'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0', 'Access-Control-Allow-Origin': '*', 'Access-Control-Allow-Credentials': 'true'}
```

```python
response_test = session_obj.get('https://httpbin.org/cookies')
response_test.text
```

```
'{\n  "cookies": {}\n}\n'
```

Figure 2.24: HTTP requests with requests.Session() objects

> **Important note**
>
> During an HTTP request, it is important that `header`, `cookies`, and `session` values obtained from DevTools or code output are monitored and updated or replaced with values such as *proxy* and *header/cookie keys* frequently. While running the scraping script, using these updated values and keys will help you bypass server exceptions and detect bots. Visit `https://requests.readthedocs.io/en/latest/user/advanced/#advanced` for more information on sessions and proxies.

You now have a basic idea of HTTP request attributes, such as cookies, headers, sessions, and methods. In the next section, we will focus on the contents of URLs or, more appropriately, HTTP responses.

Response content

We can receive pages or responses, or the page source of the requested URL, using the `content` attribute in bytes, whereas the `text` attribute returns a string object, as displayed in *Figure 2.25*:

```
response.content[0:500] #content 500 Chars
```

```
b'\n<!DOCTYPE html>\n\n<html lang="en">\n  <head>\n     <meta charset="utf-8" />\n     <meta name="viewport" content="wi
dth=device-width, initial-scale=1.0" /><meta name="generator" content="Docutils 0.17.1: http://docutils.sourceforge.ne
t/" />\n\n    <title>Requests: HTTP for Humans\xe2\x84\xa2 &#8212; Requests 2.28.1 documentation</title>\n     <link re
l="stylesheet" type="text/css" href="_static/pygments.css" />\n    <link rel="stylesheet" type="text/css" href="_stati
c/alabaster.css" />\n     <link rel="stylesheet" ty'
```

```
response.text[0:500]
```

```
'\n<!DOCTYPE html>\n\n<html lang="en">\n  <head>\n     <meta charset="utf-8" />\n     <meta name="viewport" content="wid
th=device-width, initial-scale=1.0" /><meta name="generator" content="Docutils 0.17.1: http://docutils.sourceforge.ne
t/" />\n\n    <title>Requests: HTTP for Humansâ\x84¢ &#8212; Requests 2.28.1 documentation</title>\n    <link rel="sty
lesheet" type="text/css" href="_static/pygments.css" />\n    <link rel="stylesheet" type="text/css" href="_static/alab
aster.css" />\n     <link rel="stylesheet" ty'
```

Figure 2.25: Response (content and text)

Furthermore, `requests` also returns a `raw` socket response, as shown in *Figure 2.26*. We can get our response using the `stream` argument with `get()`:

```
new_response = requests.get(link,stream=True)
```

```
new_response.raw.read(200) #first 200 Chars
```

```
b'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\xff\xff\xbcWkn\xdbF\x10\xfe\xafSlh\xa0I\x80\x90\xb4\x94
\xa6\x8e\x1dI\x80\x13\x1bp\x8a\xbc`+(\xda?\xcc\x8a\x1c\x89\xeb.w\x99\xdd\xa5,5\xc8\xbf\x9e\xa4\x87\xe8\x01z\x9
4\x9e\xa43KQ\x12m\xc9\xb0\x1b\xb4\x00m\x91\xf3\xf8f\xb8\xf3d\xa7\xff\xe0\xe4\xfd\xab\xd1\xcf\x1fNY\xee\n9\xect
```

Figure 2.26: Raw socket response, stream=True and read()

A raw response is bytes of characters that have not been transformed or decoded automatically.

After receiving a response or page content, most of the time it is quite useful to save it in a local file. This is done so we can study the content to reveal the code format, determine the DOM tree structure, or find elements. This is shown in *Figure 2.27*, and we have named this local file `test.html`:

```python
f = open('test.html',"wb") #open the file in 'w' mode
f.write(response.content) #write response.content to the file
f.close() #close file handler
```

Figure 2.27: Writing response.content to an external file

We now have responses, and they are available in a local HTML file. This file can be used to analyze the DOM structure, find query-related patterns, and many more things. We find different kinds of HTTP responses (JSON, PDF, HTML, and XML), and the next section will help us to deal with JSON responses.

Reading JSON

`requests` handles JSON very effectively with its built-in decoder. You can see the use of `json()` in *Figure 2.28*:

```
json_response = requests.get("https://feeds.citibikenyc.com/stations/stations.json").json()
for i in range(5):
    print(json_response['stationBeanList'][i]['stationName'])

W 52 St & 11 Ave
Franklin St & W Broadway
St James Pl & Pearl St
Atlantic Ave & Fort Greene Pl
W 17 St & 8 Ave
```

Figure 2.28: Loading and reading JSON content

Web scraping and crawling tasks might use any of the previously mentioned processes (HTTP headers, obtaining cookie information, status codes, or saving response content to local files) for HTTP communication and collecting or validating content. Most of the time, the methods and attributes provided by Python libraries make things convenient and easy. This is for the following reasons:

- A few logical steps (such as looping with conditions) found in some libraries are written as short snippets of code; these snippets serve specific purposes right where they are needed and are found wrapped up as functions, which are more convenient to use than procedural coding.

- Most of the new libraries overcome limitations and logical restrictions by embedding and wrapping up one or more functions, as new features that exist in old libraries are managed by the latest libraries.

- There might also be extra features in some functions inside new libraries that are lengthy and troublesome to manage. In such cases, developers choose to use old libraries.

In the next section, we will be using `requests` to implement the HTTP GET and POST methods.

Implementing HTTP methods

Generally, web-based interaction or communication between websites and users is achieved as follows:

1. The user accesses a web page or navigates through the content that is available to them.
2. The user then submits information to the website through an HTML form, by searching, logging in, registering themselves, and so on, and finally receiving the content they asked for.

In this section, we will be using the Python `requests` library to implement HTTP methods fitting the scenarios we just listed.

GET

The HTTP GET method is the default HTTP method. If no HTTP method is defined, then GET will be used by the code. We used some code earlier in this chapter that used GET. We were using the GET method unknowingly and without declaring it in the code.

By using GET, the resource's state is not altered, so it is the default and safest method. GET parameters, also known as query strings, are visible in the URL. They are appended to the domain name or path using ?, and are available as key-value pairs. Such URLs can be easily cached and bookmarked. Here are a few examples:

- `https://www.python.org`
- `https://requests.readthedocs.io/en/latest/user/quickstart/`
- `https://www.amazon.com/Hands-Web-Scraping-Python-operations/product-reviews/1789533392/ref=cm_cr_othr_d_show_all_btm?ie=UTF8&reviewerType=all_reviews`

The `get()` method from `requests` uses the HTTP GET method for communication and also accepts a few (not compulsory) arguments:

- `params`: Query string
- `headers`: HTTP request headers
- `link`: URL to access
- `cookies`: Some key-value pairs
- `proxies`: Demo IPs that will work for you
- `timeout`: The amount of time in seconds that the request will wait for the client to establish the connection

Finally, GET requests are created with some of these defined arguments, as shown in *Figure 2.29*:

```
url='https://www.anishchapagain.com'
queries={'e':'UTF8','reviewerType':'all_reviews'}
addHeaders={'user-agent':'','accept':'text/html','referer':'https://www.anishchapagain.com'}
demoCookie={'domain':'anishchapagain'}
demoIps = {'http':'http://10.10.1.10:3128',  'https': 'http://10.10.1.10:1080'}
```

```
response = requests.get(url, params=queries)
response.url
```

```
'https://www.anishchapagain.com/?e=UTF8&reviewerType=all_reviews'
```

```
response = requests.get(url,params=queries,headers=addHeaders,cookies=demoCookie,proxies=demoIps,timeout=3).content
print(response)
```

Figure 2.29: HTTP GET request with various arguments supplied

GET is the default HTTP method. Similarly, we have POST, which is a bit stricter and is explained in the next section.

POST

POST requests are known as secure requests that are made to the source or server. The requested resource's state might be altered too. Data that's posted or sent to the requested URL is not visible in the browser (unlike GET query strings); instead, it's transferred to the requested body. Requests with the POST method cannot be cached or bookmarked and have no restrictions in terms of character length.

In the example code, as shown in *Figure 2.30*, the https://httpbin.org/post URL (a simple HTTP request and response service) has been used to demo a POST request:

```
pageUrl= 'http://httpbin.org/forms/post'
postUrl= 'http://httpbin.org/post'
params = {'customer':'Mr Anish','custtel':'','custemail':'anish@anishchapagain.com','size':'medium',
          'topping':['cheese','mushroom'],'delivery':'14:45','comments':'No comments'}
headers = {'accept':'text/html, */*', 'referer':pageUrl}
```

```
response = requests.post(postUrl,data=params,headers=headers).json()
print(response)
```

```
{'args': {}, 'data': '', 'files': {}, 'form': {'comments': 'No comments', 'custemail': 'anish@anishchapagain.
com', 'customer': 'Mr Anish', 'custtel': '', 'delivery': '14:45', 'size': 'medium', 'topping': ['cheese', 'mu
shroom']}, 'headers': {'Accept': 'text/html, */*', 'Accept-Encoding': 'gzip, deflate', 'Content-Length': '14
5', 'Content-Type': 'application/x-www-form-urlencoded', 'Host': 'httpbin.org', 'Referer': 'http://httpbin.or
g/forms/post', 'User-Agent': 'python-requests/2.28.1', 'X-Amzn-Trace-Id': 'Root=1-63aef494-1499c33129671b8c7f
f6c05f'}, 'json': None, 'origin': '103.10.31.93', 'url': 'http://httpbin.org/post'}
```

Figure 2.30: HTTP POST request

pageUrl accepts data to be posted, as defined in params, to postUrl. Custom headers are assigned to headers. The post() function accepts a few arguments: the URL, for example, postUrl, data (the data to be posted), and headers, and returns a response in JSON format.

The POST request code in *Figure 2.30* automates params and headers (partially) with the required URLs as though we were submitting the actual HTML form on http://httpbin.org/forms/post and posting it to http://httpbin.org/post from the browser.

You can perform the same task manually from the browser – open DevTools and verify the parameters or **Payload** being used in the **Network** panel, as shown in *Figure 2.31*:

Figure 2.31: Payload with form data (an actual params key-value pair used in code)

As shown in *Figure 2.31*, the form data is similar to params in *Figure 2.30*. It is always recommended to learn, note, and detect the request/response sequences for any sites through the browser and DevTools.

We can get plenty of information from DevTools that can be used to deploy scripts and automate the process. In the current scenario, *Figure 2.32* displays the maximum amount of information from **Headers** in the **General** and **Request Headers** sections:

Figure 2.32: Request URL, method, and headers used in the HTTP POST request

The example and figures in this section relate to what the HTTP POST method does, as well as how POST and the parameters provided to POST differ in comparison to the GET method. Implementing the HTTP method is the core aspect of HTTP activities. For more detailed information on HTTP methods, please visit `https://requests.readthedocs.io/en/latest/api/` and `https://www.w3schools.com/tags/ref_httpmethods.asp`.

Summary

In this chapter, we have learned about Python programming, setting up a virtual environment, and installing Python libraries to send requests to web resources and collect responses and some additional information.
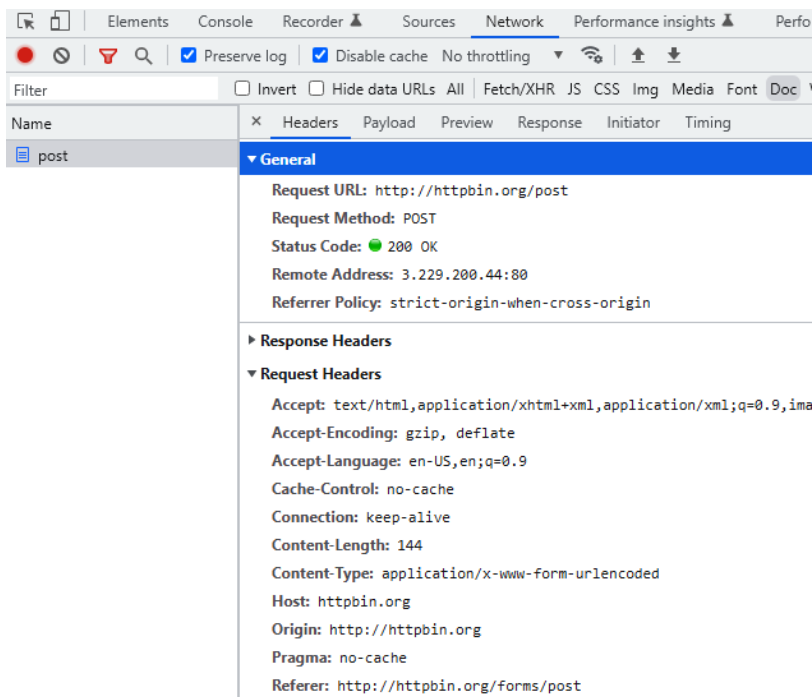
The main objective of this chapter was to demonstrate the core features of the `requests` library. Our primary aims were to learn how to deal with the HTTP request and response cycle, how to use HTTP methods (with extra parameters), how to use DevTools, and what the benefits are of using Python in this domain.

In the next chapter, we will learn and use some essential techniques to identify and extract data from web content.

Further reading

- **Python programming**:
    - `https://www.python.org/doc/`
    - `https://www.w3schools.com/python/default.asp`
- **Data and data analysis**:
    - `https://www.diesel-plus.com/the-importance-of-data-collection-10-reasons-why-data-is-so-important/`
    - `https://www.simplilearn.com/data-analysis-methods-process-types-article`
    - `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8274472/`
    - `https://scientific-publishing.webshop.elsevier.com/research-process/when-data-speak-listen-importance-of-data-collection-and-analysis-methods/`

- **requests, urllib, JSON**:

  - `https://requests.readthedocs.io/en/latest/`

  - `https://pymotw.com/3/`

- **Virtual environment**:

  - `https://docs.python.org/3/library/venv.html`

  - `https://docs.python-guide.org/dev/virtualenvs/`

  - `https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html`

  - `https://docs.anaconda.com/free/navigator/tutorials/manage-environments/`

- **Jupyter Notebooks**:

  - `https://jupyter.org/`

  - `https://code.visualstudio.com/docs/datascience/jupyter-notebooks`

- **Cookies**:

  - `https://www.cloudflare.com/en-gb/learning/privacy/what-are-cookies/`

  - `https://www.cookieyes.com/blog/session-cookies/`

- **URL and HTTP**:

  - `https://blog.hubspot.com/marketing/parts-url`

  - `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview`

Part 2:
Beginning Web Scraping

In this part, you will learn how to analyze, plan, and process a collection of desired or required data from a target website, collecting or writing the data to the desired file format. You will code a scraper using Python and its selected libraries. The chapters in this part will introduce various aspects of scraping practices that can be done effectively and efficiently.

This part contains the following chapters:

- *Chapter 3, Searching and Processing Web Documents*
- *Chapter 4, Scraping Using PyQuery, a jQuery-Like Library for Python*
- *Chapter 5, Scraping the Web with Scrapy and Beautiful Soup*

3

Searching and Processing Web Documents

So far, we have learned about web scraping, data-finding techniques, and related technologies that help us with scraping, and we've identified a few reasons to select the Python programming language.

Web- or website-based content exists as HTML elements or as a predefined document or some kind of object (JSON). For extraction purposes, we need to analyze and identify such content, patterns, and objects. HTML-based elements are generally identified with **XML Path** (**XPath**) and **Cascading Style Sheets** (**CSS**) selectors, which are traversed and processed with scraping logic for the desired content. The `lxml` library will be used in this chapter to process markup documents. We will be using browser-based **Developer Tools** (**DevTools**) for finding content and element identification.

In particular, we will learn about the following topics in this chapter:

- Introducing XPath and CSS selectors to process markup documents
- Using web browser DevTools to access web content
- Scraping using `lxml` – a Python library
- Parsing `robots.txt` and sitemap.xml

Technical requirements

The *Google Chrome* or *Mozilla Firefox* web browser will be required and we will be using Python notebooks with *JupyterLab*.

Please refer to the *Setting things up* and *Creating a virtual environment* sections of *Chapter 2* and continue using the environment we created.

The Python libraries that are required for this chapter are as follows:

- `lxml`

- `urllib`

The code files for this chapter are available online on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter03`.

> **Important note**
>
> Web- or website-based content refers to the responses or page sources that are received after processing requests to a URL. Content can be of various types, such as PDF, CSV, TXT, XML, HTML, and JSON. In general, in this chapter, we are talking about HTML, page source, or markup documents as our primary content, unless stated otherwise.

Introducing XPath and CSS selectors to process markup documents

In the *Understanding the latest web technologies* and *Data-finding techniques used in web pages* sections in *Chapter 1*, we explored and discussed **HTML** and **XML** markup documents and their availability across the web.

Normally, markup is a kind of labeling or tagging of parts, sections, or any entities in documents, which helps to identify the content and even process it using a third-party application. We call them **tags** in HTML (`https://www.w3.org/html/`) and **nodes** in XML (`https://www.w3.org/standards/xml/`). Hence, markup documents are a tree-like structure, containing tags or nodes (nested or individual), also known as an **element tree**.

> **Important note**
>
> XML documents have been pretty popular and common across the web since the start of the growing internet era. Readability, encoding support, interoperability, and data exchangeability are a few core powers of XML. XML is still supported by the latest web technologies and is the utmost backbone of markup documents.

With the brief overview of markup and XML we have received here, the next sections will introduce the **Document Object Model** (**DOM**), XPath, and CSS selectors and using them for web scraping purposes with the help of a Python library.

The Document Object Model (DOM)

A **tree-type structure** or an element tree is a base model for most markup languages and is often known as the DOM. Please visit these links for more details on the DOM: `https://www.w3.org/TR/WD-DOM/introduction.html` and `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`.

DOM processing, DOM parsing, DOM manipulation, and many more similar terms refer to activities related to the DOM. With the help of the DOM and its defined conventions, we can access, traverse, and manipulate markup documents. Plenty of web-related technologies support DOM usage, such as **JavaScript** (**JS**), template engines, and web page development tools.

DOM elements (tags or nodes) are predefined or user-defined most of the time. We can find open and closed types of conventions used in markup documents. Elements are structured or nested inside parent elements and might be the parent or child of some other elements too. This tree-like structure made up of DOM elements is based on a language or convention and can be used to build markup documents, as seen in *Figure 3.1*:

Figure 3.1: HTML DOM tree structure

Understanding the DOM or element tree (HTML tree or XML tree) is a basic step toward understanding the document as a container of elements and following the extraction process. Information can be found nested inside a tree structure and could possess additional information along with attributes representing the content. Refer to the *HTML* section of *Chapter 1* for more details.

XPath and CSS selectors are used to navigate along the DOM and are used to search for the desired content in nodes or elements found. We will explore XPath in the next section.

XPath

The XPath language is a part of XML-based technologies such as XML, XSLT, and **XML Query** (**XQuery**). XPath (`https://www.w3.org/TR/xpath/`) deals with navigating through DOM elements and locating nodes in XML (elements or tags in HTML) documents using expressions, known as XPath expressions. XPath is *like a path* (*expressions are built using and representing HTML elements and XML nodes*) that identifies nodes in documents. XPath is also a **World Wide Web Consortium** (**W3C**) (`https://www.w3.org`) recommendation.

With XPath expressions, we can navigate hierarchically through elements and reach the targeted ones. XPath is also supported and implemented by various programming languages, such as *JS*, *PHP*, *Java*, *Python*, and *C++*. Web browsers and applications also have built-in support for XPath.

XPath expressions are also identified as absolute and relative:

- **Absolute path**: This expression represents a complete path from the root element to the desired or targeted element. In an HTML document, it begins with `/html` and looks like `/html/body/div[1]/div/div[2]/div/span/p[1]`. Individual elements, such as `div` and `p`, are generally identified by their position and represented by an index number, such as `[1]` or `[2]`.

- **Relative path**: This expression is somewhat shorter and more readable in comparison with an absolute path and is often preferred over absolute expressions. It begins with certain chosen or selected elements and ends with the desired element, for example, `//*[@id="answer"]/div/span/p[@class="text"]`.

While an absolute path traces all nodes in an element tree, a relative path uses the help of elements' attributes (**name-value pair**), as we saw in the *HTML elements and attributes* section of *Chapter 1*.

XPath is also a core block of XML technologies, such as **XQuery** and **eXtensible Stylesheet Language Transformations** (**XSLT**). An XPath expression can be built using a number of built-in functions available for various data types. XPath expressions can contain code patterns, functions, and conditional statements, and also support the use of predicates.

> **Important note**
> XQuery is a query language that uses XPath expressions to extract data from XML documents. XSLT is used to render XML in a more readable format, using a stylesheet. For more details, please visit `https://www.w3.org/TR/xslt-30/` and `https://www.w3.org/TR/xquery-31/`.

Let us explore a few XPath expressions from the XML file available to us at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter03/breakfast.xml`, as seen in *Figure 3.2*:

```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <breakfast_menu>
3        <food>
4          <name>Brown Toast-Honey</name>
5          <price>$2.99</price>
6          <description>All time famous, honey and toast combination.</description>
7        </food>
8        <food>
9          <name>Oat-Pancake</name>
10         <price>$4.95</price>
11         <description>Delicious pancake, with cheese and berry on top.</description>
12       </food>
13       <food>
14         <name>Creamy Yoghurt</name>
15         <price>$3.99</price>
16         <description>Dried nut filled yoghurt, thick and creamy.</description>
17       </food>
18     </breakfast_menu>
```

Figure 3.2: XML content

As seen in *Figure 3.2*, we can deduce the following:

- Everything is a node in XML – `breakfast_menu`, `food`, `name`, and more

- An XML node can be an element itself, having start and end tags

- The `breakfast_menu` parent node has three child nodes called `food`

- Each `food` node has its own child nodes: `name`, `price`, and `description`

- There is no name-value pair or attributes in any nodes

For an XML document such as the one in *Figure 3.2*, most text or code editors provide some formatting options, for example, folding or unfolding an element or parent node and beautifying the texture, XPath query-related features, and more.

> **Important note**
>
> Numerous platforms and applications available online or offline provide XML formatting (plus code-related features), beautifying options, and XPath expression testing. A few such online providers are `https://codebeautify.org`, `https://www.freeformatter.com`, `https://xpather.com`, `https://try.jsoup.org/`, and `https://www.webtoolkitonline.com/`, among others.

In the following example, we will be using XPath Tester from Code Beautify (`https://codebeautify.org/Xpath-Tester`) to format XML content and test XPath expressions:
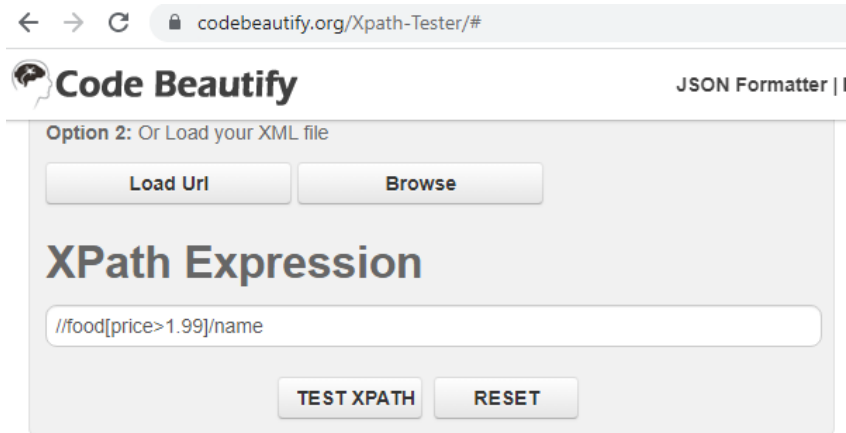
Figure 3.3: XPath Expression – //food[price>1.99]/name

As seen in *Figure 3.3*, the `//food[price>1.99]/name` expression is applied to the XML file in *Figure 3.2*.

As seen in the following output, the XPath expression applied in *Figure 3.3* returns only the `name` element of those `food` nodes where the predicate defined as `[price>1.99]` matches the `price` value greater than `1.99`:

```
<name>Brown Toast:Honey</name>
<name>Oat:Pancake</name>
<name>Creamy Yoghurt</name>
```

A predicate is used to identify a specific node or element. They are written using square brackets, which are similar in syntax to Python lists. Here are a few XPath expressions with general explanations applied on the `breakfast.xml` file seen in *Figure 3.2*:

- `//`: Document node
- `*`: All elements in the document
- `//breakfast_menu`: Root element and all elements within `breakfast_menu`
- `//food`: Selects all `food` elements
- `//food/name`: Selects the name element from all `food` elements
- `//food[1]/name/text()`: Returns only the text from `name` of the first `food` element
- `//food[1]/name`: Selects the name of the first `food` element, for example, `<name>Brown Toast-Honey</name>`
- `//food[2]/*`: Selects all elements from `food` in the second position

- `//food[position()<3]/price`: Selects the price of the `food` elements that are located in a position less than 3

- `//food[price>3.99]/name`: Filters and selects all `name` elements from `food` elements where `price` has a value greater than `3.99`, for example, `<name>Oat:Pancake</name>`

- `//food[last()]/name`: Selects the `name` element of the last `food` element

- `//food[last()]/name | //food[last()]/price`: Selects the `name` and `price` elements from the last `food` element

- `sum(//food/price)`: Sums all the values of `calories` inside `food`, which adds up to the value `11.9`

- `//food/name[contains(.,"Toast")]`: Selects the `name` element of the `food` element that contains the `"Toast"` string

- `//food/description[starts-with(.,"Dried")]`: Selects the `description` element from the `food` element that matches the option that starts with the `"Dried"` string

- `//food[price>1.99 and price<4.00]/name`: Selects the `name` element from `food` elements where the `price>1.99` and `price<4.00` filter options match

Now, with a basic idea about XPath expressions, let us consider some expressions with attributes. Attributes are extra properties that identify certain parameters for a given node or element. A single element can contain some unique or common attributes. Attributes in HTML play a significant role in efficient and effective traversing. Let us consider the `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter03/books.xml` XML file with the following content:

```
<books>
    <book id="1491946008" price="47.49">
        <author>Luciano Ramalho</author>
        <title>Fluent Python: Clear, Concise, and Effective
        Programming</title>
    </book>
    <book id="1491939362" price="29.83">
        <author>Allen B. Downey</author>
        <title>Think Python: How to Think a Computer
        Scientist</title>
    </book>
</books>
```

XPath expressions accept key attributes by adding or prepending the @ character to the key. Listed here are a few expressions using attributes with brief explanations:

- `//book/@price`: Selects the `price` attribute of all books available

- `//book[@price>30]`: Selects all elements of `book` where the `price` attribute is greater than `30`
- `//book[@price<30]/title`: Selects all the `title` elements from `book` where `price` is less than `30`
- `//book/@id`: Selects the `book` attribute of `id` and its value
- `//@id`: Selects the `id` attribute and its value
- `//book[@price="29.83"]/author`: Selects `author` from `book` where the `price` attribute matches `29.83`
- `sum(//@price)`: Returns the total of `price` attribute values, i.e., `77.3`

In this section, we have looked at a few examples and expressions to learn about XPath and how can we use such expressions to retrieve the desired content. Similar to XPath, we also have selectors based on CSS, which we will explore in more detail in the next section.

CSS selectors

In the *Understanding the latest web technologies* and *Data-finding techniques used in web pages* sections in *Chapter 1*, we learned about CSS and its use to style HTML. There are various ways to apply CSS to HTML.

CSS selectors (CSS queries or CSS selector queries) are defined patterns used by CSS to select elements. Similar to XPath expressions, which are used to find and identify elements, CSS selectors are used to select or find HTML elements and define a style for them. In the extraction process, we search for and find elements using CSS selectors.

The following code shows plain HTML with basic tags:

```
<html>
<head>
    <style>
        a{color:blue;}
        h1{color:black; text-decoration:underline;}
        #idOne{color:red;}
        .classOne{color:orange;}
    </style>
</head>
<body>
    <h1> Welcome to Web Scraping </h1> Links:
    <a href="https://www.google.com"> Google </a>  
    <a class='classOne' href="https://www.yahoo.com"> Yahoo
    </a>  
    <a id='idOne' href="https://www.wikipedia.org">
```

```
     Wikipedia </a>
</body>
</html>
```

As seen in the preceding code, we have basic HTML defined in the `https://github.com/ PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/ blob/main/Chapter03/css_selector.html` file, we can collect information such as the following:

- `<h1>` is an HTML tag, an element, and a selector
- The first selector, `<a>`, has a single `href` attribute and it contains the text `Google`
- The second `<a>` selector has multiple attributes, such as `class` with the value `classOne`
- The last `<a>` selector also has multiple `id` attributes with the value `idOne`, `href` with some values, and the text `Wikipedia`

The distinguished selectors, with or without attributes, in the preceding source code can be used to select that particular element individually or in a group. Similar to XPath Tester, there are plenty of DOM parser providers available online and offline with CSS query-related facilities. We will be using `https://try.jsoup.org`, as seen in *Figure 3.4*:
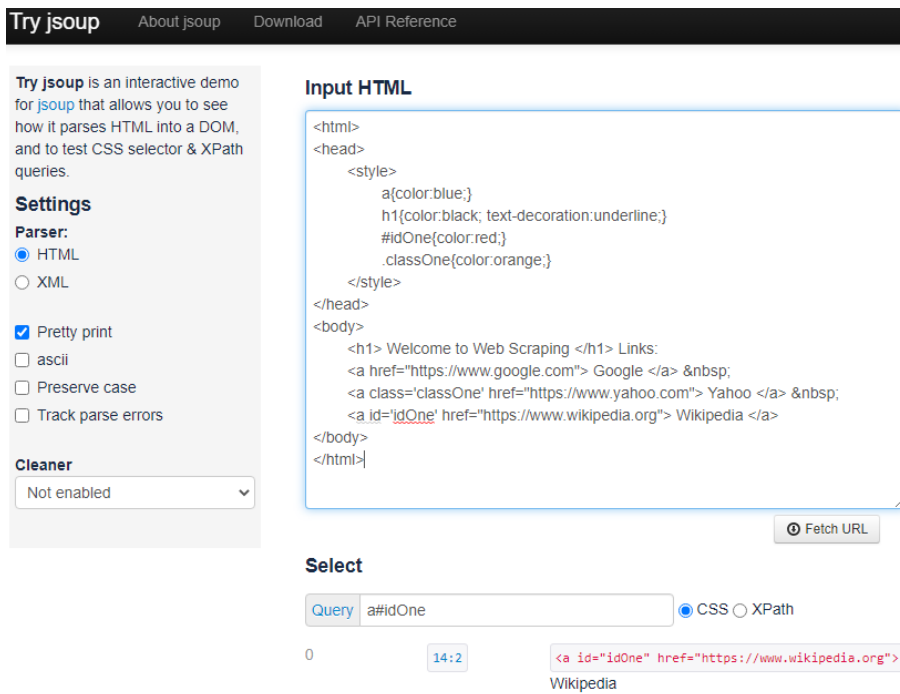

Figure 3.4: Evaluating CSS query a#idOne

CSS selectors have more variety in comparison to XPath expressions. In addition, they are categorized into four basic groups, based on the complexity, code patterns, and features available. Let's get some more information on available groups, presented in the following subsections.

Element selectors

These are some of the basic selectors, even called common selectors, that select or choose elements from HTML. Generally, these selectors are the basic tags of HTML, as seen in the following list:

- `h1`: Selects all `<h1>` elements in the document

- `a`: Selects all `<a>` elements in the document

- `body *`: Selects group of elements inside `<body>`

- `div a`: Selects all `<a>` elements inside `<div>`

- `h1,a`: Selects all `<h1>` and `<a>` elements

- `h1 + a`: Selects element `<a>` immediately after element `<h1>`

- `h1 ~ a`: Selects every `<a>` element preceded by `<h1>`

ID and class selectors

This type of selector adds additional features to the element selectors. In HTML where CSS has been used, we can find plenty of attributes, such as the ID represented by the # symbol and the class represented by `.`. These are also known as global or common attributes within HTML elements. These global attributes are mostly preferred over other attributes, as they define tags for structure and identification.

Listed here are the global attributes as found in the code and expressed as follows:

- `.classOne`: Selects an individual or group of elements with `class=classOne`

- `#idOne`: Selects an element with `id=idOne`

- `a.classOne`: Selects an individual or group of `<a>` elements with `class=classOne`

- `a#idOne`: Selects an individual `<a>` element with `id=classOne`

Attribute selectors

HTML tags contain attributes (single or multiple) that help to identify a particular element with the value it carries. These are much like the predicates used in XPath, as listed here:

- `a[href*=".org"]`: Selects `<a>` with the `href` attribute and finds `.org` in its value.

- `a[href^="https"]`: Selects `<a>` with the `href` attribute and a value that starts with `https`.

- `a[href$=".com"]`: Selects `<a>` with the `href` attribute and a value that ends with `.com`.

- `a[href~=google]`: Selects `<a>` with the `href` attribute and containing the text `google`. This is similar to `a[href*="google"]`.

- `[class=classOne]`: Selects elements that have `class=classOne`.

- `[href]`: Selects all elements with the `href` attribute name.

As seen in the preceding CSS selectors list, a few **regular expression** (**regex or Regex**) related syntaxes are also being used. This makes CSS selectors almost like regex for HTML documents. Here are some basic explanations of the symbols used in regex:

- `*` and `~`: Represent find or search, though both are used differently

- `$`: Represents the end of a string

- `^`: Represents the beginning of a string

Pseudo selectors

This type of selector is a set of handy choices when it comes to identifying or selecting elements based on their positions. Here are a few with basic explanations:

- `a:eq(1)`: Selects `<a>` at position one or the first `<a>` in the document

- `a:gt(1)`: Selects all `<a>` elements located at a position greater than one

- `:not(h1)`: Selects all elements in HTML available except `<h1>`

- `a:first-child`: Preferred with nested elements, this selects every `<a>` element that is the first child of its parent

- `a:last-child`: Selects every `<a>` element that is the last child of its parent

- `a:nth-child(1)`: Selects every `<a>` element from the first child of its parent

- `a:nth-last-child(2)`: Selects every second `<a>` element from the last child of its parent

- `a:nth-of-type(2)`: Selects every second `<a>` element from its parent

- `a:nth-last-of-type(2)`: Selects every `<a>` element in the second position from the last of its parent

- `a:last-of-type`: Selects the last `<a>` element of its parent

CSS selectors are used as a convenient alternative to XPath expressions. They are somewhat shorter in length and use both simple and regex-like patterns in their expressions. CSS selectors can be converted into XPath expressions, but not vice versa.

Many open source tools and applications that deal with XPath and CSS selectors provide facilities to convert between the two, but it's the responsibility of the developer to test such expressions and apply them accordingly in code. As we saw, in `https://try.jsoup.org`, such an option does exist.

`https://css-selector-to-xpath.appspot.com/` is quite popular among developers, shown in *Figure 3.5*:

Figure 3.5: CSS selector to XPath

In this section, we explored and learned about the most popular web-related pattern-finding techniques, XPath and CSS selectors. We can use both or either one of them as per convenience, code requirements, and availability in chosen libraries. In the next section, we will explore and learn about using browser-based **DevTools**.

Using web browser DevTools to access web content

DevTools are some of the most important tools available to us to explore response content of any type, such as HTML, JSON, XML, or TXT.

In the *Developer tools* section of *Chapter 1*, we introduced browser-based DevTools with various helpful, information-packed panels and a brief introduction to them. In this section, as the heading reads, we will be using DevTools to locate, find, or access the web content that we are seeking. Normally, we will search for and find the elements holding content in a similar way to how we dealt with XPath and CSS selectors using expressions.

We will explore web content using Google Chrome. Chrome has built-in DevTools with plenty of features that help us, with information on cookies, headers, `curl` scripts, prettifying the DOM, DOM navigation, displaying line numbers, folding/unfolding code blocks, element selection, element identification, content searching, and generating XPath and CSS selector expressions.

> **Important note**
>
> cURL, or curl, is a command-line tool and library that is used to communicate with servers and is used by machines and scripts to deal with URLs in the transfer of resources. Many programming languages use a library that communicates with curl and handles the communication with the web. For more information on curl, please visit `https://curl.se/`. You can also explore the Python interface to cURL (PycURL) at `http://pycurl.io/`.

We will now proceed with loading some URLs in Chrome and accessing some elements or navigating through the DOM using DevTools on the response content.

HTML elements and DOM navigation

The step of accessing elements and DOM navigation is important when we are codifying the concept or planning a crawler.

At this point, we should also know to be flexible enough that whatever URL we are trying to load and process could be changed or updated. Therefore, to deal with such situations occurring, we have selected a few sites that are available online for people to learn about web scraping and implement scraping. We also used those sites in the *first edition* of this book, *Hands-On Web Scraping with Python* by *Packt Publishing*. You are free to use a similar concept and steps with any other URL.

We will be using `https://toscrape.com`. This site provides two different URLs, `http://books.toscrape.com` and `http://quotes.toscrape.com/`, both loaded with plenty of categories, labeling, paginated content, DOM elements, authorization-related sections, and much more. Let us load `http://books.toscrape.com` using Google Chrome, as seen in *Figure 3.6*:
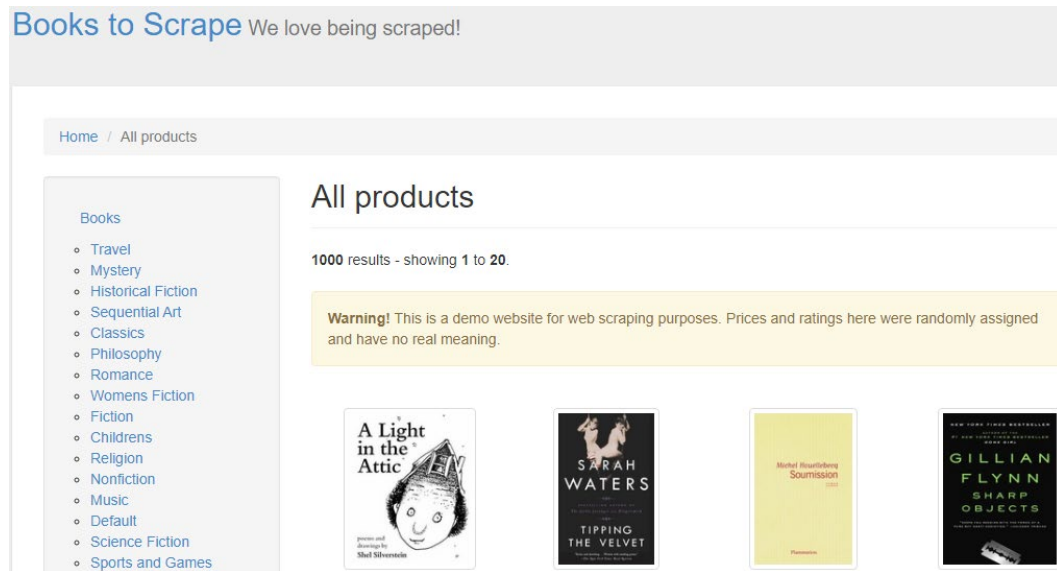

Figure 3.6: Loading books.toscrape.com

As the page content is successfully loaded, we can load DevTools with a *right-click* on the page and by selecting the **Inspect** option or by pressing *Ctrl + Shift + I*. With DevTools available, first, we can find a pointer icon listed on the left. This is used for selecting elements from the page, as shown in *Figure 3.7*. This element selector (also known as the element inspector) located in the top-left corner of DevTools, can be turned on and off by clicking it or pressing *Ctrl + Shift + C*:
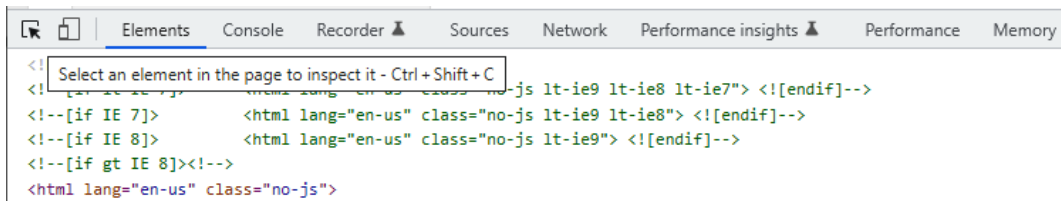
Figure 3.7: Element selector (inspector) on DevTools

We can move the mouse on the loaded web page by clicking the element inspector, and also by pressing *Ctrl + Shift + C*. The mouse or screen cursor can also be moved in the code available in the **Elements** panel.

We search for the exact HTML element, `<div class="alert alert-warning"..>`, that we are pointing to, using the element selector, as seen in *Figure 3.8*:

Figure 3.8: Using the element selector on the Warning! element

The following can be seen in *Figure 3.8*:

- The `<div class="alert alert-warning"..>` element found inside `<section>` has been selected.

- The `Warning` text is found inside `<div>` with the `<strong>` node. There is also a highlight on the page for that particular element.

We can repeat the steps discussed, shown in *Figure 3.7* and *Figure 3.8*, select any part of the page, and find the HTML tags or move across HTML and find a page section that is highlighted. We can also right-click in the element code and find options such as **Add attribute**, **Edit attribute**, **Duplicate element**, **Delete element**, **Cut**, **Copy XPath**, and **Copy selector**.

> **Important note**
>
> If we had to find the **Warning** text and the DOM element containing it without DevTools, then we would use the **page source**. To load the source of the page, we can right-click on the page and choose the **View Page Source** option or use *Ctrl + U*. This loads a new tab in the browser, such as view-source: `http://books.toscrape.com/`. On this page, we can search for the text and find it using line numbers and the DOM elements.
>
> DOM elements found using the element selector and page source should be verified, and might not be the same every time. In such cases, we should consider the element found from the page source.

So far, we have explored XPath, CSS selectors, and DevTools. In the next section, we will collect XPath and CSS selectors for the chosen element.

XPath and CSS selectors using DevTools

Let us proceed with the following steps to obtain an XPath and CSS query for the title with the text `A Light in the ...` and the `<a>` tag, as seen in *Figure 3.9*:

1. Choose **Element selector** and trace the `<a>` element.

2. Right-click on the traced element, `<a>`.

3. Select **Copy** from the available menu.

4. Select **Copy XPath** for XPath or **Copy selector** for CSS selector, as shown in *Figure 3.9*:
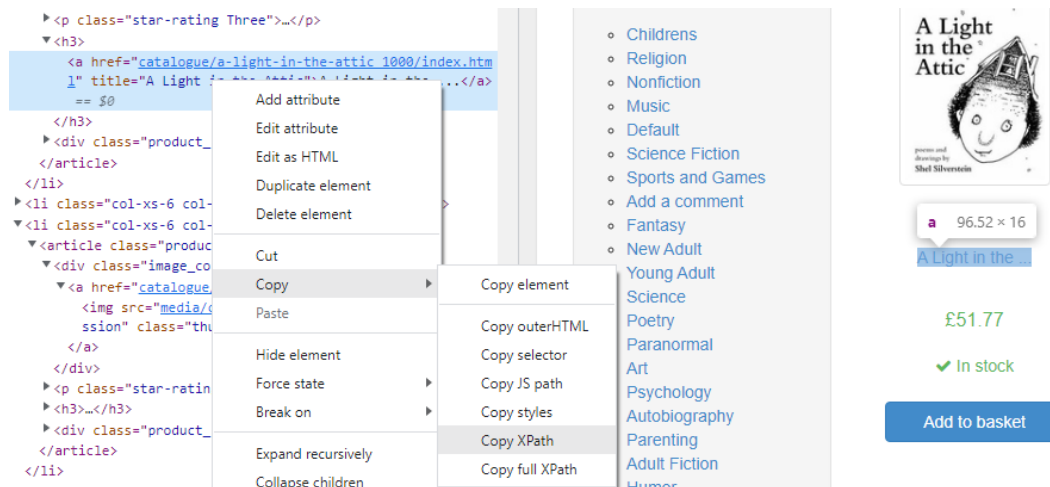

Figure 3.9: Copying XPath and CSS selector using element selector or inspector

Listed here are some results obtained from the preceding steps:

- **Copy XPath**: `//*[@id="default"]/div/div/div/div/section/div[2]/ol/li[1]/article/h3/a`

- **Copy full XPath**: `/html/body/div/div/div/div/section/div[2]/ol/li[1]/article/h3/a`

- **Copy selector**: `#default > div > div > div > div > section > div:nth-child(2) > ol > li:nth-child(1) > article > h3 > a`

- **Copy element**: `<a href="catalogue/a-light-in-the-attic_1000/index.html" title="A Light in the Attic">A Light in the ...</a>`

Similarly, you will collect the required expressions for selected elements. After collecting and verifying or testing the expressions, scraping logic is applied using Python to automate the data collection process. Understanding DOM elements (with attributes) and HTML page sources are a couple of the common areas that you must be familiar with.

There is no particular or perfect way to determine the HTML elements related to expressions. We should use tools, as discussed in this section, along with the page source available. You can choose any available tools or plugins on the market or browser-based extensions to determine the element expressions.

In this section, we inspected and explored the **Elements** panel for element identification and DOM navigation. In the next section, we will use the `lxml` Python library to scrape code and collect data from `toscrape.com` using XPath and CSS selectors.

Scraping using lxml – a Python library

The `lxml` library is an XML toolkit with a rich library set to process XML and HTML. `lxml` is preferred over other XML-based libraries in Python for its high speed and effective memory management, plus it has various other features to handle both small and large XML files.

Python programmers use `lxml` to process XML and HTML documents. There are plenty of other such libraries in Python; a few even build on top of `lxml` with extra add-ons. `lxml` is also used as a parser engine in Python libraries such as **Beautiful Soup** (`https://www.crummy.com/software/BeautifulSoup/bs4/doc/`) and **pandas** (`https://pandas.pydata.org/`).

DOM parsing, traversing element trees, XPath, and CSS selector are the features that make lxml effective and efficient enough for tasks such as web scraping. For more details on `lxml` and its documentation, please visit `https://lxml.de/`.

> **Important note**
> `lxml` provides native support to XPath and XSLT and is built on the powerful C libraries `libxml2` and `libxslt`. Its library set is normally used with XML or HTML to access XPath and parse, validate, serialize, transform, and extend features from `ElementTree` (`https://lxml.de/tutorial.html`).

Elements of markup language such as XML and HTML have start and close tags. Tags can also have attributes and can contain other elements. ElementTree is a wrapper that loads XML files as trees of elements. The Python built-in xml.etree (https://docs.python.org/3/library/xml.etree.elementtree.html) library is used to search, parse elements, and build document trees. Element objects also exhibit various accessible properties related to Python lists and dictionaries.

lxml contains important modules, listed here:

- lxml.etree (https://lxml.de/tutorial.html): For parsing and implementing ElementTree elements. Supports XPath, iterations, and more.

- lxml.html (https://lxml.de/lxmlhtml.html): Parses HTML and supports XPath, CSS selectors, HTML forms, and form submission.

- lxml.cssselect (https://lxml.de/cssselect.html): Converts CSS selectors into XPath expressions. Accepts CSS selectors or CSS queries as expressions.

With that basic introduction to lxml, let's explore some examples in the next section.

lxml by example

lxml is a core library and has a huge module set. In this section, we will explore the most common features of lxml with examples such as reading XML and HTML files and using lxml modules for scraping purposes.

Reading XML files

In this example, we will be reading the XML content available in the https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter03/breakfast.xml file using lxml, as seen in *Figure 3.10*:

Figure 3.10: Reading an XML file using lxml

As seen in *Figure 3.10*, the xml variable now contains the file content in bytes. We need to parse this to convert it into an *etree element*. We can use an lxml default parser such as etree.parse(), or as the content is an XML file here, we have chosen etree.XML(). In *Figure 3.11*, we can also find which parser is being used by using etree.get_default_parser().

```
[4]: tree=etree.XML(xml) #parsing using XML()
```

```
[5]: tree # element object
```

```
[5]: <Element breakfast_menu at 0x26cc572f9c0>
```

```
[6]: type(tree)
```

```
[6]: lxml.etree._Element
```

Figure 3.11: Parsing the file content

We can see that tree is an Element object of lxml.etree. The element tree has been defined; now we can traverse and achieve node-based results such as iterating. As seen in *Figure 3.12*, we first iterate across all tree nodes using tree.iter(). If we are targeting some specific nodes such as name and calories, then we can choose them, for example, tree.iter('name', 'calories'):

```
[ ]: for element in tree.iter(): #iteration with all elements
         print(f"{element.tag} - {element.text}")
```

Iterating among selected nodes in tree

```
[9]: for element in tree.iter('name','calories'):
         print(f"{element.tag} - {element.text}")
```

```
name - Belgian Waffles
calories - 650
name - Strawberry Belgian Waffles
calories - 900
```

Figure 3.12: Iterating through all and then selected tree nodes

In *Figure 3.13*, we are applying logic on element tags and their text, as well as applying XPath using the tree.xpath() method to acquire the desired results:

Applying condition for element 'calories' if text is less than 800

```
[ ]:  for element in tree.iter():
          if element.tag=="calories":
              if int(element.text)<800:
                  print(f"{element.text}")
```

Applying condition for element 'calories' if text is less than 800 using XPath

```
11]:  for element in tree.xpath('//food'):
          if int(element.xpath('./calories/text()')[0])<800:
              print(f"{element.xpath('./name/text()')[0]} - {element.xpath('./calories/text()')[0]}")
```
```
Belgian Waffles - 650
French Toast - 600
```

Figure 3.13: Iterating and applying logic with specific nodes and using XPath

As seen in this section, we need to parse the content first and then we can apply element-based logic to parts of content or whole content. Only basic usage of etree and its core features related to reading and traversing are used in this example. We will read and further process HTML documents in the next section.

Reading HTML documents

In this example, we will pass a request to the URL http://httpbin.org/forms/post using urllib.request, parse the web content using lxml.html, and traverse through the elements as lxml.html.HTMLElement, applying XPath and CSS selectors. Let's import and define the URL to be used, as seen in the following code:

```
from lxml import html
from urllib.request import urlopen    #loading URL
url='http://httpbin.org/forms/post'
```

Now, we will pass a request to the defined url string, using urlopen(), and parse the response received. As seen in the following code, parse() returns lxml.etree, but using getroot(), we are receiving HTMLElement as root. As this example is regarding HTML, we are interested in HTMLElement:

```
tree = html.parse(urlopen(url))    # load URL using urlopen and parse
type(tree)                         # lxml.etree._ElementTree
root = tree.getroot()              # returns the document root node <html>
type(root)                         # lxml.html.HtmlElement
```

We can process `tree` similarly to handling XML files and iterating through elements. For this example, we will use a few new functions, as listed here:

- `find()`: Used to locate the first element and returns `Element`. Similar to `find()` and `findall()`, it can be used to find and iterate through all elements.

- `text_content()`: Returns the text of the matched element.

- `findtext()`: Returns the text from the path provided.

These new functions are used in the following code:

```
tagP = root.find('.//p')                # .//p
print(tagP.text_content())
"Customer name: "
tagP1 = root.findtext('.//p/')     #.//p/
print(tagP1)
"Customer name: "
```

Now, as we are able to find and print text from elements, we will apply XPath and use CSSSelect (CSSSelect converts CSS selectors to XPath) on the scraper code.

As seen in the following code, XPath is applied as `root.xpath()` and CSS selectors as `root.cssselect()`, and the results obtained are exactly the same as for `xpath()` and `cssselect()`:

```
print(root.xpath('//p/label/input/@value'))
print(root.xpath('//legend/text()'))
print([formP.text_content().strip() for formP in root.xpath('//
form/p')])
…
print([e.get('value') for e in root.cssselect('p label
input[value]')])
print([l.text_content() for l in root.cssselect('legend')])
print([p.text_content().strip() for p in root.cssselect('form > p')])
```

The preceding code will result in exactly the same output as follows:

```
['small', 'medium', 'large', 'bacon', 'cheese', 'onion', 'mushroom']
[' Pizza Size ', ' Pizza Toppings ']
['Customer name:', 'Telephone:', 'E-mail address:', 'Preferred
delivery time:', 'Delivery instructions:', 'Submit order']
```

As there is a `<form>` element available in the HTML page source, and as seen in the expressions in the example, we can target specific elements too – say, the `<form>` element available in `root`.

As seen in the following code, there is only one `<form>` element found:

```
print(root.forms)
[<Element form at 0x1………..>]
print(root.forms[0].items())
[('method', 'post'), ('action','/post')]
print(root.forms[0].keys())
['method', 'action']
print(root.forms[0].method)
POST
print(root.forms[0].action)
http://httpbin.org/post
```

The element possesses `items()` and `keys()` and we can get more information regarding `method` (the `method` attribute with a value such as `GET/POST`) and `action` (the `action` attribute with a link, where the form values will be submitted).

In this section, we learned how to read, load, and parse content. We also looked at a few examples of element-based activities, such as iterating, finding elements, returning text, and applying XPath and CSS selector expressions. With this overview and look at code examples dealing with XML and HTML, in the next section, we will be scraping a website using `lxml`.

Web scraping using lxml

In this section, we will demonstrate an example related to web scraping using `lxml`. We will scrape and collect data from `http://books.toscrape.com`. To be more specific, we will target books from the **Childrens** category (`http://books.toscrape.com/catalogue/category/books/childrens_11/index.html`):
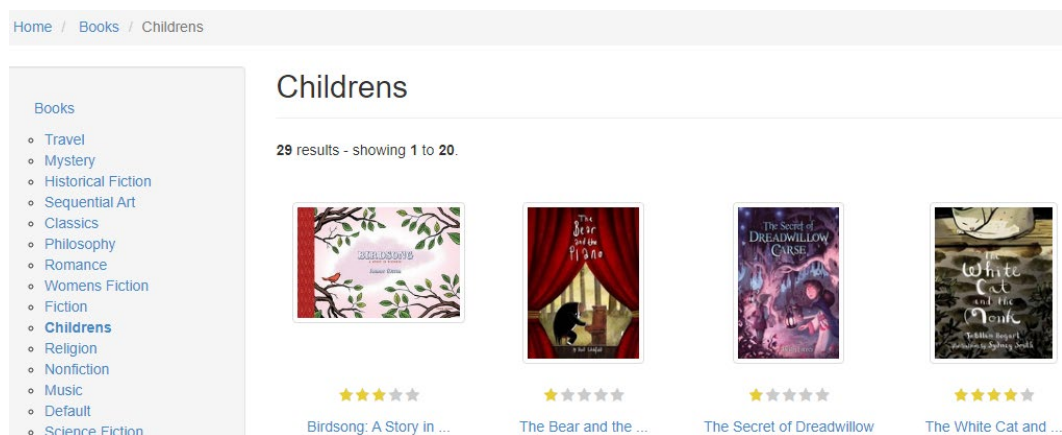

Figure 3.14: Childrens category page

As seen in *Figure 3.14*, this page has **29 results**, showing **20** on a page, so we have to implement the **pagination** concept: find the total pages, target the desired elements with expressions, and write data to a CSV file.

Important note

As we will be dealing with multiple pages for the selected category, it's good practice to explore and check the web response or page source using DevTools, targeting the page URL and information that we are looking for. A paginated URL might contain some different patterns for inner pages than the first page. Also, identify and collect the element name and XPath or CSS selector expression. In addition, while tracing content, it might be available in the page source or at some inner links with the use of an API. If an API exists, then the scraping task will be easier.

After a preliminary study of the page structure and the required URLs, we deploy the code shown here:

```
import lxml.html as web
from lxml.etree import XPath
….
baseUrl="http://books.toscrape.com/"  # URLs and Columns
bookUrl=baseUrl+"catalogue/category/books/childrens_11/index.html"
pageUrl=baseUrl+"catalogue/category/books/childrens_11/page-" # page-
1,2
columns=['title', 'price', 'stock', 'imageUrl', 'rating', 'url']  #
Columns for CSV header

dataSet=[]   # container for collected data
page=1       # default
totalPages=1 # default
```

As seen in the preceding code, we will be using `lxml`, `XPath`, the `math` library (for dealing with pagination), and `csv` (writing data to a CSV file). We have explored the pages of the site and come up with links such as `baseUrl`, `bookUrl`, and `pageUrl` for pagination. To begin with, for pagination, we have `page` and `totalPages` with the same or default values, and an empty `dataSet` list will be used to place collected data:

```
while page <= totalPages:
    source = web.parse( pageUrl + str(page)+".html")
        .getroot() # read and parse
    if page==1:        # pagination
        perpageArticles =
            source.xpath("//form[@class=\"form-
            horizontal\"]/strong[3]/text()")   # 20
        totalArticles = source.xpath( "//form[@class=
```

```
          \"form-horizontal\"]/strong[1]/text()")
          # 29
      totalPages = math.ceil( int(totalArticles[0])/
          int(perpageArticles[0])) # 1.45
  print("Processing Page "+ str(page) +" from ",
      totalPages)
```

In the preceding code, we started a loop, plus read and parsed `pageUrl` for the root element. We then used expressions and found the count of `totalArticles` (an article is a block of the DOM representing a single book element), `perPageArticles` found on a single page, and the total number of pages or `totalPages`. `math.ceil()` is used to obtain the float value for `totalPages`.

Continuing with the loop, we declare DOM expressions for each book as `articles` using XPath, then use XPath for all the required elements, such as `titlePath`: *book name*, `ratingPath`: *rating*, and so on, targeting particular elements whose data is to be obtained:

```
articles = XPath("//ol[contains(@class,'row')]/li[position()>0]")
titlePath = XPath(".//article[contains(@class,'product_pod')]/h3/a/@
title")
......
ratingPath = XPath(".//article/p[contains(@class,'star-rating')]/@
class")
```

With a handful of XPath expressions, we loop through `articles` found in the page response or `source` to obtain selected entities:

```
for row in articles(source):
        title = titlePath(row)[0].strip()
        link = linkPath(row)[0].replace('../../../',
            baseUrl+'catalogue/').strip()

        price = pricePath(row)[0]
        availability = stockpot(row)[0].strip()
        image = imagePath(row)[0].replace('../../../../',
            baseUrl).strip()
        rating = ratingPath(row)[0].replace(
            'star-rating','').strip()
```

After receiving the entities, cleaning them, and removing unwanted text (using `replace()` and `strip()`), we add them to `dataSet`:

```
        if len(title)>0:      # if title is not missing,add to dataSet
            dataSet.append( [title, price, availability,
                 image, rating, link] )
```

Finally, we write `dataSet` to a CSV file, with the help of the `csv` library:

```python
def writeto_csv(data,filename,columns):
    with open(filename,'w+',newline='',encoding="UTF-8") as
    file:
        writer = csv.DictWriter(file, fieldnames=columns)
        writer.writeheader()
        writer = csv.writer(file)
        for element in data:
            writer.writerows([element])
```

We have built the `writeto_csv()` function to convert data from `dataSet` to a CSV file. It takes three arguments, listed here:

- `data`: The data collection object or `dataSet` in this example case

- `filename`: The name of the CSV file to generate, for example, `books.csv`

- `columns`: Column names for the CSV file for those fields that are to be extracted

After appending all the desired data to `dataSet`, we finally call the `write_to_csv` function as `writeto_csv(dataSet,'books.csv',columns)`. This will create the `books.csv` file, which looks as shown in *Figure 3.15*:

	title	price	stock	imageUrl	rating	url
1	Birdsong: A Story in Pictures	£54.64	In stock	http://books.toscrape.com/media/cache/af/…	Three	http://books.toscrape.com/catalogue/birdsong-a-stor…
2	The Bear and the Piano	£36.89	In stock	http://books.toscrape.com/media/cache/cf/b…	One	http://books.toscrape.com/catalogue/the-bear-and-t…
3	The Secret of Dreadwillow Carse	£56.13	In stock	http://books.toscrape.com/media/cache/c4/…	One	http://books.toscrape.com/catalogue/the-secret-of-d…
4	The White Cat and the Monk: A Retellin…	£58.08	In stock	http://books.toscrape.com/media/cache/26/…	Four	http://books.toscrape.com/catalogue/the-white-cat-a…
5	Little Red	£13.47	In stock	http://books.toscrape.com/media/cache/80/…	Three	http://books.toscrape.com/catalogue/little-red_817/i…
6	Walt Disney's Alice in Wonderland	£12.96	In stock	http://books.toscrape.com/media/cache/28/…	Five	http://books.toscrape.com/catalogue/walt-disneys-al…
7	Twenty Yawns	£22.08	In stock	http://books.toscrape.com/media/cache/2b/…	Two	http://books.toscrape.com/catalogue/twenty-yawns_…
8	Rain Fish	£23.57	In stock	http://books.toscrape.com/media/cache/bb/…	Three	http://books.toscrape.com/catalogue/rain-fish_728/i…
9	Once Was a Time	£18.28	In stock	http://books.toscrape.com/media/cache/97/…	Two	http://books.toscrape.com/catalogue/once-was-a-ti…
10	Luis Paints the World	£53.95	In stock	http://books.toscrape.com/media/cache/85/…	Three	http://books.toscrape.com/catalogue/luis-paints-the-…
11	Nap-a-Roo	£25.08	In stock	http://books.toscrape.com/media/cache/27/…	One	http://books.toscrape.com/catalogue/nap-a-roo_567…
12	The Whale	£35.96	In stock	http://books.toscrape.com/media/cache/6c/…	Four	http://books.toscrape.com/catalogue/the-whale_501…
13	Shrunken Treasures: Literary Classics, …	£52.87	In stock	http://books.toscrape.com/media/cache/4f/…	Three	http://books.toscrape.com/catalogue/shrunken-treas…

Figure 3.15: Output from books.csv, with data from the Childrens category

It's also to be noted that CSS selector queries or XPath queries directly copied from DevTools might be different than we have used in the example in this section. We have tried to use possible short expressions with some additional and in-depth tactics, plus applying additional methods from Python programming.

In this section, we successfully scraped and collected the data from the selected URL and wrote the data to a CSV file. In the next section, we will look at two of the major resources that are available on a website. These resources hold a defined set of instructions, and also the links available across the website.

Parsing robots.txt and sitemap.xml

In this section, we will introduce `robots.txt`- and `sitemap.xml`-related information and follow the instructions or resources available in those two files. We mentioned them in the *Data-finding techniques used in web pages* section of *Chapter 1*. In general, we can dive deep into the pages, or the directory with pages, of websites and find data or manage missing or hidden links using the `robots.txt` and `sitemap.xml` files.

The robots.txt file

The `robots.txt` file, or the **Robots Exclusion Protocol**, is a web-based standard or protocol used by websites to exchange information with automated scripts. `robots.txt` carries instructions regarding site-based links or resources to web robots (**crawlers, spiders, web wanderers, or web bots**), and uses directives such as `Allow`, `Disallow`, `SiteMap`, `Crawl-delay`, and `User-agent` to direct robots' behavior.

We can find `robots.txt` by adding `robots.txt` to the main URL. For example, `robots.txt` for `https://www.python.org` can be accessed with `https://www.python.org/robots.txt`. It's also not obligatory that the `robots.txt` file is available on each website. Following the directives available in the `robots.txt` file is the ethical duty of all developers. Web crawlers should obey the directives mentioned in the file. If any access violation is caused by web crawlers or automated activities, website administration can take the following actions:

- Enhance security mechanisms to restrict any unauthorized access to the website (say, by imposing CAPTCHAs or timeouts, deploying other anti-scraping tools, and more)

- Impose a block on the traced IP address or block the site access of the account that is violating the terms and causing damage to the website or server security

- Take the necessary legal action

Figure 3.16 shows that `User-agent:Nutch` is not allowed to crawl `python.org`:

```
# Directions for robots.  See this URL:
# http://www.robotstxt.org/robotstxt.html
# for a description of the file format.

User-agent: HTTrack
User-agent: puf
User-agent: MSIECrawler
Disallow: /

# The Krugle web crawler (though based on Nutch) is OK.
User-agent: Krugle
Allow: /
Disallow: /~guido/orlijn/
Disallow: /webstats/

# No one should be crawling us with Nutch.
User-agent: Nutch
Disallow: /

# Hide old versions of the documentation and various large sets of files.
User-agent: *
Disallow: /~guido/orlijn/
Disallow: /webstats/
```

Figure 3.16: robots.txt file

Let us try to understand a few directives shown in *Figure 3.16*:

- `Allow`: Permits robots to access the link or directories defined.

- `Disallow`: Restricts robots from accessing the link or directories defined.

- `User-agent`: Agents, robots, or browser platforms mentioned should follow the directives mentioned for those User-agent objects. If `*` (an asterisk) is found mentioned in `User-agent`, it represents all agents. Sometimes, `User-agent` provides an agent name, such as `Nutch`, in *Figure 3.16*.

As mentioned in this section, we obey and follow the directives mentioned in the `robots.txt` file. Often, there might not be simple or easy-to-read content and, during automation, if we wish to test the directive's response on the fly, then parsing `robots.txt` is very helpful.

Parsing robots.txt

Using a parser, we do not need to read the file manually and follow or instruct the steps upon the code found. Also, there might be cases that require iteration, validating `User-agent` for some links, or determining the `crawl-delay` value of the `sitemap.xml` file (the prescribed time that must take place during processing). This information is extremely valuable to automate crawlers for some websites and even decide the level of information or coding tactics that might be required.

As seen in the following code, `urllib` has `robotparser`, which allows developers to read `robots.txt` and validate or verify the necessary actions:

```
import urllib.robotparser
robot = urllib.robotparser.RobotFileParser()
robot.set_url( "https://www.python.org/robots.txt" )
robot.read()
robot
# <urllib.robotparser.RobotFileParser at …….>
```

We can see examples for methods such as `can_fetch()`, which deals with the `Allow` and `Disallow` directives and returns a **Boolean** value of `True` or `False`, respectively. Similarly, if the `sitemap.xml` file exists, then `robot.site_maps()` will return the links:

```
robot.can_fetch('*','https://docs.python.org/3/library/urllib.
robotparser.html')
robot.can_fetch('Nutch','https://docs.python.org/3/library/urllib.
robotparser.html')
robot.site_maps()
```

For more information on `robots.txt` directives, please visit `http://www.robotstxt.org`. For parsing, visit `https://docs.python.org/3/library/urllib.robotparser.html` for more details.

Sitemaps

A sitemap, or `sitemap.xml`, is an XML file that holds the information related to links for a website. Links can be page URLs or to specific media. A sitemap is an easy way to inform search engines about URLs (*added, updated, modified date, removed URL, priority, changefreq, and more*) or URL management on the site. Search engine scripts crawl the links in sitemaps and use the links found for indexing and various purposes, such as **Search Engine Optimization** (**SEO**).

Similar to `robots.txt`, we can find sitemaps for a domain by appending a URL with `sitemap.xml`. Contents inside a sitemap keep changing and sitemaps may not be available for all websites. As seen in *Figure 3.17*, the sitemap of `https://www.schools.com/sitemap.xml`, there are plenty of `<url>` nodes, with a few child nodes:

```xml
▼<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ▼<url>
      <loc>https://www.schools.com/</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>1.0</priority>
   </url>
  ▼<url>
      <loc>https://www.schools.com/online-colleges/south-carolina</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>0.9</priority>
   </url>
  ▼<url>
      <loc>https://www.schools.com/online-colleges/west-virginia</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>0.9</priority>
   </url>
```

Figure 3.17: sitemap.xml

Information in the child nodes, such as `<lastmod>`, notifies us when the last modification took place for the `<loc>` child element. Similarly, `<changefreq>` is the changing duration of the `<loc>` element's contents. We can deal with sitemap content or parse a sitemap as we did in the *Reading XML files* section.

> **Important note**
>
> There are plenty of Python libraries available that read and parse *sitemap.xml* and `robots.txt`. Among those, `https://pypi.org/project/advertools/` is one of the popular ones among the developer community. We also can parse and apply logic in sitemaps using `lxml`. For more details on sitemaps, explore `https://www.sitemaps.org/`.

Summary

In this chapter, we learned about DOM navigation, XPath, and CSS selectors using the page source and DevTools. We also learned about reading and accessing XML and HTML files and defining and using XPath and CSS selector expressions for content extraction.

We also looked at various aspects of content extraction, plus the benefits and restrictions imposed by `robots.txt` and sitemaps. The main objective of the chapter was to demonstrate core features related to nodes, element identification from HTTP responses received, using the `lxml` and `urllib` libraries as required, and dealing with XML and HTML files. Finally, web scraping techniques were deployed using an example and data was collected and written to a CSV file.

In the next chapter, we will learn more about web scraping techniques and about some new Python libraries.

Further reading

- **XPath**:

  - `https://www.w3.org/TR/xpath-31/`

  - `https://www.w3schools.com/xml/xpath_intro.asp`

- **CSS selectors**:

  - `https://www.css3.info/`

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

  - `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors`

- **lxml**: `https://lxml.de/`

- **XML**:

  - `https://www.w3.org/XML/`

  - `https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction`

- **HTML DOM**:

  - `https://html.spec.whatwg.org/multipage/`

  - `https://www.w3schools.com/html/`

  - `https://www.w3schools.com/xml/dom_intro.asp`

- **robots.txt**:

  - `https://developers.google.com/search/docs/crawling-indexing/robots/intro`

  - `https://www.robotstxt.org/robotstxt.html`

- **Sitemaps**:

  - `https://www.sitemaps.org/`

  - `https://www.wordstream.com/blog/ws/2022/11/14/what-is-a-sitemap`

  - `https://developers.google.com/search/docs/crawling-indexing/`
    `sitemaps/overview`

- **DevTools**:

  - `https://developer.chrome.com/docs/devtools/`

  - `https://developer.mozilla.org/en-US/docs/Learn/Common_questions/`
    `Tools_and_setup/What_are_browser_developer_tools`

4

Scraping Using PyQuery, a jQuery-Like Library for Python

In the previous chapters, you learned about the basics of web scraping, the technologies involved, data-finding techniques, and traversing markup documents for data with the help of some **Python** code.

Web scraping is a handful of tasks that involve **reverse engineering** techniques. It entails exploring a website (examining its content-related structure, **DevTools**, paginations, and more), preparing or installing tools (libraries and so on), coding and testing, and finally, collecting data in files, clouds, databases, and many more places. In this chapter, we will be learning about web scraping using the **PyQuery** Python library; this library assists us in the use of concepts such as **XPath**, **CSS selectors**, and parsing markup documents. PyQuery provides a **jQuery**-like ability to *write less, do more*, which is very significant when coding web scrapers.

In particular, this chapter will cover the following topics:

- PyQuery overview
- Exploring PyQuery
- Web scraping using PyQuery

Technical requirements

A web browser (**Google Chrome** or **Mozilla Firefox**) will be required, and we will be using **Jupyter notebooks** for code using **JupyterLab**.

Please refer to the *Creating a virtual environment* section in *Chapter 2* to continue setting up and using the created environment.

The Python libraries that are required for this chapter are as follows:

- `pyquery`
- `urllib`
- `requests`

The code files for this chapter are available online on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter04`

PyQuery overview

PyQuery is a jQuery-like library for Python that facilitates the easy implementation and use of `lxml` and CSS selectors.

As the name suggests, PyQuery enhances query-related procedures (XPath and CSS selector expressions) with short and readable lines of code. Web scraping, as you all are aware, requires parsing and traversing features that reside on top of various types of web documents.

PyQuery provides additional features related to DOM and ElementTree, and uses CSS selectors to perform queries. The purpose of using PyQuery expressions (or queries) is similar to that of XPath or CSS selector-based expressions. PyQuery is almost the same as jQuery for web documents.

The following list contains a basic comparison of expressions that collect the `href` attribute from the `<a class="main">` element:

- PyQuery: `response.find('a.main').attr('href')`
- lxml XPath: `response.xpath(".//a[contains(@class,'main')]/@href")`
- lxml cssselect: `response.cssselect("a.main").get('href')`
- jQuery: `$('a.main').attr('href');`

> **Important note**
>
> For more details on PyQuery and its latest documentation, please visit `https://pyquery.readthedocs.io/en/latest/` and `https://pypi.org/project/pyquery/`. Similarly, for `lxml` (XPath and `cssselect`), visit `https://lxml.de/index.html`.

After our short introduction to PyQuery and its similarities with XPath and CSS selectors, let's go over the basics of jQuery.

Introducing jQuery

jQuery (*write less, do more*; see `https://jquery.com/`) is one of the most popular JavaScript libraries and is lightweight, CSS3-complaint, cross-browser compatible, and quick. It supports plenty of features related to web documents, such as the DOM, HTML, and CSS.

Web-based document traversal, manipulation, event handling, animation, and AJAX are some of the core features jQuery is applied to. PyQuery expressions are similar to jQuery expressions, so anyone with prior jQuery and JavaScript knowledge will find it easy to use PyQuery with Python.

Throughout the chapter, you will be focused on using Python libraries only. Web development today involves JavaScript in relation to dynamism, development and testing, and many other aspects. For a full stack developer or web-related programmer, knowledge of JavaScript, along with AJAX and jQuery, is almost compulsory. Please visit `https://learn.jquery.com/` for more information on jQuery.

You've received a basic introduction to PyQuery and jQuery in this section. The next section will explore PyQuery in depth and show you how to use it to write a crawling script or scrape a website.

Exploring PyQuery

PyQuery addresses **DOM**-based expressions or queries in a quick, easy, and effective manner. In this section, we will install PyQuery and explore a few of its important features so that we are ready to develop a web scraping script.

> **Important note**
>
> You can find plenty of Python libraries that are similar to PyQuery. A few examples are `parsel` (`https://pypi.org/project/parsel/`), `beautifulsoup` (`https://pypi.org/project/beautifulsoup4/`), `selectolax` (`https://pypi.org/project/selectolax/`), and, of course, `lxml` (`https://pypi.org/project/lxml/`).

Installing PyQuery

Please refer to the *Technical requirements* section before proceeding with installing the PyQuery library. With the help of the virtual environment (`secondEd`) you created in *Chapter 2*, install or update PyQuery using `pip`:

```
C:\HOWScraping2E>secondEd\Scripts\activate

(secondEd) C:\HOWScraping2E>pip install pyquery
Requirement already satisfied: pyquery in c:\howscraping2e\seconded\lib\site-packages (2.0.0)
Requirement already satisfied: lxml>=2.1 in c:\howscraping2e\seconded\lib\site-packages (from pyquery) (4.9.2)
Requirement already satisfied: cssselect>=1.2.0 in c:\howscraping2e\seconded\lib\site-packages (from pyquery) (1.2.0)

(secondEd) C:\HOWScraping2E>_
```

Figure 4.1: Installing PyQuery

As shown in *Figure 4.1*, installing PyQuery also installs or updates the `lxml` and `cssselect` libraries. To verify the installation, let's import PyQuery using the following code:

```
import pyquery
print(dir(pyquery)) # Explore pyquery
# ['PyQuery', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__', '__spec__',
'cssselectpatch', 'openers', 'pyquery', 'text']
```

In the preceding code, we imported PyQuery and checked the resources of the library using `dir()`. There are some functions and classes; here, we will be mostly using the `PyQuery` class:

```
print(dir(pyquery.PyQuery)) #Class PyQuery provides DOM related
features
['Fn', '__add__',….,'add_class', 'after', 'append', 'appendTo',
'append_to', 'attr', 'base_url', 'before', 'children', 'clear',
'clone', 'closest', 'contents', 'copy', 'count', 'css', 'each',
'empty', 'encoding', 'end', 'eq', 'extend', 'filter', 'find', 'fn',
'hasClass', 'has_class', 'height', 'hide', 'html', 'index', 'insert',
'insertAfter', 'insertBefore', 'insert_after', 'insert_before', 'is_',
'items', 'length', 'make_links_absolute',…, 'parent', 'parents',
'pop', 'prepend', 'prependTo', 'prepend_to', 'prev', 'prevAll',
'prev_all', 'remove', 'removeAttr', 'removeClass', 'remove_attr',
'remove_class', 'remove_namespaces', 'replaceAll', 'replaceWith',
'replace_all', 'replace_with', 'reverse', 'root', 'serialize',
'serializeArray', 'serializeDict', …., 'show', 'siblings', 'size',
'sort', 'text', 'toggleClass', 'toggle_class', 'val', 'width', 'wrap',
'wrapAll', 'wrap_all', 'xhtml_to_html']
```

In the preceding code, we explored the `PyQuery` class and found many of the methods and attributes that somewhat resemble DOM concepts such as `attr`, `children`, `eq`, `root`, `parent`, `next`, and `hasClass`.

Important note

Code and outputs are copied from the Jupyter notebook. A single-line comment with # has been used to present short outputs and comments. If the output is spread across multiple lines, then a new line with # has been used. Also, multi-line outputs will be shortened as required.

In this section, you installed PyQuery and learned about a few of its features. In the next section, you will explore some examples of using PyQuery.

Loading a web URL

We will now practically explore PyQuery's features with the help of some code examples. Various types of web documents are available. We will be targeting HTML and XML content using the `requests` library as follows:

```
from pyquery import PyQuery as pq
import requests
response = requests.get
    ("https://webscraper.io/test-sites/")
type(response)      # requests.models.Response
source = pq(response.content) # convert HTTP Response to pyquery
type(source) # pyquery.pyquery.PyQuery
```

As you can see in the preceding code, we have used the `requests` library for HTTP communication. In this example, we are processing the `https://webscraper.io/test-sites/` URL, and we are checking the type of `response` and `source`. We need to convert the page source or HTTP response into a PyQuery object before processing the DOM using `pyquery`. `source = pq(response. content)` converts the `response.content` page source from the web into a PyQuery object.

With the PyQuery object, `source`, in hand, we are now ready to move on to the next section, where we will explore more features of PyQuery.

Element traversing, attributes, and pseudo-classes

With the `source` PyQuery object from the previous section, we will explore some of the attributes and methods that PyQuery provides. You are also advised to go through the page source in the browser, or to identify or inspect elements using DevTools, before implementing the code.

These examples are handy for analyzing the elements and their contents in the page source:

```
source.find('title')            # [<title>]
source.find('title').text()   # Web Scraper Test Sites

source.find('a')                 # source('a')
# [<a>,….,<a.menuitm>, <a.menuitm>, <a.menuitm>, <a.menuitm.dropdown-
toggle>, <a>,………, <a>, <a.btn-menu1.install-extension>, <a.btn-menu2>,
<a>, <a>, <a>, <a>,…….,<a>, <a>, <a>]
```

The preceding code uses the `find()` method to look for the element, tag, or CSS selector provided to it and identifies those elements. `find(a)` returns the list of `<a>` elements along with a few class names, such as `menuitm` and `btn-menu`. After finding the elements, there are various methods, such as `text()`, `attr()`, and `html()`, that return the desired content or capture the `key:value` using attribute names or expressions. In the preceding case (`source.find('title').text()`), `text()` returns the contents found in `<title>` tags.

Here, we are trying to find information in the `content` attribute of the `<meta>` tag that has a `name` attribute with values of `keywords` and `description` respectively:

```
source.find('meta[name="keywords"]').attr('content')
# web scraping, Web Scraper, Chrome extension, Crawling,
  Cross platform scraper
source.find('meta[name="description"]').attr('content')
# You need to train your web scraper? We have created
  simple test sites that allow you to try all corner cases
  and proof test your scraper. Try it now.
```

The following code demonstrates the use of the `html()` function. It returns the HTML content of the provided expression:

```
source.find('ul.dropdown-menu').html()
# \n\t…\t<li>\n\t…\t<a href="/documentation">Documentation</a>\n\t….\
t</li>\n\t….\t<li>\n\t…\t<a href="/tutorials">Video Tutorials</a>\
n\t…\t</li>\n\t…\t<li>\n\t…\t<a href="/how-to-videos">How to</a>\n\t…\
t</li>\n\t…\t<li>\n\t…\t<a href="/test-sites">Test Sites</a>\n\t…\t</
li>\n\t…\t<li>\n\t…\t<a href="https://forum.webscraper.io/" target="_
blank" rel="noopener">Forum</a>\n\t…\t</li>\n\t\t\t\t\t\t\t
```

This technique is quite handy for identifying inner or child elements along with their attributes and contents without using the DevTools element selector or for verifying the output from DevTools.

> **Important note**
>
> The `class` and `id` CSS attributes are represented with `.` and `#` respectively. For example, `<a class="main" id="mainLink">` will be expressed as `a.main` and `a#mainLink`.

PyQuery also contains pseudo-classes or *pseudo-elements* that are used with expressions. These elements begin with the `:` character, such as `:eq()`, `:last`, and `:first`. They are like predefined constants that implement some shortcut. Most of the time, pseudo-elements are used with identified **iterable** (collection) types of elements. We will cover iteration in more detail in the *Iterating using PyQuery* section.

The code in the following block shows the use of PyQuery's pseudo-elements. These elements are similar to shortcuts, and most of the time they erase the need for iteration:

```
source.find('a:eq(0)').text()      # Toggle navigation
source.find('a.menuitm:first').text() # Web Scraper
source.find('a.menuitm:last').text()   # Learn
source.find('a.menuitm:eq(1)').attr('href')
# /cloud-scraper
```

Let's discuss a few pseudo-elements that were used in the preceding code:

- `:eq()`: Accepts an index number (and in Python, indexes start at 0). It evaluates the matched expression and collects the text associated with it. This can also be used as `source.find('a').eq(0).text()`. There are also a few elements that are similar to `:eq()`:

  - `:lt()`: Accepts an index number and only returns values *less than* the provided index number, for example, `source.find('a:lt(2)')`

  - `:gt()`: Accepts an index number and only returns values *greater than* the provided index number, for example, `source.find('a:gt(0)')`

- `:first`: Returns the *first occurrence* of an element for the matched expression and is the same as `:eq(0)`.

- `:last`: Returns the *last occurrence* of an element for the matched expression.

There are a few more *pseudo-elements*, though they are a bit different from the ones listed in the previous code block:

```
source.find(':input')
# [<button.navbar-toggle.pull-right.collapsed>]


source.find(':header')
# [<h1>, <h2.site-heading>, <h2.site-heading>, <h2.site-heading>,
<h2.site-heading>, <h2.site-heading>, <h2.site-heading>, <h2.site-
heading>]


source.find(':empty')
# [<meta>, <meta>, <meta>, <meta>, <link>, <meta>, <link>, <link>,
<link>, <script>, <iframe>, <span.icon-bar.top-bar>, <span.icon-bar.
middle-bar>, <span.icon-bar.bottom-bar>, <img>, <a>, …. <hr.test-
site-divider>, <img>, <hr.test-site-divider>, <img>, <hr.test-site-
divider>, <img>, <hr.test-site-divider>, <img…. <div.clearfix>, <div.
push>, <br>, <img>, <img>]


source.find('meta:empty')
# [<meta>, <meta>, <meta>, <meta>, <meta>]


source.find(':empty:even')
# [<meta>, <meta>, <link>, <link>, <link>, <iframe>, <span.icon-bar.
middle-bar>, <img>, <div.crta>, <div.crta>, <hr.test-site-divider>,
<hr.test-site-divider>, <hr.test-site-divider>, <hr.test-site-
divider>, <hr.test-site-divider>, <hr.test-site-divider>, <hr.test-
site-divider>, <div.clearfix>, <br>, <img>]


source.find(':header:odd')
# [<h2.site-heading>, <h2.site-heading>, <h2.site-heading>, <h2.site-
heading>]
source.find('meta:empty:odd')    # [<meta> ,<meta>]
```

Let's explore the *pseudo-elements* that were used in the preceding code:

- `:input`: Returns all the `<form>` input elements or elements that receive some action, such as `<button>`. `source(':input')` is equivalent to `source.find(':input')`. There is also the `<input>` element in HTML, which can be expressed as `source.find('input')`.

- `:header`: Returns the HTML header elements (`<h1>`, `<h2>`…. `<h6>`) found.

- `:empty`: Returns all the elements that don't have any child elements. This is helpful to check before proceeding with the iteration of inner elements.

- `:even`: Returns all the elements that are evenly indexed. They can be used with other *pseudo-elements* too. Here are two examples: `source.find('meta:empty:even')` and `source('meta;empty:even')`.

- `:odd`: Similar to `:even`, but returns elements that are oddly indexed.

Apart from index-related tasks, and finding elements, *pseudo-elements* can also be used to search for elements with the provided text:

```
source.find('a:contains("Web")')
# [<a.menuitm>, <a>, <a>, <a>, <a>]

source.find('a:contains("Web"):last').text()
# 'Web Scraper'
source.find('a:contains("Web"):last').attr('href') # '#'
```

As shown in the preceding code, `:contains()` accepts a string parameter, which is case sensitive and returns the matching elements.

PyQuery also has some verification functions. These functions are quite effective in many circumstances (such as updating an existing crawler) for searching for elements with attributes and confirming the attributes' values.

Figure 4.2 shows a block of the page source from `https://webscraper.io/test-sites` with `<p class="copyright">`:

Figure 4.2: Page source view

The following code tries to confirm that the `<p>` element, as shown in *Figure 4.2*, has the `class` attribute with the value `copyright`, and `<a>` inside `<p.copyright>` has the attribute `id` with the value `link`:

```
source.find('div.row div p').is_('.copyright') # True
source.find('div.row div p').hasClass('copyright') # True
source.find('div.row div p.copyright a').is_('#link') # False
```

In the preceding code, we can see the use of a couple of function-type elements that are explained here:

- `is_()`: Accepts a selector as an argument and returns `True` or `False`. You can try using a class name or class-like attribute, for example, *href* in `<a>`.

- `hasClass()`: Checks whether an expression provided has a *class* attribute (you pass the class name as an argument) and returns `True` or `False` accordingly.

We have now identified and learned about a few core features of PyQuery, such as dealing with individual elements and groups of similar elements. In the next section, we will explore iteration, which will help us prepare for scraping content.

Iterating using PyQuery

Iterating, iteration, or looping is a task that involves repetition. During web scraping planning and studying the format of your web content, you will come across the concept of looping many times.

In the *Element traversing, attributes, and pseudo-classes* section, you encountered many outputs where, in Python lists, for example, there might be a collection of `<a>`, `<img>`, and `<meta>` tags with targeted content. If we manage to find the pattern or expression for all such collection-related elements and iterate through them, it will be easy to manage and process the content:

```
[item.attr('href') for item in
    source.find('a.menuitm').items()]
# ['/', '/cloud-scraper', '/pricing', '#section3']
```

The `items()` method is used for iteration with PyQuery, as shown in the preceding code. In `source.find('a.menuitm').items()]`, `<a>` with the `class="menuitm"` attribute has been traced with `items`. Iterating through `items()` for each `a.menuitm href` attribute reveals four URLs or paths:

```
[item.attr('href') for item in
    source.find('a.menuitm,a[class*="btn-menu"]').items()]
# ['/', '/cloud-scraper', '/pricing', '#section3', 'https://chrome.
google.com/webstore/detail/web- scraper/…….?hl=en ', 'https://cloud.
webscraper.io/']
```

In the preceding code, `source.find('a.menuitm,a[class*="btn-menu"]').items()` collects elements matching the expressions `a.menuitm` and `a[class*="btn-menu"]`. Iterating through them found `items()` for the `href` attribute; we have a total of six results.

We encountered many `<meta>` tags in the *Element traversing, attributes, and pseudo-classes* section. Let's collect the values of the common `<meta>` tag's name and `content` attributes, if they are present:

```
meta=[]
count=0
for item in source.find('meta').items():
    if item.attr('name') and item.attr('content'):
        meta.append({'index':count,'name':item.attr('name')
            , 'content':item.attr('content')})
        count+=1
```

The preceding code iterates through `source.find('meta').items()`. For each `<meta>` tag or `item`, a logical step has been applied to check whether both the `name` and `content` attributes exist. If they exist, they are appended as Python dictionary or `dict()` objects to the `meta` output list. The final `meta` output list looks as follows:

```
[ {'index': 2,  'name': 'keywords',  'content': 'web scraping,Web
Scraper,Chrome extension,Crawling,Cross platform scraper'},
{'index': 3,  'name': 'description',  'content': 'You need to train
your web scraper? We have created simple test sites that allow you to
try all corner cases and proof test your scraper. Try it now.'},
{'index': 4,  'name': 'viewport',  'content': 'width=device-width,
initial-scale=1.0'}]
```

The output in the preceding code provides information such as the following:

- There are a total of five `<meta>` tags (indexes 0 to 4)

- Out of the five `<meta>` tags, the first and second tags either do not have both `name` and `content` attributes or don't have one of them

In this section, we've learned about deploying PyQuery features to obtain specific results and tested techniques such as iteration and pseudo-classes. In the next section, we will learn to how implement all the concepts from earlier chapters and the preceding sections to scrape data from the web.

Web scraping using PyQuery

This section is loaded with examples that explain how to code a web scraping script. We will be using the PyQuery and `requests` libraries and dealing with HTML and XML content from the web. We will write the data collected in these examples to CSV or JSON files.

Example 1 – scraping book details

In this example, we will be scraping books listed on `http://books.toscrape.com` in the **Childrens** category (`http://books.toscrape.com/catalogue/category/books/childrens_11/index.html`). This example is similar to another provided in *Chapter 3*, where the `lxml` library was used. The code for this example is available on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter04/example_1.ipynb`.

As you can see in *Figure 4.3*, the **Childrens** category contains **29** results (a single page shows only **20** results):

Figure 4.3: Childrens category page with 29 results

> **Important note**
>
> You are recommended to browse the pages for the selected category using DevTools and find the differences in URLs or content by using the *element inspector* tools and moving across the page, or exploring the *page source* and identifying the targeted elements.

After studying the basics of processing pages and URLs, we can move on to the script. The following code defines the required libraries, URLs, default pagination value, and an empty `dataSet` to collect the data:

```
from pyquery import PyQuery as pq
import requests, math
siteUrl="http://books.toscrape.com/"
baseUrl=siteUrl+"catalogue/category/books/childrens_11/index.html"
pageUrl=siteUrl+"catalogue/category/books/childrens_11/page-"
```

```
dataSet=[]
page=1
totalPages=1
```

pageURL holds the pattern for the pagination link that is found while browsing the **Childrens** pages (for example, `../page-1.html` and `../page-2.html` each represent a page's URL).

With pagination loops, `pageUrl` is processed using `requests` and, finally, a PyQuery object is generated called `source`:

```
response = requests.get(pageUrl+str(page)+".html")
source = pq(response.content)
if page==1:
    pageValues=[value.text() for value in
        source.find('form.form-horizontal strong').items()]
    if len(pageValues)>0:
        pageValues = list(map(int,pageValues)) # converts
        to int [29 1 20]
        totalPages = math.ceil(pageValues[0]/pageValues[2])
print(f"Page {page} from Total {totalPages}")
```

With `source`, we can now find the exact block of elements that we want and iterate on them for the desired values. `pageValues` results in three elements, `['29','1','20']` in this case. Using `map(int,pageValues)`, those string `pageValues` results are converted to `int` (integer) values and `totalPages` is calculated.

Using the `find()`, `attr()`, and `text()` PyQuery methods, the required values are traced and finally cleaned and exploded using `strip()` and `split()`:

```
books = source.find('article.product_pod') #book
for book in books.items():  #iterate
    image = book.find('.image_container a img').attr('src')
    rating = book.find('p.star-rating').attr('class')
        .split()
    title = book.find('h3:first a').attr('title').strip()
    url = book.find('h3:first a').attr('href')
    price = book.find('p.price_color').text().strip()
    stock = book.find('p.availability').attr('class')
        .split()
```

As shown in the following code, after collecting the desired values and cleaning and processing them, they are finally added to `dataSet` as a Python dictionary:

```
dataSet.append({
    'name':title,
    'price':price.replace('£',''),
    'stock':stock[0],
    'rating':rating[1],
    'image':image.replace('../../../../',
        'http://books.toscrape.com/catalogue/'),
    'url':url.replace('../../../',
        'http://books.toscrape.com/catalogue/')
})
```

After the addition of the data to `dataSet`, the following code creates a `childrens_books.json` file with data from `dataSet` using `json.dump()` from the `json` library:

```
import json
with open("childrens_books.json", "w") as file:
    json.dump(dataSet, file, indent=4, sort_keys=False)
```

For proper, indented JSON content, any code editor or text editor supporting the JSON file format can be used.

> **Important note**
>
> For more information on the `json` library, please visit `https://docs.python.org/3/library/json.html`.

In this example, we learned how to scrape data from an HTML document that exists on multiple pages, how to handle the URLs of multiple pages, and how to maintain the `items()` loop. Some basic data cleaning activities were also carried out, and finally, we created a JSON file from our collection object. We will deal with different content formats in the next example.

Example 2 – sitemap to CSV

In this example, we will parse a sitemap or XML file, traverse through it, and generate a CSV file. The code for this example is available on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter04/example_2.ipynb`. The URL we are loading is `https://www.schools.com/sitemap.xml`, the contents of which are shown in *Figure 4.4*:

```
▼<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ▼<url>
      <loc>https://www.schools.com/</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>1.0</priority>
    </url>
  ▼<url>
      <loc>https://www.schools.com/online-colleges/south-carolina</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>0.9</priority>
    </url>
  ▼<url>
      <loc>https://www.schools.com/online-colleges/west-virginia</loc>
      <lastmod>2021-02-03</lastmod>
      <changefreq>daily</changefreq>
      <priority>0.9</priority>
    </url>
```

Figure 4.4: sitemap.xml

The <urlset> node contains <url> child nodes, which contain children such as <loc>, <lastmod>, <changefreq>, and <priority>. With distinct nodes in place, we will move through each child <url> and collect their children's information in a list and, finally, write it to a CSV file:

```
from pyquery import PyQuery as pq
import requests, csv
url = "https://www.schools.com/sitemap.xml"
columns=['loc','lastmod','changefreq','priority']
#Columns for CSV header
```

The required libraries, the url attribute, and the desired column names (columns) for the CSV format have been declared in the code.

As shown in the following code, a PyQuery object is supplied with an additional argument of parser='html'. This allows PyQuery to interpret the nodes and child nodes encountered in the source page as HTML documents:

```
xmlFile = requests.get(url).content  #loading the url
urlXML = pq(xmlFile, parser='html')  #parser
print("Child-Length: ",urlXML.children().__len__())
# Child-Length: 530
```

`urlXML` refers to the `<urlset>` node and `urlXML.children()` refers to the `<url>` child nodes of `<urlset>`, which total 530.

Python provides the very useful `range(start,stop,step)` function. This provides us with the difference between `start` and `stop` (0 and 530 in our code). As there are a total of 530 targeted `<url>` elements, a loop has been initiated by providing the index number as `eq(loop)`:

```
dataSet=[]
loops = range(0, urlXML.children().__len__())
# range(start, stop)
for loop in loops:
    child = urlXML.children().eq(loop)  #0, 1, 2, 3.,.529
    dataSet.append([
        child.find('loc').text(),
        child.find('lastmod').text(),
        child.find('changefreq').text(),
        child.find('priority').text()
    ])
```

With the respective `<url>` instances identified, their `<loc>`, `<lastmod>`, `<changefreq>`, and `<priority>` values are appended to `dataSet`.

The following code defines a function that is also used in *Chapter 3*. It accepts arguments such as `data`, or in our case, the collection object, `dataSet`; the `filename` to be created; and the `column` list for the first row of the CSV file (`columns`) as follows:

```
def writeto_csv(data,filename,columns):
    with open(filename,'w+',newline='',encoding="UTF-8") as
        file:
        writer = csv.DictWriter(file, fieldnames=columns)
        writer.writeheader()
        writer = csv.writer(file)
        for element in data:
            writer.writerows([element])
```

A call to the `writeto_csv(dataSet,'schoolXML.csv',columns)` function with the right arguments creates the `schoolXML.csv` file, as shown in *Figure 4.5*:

	loc	lastmod	changefreq	priority
1	https://www.schools.c…	2021-02-03	daily	1.0
2	https://www.schools.c…	2021-02-03	daily	0.9
3	https://www.schools.c…	2021-02-03	daily	0.9
4	https://www.schools.c…	2021-02-03	daily	0.9
5	https://www.schools.c…	2021-02-03	daily	0.9
6	https://www.schools.c…	2021-02-03	daily	0.9
7	https://www.schools.c…	2021-02-03	daily	0.9
8	https://www.schools.c…	2021-02-03	daily	0.9
9	https://www.schools.c…	2021-02-03	daily	0.9
10	https://www.schools.c…	2021-02-03	daily	0.9

Figure 4.5: schoolXML.csv

In this example, we have collected data from XML nodes and created a CSV file. This proves to be very useful when representing and visualizing XML parameters such as `lastmod` (last modified date) and `priority` for a certain `loc` (URL or location). In the next example, we will incorporate a few more concepts that are helpful during web scraping.

Example 3 – scraping quotes with author details

In this example, we are going to collect data or quotes from `http://quotes.toscrape.com/tag/books` under the **books** tag. The code for this example can be found on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter04/example_3.ipynb`. This example deals with a few scenarios that are a bit different from those in *Example 1 – scraping book details* and *Example 2 – sitemap to CSV*:

- We need to handle pagination, which is different from *Example 1 – scraping book details*.

- We need to go deeper into the individual pages or links in the listings to collect a few additional details.

- There are also multiple quotes available from each author. This example deals with looping through the same page repeatedly, which might consume memory and time.

- In addition, we will create separate files for **authors** and **quotes**.

The main page (`http://quotes.toscrape.com/tag/books`) is shown in *Figure 4.6*:

Quotes to Scrape

Viewing tag: books

> "The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."
> by **Jane Austen** (about)
> Tags: aliteracy books classic humor

> "Good friends, good books, and a sleepy conscience: this is the ideal life."
> by **Mark Twain** (about)
> Tags: books contentment friends friendship life

Top Ten tags

love
inspirational
life
humor
books
reading

Figure 4.6: Base page to be scraped

Data related to authors (from the **about** link), **tags**, author names, and quote statements are the main targets.

The following code shows the declaration of basic requirements, with the format of pagination URLs to be `http://quotes.toscrape.com/tag/books/page/1/` and `http://quotes.toscrape.com/tag/books/page/2/`:

```
from pyquery import PyQuery as pq
import requests, csv
url = "http://quotes.toscrape.com/tag/books/page/"
columns=['id', 'author', 'quote', 'tags', 'quote_length',
    'born_date', 'born_location','author_url']
authorSet=dict()
dataSet=list()
page=1
nextPage=True
uid=0
```

Two different types of collection objects (dictionaries and lists) are also defined (`authorSet` and `dataSet`), along with some default values carrying variables.

In the following code, `while (nextPage)` initiates the loop, and the value of `nextPage` is now `True`:

```
while (nextPage):
    response = requests.get(url+str(page))
    source = pq(response.content)
    if source.find('ul.pager li.next a:contains("Next")'):
```

```
        nextPage=True
    else:
        nextPage=False
```

While browsing through the pages, we did find the URL pattern, but there is no mention of how many total quotes are available. This restricts us to calculating values such as the final page number or the total number of pages to maintain the loop. To do this, we have used the `nextPage` Boolean variable, which is always `True` if the browsed page contains the text `Next`. If the browsed page does not have the text `Next`, then we assume that the browsed page is the final page and maintain the loop accordingly by updating `nextPage` to `False`. Maintaining the loop is always a priority and needs to be checked while browsing each page.

The following code iterates on the class attribute value `quote` and collects the required data with some basic data-cleaning activities:

```
for quotes in source.find('.quote').items():
    quote = quotes.find('[itemprop="text"]').text().strip()
    author = quotes.find('[itemprop="author"]').text()
        .strip()
    tags = quotes.find('[itemprop="keywords"]')
        .attr('content').strip()
    authorUrl = quotes.find('a[href*="/author/"]')
        .attr('href')
```

In each iteration over `quote`, `authorUrl` is collected. During the preliminary study, we found that there are a few different quotes from the same author on the page.

As we have to browse each `authorUrl` encountered, some system resources are wasted when loading duplicate URLs, and even in the final output, duplicate values will be available. To deal with this situation, we must verify whether `authorUrl` has been accessed or not.

The following code inserts the formatted value of `author` (author name) or `authorKey` in `authorSet`:

```
if authorUrl:
    authorKey = author.replace('.','_').replace(' ','_')
        .strip()

if authorUrl and authorKey not in authorSet.keys():
    authorUrl = "http://quotes.toscrape.com"+authorUrl
    source_author = pq(requests.get(authorUrl).content)
    bornDate = source_author.find('.author-born-date')
        .text()
    bornLocation = source_author
        .find('.author-born-location').text()
            .replace('in','').strip()
```

```
        authorSet[authorKey]={
            'name':author,
            'url':authorUrl,
            'date':bornDate,
            'location':bornLocation
        }
    else:
        print(f"Author ({authorKey}) details already found!")
```

Python dictionaries do not allow duplicate keys. We are implementing the same logical step to restrict possible duplication.

Important note

Detecting and removing duplicate values from any dataset is a key activity in web scraping. Developers often use some unique values for data (randomly generated or using a pattern), store them in files (log or temp files) or database tables, and compare them each time with the next record to pass or fail the final insertion steps.

In the preceding code, `if authorUrl and authorKey not in authorSet.keys():` verifies that `authorUrl` is there but `authorKey` has not already been processed and added to `authorSet`. This condition enables us to load a unique `authorUrl` string and access the author detail page, as shown in *Figure 4.7*:

Figure 4.7: Author page for Mark Twain

> **Important note**
>
> Targeting only required values and unique values and cleaning and formatting the data before inserting it into its final destination (files, database tables, or cloud storage) will help the **quality analysis** (**QA**) and **data analysis** processes a lot. These processes, also known as data verification, are among the basic steps to take with a dataset before carrying out **machine learning** (**ML**) and data visualization, and **exploratory data analysis** (**EDA**) helps us to find out how much data verification is required.
>
> These activities are often forwarded to the QA team to find issues (duplicate, missing, and unformatted data) in the dataset or for preprocessing (finding duplicates, cleaning, and formatting) the dataset.

After browsing the author detail page and collecting data on individual quotes, the desired, cleaned values are appended to `dataSet`:

```
dataSet.append ([
    uid,
    author,
    quote,
    tags.replace(',','|'),
    len(quote),
    authorSet[authorKey]['date'],
    authorSet[authorKey]['location'],
    authorSet[authorKey]['url']
])
page+=1  # continue with pagination value

# After looping through all pages
writeto_csv(dataSet,'quotes.csv',columns)
```

After collecting all the data and progressing through all the pages, the `dataSet` content is finally written to a CSV file, as shown in *Figure 4.8*, using the `writeto_csv()` function, as described in *Example 2 – sitemap to CSV*:

id	author	quote	tags	quote…	born_date	born_location	author_url
1	Jane Austen	"The person, be it gentlema…	aliteracy|boo…	104	December 16, 1775	Steventon …	http://quotes.toscrape.com/author/Jane-Austen
2	Mark Twain	"Good friends, good books,…	books|conten…	76	November 30, 1835	Florida, Mis…	http://quotes.toscrape.com/author/Mark-Twain
3	Jorge Luis Borges	"I have always imagined th…	books|library	65	August 24, 1899	Buenos Air…	http://quotes.toscrape.com/author/Jorge-Luis-Borges
4	C.S. Lewis	"You can never get a cup of…	books|inspira…	79	November 29, 1898	Belfast, Irel…	http://quotes.toscrape.com/author/C-S-Lewis
5	Haruki Murakami	"If you only read the books …	books|thought	110	January 12, 1949	Kyoto, Japan	http://quotes.toscrape.com/author/Haruki-Murakami
6	Ernest Hemingway	"There is no friend as loyal …	books|friends…	40	July 21, 1899	Oak Park, Il…	http://quotes.toscrape.com/author/Ernest-Hemingway
7	J.D. Salinger	"What really knocks me out…	authors|book…	241	January 01, 1919	Manhattan,…	http://quotes.toscrape.com/author/J-D-Salinger
8	Mark Twain	"'Classic' - a book which pe…	books|classic…	56	November 30, 1835	Florida, Mis…	http://quotes.toscrape.com/author/Mark-Twain
9	Jane Austen	"I declare after all there is n…	books|library|…	203	December 16, 1775	Steventon …	http://quotes.toscrape.com/author/Jane-Austen

Figure 4.8: Output (quotes.csv)

As defined in the following code, we are also exporting the author details from `authorSet` to a JSON file (`quotes_author.json`) using the `json` library:

```python
import json
# Writing Dictionary authorSet to JSON file using json.dump()
with open("quotes_author.json", "w") as file:
    json.dump(authorSet, file, indent=4, sort_keys=False)
```

The final output of the JSON file will look as shown in *Figure 4.9*:

```
▼ root:
  ▶ Jane_Austen:
  ▶ Mark_Twain:
  ▶ Jorge_Luis_Borges:
  ▶ C_S__Lewis:
  ▶ Haruki_Murakami:
  ▶ Ernest_Hemingway:
  ▼ J_D__Salinger:
      name: "J.D. Salinger"
      url: "http://quotes.toscrape.com/author/J-D-Salinger"
      date: "January 01, 1919"
      location: "Manhattan, New York, The United States"
  ▶ Madeleine_L'Engle:
  ▼ George_R_R__Martin:
      name: "George R.R. Martin"
      url: "http://quotes.toscrape.com/author/George-R-R-Martin"
      date: "September 20, 1948"
      location: "Bayonne, New Jersey, The United States"
```

Figure 4.9: JSON data (quotes_author.json)

In this section, we used some scraping techniques to extract the desired content from a website. Content identification on the web and a preliminary study before scraping is compulsory and is based on the structure of the website. Libraries such as PyQuery provide the necessary tools and techniques to scrape effectively and efficiently.

Summary

In this chapter, we explored and learned about PyQuery, applying various scraping techniques with examples. Short, simple code that is easy to deal with is always in demand, and PyQuery provides and assists with this. PyQuery supports CSS selectors and its applicability across various markup documents is one of its major advantages.

In the next chapter, we will learn more about web scraping techniques and some new Python libraries.

Further reading

- PyQuery, a jQuery-like library for Python: `https://pyquery.readthedocs.io/en/latest/`

- jQuery: `https://jquery.com/`

- CSS:

  - `https://www.w3.org/Style/CSS/Overview.en.html`

  - `https://developer.mozilla.org/en-US/docs/Web/CSS`

- CSS selectors:

  - `https://www.w3schools.com/css/css_selectors.asp`

  - `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors`

- JSON: `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON`

- Sitemaps: `https://sitemaps.org/`

5

Scraping the Web with Scrapy and Beautiful Soup

In previous chapters, we learned about web scraping-related technologies, data-finding techniques, and using various Python libraries to scrape data from the web.

In this chapter, we will explore and learn practically about two popular Python libraries, Scrapy and Beautiful Soup. Scrapy is a web crawling framework for Python and provides a project-oriented scope for web scraping. Beautiful Soup, on the other hand, deals with document or content parsing. Parsing a document is normally done to effectively traverse and extract content. Apart from this, both libraries are heavily loaded with DOM-related features.

In particular, we will learn about the following topics in this chapter:

- Web parsing using Python
- Web scraping using Beautiful Soup
- Web scraping using Scrapy
- Deploying a web crawler

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required and we will be using Python notebooks for the code using *JupyterLab*.

Please refer to the *Setting things up* and *Creating a virtual environment* sections in *Chapter 2* to continue setting up and using the environment created.

The Python libraries that are required for this chapter are as follows:

- `lxml`
- `urllib`

- requests

- html5lib

- beautifulsoup4

- scrapy

The code files for this chapter are available online on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter05`

Web parsing using Python

In earlier chapters (in both the explanations and code examples), we learned that web scraping is a procedure for extracting data from websites, as per our requirements and choice. Data collection can be smooth and error-free from a coding perspective with the use of some Python libraries, but still, identifying content and traversing through elements (individual or nested) are required, at a minimum, to carry out the task.

To ensure high-quality data is collected, the content on the web must be complete and error-free. We use CSS or XPath-based expressions in the DOM structure. If the DOM's structure is somehow imperfect or it contains bugs, such as incomplete tags, missing closing tags, or spelling errors in tags, then the code expressions and query paths that are deployed will not be directed to the original nodes or elements of the DOM. This will lead to the extraction of incomplete or unnecessary content, which might then require extra tasks, such as data cleaning or preprocessing, or even rejecting the data as it does not comply with the schema or data verification rules.

Web parsing, or more precisely web document (such as HTML or XML) parsing, refers to the activity of breaking down, exposing, or identifying components such as elements, nodes, or HTML tags in content. Web parsing is done to structure web content and overcome the issue of unstructured data elements in web content. Parsing is important because it makes it easier to traverse, search, and collect information from the desired elements. Using effective parsers (tools used to parse) on web content makes the scraping task more efficient and trouble-free.

While dealing with `lxml` in *Chapter 3* and `pyquery` in *Chapter 4*, we used the default parser provided by the libraries. In *Chapter 3*, we also used a special parser to parse `robots.txt`.

Important note

`lxml` is still used widely in Python because of its various DOM-based features. It is the default parser and has memory-efficient features and collaborates with other libraries. For more information on `lxml`, visit `https://lxml.de/index.html`.

In Python programming, the task of web-based document parsing is mostly carried out by Beautiful Soup, also known as the `bs4` Python library. In the next section, we will explore this topic further.

Introducing Beautiful Soup

Beautiful Soup is the Python parsing library for dealing with HTML and XML documents. It generates a parsed tree similar to that of the Python `lxml` library (using attributes and classes such as `lxml.etree` and `ElementTree`), which is further used to traverse, search, and identify elements to extract data and scrape the web.

Beautiful Soup provides the same parsing-related features that are available using the `lxml` and `html5lib` libraries. Large, simple, and easy-to-use methods and properties are available to deal with DOM-related activities in Beautiful Soup.

A few distinguishing features of Beautiful Soup over other Python libraries are listed here:

- It can parse documents with broken, incomplete, misspelled, or missing tags.
- Unlike other parsers, it allows handling duplicate and multi-valued attributes.
- Specific selected portions or sections of the content can also be parsed, saving memory and time.
- Document-based encoding is handled automatically. Encoding details can also be provided to the Beautiful Soup constructor.

For more detailed information about Beautiful Soup, please visit the official documentation at `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`. Now that we have been introduced to Beautiful Soup in this section, we will install and further explore it in the upcoming sections.

Installing Beautiful Soup

Please refer to the *Technical requirements* section in this chapter, and the `https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-beautiful-soup` link to install Beautiful Soup.

As seen in *Figure 5.1*, `pip install beautifulsoup4` installs or updates the `bs4` library and its dependent libraries, such as `soupsieve`:

```
(secondEd) C:\HOWScraping2E>pip install beautifulsoup4
Requirement already satisfied: beautifulsoup4 in c:\howscraping2e\seconded\lib\site-packages
 (4.11.1)
Requirement already satisfied: soupsieve>1.2 in c:\howscraping2e\seconded\lib\site-packages
(from beautifulsoup4) (2.3.2.post1)

(secondEd) C:\HOWScraping2E>cd Chapter05

(secondEd) C:\HOWScraping2E\Chapter05>jupyter-lab
```

Figure 5.1: Installing Beautiful Soup

Let's verify the installation and version of bs4 (the complete code is available on GitHub: https:// github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter05/bs4_setup.ipynb):

```
import bs4
bs4.__version__         # 4.11.1
dir(bs4)
# ['BeautifulSoup', 'BeautifulStoneSoup', 'CData', 'Comment',
'Counter', 'DEFAULT_OUTPUT_ENCODING', 'Declaration', 'Doctype',…,
'SoupStrainer', 'StopParsing', 'Stylesheet', 'Tag',…, '__version__',
'_s', '_soup', 'builder', 'builder_registry', 'dammit', 'element',
'formatter', 'os', 're', 'sys', 'traceback', 'warnings']
```

The preceding code imports bs4 (Beautiful Soup), prints its version information, which is 4.11.1, and explores bs4. We can see in the output that there are a few important classes available, such as BeautifulSoup and SoupStrainer.

In this section, we have installed Beautiful Soup and verified its installation along with its version information. In the next section, we will explore various features of Beautiful Soup.

Exploring Beautiful Soup

We will now explore a few select, important features and classes in Beautiful Soup, also known as the bs4 Python library. Activities such as parsing web documents and using Beautiful Soup for traversing, searching, and iterating will be covered in this section.

In carrying out parsing and using the features of bs4, we will be using the BeautifulSoup and SoupStrainer classes to target certain sections or elements of web documents:

```
from bs4 import BeautifulSoup as BSoup
from bs4 import SoupStrainer
```

As seen in the preceding code, the BeautifulSoup and SoupStrainer classes are imported from bs4. In the next section, we will demonstrate parsing using bs4.

Parsing

As discussed in previous sections, parsing is an activity that rectifies errors in content. Using a parser is always recommended to obtain similar results from web content across platforms and systems. Also, we need to plan for or be ready to accept that the project will take longer than usual due to parsing when using BeautifulSoup.

HTML content or code blocks will be passed to BeautifulSoup, also known as the BSoup constructor, along with the selected parser or a parser similar to BSoup (that can be a string or HTML parser). If no parser is specified, the system will use the default HTML parser, which is usually lxml.

Normally, two types of parsers are supported by `BeautifulSoup`:

- Type of markup: `html`, `xml`, and `html5`

- Name of parser library: `lxml`, `html5lib`, and `html.parser` (`lxml` and `html5lib` must be installed to be used; `html.parser` is a built-in HTML parser)

> **Important note**
>
> `lxml` is recommended as the best parser to be used with `BeautifulSoup`, because of its memory and speed. `html5lib` is usually the second preference, but the parsed output can also be used to determine the preference of the parser to be used based on the content or DOM requirements.

It's also to be noted that the parsed tree (output) or parsed content generated will differ based on the parser used. To demonstrate parsing, here, a `markup` variable with an incomplete error code has been declared:

```
markup = "<a></p></a></p>"
```

The following code uses the default parser for `markup`:

```
BSoup(markup)    # <html><body><a></a></body></html>
```

The `html.parser` parser has been used and only `<a>` has been parsed; the `</p>` HTML tag has not been returned from the `markup` string variable:

```
BSoup(markup,"html.parser")    # <a></a>
```

The `lxml` parser and the default parser outputs look similar:

```
BSoup(markup,"lxml")    # <html><body><a></a></body></html>
```

The `html5lib` parser seems to provide more detailed information in comparison to other parsers, such as `lxml` and `html.parser`:

```
BSoup(markup,"html5lib")
# <html><head></head><body><a><p></p></a><p></p></body>
   </html>
```

All incomplete tags have been listed in the output, and the complete HTML structure (`<html>`, `<head>`, `<body>`) has also been maintained.

markup is HTML code, but parsing it with `xml` generates XML output with a single node, `<a/>`:

```
BSoup(markup,"xml")
# <?xml version="1.0" encoding="utf-8"?>
# <a/>
```

The `BeautifulSoup` constructor also accepts a third argument, `parse_only`, which is used with an object from `SoupStrainer`. The `SoupStrainer` class targets a part of the documents. This option is quite handy when the document is quite large and we want to shift focus to a particular section, saving processing time and memory. For more information on `SoupStrainer`, please visit `https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html#soupstrainer`.

The following code uses an HTML string, to be parsed using `lxml` and `html.parser`, but targeting only `<a>` from the HTML string. Both lines of the following code result in the output `<a></a>`:

```
BSoup("<p><a></img></a></p>", 'lxml',
    parse_only=SoupStrainer("a"))
BSoup("<p><a></img></a></p>",'html.parser',
    parse_only=SoupStrainer("a"))
```

As seen in the following code, the only difference is the parser used, which in this case is `html5lib`:

```
BSoup("<p><a></img></a></p>", 'html5lib',
    parse_only=SoupStrainer("a"))
```

Using `html5lib` results in the output `<html><head></head><body><p><a></a></p></body></html>`, and the warning **"UserWarning: You provided a value for parse_only, but the html5lib tree builder doesn't support parse_only. The entire document will be parsed."**. Therefore, `html5lib` is not effective when used with `SoupStrainer`.

With this basic introduction to parsing and using various parsers, we will now move on to deploy `bs4` for searching, traversing, and iterating through the parsed tree, looking for elements, in the upcoming section.

Searching, traversing, and iterating

We can find plenty of methods and properties to traverse and search elements when using Beautiful Soup. An important feature of Beautiful Soup is its easy-to-use and large collection of DOM-based features.

To demonstrate and learn about the advanced features of Beautiful Soup along with some examples, we will be using the HTML content seen in *Figure 5.2* (the file is also available on GitHub at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter05/bs4_traverse.ipynb`):

```html
<html>
    <head><title>The Dormouse's story</title></head>
    <body>

        <p class="title"><b>The Dormouse's story</b></p>

        <p class="story">Once upon a time there were three little sisters; and their names were
            <a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
            <a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
            <a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
            and they lived at the bottom of a well.
        </p>
        <p class="story">...</p>

        <h1>Secret agents</h1>

        <ul>
            <li data-id="10784">Jason Walters, 003: Found dead in "A View to a Kill".</li>
            <li data-id="97865">Alex Trevelyan, 006: Agent turned terrorist leader; James' nemesis in "Goldeneye".</li>
            <li data-id="45732">James Bond, 007: The main man; shaken but not stirred.</li>
        </ul>
    </body>
</html>
```

Figure 5.2: HTML source

The following code imports the bs4 and re (Python regex, or regular expression) libraries:

```python
from bs4 import BeautifulSoup as BSoup
import re            # regex
html = open("testHTML.html","r").read()  # html file
soup = BSoup(html) # default parser
```

HTML content, as seen in *Figure 5.2*, is loaded with the designated path and converted into a soup object from the BeautifulSoup library using the default parser.

With the bs4 soup object, now we can further process the DOM or HTML content. The HTML content we are dealing with can be singular or nested. Individual elements can be accessed in the same way as accessing attributes, as seen in the following code:

```python
soup.a   # <a class="sister" href="http://example.com/elsie"
            id="link1">Elsie</a>
soup.h1 # <h1>Secret agents</h1>
```

soup.a is similar to soup.find('a'); it finds the first <a> element by default and displays the complete HTML (text and attributes) of that element.

Plenty of elements have various, or at least a single, attributes. Element attributes are important resources as they provide identification of the content.

In the following code, some of the major functions and attributes of the soup object are used:

```python
soup.a.attrs
# {'href': 'http://example.com/elsie', 'class': ['sister'],
   'id': 'link1'}
```

```
soup.a.has_attr('class')      # True
soup.a.has_attr('name')       # False
soup.a.get('class')           # ['sister']
soup.a.get('href')      # 'http://example.com/elsie'
```

A brief explanation of the functions and attributes used in the preceding code is listed here:

- `attrs`: Lists all the attributes of any element found

- `has_attr()`: Accepts the attribute name as an argument to check whether it exists; returns a Boolean (true/false) answer upon searching the query

- `get()`: Accepts the attribute name as an argument and returns the value of the provided attribute

A string value or content from any element can be retrieved using `text` and `string` attributes, as well as the `get_text()` function, as used in the following code:

```
soup.a.text           # Elsie
soup.a.get_text()     # Elsie
soup.a.string         # Elsie
```

Here, we explored individual elements with their attributes. In the following section, we will be searching for elements.

find()

The `soup` object also supports the `find()` method, which returns a single result as per the arguments provided. If there's no result, then `find()` returns `None`.

In the following code, various forms of `find()` have been displayed, identifying or selecting `<a>` by default, with the output `<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>`:

```
soup.find('a')
soup.find('a',string="Elsie")
soup.find('a',{'id':"link1"})
soup.find('a',"sister")
soup.find("a",attrs={'class':'sister'},text="Elsie")
```

Let's try to understand the various forms of `find()`, as listed here:

- `find('a')`: Finds the first `<a>` available.

- `find('a',string="Elsie")`: Finds the first `<a>` that has the `string` value `Elsie`.

- `find('a',{'id':"link1"})`: Finds the first `<a>` that has the `id` attribute with the value `link1`.

- `find('a',"sister")`: Finds the default `<a>` that has a class with the value `sister`. This is the same as `find('a',attrs={'class':"sister"})` and `find('a',{'class':"sister"})`.

- `find("a",attrs={'class':'sister'},text="Elsie")`: Finds the first `<a>` that has the text value `Elsie` and also possesses an attribute named `class` with the value `sister`.

The examples coming up show some common and distinguishing uses of `find()`.

In the following code, the `<ul>` element is searched by default, which results in its content, including its child `<li>` elements, being returned:

```
soup.find('ul')# <ul>
<li data-id="10784">Jason Walters, 003: Found dead in "A
    View to a Kill".</li>
<li data-id="97865">Alex …."Goldeneye".</li>
<li data-id="45732">James Bond, 007: The main man; shaken
    but not stirred.</li>
</ul>
```

As seen in the following code, `find()` has been used for concatenation. The code searches `<ul>` and then finds `<li>` (the first child of `<ul>` that is found):

```
soup.find('ul').find('li')
# <li data-id="10784">Jason Walters, 003: Found dead in "A
  View to a Kill".</li>
```

The following code uses an extra filter option while finding `<li>` by mentioning its `data-id` attribute and revealing its content using the `get_text()` method:

```
soup.find('ul').find('li',attrs={'data-id':'97865'})
    .get_text()
# 'Alex Trevelyan, 006: Agent turned ….. nemesis in
  "Goldeneye".'
```

We dealt with single-element searching in this section. Now, we will learn how to find all matching elements in the next section.

find_all()

The `soup` object also contains the `find_all()` method, which returns one or more results, or an empty `list()` object if the query doesn't match.

The following code block uses the `find_all()` method along with the main parameters:

```
soup.find_all('a') # soup("a")
soup.find_all('a',attrs={'id':re.compile(r"link")})
soup.find_all('a',attrs={'href':re.compile(r".*ie$")})
soup.find_all("a",text=re.compile(r".*ie$"))
```

The preceding code, using the soup `find_all()` method, results in the following output:

```
[<a class="sister" href="http://example.com/elsie"
    id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie"
    id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie"
    id="link3">Tillie</a>]
```

Let's try to understand the various aspects of the code using `find_all()`:

- `find_all('a')`: Finds all `<a>` elements. This code is similar to `soup("a")`, `find_all('a','sister')`, and `find_all('a',{'class':'sister'})`.

- `find_all('a',attrs={'id':re.compile(r"link")})`: Here, the `re` Python library has been used, to match the `<a>` element with the pattern found in the value of the `id` attribute, which contains the text `link`. Similarly, a regex pattern can be matched on values of the `href` attribute that end with the characters `ie` with `find_all('a',attrs={'href':re.compile(r".*ie$")})`.

- `find_all("a",text=re.compile(r".*ie$"))`: Similar to regex matched on attributes (`id` or `href`), a pattern can also be formed among text arguments that end with the characters `ie`.

The upcoming examples in this section show some extra features using `find_all()`.

The following code returns a list of the `<a>` and `<title>` elements:

```
soup.find_all(['a','title'])
# [<title>The Dormouse's story</title>,
<a class="sister" href="http://example.com/elsie"
    id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie"
    id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie"
    id="link3">Tillie</a>]
```

The following code finds all <p> elements that have the class attribute with the title or story value:

```
soup.find_all("p",attrs={'class':["title","story"]})
# [<p class="title"><b>The Dormouse's story</b></p>,
<p class="story">Once upon a time there were three little
sisters; …
<a class="sister" href="http://example.com/elsie"
    id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie"
    id="link2">Lacie</a> …
<a class="sister" href="http://example.com/tillie"
    id="link3">Tillie</a>;
…. well. </p>,
<p class="story">...</p>]
```

find_all() also supports limiting the results by using the limit argument, as shown here:

```
soup.find_all('a',limit=2)
# [<a class="sister" href="http://example.com/elsie"
  id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie"
    id="link2">Lacie</a>]
```

As seen in the preceding example, only two <a> elements are returned.

> **Important note**
>
> *CSS selectors* can also be used with BeautifulSoup objects. There are two methods that allow the use of CSS selectors with bs4: select(), which is similar to find_all(), and select_one(), which is similar to find(). For more information, please visit https://www.crummy.com/software/BeautifulSoup/bs4/doc/#css-selectors.

We looked at searching for elements in this section, which can also be used for the purpose of iteration (please refer to the *Iteration* section). In the next section, we will cover a combination of searching and traversing along DOM content.

next_element and previous_element

Traversing along a DOM is quite effective when moving back and forth between an element. It is achieved using next_element and previous_element. next_elements and previous_elements can also be used to return multiple elements. Before we start to understand traversing, understanding the concept of the parsed tree, or DOM tree, is very important. The DOM tree normally reveals the HTML code structure, parent-child tag combinations, and nested or chained tags.

The following code shows the use of `next_element`. While traversing along a DOM (normally from top to bottom), various elements can be found with `next_element`:

```
soup.find('p','story').next_element
# 'Once upon a time there were three little sisters; and
  their names were\n  '

soup.find('p','story').next_element.next_element
# <a class="sister" href="http://example.com/elsie"
  id="link1">Elsie</a>
```

Let us understand the details of the preceding code:

- `find('p','story').next_element`: Searches for a `<p>` element for the `story` value in `class`, and points to the next element. Here, the next element after `<p class="story">` is the text itself, as seen in the preceding code output.

- `find('p','story').next_element.next_element`: Points to the `<a>` child element with the text `Elsie`.

`previous_element` is similar to `next_element`, but the direction of traversing is reversed.

The following code shows the use of `previous_element`, which can be extended or chained:

```
soup.find('b').previous_element
# <p class="title"><b>The Dormouse's story</b></p>

soup.find('li',{'data-id':'45732'}).previous_element
    .previous_element
# 'Alex Trevelyan, 006: Agent turned terrorist leader;
  James\'... in "Goldeneye".'
```

Regarding the preceding code, let us understand a few details:

- `find('b').previous_element`: There's only one `<b>` element in the HTML content inside `<p>`, so it returns the element at `previous_element` or `<p class="title">`

- `find('li',{'data-id':'45732'}).previous_element.previous_element`: With `<li data-id="45732">`, we find that the previous element is `<li data-id="97865">`, hence we get the text

The `next_element` and `previous_element` attributes can be chained, as seen in the preceding code.

We have explored a lot regarding searching and traversing so far in this section. The next section demonstrates performing iteration.

Iteration

We have already come across the concept of iteration in previous sections. For example, `find_all()`, `next_elements`, and `previous_elements` result in one or more elements, where a loop can be used.

The following code iterates on the child elements of the `<ul>` elements, which are `<li>` using `find_all()`, and uses the `name` attribute to show the element name:

```
for li in soup.ul.find_all('li'):
    text = li.get_text().strip()
    print(f"{li.name} : {li.get('data-id')} :
        ({len(text)}) - {text}")
```

Similarly, the `get_text()` function returns the `text` string of an element, and the `get('data-id')` function returns the values of the `data-id` attribute. Printing all those return values with their text lengths, the output looks as follows:

```
li : 10784 : (53) - Jason Walters, 003: Found dead in "A
    View to a Kill".
li : 97865 : (82) - Alex Trevelyan, 006: …; James' nemesis
    in "Goldeneye".
li : 45732 : (54) - James Bond, 007: The main man; shaken
    but not stirred.
```

Various alternate mechanisms exist to perform iteration and the extraction process.

In this and previous sections, we explored the main concepts related to `BeautifulSoup`. We will scrape content using `BeautifulSoup` in the upcoming section.

> **Important note**
>
> `BeautifulSoup` has various other methods related to the *find* activity, such as `find_all_next`, `find_next`, and `find_previous_siblings`, to deal with *parent-child* and *traversing*-related instances. There have been updates to the names and supporting arguments of methods in `bs4`. For the complete and updated documentation, please visit `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`.

Web scraping using Beautiful Soup

In this section, we will build and execute a web crawler using Beautiful Soup. To set things up, we have chosen to scrape quotes from `http://quotes.toscrape.com`. Specifically, we will be scraping from the page `http://quotes.toscrape.com/tag/inspirational`, as seen in *Figure 5.3*:

Figure 5.3: Category "inspirational"

The example we are dealing with is similar to *Example 3 – scraping quotes with author details* in *Chapter 4*. Only the links and compositions have changed, by carrying out a few additional logical steps. The code for the example can be found on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter05/bs4_scraping.ipynb`.

The following code declares the paginated link as `url`, and `columns` contains a column header for the CSV file to be generated:

```
url = "http://quotes.toscrape.com/tag/inspirational/page/"
columns=['id','author','quote','tags','quote_length',
     'born_date','born_location','author_url']
```

Inside the paginated loop, a response from `url` has been collected and provided to `BSoup` to be parsed with the default parser:

```
response = requests.get(url+str(page))
source = BSoup(response.content)
    if source.find('ul','pager').find('li','next'):
        txtNext = source.find('ul','pager')
            .find('li','next').find('a').get_text()
```

`txtNext` holds the value for the next page.

As seen in the following code, the nextPage pagination variable is controlled based on the "Next" string if it exists for txtNext:

```
if txtNext and
re.findall(r".*(Next).*",txtNext)[0]=="Next":
    nextPage=True
```

Multiple variables, such as quote, author, and tags, are created with the content available from iterating <div> with the quote class value:

```
for quotes in source.find_all('div','quote'):
    quote = quotes.find(attrs={'itemprop':'text'})
        .get_text().strip()
    author = quotes.find(attrs={'class':'author'})
        .get_text().strip()
    tags = quotes.find(attrs={'itemprop':'keywords'})
        .get('content').strip()
    authorUrl = quotes.find(href=re.compile(r"/author/"))
        .get('href')
```

Similarly to the quote-based information, a unique authorUrl string has been identified and its content is parsed to collect author-related information, such as bornDate and bornLocation:

```
if authorUrl and authorKey not in authorSet.keys():
    authorUrl = "http://quotes.toscrape.com"+authorUrl
    source_author = BSoup(requests.get(authorUrl).content)
    bornDate = source_author.find(attrs=
        {'class':'author-born-date'}).get_text().strip()
    bornLocation = source_author.find(attrs=
        {'class':'author-born-location'}).get_text()
            .replace('in','').strip()
```

Finally, author-related information is collected in authorSet, and dataSet holds quote and author details:

```
    authorSet[authorKey]={
        'name':author,
        'url':authorUrl,
        'date':bornDate,
        'location':bornLocation
    }
    dataSet.append([
        uid,
        author,
        quote,
```

```
        tags.replace(',','|'),
        len(quote),
        authorSet[authorKey]['date'],
        authorSet[authorKey]['location'],
        authorSet[authorKey]['url']
    ])
```

The collected set of information (author, quote, tags, and a few more) is finally written into JSON (`quotes_author_inspirational.json`) and CSV (`quotes_inspirational.csv`) files, as seen in *Figure 5.4*:

id	author	quote	tags	quote_len...	born_date	born_location	author_url
1	Albert Einstein	"There are only two w...	inspirationa...	131	March 14, 1879	Ulm, Germany	http://quotes.toscrape.com/author/Alb...
2	Marilyn Monroe	"Imperfection is beaut...	be-yourself|...	111	June 01, 1926	The United States	http://quotes.toscrape.com/author/Ma...
3	Thomas A. Edison	"I have not failed. I've ...	edison|failu...	65	February 11, 1847	Milan, Ohio, The Un...	http://quotes.toscrape.com/author/Th...
4	Marilyn Monroe	"This life is what you ...	friends|hear...	1084	June 01, 1926	The United States	http://quotes.toscrape.com/author/Ma...
5	Elie Wiesel	"The opposite of love ...	activism|ap...	224	September 30, 1928	Sighet, Romania	http://quotes.toscrape.com/author/Eli...
6	J.K. Rowling	"To the well-organized...	death|inspir...	68	July 31, 1965	Yate, South Glouce...	http://quotes.toscrape.com/author/J-K...
7	George Eliot	"It is never too late to ...	inspirational	54	November 22, 1819	South Farm, Arbury...	http://quotes.toscrape.com/author/Ge...
8	C.S. Lewis	"You can never get a ...	books|inspir...	79	November 29, 1898	Belfast, Ireland	http://quotes.toscrape.com/author/C-...
9	Martin Luther King Jr.	"Only in the darkness ...	hope|inspir...	45	January 15, 1929	Atlanta, Georgia, T...	http://quotes.toscrape.com/author/Ma...
10	Helen Keller	"When one door of ha...	inspirational	153	June 27, 1880	Tuscumbia, Alabam...	http://quotes.toscrape.com/author/Hel...
11	George Bernard S...	"Life isn't about findin...	inspirationa...	69	July 26, 1856	Dubl, Ireland	http://quotes.toscrape.com/author/Ge...
12	John Lennon	"You may say I'm a dr...	beatles|con...	117	October 09, 1940	Liverpool, England, ...	http://quotes.toscrape.com/author/Joh...
13	Dr. Seuss	"A person's a person, ...	inspirational	43	March 02, 1904	Sprgfield, MA, The ...	http://quotes.toscrape.com/author/Dr-...

Figure 5.4: CSV data from the inspirational category

> **Important note**
>
> `BeautifulSoup` also supports CSS selectors, and there is a short, small crawler available as an example in the GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter05/bs4_scraping_css_select.ipynb`.

In this section, we have developed Beautiful Soup-based crawlers and extracted data into files. In the next section, we will learn about the Python `scrapy` library and develop a crawler using it.
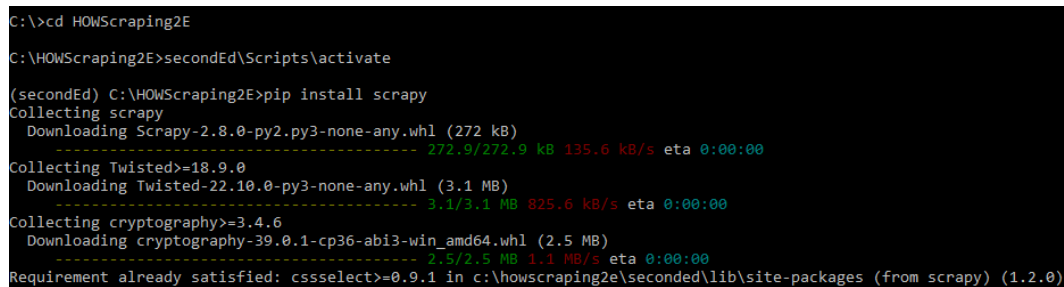
Web scraping using Scrapy

We have learned about, explored, and used different Python libraries for web scraping in the current and previous chapters. Scrapy is one of the few open source web crawling frameworks written in Python that allows dynamic adaptation, a project-based scope, and modular extensibility for web scraping tasks.

As per Scrapy's official website, `https://scrapy.org/`, it is *simple, fast, collaborative, and yet extensible*. Scrapy was previously maintained by **Scrapinghub**, but now it is maintained by **Zyte** (`https://www.zyte.com/`) and some other contributors.

Listed here are a few important features that make Scrapy popular and make it stand out among the Python web crawling frameworks:

- Built-in support for *parsing, traversing, XPath, CSS selectors, and regex*

- Handles HTTP requests and responses using built-in libraries

- Modular structure and components allow developers to focus on a specific task and manage coding collaboratively

- Provides a **Command-Line Interface** (**CLI**) to deal with the project, data exporting, managing the database, and much more

- Plenty of middleware and extensions are available, which allows for the easy processing of *cookies, sessions, authentication, robots.txt, project log, usage statistics, emails, and much more*

To begin trying out some scraping tasks using Scrapy, let's install it in the preset environment using `pip`, as seen in *Figure 5.5*, or follow the instructions at `https://docs.scrapy.org/en/latest/intro/install.html`:


```
C:\>cd HOWScraping2E

C:\HOWScraping2E>secondEd\Scripts\activate

(secondEd) C:\HOWScraping2E>pip install scrapy
Collecting scrapy
  Downloading Scrapy-2.8.0-py2.py3-none-any.whl (272 kB)
     ---------------------------------------- 272.9/272.9 kB 135.6 kB/s eta 0:00:00
Collecting Twisted>=18.9.0
  Downloading Twisted-22.10.0-py3-none-any.whl (3.1 MB)
     ---------------------------------------- 3.1/3.1 MB 825.6 kB/s eta 0:00:00
Collecting cryptography>=3.4.6
  Downloading cryptography-39.0.1-cp36-abi3-win_amd64.whl (2.5 MB)
     ---------------------------------------- 2.5/2.5 MB 1.1 MB/s eta 0:00:00
Requirement already satisfied: cssselect>=0.9.1 in c:\howscraping2e\seconded\lib\site-packages (from scrapy) (1.2.0)
```

Figure 5.5: Scrapy installation

After this basic introduction to and successful installation of Scrapy in the system, next, we are going to set up a project.

Setting up a project

Scrapy, as mentioned in the preceding section, provides a command-line tool, `scrapy`. This tool is loaded with plenty of commands that help with setting up the project and data extraction-related activities. The code is available on GitHub: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter05/bookScrapy`.

Developing a crawler with `scrapy` is detailed in the following steps:

1. Scrape the `http://books.toscrape.com` site for all book details across all pages.

2. For each book, extract fields such as *title*, *price*, *rating*, *stock*, *url*, and *image*.

3. After collecting the URL or book link in *step 2*, navigate to it.

4. Parse fields such as *category*, *upc*, and *no_review* (number of reviews).

5. Repeat *steps 2*, *3*, and *4* for all books, and parse through all the pages available.

6. Export all the collected data to JSON and CSV files.

To create a project, we can use the `scrapy startproject projectname projectfolder` or `scrapy startproject books bookScrapy` commands. These commands will create a project named `books` inside the `bookScrapy` folder, as seen in *Figure 5.6*:

```
(secondEd) C:\HOWScraping2E>cd Chapter05

(secondEd) C:\HOWScraping2E\Chapter05>scrapy startproject books bookScrapy
New Scrapy project 'books', using template directory 'C:\HOWScraping2E\secondEd\Lib\site-packages\scrapy\templates\project'
, created in:
    C:\HOWScraping2E\Chapter05\bookScrapy

You can start your first spider with:
    cd bookScrapy
    scrapy genspider example example.com

(secondEd) C:\HOWScraping2E\Chapter05>
```

Figure 5.6: scrapy startproject

Inside the `bookScrapy` folder, `scrapy` also creates a `books` subfolder and a `scrapy.cfg` configuration file. The `scrapy.cfg` file contains default project-related settings. These settings can be updated as per the crawler's needs. Please refer to `https://docs.scrapy.org/en/latest/topics/commands.html` for more details.

After the successful creation of the project folder (`bookScrapy`), as seen in *Figure 5.7*, we need to create a spider script:

```
(secondEd) C:\HOWScraping2E\Chapter05>cd bookScrapy

(secondEd) C:\HOWScraping2E\Chapter05\bookScrapy>scrapy genspider booklist books.toscrape.com
Created spider 'booklist' using template 'basic' in module:
  books.spiders.booklist

(secondEd) C:\HOWScraping2E\Chapter05\bookScrapy>
```

Figure 5.7: scrapy genspider

The `scrapy genspider booklist books.toscrape.com` command creates a spider named `booklist.py` inside the `bookScrapy\books\spiders` subfolder, with `books.toscrape.com` set for the `allowed_domains` argument:

```python
import scrapy
class BooklistSpider(scrapy.Spider):
    name = "booklist"
    allowed_domains = ["books.toscrape.com"]
    start_urls = ["http://books.toscrape.com/"]
    def parse(self, response):
```

As seen in the preceding code, the newly created `booklist.py` spider contains the default code. To develop a crawler, we have to implement the necessary code and logic in the `booklist.py` file using `parse()`. The `bookScrapy\books` folder also contains a few additional files: `items.py`, `middlewares.py`, `pipelines.py`, and `settings.py`, along with the `spiders` folder.

> **Important note**
>
> Spider is a Python class file that contains code used for scraping and data collection logic. Multiple spider classes can exist targeting specific scraping activities. The spider class is used in the crawling process. Commands such as `scrapy list` and `scrapy list spider` list the spiders of a project. Please visit `https://docs.scrapy.org/en/latest/intro/tutorial.html?highlight=Spider#our-first-spider` for more information.

Each of these files has its own control and implementation over the final crawler. The following are brief explanations of these files:

- `items.py`: An item is like a Python dictionary that holds keys and values (column and value). For more details, visit `https://docs.scrapy.org/en/latest/topics/items.html`.

- `pipelines.py`: After collecting data, scraped items are sent to the pipeline to perform additional actions, such as cleaning and dropping. For more details, please visit `https://docs.scrapy.org/en/latest/topics/item-pipeline.html`.

- `settings.py`: Project-related settings can be controlled and added. For more details, please visit `https://docs.scrapy.org/en/latest/topics/settings.html`.

- `middlewares.py`: We can specify some hooks or extensions that can perform additional tasks with spiders (processing inputs and output). Visit `https://docs.scrapy.org/en/latest/topics/spider-middleware.html` for more details.

> **Important note**
>
> Dealing with or updating the files we just mentioned is not compulsory. These files are there to implement a full-fledged crawling project, by allowing users to use the in-depth facilities and variations provided by Scrapy.

We now have the project and default spider files ready to begin coding or further processing. In the next section, we will be dealing with `scrapy.Item` using the `items.py` file.

Creating an item

An item is normally understood as a column name or key for a dictionary object that is used to collect values to implement in the spider. The `items.py` file contains a default class that implements `scrapy.Item`. This class (`BooksItem`) is generated during the processing of the `scrapy startproject` command.

The following code block explores the `BooksItem` item class with a `scrapy.Field()` type of object associated with it:

```
import scrapy
class BooksItem(scrapy.Item):     # define the fields for
your item here like:
    title = scrapy.Field()
    no_review = scrapy.Field()
    …..
    url = scrapy.Field()
    image = scrapy.Field()
```

As per our crawler plan, `items.py` will contain the default object of `scrapy.Field()`. These fields will get their values from the spider or `spiders\booklist.py`, as we'll discover in the next section.

Implementing the spider

With the planning and collection of items ready, we will implement the spider to use `items.py` and further proceed with the code implementation. The code is available on GitHub: https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter05/bookScrapy/books/spiders/booklist.py.

Crawling-related logic and code is provided by the `parse()` function, which is present in `booklist.py`, as follows:

```
def parse(self, response):
    listings =
        response.xpath("//article[@class='product_pod']")
```

```
        #root
    nextPage = response.xpath
        ("//ul[@class='pager']/li[@class='next']/a/@href")
    for listing in listings:
```

The preceding code identifies `listings` from `http://books.toscrape.com` and will iterate using XPath. A link for the new page, `nextPage`, has also been parsed. Both CSS selectors and XPath expressions are available to use with `response` (HTTP response to `start_urls`).

As seen in the following code, the Python `BooksItem` class from `items.py`, which has the `column` object, has been created and identified or planned fields are being parsed and added to `column`:

```
column = BooksItem() # from books.items import BooksItem
column['title'] = listing.xpath
    (".//h3/a/@title[normalize-space()]").extract_first()
column['url'] =
    listing.xpath(".//h3/a/@href").extract_first().strip()
….
yield response.follow
    ('http://books.toscrape.com/catalogue/'+column['url'],
        callback=self.parse_page, meta={'column':column})
```

To parse the detail listing page using the link made available by `column['url']`, Scrapy provides the `follow()` function, which links to the parse function to deal with (`parse_page`) as a callback, and the items collected are added to the `BooksItem` class's `column` object.

The following code implements loop-based logic and the `parse()` call function each time a link to the next page is available:

```
if nextPage:
    nextPage =
        'http://books.toscrape.com/catalogue/'+nextPage
    yield scrapy.Request(nextPage, callback=self.parse)
```

This spider (`booklist`) created can be executed using the `scrapy crawl booklist` command. It lists lots of logs, such as elapsed time and dates.

With the spider ready to process or crawl, we will export data in the next section.

Exporting data

The `scrapy` command has made exporting data to files very easy. We can use these commands:

- `scrapy crawl booklist -o bookRecords.csv`
- `scrapy crawl booklist -o bookRecords.json`

These commands execute the `booklist` spider and export scraped data items as `column` or `BooksItem` to CSV and JSON. These files will be available in the project folder or inside `bookScrapy`.

We have created a Scrapy-based crawler. Now, we will try to deploy this crawler on **Zyte** (`https://app.zyte.com/`), and use its live features in the next section.

Deploying a web crawler

We have successfully implemented a crawler and extracted and exported data to external files using Scrapy (with the help of the `scrapy` CLI tool). This process has been done on a local machine or **Personal Computer** (**PC**). Deploying a crawler online or on a server is the only option for most developers. The deployed crawler benefits from multiple features of the server (such as having access anytime and anywhere, speed, and ample storage), as well as its dynamic nature.

We can choose any cloud platform, web hosting server, or internet-based service to upload our code and execute it. Most of these services are not 100% free; we have to pay a certain amount for the desired configuration and services.

Scrapy, from the beginning, has been famous for its architecture. There were and are still multiple web-based platforms that allow users to run their Scrapy-based projects. One of these is **Scrapinghub** (now **Zyte**). **Zyte Scrapy Cloud** (`https://www.zyte.com/scrapy-cloud/`) provides many free additional infrastructures for Scrapy and other projects.

> **Important note**
>
> Web services and platforms such as `https://apify.com/apify/scrapy-executor` and `https://scrapeops.io/` can be found on the web. Some provide limited resources, while others provide additional services (proxies, additional RAM and storage, schedulers, exporting options, databases, and more).

Let's follow a few steps to move ahead with deploying our Scrapy-based spider on Zyte:

1. Register for an account at `https://app.zyte.com/`.

2. After registration and login, the system will route us to a default dashboard. The dashboard will list multiple projects that exist and is loaded with plenty of other features.

3. We need to create a project first, so click on **Create Project** and provide a project name, for example, `BooksToScrape`.

4. As the project is created, Zyte will route you to the **Deploy** section, or you can choose the option from the menu. Here, we have to specify how we are going to load the project to app. zyte.com. Multiple options are available, as seen in *Figure 5.8*:

< Deploy Options   Deploy

Command Line
Deploy code from your own computer.

SCRAPY	CUSTOM IMAGE	

```
$ pip install shub
$ shub login
API key: 3516a87e569c4bb39c44de261bbd5598
$ shub deploy 638593
```

Don't have a project yet? **Clone a public repository**

Figure 5.8: Code & Deploy – Scrapy option

5. As per the instructions found under **SCRAPY** from *step 4*, we will install shub using pip install shub, login, and deploy with the code shown in *Figure 5.9*. A successful installation and login will create an API key, provide the deployment ID (638593), and add a few files, such as scrapinghub.yml and setup.py, to the project folder:

```
(secondEd) C:\HOWScraping2E\Chapter05\bookScrapy>shub deploy 638593
C:\HOWScraping2E\secondEd\Lib\site-packages\_distutils_hack\__init__.py:33: UserWarning: Setuptools is replacing
distutils.
  warnings.warn("Setuptools is replacing distutils.")
Packing version 1.0
Deploying to Scrapy Cloud project "638593"
Run your spiders at: https://app.scrapinghub.com/p/638593/
{"status": "ok", "project": 638593, "version": "1.0", "spiders": 1}
```

Figure 5.9: shub (install, deploy)

6. *Step 5* will load the deployed code or project as jobs that will be available to execute, run, schedule, and many more options under **Jobs Dashboard**, as seen in *Figure 5.10*:

Figure 5.10: Listing jobs and Spider links

7. *Step 6* shows that **1 spider** exists. Click on the spider, which will take you to **Spider Details**, and provide some information on the spider. We now can run the spider by clicking the **Run** option from **Jobs Dashboard**, and then on **Spider Details**.

8. You can verify that the run was successful by clicking on the spider name (**booklist**) in the **Completed**, **Deleted**, **Running**, and **Next** sections, as seen in *Figure 5.11*:

Job	Spider	Items	Requests	Errors	Logs	Runtime	Started	Finished	o
1/2	booklist 1.0	240	252	0	811	00:01:01	2023-02-21 15:53:56 UTC	2023-02-21 15:54:57 UTC	finished
1/1	booklist 1.0	0	0	1	7	00:00:15	2023-02-21 15:49:57 UTC	2023-02-21 15:50:12 UTC	failed

Figure 5.11: Spider jobs completed

9. As seen in *Figure 5.11*, details such as **Items**, **Requests**, **Runtime**, and **Errors** are also available and can be clicked on for a more detailed investigation. *Figure 5.12* shows us the information under **Items**:

Figure 5.12: Job Items (loading Item 0 or the first item)

10. There are multiple options, such as **Download**, **Exporting to Files (CSV, JSON, XML)**, **Filtering**, **Dataset Publishing**, and **Samples**, that are available under the **Jobs Items** section, as seen in *Figure 5.12*.

With the steps mentioned, we successfully deployed our Scrapy project to `app.zyte.com` and extracted the details as per the code available in the spider (`booklist`).

> **Important note**
>
> It's also advisable that you test the crawler on your local machine for any exceptions and errors that might occur, testing all features (such as running the crawler and exporting data) and using the latest libraries (if possible). Regarding deployment or any issues in Zyte, please visit `https://support.zyte.com`.

In this section, we focused on the deployment of our crawler in the server.

Summary

In this chapter, we explored and learned about parsing and extracting data from the web using *Beautiful Soup* and *Scrapy*.

So far, in this book, we have identified many libraries and techniques that are effective and suitable for web scraping. Beautiful Soup equips developers with a handful of features to parse, traverse, and create a crawler. Scrapy provides the same features as Beautiful Soup and can be used for data extraction, but is more of a project-based framework that uses lots of libraries behind the scenes and enables you to focus only on your tasks. Because of Scrapy's easy-to-implement and collaborative architecture, it's quite popular among beginners, professionals, and even web-based service providers such as *Zyte*, *ScrapeOps*, and *Apify*.

In the next chapter, we will learn about and explore more scraping techniques and security-related issues.

Further reading

- Beautiful Soup: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`
- `lxml` – XML and HTML with Python: `https://lxml.de/`
- Web crawlers:
  - `https://www.cloudflare.com/en-gb/learning/bots/what-is-a-web-crawler/`
  - `https://research.aimultiple.com/web-crawler/`
- Web scraping: `https://www.zyte.com/learn/`
- robots.txt: `https://www.robotstxt.org/robotstxt.html`
- Scrapy Cloud: `https://www.zyte.com/scrapy-cloud/`
- Scrapy: `https://docs.scrapy.org/en/latest/index.html`

Part 3: Advanced Scraping Concepts

In this part, you will learn how to work with secure or security-enabled websites. You will learn to scrape data from web APIs and use browser automation tools such as Selenium to bypass certain imposed security. There are practical examples across the chapters in this part, which will help you to extract data from PDF files and even unstructured formats using regular expressions.

This part contains the following chapters:

- *Chapter 6, Working with the Secure Web*
- *Chapter 7, Data Extraction Using Web APIs*
- *Chapter 8, Using Selenium to Scrape the Web*
- *Chapter 9, Using Regular Expressions and PDFs*

6

Working with the Secure Web

So far, we have learned about the web, web content, reverse-engineering techniques, data-finding techniques, a few Python libraries, and a framework that we can employ to access and scrape the desired web content.

Plenty of security-related concerns exist on web platforms today, along with measures to ensure security. Lots of web applications, extensions, and even web-based service providers exist to protect us and our web-based systems against unauthenticated usage and unauthorized access.

The growing use of internet applications and e-commerce activity demands a secure web (or web-based security-enabled features) as a high priority to deal with actions that are harmful or even illegal. We often receive irrelevant emails or spam, containing information that we did not ask for.

It's quite challenging from a web scraping perspective to deal with such issues, but the concept of ethical hacking makes it more viable, even from an application and security perspective. We will cover a few basic concepts that deal with security on the web and proceed to look at web scraping.

In this chapter, we will learn about the following topics:

- Exploring secure web content
- HTML `<form>` processing using Python
- User authentication and cookies
- Using proxies

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required, and we will also use *JupyterLab* for Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections in *Chapter 2* to continue using the environment created.

The Python libraries that are required for this chapter are as follows:

- `requests`
- `pyquery`

The code files for this chapter are available online in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter06`.

Exploring secure web content

Today's web and internet technologies are quite vulnerable in terms of web security (content, authorization, illegal access, and so on). We want the web to be safe and the content that we browse, search, or view to be genuine, not violating any legal or ethical standards or affecting people's human rights.

The web must be accessible and available to everyone who seeks information, effectively and by following ethical practices. We often encounter web content that is not exactly what we were looking for, or hear of web content that has been tampered with or hacked into or that private and sensitive information has been leaked illegally, and so on. Although a lot of these cases are beyond our control, we can reduce vulnerabilities and make the web a much safer place to be.

In many new technologies, security-related concerns have been identified and solutions have been implemented. There are even applications and organizations concerned with network-based security, which are growing in demand and using the security enabled technologies. In government departments, developers use up-to-date technologies to make the web a safer place. With such improvements in technology, scraping and crawling are getting more challenging. Web scraping, or extracting the information available on the web, is not harmful when ethical concerns are taken into account.

In the upcoming sections, we will introduce some basic security-related concepts that can be implemented and are easily available on the web, and then explore, implement, and access them using Python.

Form processing

Form or HTML `<form>` processing, also known as form handling, is used in most cases to pass or submit collected or user-provided data to a backend or server. There are plenty of cases of web-based forms (such as login, contact us, registration, and subscription forms) being submitted by someone impersonating someone else.

From a user perspective, a form is simply a data collector that has some `<form>` elements, such as `<select>`, `<option>`, `<textarea>`, and `<input>`, asking for some input or values. When data is provided, it gets validated at some level and is processed by the server or concerned system, and any necessary further action takes place (such as user login, user registration, resetting a forgotten password, QR processing, and image validation).

Even from a user security perspective, when forms are used to deal with personal or private information with banks, e-commerce sites, and so on, some security-related measures are embedded in or linked to forms, such as providing a **One-Time Password** (**OTP**) via SMS or email or asking the user to fill in human-readable (but challenging) CAPTCHAs. Blocking or denying someone access to the web for some time or period if incorrect details have been submitted after multiple attempts is a common security measure on the web.

HTML forms use a dedicated HTTP method (`GET`, `POST`, `PUT`, and so on) and path to process submitted information. In addition to the available content in forms, they also contain some hidden fields (`<input type="hidden" ....>`) that are not visible in the browser but contain some values (such as page identification, system variables, third-party code, marketing-analysis IDs, user IDs, and security). There can be one or more of these fields and they can be found through the page source or using DevTools.

> **Important note**
>
> Form data submitted using the `GET` method is visible in the URL as a "key-value" pair. Please visit the *Implementing HTTP methods* section of *Chapter 2* or `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods` for more information on HTTP methods and `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form` for information on HTML forms. For more information on DevTools, visit the *Using web browser development tools* section of *Chapter 3*.

HTML forms are an effective way to transfer or share information on the web or across web pages. Now that we have given you a basic introduction to HTML `<form>` processing in this section, we will explore cookies and sessions in the next section.

Cookies and sessions

Cookies and sessions are two web-based buzzwords that most users associate with web-related security. They are mostly used for online privacy, imposing restrictions, filtering content, and preventing identity theft on websites. Nowadays, most websites even ask users to accept or deny cookies while browsing them.

> **Important note**
>
> We briefly discussed obtaining and locating cookies and sessions in the *Data-finding techniques used in web pages* section of *Chapter 1* and the *URL handling and operations* section of *Chapter 2*.

Cookies

Cookies are pieces of information created by websites. The created information exists as a key-value pair, similar to a JSON string or a Python dictionary, which is stored in text format on a user's **Personal Computer** (**PC**) or machine.

Generally, a cookie is a secure piece of information generated by a website. It identifies and saves a user's profile and their machine details, browser details, login credentials, browsing habits, searched keywords, and similar information.

This collected information and data is used to verify the user, log them in directly, and redirect to or present them with pages or locations of interest on their next visit to the website. Hence, information inside cookies acts somewhat like shortcut options or a memory-related tool that recognizes and remembers user actions and preferences.

There are various types of cookies; a few of the important ones are listed here:

- **Persistent**: Also known as a **first-party cookie**, this kind of cookie tracks and remembers a user's actions, preferences, and choices and presents those pages or choices on the user's next visit to the website. They also have a long expiration date and stay active for a long period of time, until they are deleted or expire. These cookies also contain information regarding the domain; in this case of first-party cookie, it is the parent domain that was visited.

- **Session**: In comparison to a persistent or first-party cookie, this is a temporary type of memory that remembers user activity. The session contains encrypted information that is stored on a server, identifying the client. As soon as a browsed website is closed or a user leaves a website, the session-based information is deleted.

- **Third-party**: Also known as **tracking cookies**, the original domain for this kind of cookie might not be the domain itself. They contain third-party domains or API URLs. These cookies normally collect information on a client's online behavior, searched terms, URLs visited, and so on. This collected information is then used to display customized advertisements, marketing popups, and so on.

Cookies normally contain different key:value pairs for various aspects; we will explore this in later chapters with practical examples. They also contain an expiration date that destroys the collected information after some time. This time might vary from a few minutes to a month or more. Most of the time, the information inside cookies is essential to bypass some web-based securities, and from a scraping perspective, it is necessary to have such information and manage or modify (by including cookies) an HTTP request accordingly.

With this information on cookies and their types, we will explore sessions in the next section.

Sessions

A session is used to maintain secure activity – for example, a user being identified through their profile across different pages of some website. A session, as explained in an earlier section, is a type of temporary memory that gets destroyed when a browser is closed.

Sessions are properties that enforce and share security identification between two or more systems. This could be an encrypted value or ID (unique identification values) generated on behalf of the user

(such as the machine, profile, or stored information on a server). Such IDs are used to identify and track user-related behavior, and they get updated every time a session ends.

In *Figure 6.1*, we can see certain rows that possess session-related names (such as `_gh_sess` and `user_session`) with the corresponding columns for **Value**, **Domain**, **Expires / Max-Age**, **Priority**, and so on:

× Headers Preview Response Initiator Timing Cookies											
Request Cookies ☐ show filtered out request cookies											
Name	Value ▲	Domain	P...	Expires / Max-Age	S...	H..	S...	S...	S...	P...	Priority
color_mode	%7B%22color_mode%22%3A%22auto%22%2C%22light...	.github.com	/	Session	2...		✓	L...			Medium
has_recent_activity	1	github.com	/	2023-03-04T14:15:24.375Z	20	✓	✓	L...			Medium
__Host-user_session_same_site	3AOWgHosLSz-TGOn9xVqmK6kNofrFLSInxY4tiGa9EXnC...	github.com	/	2023-03-18T12:46:03.150Z	77	✓	✓	S...			Medium
user_session	3AOWgHosLSz-TGOn9xVqmK6kNofrFLSInxY4tiGa9EXnC...	github.com	/	2023-03-18T12:46:03.150Z	60	✓	✓	L...			Medium
_device_id	92304b360b52e4eeddac56aa1ca95d37	github.com	/	2024-01-24T10:21:35.673Z	42	✓	✓	L...			Medium
tz	Asia%2FKatmandu	.github.com	/	Session	17		✓	L...			Medium
_octo	GH1.1.271232079.1674527553	.github.com	/	2024-01-24T02:31:20.877Z	31		✓	L...			Medium
_gh_sess	Qa6Kwm6KOgVkMFfEFy0GA%2F71UzuzLNKlj8VDxyT%2...	github.com	/	Session	3...	✓	✓	L...			Medium

Figure 6.1: Session-related information in cookies

Since information will be stored on a server and sharing or identifying information between a client and server is necessary, a cookie is the best place to store information. Session-related information is stored in cookies and is supplied to desired or accessed web pages, maintaining a stable and authenticated state.

> **Important note**
>
> Note that we can deny a website from storing information in cookies by refusing to accept the cookies, or even deleting stored cookies using the browser or DevTools. For more information on cookies and sessions, please visit `https://www.aboutcookies.org`, `https://developer.mozilla.org/en-US/docs/web/HTTP/Cookies`, and `https://securiti.ai/blog/session-cookies/`.

So far in this section, we have presented some basic concepts of cookies and sessions. In the next section, we will talk about user authentication and using different types of web-based security measures.

User authentication

Authentication or web-based user authentication is a process related to the handling and managing of user identification. This is often a combined process where HTML form processing and cookie-related information and activities need to be used.

User login or registration is done using HTML form processing. The information collected is stored and inserted into backend databases or secure files. Nowadays, **Two-Factor Authentication** (**2FA**) is often used across the web. Normally, registered users are verified and permitted to log in or out of a system, validating their username and password while also solving some CAPTCHAs. This whole process is

known as **Single-Factor Authentication** (**SFA**). Conversely, 2FA imposes additional verification on top of SFA, which involves the following:

- Sending an *activation code* or OTP to a cell phone (through a message or app) or email address

- Sending a secure *one-time clickable link* for verification (login, a forgotten password, updating information, and so on) to a registered email address or cell phone (through an app or message)

These authentication processes need a combination of HTML form processing, within some preconfigured or allocated time, and some secure identification IDs or a *key:value* pair that manages the session or session management, cookies, and so on, helping to complete the authentication process.

Now that you have been introduced to the basics of authentication, in the next section, we will deal with security and authentication-related web-based cases using the Python programming language.

HTML <form> processing using Python

Form processing has various titles and functions, such as search, filter, login, registration, submission, and verification. In this section, we will explore `http://quotes.toscrape.com/search.aspx`, as shown in *Figure 6.2*, and process or use the forms available on the page to extract or filter out the results, based on a choice of options (**Author** or **Tag**):

Figure 6.2: The search form (Author and Tag)

> **Important note**
>
> We need to investigate the available `<form>` tag, the options or form elements that exist, and the resulting pages as they appear on form submission. It's also advisable to explore form-related actions using DevTools, as this will help you to understand the flow of systems, links, availability, and resources such as HTTP headers, cookies, and payload.

During page analysis, we found that only a single form, named `filterform`, exists, and options are available for the **Author** name only; **Tag** doesn't have any options. **Tag**-related options are loaded dynamically if we select the **Author** option (`http://quotes.toscrape.com/filter.aspx` gets executed using the HTTP POST method and loads the options for **Tag**), and the final search results are loaded under `http://quotes.toscrape.com/filter.aspx` using HTTP POST, when both the **Author** and **Tag** options are chosen, and the submit button (**Search**) is pressed or clicked.

To initiate form-based activity through code, we have to identify and list the URLs that are used or loaded:

```
baseUrl="http://quotes.toscrape.com/search.aspx"
# POST URL: Tag and result.
filterUrl="http://quotes.toscrape.com/filter.aspx"
```

Let's break down this `<form>` processing step by step and capture the final result:

1. During analysis, we noticed that only the **Author** option was listed and the form is submitted automatically when we choose any option for **Author**. An HTTP POST request is processed with the payload, as shown in *Figure 6.3*:

Figure 6.3: HTTP POST (payload and form data)

2. From *step 1*, it's clear that we need to have at least three values (`author`, `tag`, and `__` `VIEWSTATE`); they are collected and processed with the following code:

```
author = response.find('select#author option:gt(0)')
    .attr('value')   # 'Albert Einstein'
tag = response.find('select#tag option').attr('value')
search = response.find('input[name="submit_button"]')
    .attr('value')   #'Search'

viewState = response.find('form[name="filterform"]
    input#__VIEWSTATE' ).attr('value')
formData={'author':author, 'tag':tag,
    '__VIEWSTATE':viewState}  # Payload info
# process internal URL: filterUrl to load Tag option

responseA = requests.post(filterUrl, data=formData)
# POST request
tag = responseTag.find('select#tag option:gt(0)')
    .attr('value')    # change
```

> **Important note**
>
> `__VIEWSTATE` or `viewState` is a unique, random value that is generated by websites to identify individual states of a page, which are often found as hidden `<input>` values. This `<form>` value exists in most websites that use **ASP.NET** technologies. The `viewState` value is used on the client side, and it preserves or retains the value of `<form>` elements, alongside page identity. For more details on **ASP.NET**-based form management, please visit `https://www.c-sharpcorner.com/article/Asp-Net-state-management-techniques`.

   `filterUrl` with the `http://quotes.toscrape.com/filter.aspx` value is an internal URL that is processed as soon as there is a change or selection within the **Author** option, and also during final result loading. Direct access to `filterUrl` will result in a `405` HTTP status code (**Method Not Allowed**). For more details on HTTP status codes, please visit `https://developer.mozilla.org/en-US/docs/web/HTTP/Status`.

3. Continuing from *step 2*, as **Author** is selected, **Tag** options are listed. As shown in *Figure 6.4*, we have now selected the option for **Tag** (**change**) and have submitted the form button with the text **Search**. This submission results in some changes to **Form Data** under **Payload**, as shown in *Figure 6.4*:

Figure 6.4: HTTP POST (Payload and Form Data) during submission

So, for the results, we need values such as `author`, `tag`, `submit_button`, and `__VIEWSTATE`. Submitting these values as form data to `filterUrl` will load and update `<div class="results">` with the `quote` element, as shown in the following code:

```
finalFormData={ 'author':author, 'tag':tag,
    'submit_button':search, '__VIEWSTATE':viewState}
    # payload

responseB = requests.post(filterUrl,
    data=finalFormData)    # HTTP POST filterUrl

responseFinal = pq(responseB.content)
result = responseFinal.find('.results .quote').text()
result
# '"The world as we have created it is a process of
  our thinking. It cannot be changed without changing
  our thinking." - Albert Einstein (change)'
```

The results will be listed as seen in *Figure 6.5*:

Figure 6.5: Result after form submission (with Author and Tag)

There might be one or multiple results, as seen in *Figure 6.5*. The result is found in the `<div class="quote">` element, as seen in the following code:

```
<div class="results">......
  <div class="quote">
    <span class="content">"The world as we have
      created it is a process of our thinking. It
      cannot be changed without changing our
      thinking."</span> -
    <span class="author">Albert Einstein</span>
      (<span class="tag">change</span>)
  </div>
</div>
```

If there are multiple results, then multiple instances of `<div class="quote">` might exist, and we need to extract those results accordingly.

4. We can repeat *steps 1-3* for different **Author** and **Tag** values.

From these steps, it's clear that `<form>` processing is an important concept that requires proper planning and analysis. We successfully managed and processed a multi-step type of form, loaded with hidden elements. The complete code for this example can be found on GitHub: `https://github.com/PacktPublishing/Hands-On-web-Scraping-with-Python-Second-Edition/blob/main/Chapter06/chapter06_htmlform.ipynb`.

In the next section, we will look at the authentication (user login and logout) process.

User authentication and cookies

User authentication (managing and handling user credentials) is another form of web security. In this section, we will use the user authentication feature available on the `http://quotes.toscrape.com/login` site, along with cookies.

As seen in *Figure 6.6*, `http://quotes.toscrape.com/login` contains a page with the **Login** text, and a form asking for **Username** and **Password**:

Figure 6.6: User login page

Analyzing the page source, there's only a single `<form>` element found, with one `hidden` input element named `csrf_token`, two text input elements with `type="text"` (username and password), and finally, an object with the `submit` type and the value `Login`, as seen in *Figure 6.7*:

```
27    <form action="/login" method="post" accept-charset="utf-8" >
28        <input type="hidden" name="csrf_token" value="KGVFwPOucWyElYtaReDdkrBSbHCJjQsLzvTMfxgiopIZnUAhNXmq"/>
29        <div class="row">
30            <div class="form-group col-xs-3">
31                <label for="username">Username</label>
32                <input type="text" class="form-control" id="username" name="username" />
33            </div>
34        </div>
35        <div class="row">
36            <div class="form-group col-xs-3">
37                <label for="username">Password</label>
38                <input type="password" class="form-control" id="password" name="password" />
39            </div>
40        </div>
41        <input type="submit" value="Login" class="btn btn-primary" />
42
43    </form>
```

Figure 6.7: Login page – <form> source code

> **Important note**
> `http://toscrape.com` is a demo site, enriched with content for scraping-related purposes. The site does not provide a user registration link. To practice authentication, we can use any credentials for the username and password. Successful login loads `http://quotes.toscrape.com` with a user logout option, or we can use the `http://quotes.toscrape.com/logout` link.

As seen in the following code, the required links are defined. We now need to collect form-related data from the HTML form available at `loginUrl` (as seen in *Figure 6.7*):

```
import requests
from pyquery import PyQuery as pq
baseUrl="http://quotes.toscrape.com/"    # main URL
loginUrl="http://quotes.toscrape.com/login" # POST URL (Credentials
and results)

logoutUrl="http://quotes.toscrape.com/logout"
```

To process the login step, we have the necessary URLs already declared. In the following code, we have defined the `user` and `passw` variables with some test values. These two values are there to be submitted in the payload. `loginUrl` has been processed, and we have collected the values for `csrf_token`, `username`, and `password`:

```
user = "test"
passw = "password"
html = requests.get(loginUrl)  # load loginUrl
response = pq(html.content)
# collect <form> data,required for submission

csrf = response.find('form:first input[name="csrf_token"]')
    .attr('value')
username = response.find('input[id="username"]')
    .attr('name')
password = response.find('input[id="password"]')
    .attr('name')
```

With the form data in hand, payload information is to be created: `payload = {"csrf_token":csrf, username:user, password:passw}`. This payload will be submitted to the path mentioned in the form attribute named `action` or `loginUrl`, in this case. Upon investigation (using DevTools), it looks like `loginUrl` is processed with `payload` when the form is submitted, but it gets redirected (as seen in *Figure 6.8*) to `baseUrl`, loading the link for `logoutUrl`:

Figure 6.8: HTTP status code 302

> **Important note**
>
> The original URL produces a **Status Code: 302 Found** error and gets redirected to another URL. For more information, please visit `https://developer.mozilla.org/en-US/docs/web/HTTP/Status/302`.

As seen in the following code block, the required `payload` dictionary object with all information found and confirmed using DevTools is posted to `loginUrl`. On submitting the form as conveyed by *Figure 6.8*, redirection must happen, but it (`postHTML.url`) is still showing `loginUrl`:

```
payload = {"csrf_token":csrf, username:user,
    password:passw}

postHTML = requests.post(loginUrl, data=payload)
postHTML.url
# http://quotes.toscrape.com/login
```

For the current scenario, there must be either a problem with the payload or some code-related error. Upon conducting multiple attempts and verifications, it is found that the code processed so far has no issues but the POST request is also looking for session-related data that is available in the cookies to be submitted.

As seen in *Figure 6.8*, we can see that the **Request Headers** tab has an entry for **Cookie** with some *session*-related values. We are required to collect the HTTP headers and cookies too for successful form processing. For details on HTTP headers and cookies, please refer to the *URL handling and operations* section of *Chapter 2*.

To progress with the authentication part, as seen in *Figure 6.9*, `html.headers` outputs the HTTP request headers:

```
html = requests.get(loginUrl)
response = pq(html.content)
```

```
html.headers
```

```
{'Date': 'Mon, 06 Mar 2023 15:00:20 GMT', 'Content-Type': 'text/html; charset=utf-8', 'Content-Length':
'1869', 'Connection': 'keep-alive', 'Vary': 'Cookie', 'Set-Cookie': 'session=eyJjc3JmX3Rva2VuIjoidVJsdGVn
WWZKRlZ5cHFOY29XeEtFRFVqaVpNSVBPbnZTbUxrrQ1RCR3piZFhRSGF3cNoQSJ9.ZAYABA.P_mWHn2rSrmmdbYhYhaSPp-cS-o; Http
Only; Path=/'}
```

```
html.headers['Set-Cookie']
```

```
'session=eyJjc3JmX3Rva2VuIjoidVJsdGVnWWZKRlZ5cHFOY29XeEtFRFVqaVpNSVBPbnZTbUxrrQ1RCR3piZFhRSGF3cNoQSJ9.ZAY
ABA.P_mWHn2rSrmmdbYhYhaSPp-cS-o; HttpOnly; Path=/'
```

Figure 6.9: Collecting HTTP headers and desired values

We are trying to collect the value of `Set-Cookie` (`html.headers['Set-Cookie']`) from the returned `headers` data.

As seen in the following code, the Python `requests` library supports various parameters, such as `data` and `headers`:

```
payload = {"csrf_token":csrf, username:user,
    password:passw}

postHTML = requests.post(
    loginUrl,
    data=payload,
    headers={'Cookie':html.headers['Set-Cookie']}
    )      # POST request

postHTML.url      # http://quotes.toscrape.com/
verify =
    pq(postHTML.content).find('a[href="/logout"]').text()
    # Logout
```

The HTML POST request to `loginUrl` with the `headers={'Cookie':html.headers['Set-Cookie']}` parameter seems to be working now, which shows the redirected URL, `http://quotes.toscrape.com`, and we can also find the URL for logging out (`verify`) there. The complete code for user authentication can be found on GitHub: `https://github.com/PacktPublishing/Hands-On-web-Scraping-with-Python-Second-Edition/blob/main/Chapter06/chapter06_authentication.ipynb`.

The authentication process has been completed successfully, with the help of cookies and sessions. It's the ethical duty of the user to log out from a session after the completion of any task. `requests.get(logoutUrl)` simply terminates the existing user session.

> **Important note**
>
> It's also an important factor to consider that, in the current case, only the `Cookie` header value was sufficient for successful form submission. There are plenty of other keys inside HTTP headers (such as `Accept`, `Origin`, `Host`, and `User-Agent`) that are also important and must be passed along with HTTP `GET` or `POST` parameters for successful form submission.

Apart from the form elements, HTTP headers and cookie values do play a significant role in form submission, and even in receiving the expected response from the processed link. In the next section, we will introduce and learn how to use the HTTP proxy.

Using proxies

A proxy (HTTP proxy or web proxy) is considered middleware for the web. A proxy is a gateway that is used to communicate between a client and a server. Put simply, a proxy is an **Internet Protocol** (**IP**) address with some random ports assigned to it. On the web, an HTTP request starts from the client end with its own IP, which, when routed through the proxy, gets updated as a new IP and is then forwarded to the actual destination looking for a response. So, a proxy here works as a transport layer that resides between the client and the server or destination.

On the internet, there are plenty of services (paid and free) offered by various organizations providing proxies that work as a filter layer for their customers and deal with different kinds of security-related threats, malware, content filtering, spy protection, and many other activities. There are plenty of benefits to using proxies; a few of them are listed here:

- **Privacy**: The destination server does not identify the actual client. Privacy issues are mitigated as the client has initiated a request using the web proxy.

- **Security**: With privacy in mind, client information is not shared with the end location, which is the server. Security features are implemented, such as blocking the scope of multiple related requests and blocking and blacklisting the IP for a certain duration.

- **Content filter**: Along with security and privacy, the proxy server or middle parties act as middleware between the client and server. In this process, middle parties can act as a secure layer that filters out content to the client based on some preconfigured settings. For example, parents can use settings regarding the content that's being delivered to their child (based on age, movie certificates, and many more factors). In addition, such a facility provides security to the end user against malware, spyware, geo-location-related spam, and anything that is detected as harmful or could be harmful to the client machine.

- **Speed**: Most proxy providers use their servers to accept client requests and perform a task on their behalf. These servers also possess in-built cache management systems that transfer the content immediately back to the client if it already exists in the cache. This speeds up the time taken for requests and the response cycle.

Using proxies or involving them in scraping tasks or projects is considered a performance enhancer and is used to tackle or bypass web-related security measures (such as CAPTCHAs), IP blockage, geo-location barriers (some sites, for example, US-based government or organization sites demand a US IP; otherwise, access to the site is denied), and plenty more.

An HTTP proxy hides the client IP address from the server. For example, when the client IP address 192.168.0.1 passes or uses a proxy, it is converted to 11.xyz.4x.y2, or something similar, which then forwards the request to the server. In this case, the server identifies that the request is coming from 11.xyz.4x.y2 and not 192.168.0.1.

In the *User authentication and cookies* section, we came across the need for HTTP headers in making requests. Web proxies are used to generate random HTTP headers (combining different header keys, such as `User-Agent`, `Accept`, `Origin`, and `Referer`), which, when processed, act like an entirely new request to the server, and the proxies completes the associated tasks successfully (most of the time).

> **Important note**
> It's a widely accepted fact among developers that forwarding a random combination of HTTP headers, with or without proxies, to HTTP requests will work most of the time as it will bypass certain web-based imposed restrictions.

From a scraping perspective, proxies are mostly divided into two types:

- **Residential proxy**: These are considered the best, safest, and most reliable among the proxy types. These are most often the IP addresses of real devices, which are difficult to detect as fake IPs or proxy addresses. The server or end user's machine generally treats requests from such proxies as being genuine and does not promptly respond with security measures. These are very costly in comparison to other types of proxies, and many providers put conditions in place, such as limits on the number of attempts, the usage per hour or per day, and the total consumption of bandwidth.

- **Rotating or rotational proxy**: A rotating proxy, sometimes also known as a **data center proxy**, is normally considered to be a group of various types of proxies (*residential*, *shared*, *public*, *freely available*, and many more). If any address is blocked, then some other address is forwarded. The server or end machine will treat the request as new and from a different client, which bypasses certain restrictions. Because plenty of proxy types are mixed, the chances of a device being recognized by the server as an agent or proxy are not high for this type.

> **Important note**
>
> There are various types of proxies. Here are a few links for more information:
>
> `-https://developer.mozilla.org/en-US/docs/web/HTTP/Proxy_servers_and_tunneling`
>
> `-https://blog.apify.com/types-of-proxies/`
>
> `-https://www.zyte.com/blog/python-requests-proxy/`
>
> `-https://smartproxy.com/proxies`
>
> `-https://scrapfly.io/blog/how-to-rotate-proxies-in-web-scraping/`
>
> `-https://www.webscrapingapi.com/best-shared-dedicated-proxy-providers`
>
> Plenty of proxy providers can be found easily via a Google search (`https://www.google.com/search?q=web+proxy+provider`).
>
> Freely available or shared proxies found on the web are not recommended or safe to use.

Depending on the type of proxy, we can find the IP from proxy providers, and collect and use the proxies while making HTTP requests. In addition, many providers ask you to register and pay to receive an API key, which can be used to make HTTP requests.

The following code displays an example, with demo links, of how a proxy can be obtained and used to make HTTP requests and progress the task at hand:

```
import requests, random
from pyquery import PyQuery
urlA = "https://www.somewebsite.com"   # demo URL
urlB = "http://www.anotherdomain.io"   # demo URL
proxyPool = ["3.168.X.X", "104.X.X.12", "107.194.X.X",
    "202.45.X.X"]  # demo
proxyAPI =
    "https://www.someprovider.com/proxy?key=API_KEY"
    # demo
```

Proxy providers generally provide their clients with some IP addresses or API URLs from a list of IPs, or a single IP (which can be used for a certain duration) is provided. `proxyPool` returns a list of available IPs, whereas `proxyAPI` returns a proxy upon being called.

It's also quite a common scenario, and recommended, that HTTP headers are provided to the HTTP request when we use proxies. We can provide HTTP headers to any HTTP request. It almost impersonates the HTTP request. An HTTP header's values can be obtained using DevTools and reused with basic formatting and updates:

```
headersHTTP = {
    "accept" :
        "text/html,application/xhtml+xml,application/xml;
        q=0.9, ….b3;q=0.7",
    "accept-encoding": "gzip, deflate, br",
    "accept-language":"en-US,en;q=0.9",
    "cache-control: "max-age=0",
    "upgrade-insecure-requests":"1",
    "referer":"https://www.somesite.com",
    "user-agent": "Mozilla/5.0 (Windows NT 10.0;
        Win64; x64) …….",
}  # HTTP Header (demo)
```

> **Important note**
>
> To obtain HTTP headers or other values, we can choose any URL in DevTools. Right-click the URL and select the **Copy as a cURL (bash)** option under the **Copy** option. We will receive a `curl`-formatted HTTP request, which can be cleaned, modified, updated, and changed to a usable format for the required programming language. For more information on `curl`, please visit `https://curl.se/`.

The Python `random` built-in library has the `sample()` method, which returns a random option from the arguments provided to `random.sample()`, such as how much output is to be returned. In the following code, we have chosen a random proxy from `proxyPool` and used it to execute an HTTP request that returns a response as `responseA`:

```
#1. Using proxyPool
proxyA = random.sample(proxyPool,k=1) # select one random IP
responseA = requests.get(urlA, headers=headersHTTP,
    proxies={'https':proxyA[0]})
print(responseA)
```

The following code uses a proxy (`proxyB`) obtained from `proxyAPI`, and uses it to make an HTTP request and obtain the response, `responseB`:

```
#2. Using proxyAPI
proxyB = requests.get(proxyAPI).content  # returns dynamic IP
responseB = requests.get(urlB, headers=headersHTTP,
    proxies={'http':proxyB})
print(responseB)
```

This section has covered a basic introduction to proxies, illustrating their use with some code examples. Proxy usage is a growing market considering the reliability and security features they offer on the internet, network services, and service providers.

> **Important note**
>
> On the market, there are plenty of service providers for proxies. They offer various paid offers and can even be contacted for customized deals if required. A few of them are `https://proxyempire.io/`, `https://oxylabs.io/`, `https://brightdata.com/`, `https://mobilehop.com/`, and `https://smartproxy.com/`, but there are many more.

Summary

Web security is a compulsory component of the current internet revolution. We need to participate in web scraping and extraction processes ethically for the betterment of the information that's available everywhere on the web. We have learned about the basics of processing with HTML forms, cookies, and sessions, and using proxies with the help of the Python programming language, from a web scraping perspective.

In the next chapter, we will use web-based APIs to collect relevant data.

Further reading

- Cookies and sessions:

  - `https://www.kaspersky.com/resource-center/definitions/cookies`

  - `https://developer.mozilla.org/en-US/docs/web/HTTP/Cookies`

  - `https://www.ibm.com/docs/en/sva/9.0?topic=cookies-session-concepts`

- HTTP headers:

  - `https://jkorpela.fi/http.html`

  - `https://developer.mozilla.org/en-US/docs/web/HTTP/Headers`

- HTML forms:

  - `https://www.w3schools.com/html/html_forms.asp`

  - `https://developer.mozilla.org/en-US/docs/web/HTML/Element/form`

  - `https://www.w3.org/TR/html401/interact/forms.html`

- User authentication:

  - `https://www.sciencedirect.com/topics/computer-science/user-authentication`

  - `https://www.ibm.com/docs/en/aix/7.2?topic=passwords-user-authentication`

- Proxy:

  - `https://www.fortinet.com/resources/cyberglossary/proxy-server`

  - `https://smartproxy.com/proxies`

  - `https://www.upguard.com/blog/proxy-server`

  - `https://www.zyte.com/learn/use-proxies-for-web-scraping/`

7

Data Extraction Using Web APIs

So far, we have learned about the web, web-based technologies, techniques for locating and extracting data, and plenty of data-related services in various chapters of this book.

Web **application programming interfaces** (**APIs**) are built to interact with queried information. Web APIs provide interfaces to query and access formatted data (normally in JSON format) or structured data that is easy to use and process. Depending on the type of information (HTTP POST payload, JSON content, third-party authentication, collective metadata, or middleware, for example) being carried and the type of service being engaged with, authentication and some form of tokenization are also required.

In general, a web API is a medium for loading, seeking, transferring, and embedding information in pages or other APIs.

In this chapter, we will learn about the following topics:

- Introduction to web APIs
- Data formats and patterns in APIs
- Web scraping using APIs

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required, and we will be using *JupyterLab* for our Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections in *Chapter 2* to continue with setting up and using the environment we created.

The Python libraries that are required for this chapter are as follows:

- `requests`

- `csv`

The code files for this chapter are available online in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter07`.

Introduction to web APIs

Information sharing and processing are in huge demand on the web and across web-based services. A web API is a service available on the web that provides access to resources such as raw data, filtered information, and embedded and dynamic content, normally in a ready-to-use and exchangeable format such as JSON, CSV, or XML.

Information exchange or transfer is the core purpose of web APIs. We can consider an API as a web- or browser-based service or **user interface** (**UI**) for retrieving, sharing, or accessing information based on requests made to a web server or API provider. With plenty of security-related vulnerabilities, many providers ask for authentication in the form of an *API key* (a unique block of identification with a certain length that identifies the user or client with the API or server) or some code to make API processing more efficient and effective.

Web APIs are not dependent on any single programming language (such as ASP, PHP, or JSP); users and developers can read data from an API and copy it into their system or platforms using any tools or languages they wish. Data collection and exchange, even in huge volumes, has been made possible with the use of APIs, and also, APIs incorporate some security when required. Many API providers found on the web provide their APIs as **Software-as-a-Service** (**SaaS**) and **Product-as-a-Service** (**PaaS**) to their clients.

> **Important note**
>
> As developers and data collectors, we should always be analyzing web links (using DevTools) for any possible APIs or any other related links, if available. If we use DevTools, then scraping activities become quite easy and fast compared to dealing with HTML code using regex, PyQuery, lxml, and so on.
>
> While using DevTools for scraping, there might be situations that require some code-level processing (such as HTTP requests) to be handled, such as HTTP headers, HTTP methods, and setting up requests with proxies.

The number of web APIs on the market for web-related services and products has grown tremendously. Product and service users are often provided with API tokens and API keys, and they can use APIs for their tasks (banking, fetching logs, HTTP request analysis, web traffic analysis, accessing HTTP

status code logs, customer support, and accessing contact us and sitemap pages, dealer lists, and more). Growth in the number of APIs has powered mobile app systems. Many service providers use APIs in their mobile or associated systems and use content from APIs to create or generate various kinds of messages and display that generated information in their mobile apps.

From a user's point of view, if it's only information that users are seeking, APIs are a huge relief because users do not have to learn complex systems or programming languages. Service providers will make API links available, which can be used to load and view results in real time and as required.

APIs extend the possible uses of browser-based features, and serving data. In the next section, we will briefly discuss API types.

Types of API

Web APIs, when normally used through the browser, accept some input, authorization credentials or keys, and information for existing keys or key:value pairs. The API result generated shows the data, and even transfers or exchanges the data to configured systems. Developers can also extend the functionality and features of web APIs, integrating essential logic as required.

Based on the *web as a service*, APIs can be classified into four different types:

- **Public API**: This is the most common type of API found on the web. These APIs are the services that websites provide to their clients and users. They are service-oriented and are used for information exchange. There are plenty of freely available public APIs that are found on the web relating to Twitter, weather forecasting, the stock market, NBA games, and plenty more. Also, most of the free APIs available on the web apply usage limits to each IP address or machine, or time limits.

- **Partner API**: These are a restricted type of public API. Service providers allow their approved clients or registered users to use this API based on authorization, paid services, and requirement demand. Service providers provide an API key or credentials, and the client has to use the key or credentials while using the API.

- **Private or Internal API**: These APIs are used for internal, company, or organizational purposes. They are also known as in-house APIs and serve organizational features such as calendars, **human resources** (**HR**), leave management, notices, and many more for employees and staff.

- **Composite API**: Also known as unified APIs, these APIs incorporate multiple different APIs from one system. There might be task-specific APIs implemented in a system that need to be implemented as a process in a certain order (that is, the output from one API becomes the input for another API); a unified API comes in handy in such cases. The client, when accessing a composite API, is not sure where the content comes from because redirection and time-related issues might occur with this type of API.

APIs provide data and perform services as requested. In a web-based communication *protocol and design architecture* scenario, they can be divided into two types:

- **REST**: The most common and popular API architecture, **representational state transfer** (**REST**). It is stateless (data is not stored between requests), it supports caching, and it is considered secure. It uses standard HTTP methods (GET/POST) to provide services that are not exposed. HTTP status codes are tracked and can also be provided as responses. It supports various response formats, such as JSON, XML, and CSV. External library support is not required when used with HTTP. *RESTful* (this is what REST APIs for web services are called) APIs provide an interface for the user and resources to communicate; many public APIs are RESTful. In the majority of cases, REST APIs perform **Create, Retrieve, Update, and Delete** (**CRUD**) activities.

- **SOAP**: **Simple object access protocol** (**SOAP**) is normally a messaging protocol for the web. It supports various other protocols on the web, such as HTTP, **Simple Mail Transfer Protocol** (**SMTP**), **Remote procedure calls** (**RPC**), and **Transmission control protocol/Internet protocol** (**TCP/IP**) to exchange or share information across the web. SOAP is independent of platforms and programming languages. SOAP is popular in distributed enterprise environments because of its function drive approach, strict rules, structure, and controlled standard. SOAP supports XML as a messaging service and is known as an XML-based protocol. The HTTP POST method is used for information exchange and is also considered difficult to use by developers.

We now have looked at the different types of APIs based on their architectures and the services that are integrated. With the growing demand for data, web APIs are already widely used because of their applicability and usage among diverse web systems, applications, and users.

> **Important note**
> For more information on web APIs, REST, and SOAP, please visit the following URLs: `https://www.w3.org/api/`, `https://www.w3.org/TR/soap/`, `https://restfulapi.net/`, `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, and `https://www.altexsoft.com/blog/engineering/`.

In the next section, we will discuss the benefits and limitations of web APIs from a data collection point of view.

Benefits of web APIs

Though web APIs are not exactly a solution to web scraping, APIs are among the preferred and most widely searched data sources that can be used to retrieve data. APIs not only process web queries (URLs with parameters); they also display information that has been requested.

We can access an API as an information source, transformer or exchanger, query engine, in-house web system, and plenty of other forms and formats where data communication is needed. We just need the web browser, API access, and the API's URL, and we are ready to go. With the growing demand from developers and users, web APIs are being deployed, or are being used directly or indirectly.

Listed here are a few reasons why APIs are preferred to other programming concepts from a web scraping perspective:

- **Integration**: APIs return data in a standard format, such as JSON, that can be saved and processed further with further additions or as required.

- **Automation**: Executing APIs and collecting output is a part of automation as APIs contain pre-configured code, such as calls to their internal processes or procedures, and necessary dependencies, which work as automation.

- **Cross-platform**: Each API requires a URL with supported or necessary arguments that need to be executed or loaded in the web browser. Machine, OS, and web browser dependencies are not factors to consider when receiving output.

- **Data format**: Data received via APIs is in a certain format, such as JSON. Structured and semi-structured data will consume less processing time and is much easier to handle.

- **Scalability and flexibility**: APIs use internal mechanisms to process and output content. Certain URLs with some parameters will be dedicated to some sort of planned output. APIs' links with parameters and key:value pairs can be prepared and provided with similar or matching values (other than existing values) to obtain the content. These manipulations of the values are deployed and tested and are used for activities such as scraping.

- **Data wrangling/preprocessing**: There will certainly be some structured and semi-structured output from APIs. Information from APIs is self-explanatory in comparison to content in paragraph or textual formats. Data is available as *key:value* (Python dictionary or `dict()`) pairs, and is easy to identify, iterate through, and process further. This saves time, money, or both in cleaning, pre-processing, wrangling, and many more data processing activities.

- **Time**: APIs are easy to process, call, and load in the browser. These activities take less time in comparison to whole machine or virtual machine setups, which require more time to further process the obtained data.

- **Request-specific**: Data returned from APIs is specific to the arguments supplied; there might be more information returned than is required. Users and developers can access specific data by applying specific filters or as provided by the API.

- **Secure**: API processing using a subscription and API keys is considered secure. But this security can be bypassed with HTTP headers and proxies.

While there are plenty of benefits, there are also some cons to look for while using APIs for scraping purposes. Some of them are listed here:

- **Availability**: APIs might not exist for all targeted websites, or if one is available, it might not be accessible to users.

- **Validation**: Data retrieved by an API might require verification and validation. Normally, API content does not reveal the date or period of the data returned, though it's assumed to provide up-to-date information.

- **Limitation**: APIs can filter and query their output, but there are limitations in the resources they return. An API might return 20 results per page but only 100 in total. APIs are not replacements for web-related content. In addition, the parameters supported by an API might be limited, which in turn limits the expected results.

- **Irrelevant data**: Data returned by an API might not exactly fit your purpose. There might be a huge amount of irrelevant data, which requires extra cleaning and filtering.

- **Incomplete data**: Data returned by an API may fulfill a few of your scraping requirements but might not completely or exactly fulfill the expectations or requirements. For example, basic information on e-commerce data might exist, but detailed information may not be available through APIs, so it might be necessary to crawl through individual product URLs. Data collection using APIs that are available across categories and in multiple pages might be tedious.

With these pros and cons, API usage is growing on the internet. Various sites allow or use API-based data communication. From a web scraping point of view, APIs might fulfill certain data needs, but not completely. The data available can be helpful and can be used to tackle or manage the urgent need of data in JSON type format or even used for inspection purposes.

> **Important note**
> Web APIs are not an alternative to web scraping. They do provide timely support and various filters. Plus, if we find hidden API links, we can access data directly.

In the next section, we will look at using APIs, loading requests with different parameters, and exploring APIs' format, URLs, and more.

Data formats and patterns in APIs

Data available through APIs might be different from what you expected or might not fit the plan exactly – in the *Benefits of web APIs* section, we covered a few of the ways the data might be different (limitation, irrelevance, and more).

Acquiring data from APIs is a straightforward process with or without API authorization. Content received via an API may appear in many formats, such as key names, nested lists and dictionaries, named or numerical indexing blocks, and many more. We generally find API content in JSON format, comprising Python lists and dictionaries.

Before moving on to the data formats, patterns, and results of APIs or API content, it is important to demonstrate how APIs are called or used. Most of the time, API service providers keep an updated

version of their APIs in their documentation. Listed here are a few examples that web users normally use to access data from an API (example URL: `exampledomain.com`):

- `http://api.exampledomain.com`

- `https://api.exampledomain.com/resource?key1=value1`

- `https://demo.exampledomain.com/api/v1/holidays/2023`

- `https://api.exampledomain.com?query=somevalue&limit=10&sort=asc`

- `http://content.exampledomain.com/search?key=value&key_a=value_a&fields=field1,field2,field3`

- `https://api.exampledomain.com?start=0&end=10&step=2`

- `https://api.exampledomain.com/find?apiKey=somevalue&key1=value1&format=json`

- `https://api.sunrise-sunset.org/json?lat=36.7201600&lng=-4.4203400&date=today`

- `https://api.sunrise-sunset.org/json?lat=36.7201600&lng=-4.4203400&formatted=0`

Here are a few pointers from the preceding list of example URLs:

- Various types of URLs can be found, and they look like API links.

- Key=value arguments are used to filter the data to be more specific. If an unwanted combination of *key* and *value* is provided, the API will return an error message or a null schema.

- Various HTTP methods (*GET/POST/PUT* and more) can be used to load the URLs. HTTP GET method is the default method to access the URLs.

With this brief overview of probable API URL formats, let's explore a few examples and their responses in the following examples.

Example 1 – sunrise and sunset

`https://sunrise-sunset.org/api` provides a REST API that returns sunrise and sunset times based on the geo-coordinates supplied. The API supports the GET HTTP method and a number of other parameters, along with the coordinates.

For example, `https://api.sunrise-sunset.org/json?lat=36.7201600&lng=-4.4203400` is provided with values for `lat` (latitude) and `lng` (longitude). This request results in JSON content, as shown in *Figure 7.1*:

```
▼ {
    ▼ "results": {
          "sunrise": "6:26:01 AM",
          "sunset": "6:26:43 PM",
          "solar_noon": "12:26:22 PM",
          "day_length": "12:00:42",
          "civil_twilight_begin": "6:01:34 AM",
          "civil_twilight_end": "6:51:10 PM",
          "nautical_twilight_begin": "5:31:31 AM",
          "nautical_twilight_end": "7:21:13 PM",
          "astronomical_twilight_begin": "5:01:11 AM",
          "astronomical_twilight_end": "7:51:33 PM"
      },
      "status": "OK"
  }
```

Figure 7.1: API result (sunrise-sunset)

The response, as shown in *Figure 7.1*, is available with two keys: `results` and `status`.

Example 2 – GitHub emojis

`https://api.github.com/emojis` provides a list of emojis that can be used in GitHub comments and content. There are a total of 1,876 items; we can see a few of them in *Figure 7.2*:

```
▼ {
    "100": "https://github.githubassets.com/images/icons/emoji/unicode/1f4af.png?v8",
    "1234": "https://github.githubassets.com/images/icons/emoji/unicode/1f522.png?v8",
    "+1": "https://github.githubassets.com/images/icons/emoji/unicode/1f44d.png?v8",
    "-1": "https://github.githubassets.com/images/icons/emoji/unicode/1f44e.png?v8",
    "1st_place_medal": "https://github.githubassets.com/images/icons/emoji/unicode/1f947.png?v8",
    "2nd_place_medal": "https://github.githubassets.com/images/icons/emoji/unicode/1f948.png?v8",
    "3rd_place_medal": "https://github.githubassets.com/images/icons/emoji/unicode/1f949.png?v8",
    "8ball": "https://github.githubassets.com/images/icons/emoji/unicode/1f3b1.png?v8",
    "a": "https://github.githubassets.com/images/icons/emoji/unicode/1f170.png?v8",
    "ab": "https://github.githubassets.com/images/icons/emoji/unicode/1f18e.png?v8",
    "abacus": "https://github.githubassets.com/images/icons/emoji/unicode/1f9ee.png?v8",
    "abc": "https://github.githubassets.com/images/icons/emoji/unicode/1f524.png?v8",
    "abcd": "https://github.githubassets.com/images/icons/emoji/unicode/1f521.png?v8",
```

Figure 7.2: GitHub emojis

As shown in *Figure 7.2*, the API result is in JSON format. Each entity is a distinct key:value pair. There are no nested or array results from the API.

Example 3 – Open Library

`https://openlibrary.org/search.json?q=python&author=Wes` provides data that's filtered using two parameters: `q=python` and `author=Wes`. We are querying the `python` text for authors whose first name is or contains the text `Wes`. **Open Library** is an experimental search API (`https://openlibrary.org/dev/docs/api/search`). We can see that there is no mention of the term API in the URL in this example. The URL looks like a regular HTTP GET request with a few GET parameters.

Figure 7.3 shows the result, which totals `11`. Each result is an element of possible arrays found inside the `docs` block. There are also multiple key:value pairs and a few list and array entities inside `docs` with some items inside them.

```
▼ {
      "numFound": 11,
      "start": 0,
      "numFoundExact": true,
   ▼ "docs": [
      ▼ {
            "key": "/works/OL17422847W",
            "type": "work",
          ▶ "seed": [ … ], // 10 items
            "title": "Python For Data Analysis",
            "title_suggest": "Python For Data Analysis",
            "title_sort": "Python For Data Analysis",
            "edition_count": 4,
          ▶ "edition_key": [ … ], // 4 items
          ▶ "publish_date": [ … ], // 3 items
          ▶ "publish_year": [ … ], // 2 items
            "first_publish_year": 2012,
            "number_of_pages_median": 550,
          ▶ "lccn": [ … ], // 1 item
          ▶ "oclc": [ … ], // 2 items
          ▶ "lcc": [ … ], // 2 items
          ▶ "isbn": [ … ], // 8 items
```

Figure 7.3: Open Library API

> **Important note**
>
> Developers or users are supposed to open or keep investigating the browser's DevTools and open the **Network** panel for content and information such as HTTP methods, HTTP headers, cookies, content type, and to see if any API links can be found. Multiple APIs might also get used behind some web-based logic for generating content.

In this section, we have explored various types of data that are found through APIs. Nested, unnested, key:value pairs, and combined output are generally expected from web APIs. Apart from this, API-related links are also difficult to find, until and unless the website provides them, otherwise pages such as `robots.txt` and `sitemap.xml`, as well as the *DevTools*, are there to be thoroughly investigated.

In the next section, we will use some freely available web APIs. We will use Python to scrape their contents.

Web scraping using APIs

Technically, obtaining data from APIs is easy, and is also different from the web scraping scenarios. Most of the data is in JSON format, and there is no use of XPath, CSS Selector, or any other parsing libraries. Some data found using APIs might contain a lot of HTML code. To deal with this content, XPath and CSS Selector might be required.

> **Important note**
>
> In the examples in this section, we have tried to omit APIs that require an API key (`walmartlabs.com`, `nasa.gov`, `nytimes.com`, `maps.googleapis.com`) or user authentication tools. An API key is an authenticated value provided by service providers (after user registration) that identifies the user using the API. Freely available APIs are being used in these examples.

With the help of a few examples, we will collect data returned via some APIs. This collected data will then be exported as JSON and CSV files.

Example 1 – holidays from the US calendar

`https://date.nager.at/` contains information on worldwide public holidays. `https://date.nager.at/PublicHoliday/Country/US` displays a list of public holidays in the US. Tabular data is available for scraping, but we are going to use the `https://date.nager.at/api/v3/PublicHolidays/2023/US` API for the holidays in the year 2023.

The following code declares `apiUrl` with the content URL:

```
apiUrl= "https://date.nager.at/api/v3/PublicHolidays/2023/US"
headers= {           # HTTP Request Header
'accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,  image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7',
'accept-language': 'en-US,en;q=0.9',
'cache-control':'max-age=0',
'upgrade-insecure-requests':'1',
'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0 Safari/537.36'
}
response = requests.get(apiUrl, headers=headers).json()
```

The response is collected using the `json()` function from the `requests` library. HTTP request headers, `headers`, are also provided to `requests.get()`. Request header entities or `headers` have been collected from *DevTools* via **Copy | Copy as cUrl (bash)**.

As shown in *Figure 7.4*, `apiUrl` returns the HTTP response in JSON format. To deal with JSON content, the `requests` library has a method called `json()` (this does the loading and dumping of JSON content, so there's no need to import the `json` Python library) for processing JSON content.

Figure 7.4: JSON response from the API

The JSON response looks plain and straightforward and is easy to process. It's been decided to collect only `'date'` and `'name'` from all JSON listings.

The following code block shows that the `'date'` and `'name'` entries for each holiday are being collected in the `lists` collector:

```
lists=[]                    # empty list
for holiday in response:
    lists.append([holiday['date'],holiday['name']])

lists                       # final list
writeto_csv(lists,'holidays.csv',['Date','Name'])        # Export to CSV
```

Finally, `lists` is exported to a file named `holidays.csv` using the predefined `writeto_csv()` function, with `Date` and `Name` column headers.

Example 2 – Open Library book details

Open Library (`https://openlibrary.org/developers/api`) provides an experimental search API. It has various API-related options for its collection of books and similar content. For this example, `https://openlibrary.org/dev/docs/api/search` will be used to find book data with the subject `python` and the author `Wes` (the result is available via `https://openlibrary.org/search.json?q=python&author=Wes`).

As defined in the following code, `apiUrl` searches `openlibrary.org` for the defined `subject` and `author` variables. The response from `apiUrl` is in JSON format but is nested in structure. The HTTP request to `apiUrl` is processed along with a few selected HTTP headers. The `json()` method and `requests.get()` reveals that the content is in JSON format:

```
subject="python"
author="Wes"
apiUrl="https://openlibrary.org/search.
json?q="+subject+"&author="+author

headers={
'Upgrade-Insecure-Requests': '1',
'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0 Safari/537.36'
}
response = requests.get(apiUrl, headers=headers).json()
```

Information about books is provided under various indexes, titles, or keys, found inside `response['docs']`. Values for fields such as `'book_title'`, `'published_date'`, `'publisher'`, `'author'`, and `'subjects'` are collected (if available) in a list named `lists`:

```
lists=[]  # Collector
for book in response['docs']:
    if 'subject_key' in book: # check if subject_key exists
        lists.append([book['title'],
        book['publish_date'][0],
        book['publisher'][0],
        book['author_name'][0],
        "|".join(book['subject_key'])])
    else:
        lists.append([book['title'],
        book['publish_date'][0], book['publisher'][0],
        book['author_name'][0], ''])
```

With all the values collected, the data is then written to the CSV file `python_wes.csv`:

```
writeto_csv(
lists,
'python_wes.csv',
['book_title', 'published_date', 'publisher', 'author', 'subjects']
)
```

The collected data is shown in *Figure 7.5*:

book_title	published_date	publisher	author	subjects
Python For Data Analysis	Oct 04, 2022	O'Reilly Media, Incor…	Wes McKinney	data_mining|programming…
Python Programming f…	2019	Independently Publis…	Jason Wes	
Python for Data Analysis	2019	Independently Publis…	Clark Wes	
Python for Data Analysis	2012	O'Reilly Media, Incor…	Wes Mckinney	
Python for Data Science	2019	Independently Publis…	Jason Wes	
Python Programming L…	2019	Independently Publis…	Clark Wes	
Python Para Análise d…	Sep 07, 2000	Novatec	Wes McKinney	
Genuine _ use Python …	Jan 01, 2014	Machinery Industry P…	MAI JIN NI (Wes McK…	
Python Programming f…	2019	Independently Publis…	Tony Wes	
Python for Data Analys…	Oct 20, 2017	O'Reilly Media, Inc.	Wes McKinney	computers|data_mining|da…
Python for Data Analysis	2019	Independently Publis…	Tony Wes	

Figure 7.5: Book details from the search API

Though there were multiple values for `publish_date`, `publisher`, and `author_name`, only the first data to be found is collected. The API used in this example has many nested and similar data blocks. It's the developer's duty to decide which elements are to be captured.

Example 3 – US cities and time zones

In this example, we are using the free `https://api.travelpayouts.com/data/en/cities.json` API. This URL contains lists of global cities and their coordinates, country plus city codes, and time zones. Data from the API is in JSON format, which is converted to a list of Python `dict` objects. There are more than 9,000 records.

From the global records, we are only interested in collecting details for the US. Also, upon brief analysis, you will notice some different time zones for cities in the US, so we will be collecting data in two datasets. The `zone_america` dataset contains records that contain a time zone with the word

America, and the `zone_not_america` dataset contains records for those cities with a time zone that does not contain the word `America`, as shown in the following code block:

```
apiUrl="https://api.travelpayouts.com/data/en/cities.json
response = requests.get(apiUrl).json() # Load API
zone_america=[] # time zone like "America"
zone_not_america=[] # time zone without word "America"
for info in response:
      if info['country_code']=="US" and "America" in
      info['time_zone']:
            zone_america.append([info['country_code'],
            info['code'],info['name'],info['time_zone']])

      if info['country_code']=="US" and "America" not in
      info['time_zone']:
            zone_not_america.append([info['country_code'],
            info['code'],info['name'],info['time_zone']])
```

Data, as per the conditions implemented in the preceding code, is collected from the iteration. The data is written to the `zone_america.csv` and `zone_not_america.csv` files:

```
# Write collected data to CSV
writeto_csv(zone_america,'america_timezone.csv',['country_code','city_
code','city_name','time_zone'])
writeto_csv(zone_not_america,'not_america_timezone.csv',['country_
code','city_code','city_name','time_zone'])
```

The collected data is shown in *Figure 7.6*:

1 country_code,city_code,city_name,time_zone	1 country_code,city_code,city_name,time_zone
2 US,IYK,Inyokern,America/Los_Angeles	2 US,BLW,Waimanalo,Pacific/Honolulu
3 US,BTN,Bennettsville,America/New_York	3 US,HNM,Hana,Pacific/Honolulu
4 US,HTL,Houghton,America/Detroit	4 US,HDH,Oahu,Pacific/Honolulu
5 US,ABQ,Albuquerque,America/Denver	5 US,BSF,Pohakuloa,Pacific/Honolulu
6 US,WBN,Woburn,America/New_York	6 US,NAX,Kapolei,Pacific/Honolulu
7 US,BXK,Buckeye,America/Phoenix	7 US,MUE,Kamuela,Pacific/Honolulu
8 US,MHT,Manchester,America/New_York	8 US,JRF,Kapolei,Pacific/Honolulu
9 US,SPW,Spencer,America/Chicago	9 US,LUP,Kalaupapa,Pacific/Honolulu
10 US,LWS,Lewiston,America/Los_Angeles	10 US,BKH,Kekaha,Pacific/Honolulu
11 US,PCD,Prairie Du Chien,America/Chicago	11 US,JHM,Kapalua,Pacific/Honolulu
12 US,RRL,Merrill,America/Chicago	12 US,HNL,Honolulu,Pacific/Honolulu
13 US,CLT,Charlotte,America/New_York	13 US,WKL,Waikoloa,Pacific/Honolulu
14 US,AYE,Fort Devens,America/New_York	14 US,HHI,Wahiawa,Pacific/Honolulu
15 US,OXR,Ventura,America/Los_Angeles	15 US,OGG,Kahului,Pacific/Honolulu
16 US,CHA,Chattanooga,America/New_York	16 US,UPP,Upolu Point,Pacific/Honolulu
17 US,HCC,Hudson,America/New_York	17 US,ITO,Hilo,Pacific/Honolulu
18 US,WSX,Westsound,America/Los_Angeles	18 US,LNY,Lanai,Pacific/Honolulu
19 US,CUS,Columbus,America/Denver	19 US,KOA,Kailua-Kona,Pacific/Honolulu
20 US,BLU,Blue Canyon,America/Los_Angeles	20 US,USA,Concord,US/Eastern
21 US,BDL,Hartford,America/New_York	21 US,PAK,Hanapepe,Pacific/Honolulu
22 US,LUR,Cape Lisburne,America/Nome	22 US,MKK,Hoolehua,Pacific/Honolulu
23 US,KQA,Akutan,America/Nome	23 US,LIH,Kauai Island,Pacific/Honolulu
24 US,BNA,Nashville,America/Chicago	24
25 US,PCU,Picayune,America/Chicago	

Figure 7.6: US – time zones with city names and codes

In this example, we filtered the API results to return only cities in the US, and filtered the time zone text with and without the word `America`. With the desired data collected in a CSV file, it's much easier to perform data analysis tasks and deal with data visualization.

Summary

Web APIs provide many benefits for finding and scraping data. Because of the predefined format of data returned by APIs, you don't need to perform additional query-related tasks using XPath or CSS selectors, which saves time. Web APIs are structured, easily accessible, filterable, and ready to exchange data between machines, platforms, and protocols.

In the next chapter, we will be learning about automation using Selenium, and we will use it to scrape data.

Further reading

- APIs:

  - `https://restfulapi.net/`

  - `https://developer.mozilla.org/en-US/docs/Web/API`

  - `https://auth0.com/blog/developing-restful-apis-with-python-and-flask/`

- JSON:

  - `https://www.json.org/json-en.html,`

  - `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON`

- HTTP methods: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`

- SOAP: `https://www.soapui.org/learn/api/soap-vs-rest-api/`

- REST:

  - `https://www.ibm.com/topics/rest-apis,`

  - `https://pythonbasics.org/flask-rest-api/`

8

Using Selenium to Scrape the Web

So far, we have learned about some Python libraries, web and API-based technologies, data-finding and locating elements, extraction techniques, and plenty of data-related services in *Chapters 1* to *7*.

Selenium automates browsers – a quote from `https://www.selenium.dev/`, and it is primarily a collection of tools also known as a testing framework. Selenium is used to automate the web (applications, website forms, and much more) for testing purposes. Along with testing using automation, there are many potential service cum task-based scenarios that can be performed and handled using Selenium. The Selenium framework consists of various modules or components. We will be using **Selenium WebDriver**.

In general, we will install and learn about Selenium WebDriver, use WebDriver to automate websites, and use Selenium to scrape data from the web.

In this chapter, we will cover the following topics:

- Introduction to Selenium
- Using Selenium WebDriver
- Scraping using Selenium

Technical requirements

You will require a web browser (*Google Chrome* or *Mozilla Firefox*) and we will be using *JupyterLab* for Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections of *Chapter 2* to continue with setting up and using the environment created. We will be using **Google Chrome** with **Selenium WebDriver v4.10.0**.

To install the `selenium` Python library, the following links will be very helpful:

- `https://selenium-python.readthedocs.io/installation.html`

- `https://www.selenium.dev/documentation/webdriver/getting_started/install_library/`

- `https://pypi.org/project/selenium/`

We will require the following Python libraries for this chapter:

- `requests`

- `selenium`

- `re`

- `csv`

- `json`

The code files for this chapter are available online in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter08`.

Introduction to Selenium

The testing of web-based applications or systems is compulsory according to the **system development life cycle** (**SDLC**), and this step is done prior to the launch of applications on the web. Selenium, an open source project, uses a web browser as an interface for automation and can be used for web-related or web-based activities.

Dynamic and secure web applications using JavaScript (JS), cookies/sessions, other JS scripts, and many more web components can be handled, processed, automated, and crawled with the use of Selenium. "*Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.*" (`https://www.selenium.dev/documentation/overview/`). Though primarily used for browser-based automation, different cases that can be managed or used with a browser can be tackled using Selenium. This makes Selenium the most popular and appreciated automation-related browser-based tool.

Selenium is mostly used for web testing. In the following sections, we will explore the benefits and usages of Selenium, and how we can use it for web scraping.

Advantages and disadvantages of Selenium

Selenium supports different web browsers through one of its components, called WebDriver. There are many benefits that make Selenium popular, and a few are as follows:

- Easy to implement

- Cross-browser support

- Open source and free

- Supports parallel testing

- Supports multiple **operating systems** (**OSs**)

- Supports multiple languages (Java, Python, Ruby, PHP, and others)

- A huge collection of docs and resources is available

- Supports remote servers and cloud devices

Using and implementing Selenium in application testing has many advantages, but there also exist some limitations or disadvantages:

- Working with multiple tabs and frames

- Low execution speed (depending on the machine)

Despite the large number of advantages and extendibility that Selenium offers, it is slow in terms of processing compared to some of its competitors (**Playwright**, **Puppeteer**), and its large memory consumption is still a debatable issue, which is discussed in web-based groups and communities.

> **Important note**
> For more details on a few selected automation libraries and frameworks, please refer to Selenium `https://www.selenium.dev/about/`, Puppeteer `https://pptr.dev/`, and Playwright `https://playwright.dev/`.

Use cases of Selenium

There are plenty of cases for the use of Selenium in projects. From a web-scraping perspective, Selenium can be used in both normal and complex cases. For normal scraping cases, we can definitely use other libraries and techniques that we learned about earlier, in *Chapters 2* to *7*.

Selenium is used for automation and testing on the web. The use of Selenium is preferred (many times, even as a last option) in specific cases or when scraping is not possible with other libraries and techniques. Some of the cases in which the use or involvement of Selenium for scraping tasks may be required are as follows:

- Handling alerts, iframes, and popups (time-bound)

- Collecting and using cookies and sessions

- Addressing scrolling and clicking activity (ensuring anti-bot measures on websites)

- Working on JavaScript-based websites (websites with dynamic values or elements)

- Taking screenshots

- Using headless mode (less consumption of resources)

- Bypassing basic authentication (hidden or dynamic values)

- Dealing with HTML forms

- Executing and injecting JavaScript code

- Impersonating human action on a page

Selenium is a framework or collection of various task-based modules (libraries). We will be introducing these modules in the next section.

Components of Selenium

Application (web-based) testing is done at various stages of the development cycle and even multiple times. This type of testing ensures that the development is requirement-specific or is progressing as planned. It also helps to find any possible bugs or errors and to overcome or document them.

Generally, testing is done manually (by users) and/or using automated tools such as Selenium. Automating testing and verification tasks is one of the core components of the application life cycle. Selenium consists of the following three major components or projects (`https://www.selenium.dev/projects/`):

- **Selenium WebDriver**: This is an **Object-Oriented** (**OO**) API and one of the main components that is used to automate the browser. Browser automation is done by providing commands with the help of in-built APIs and other languages (JAVA, PHP, and Python) to third-party browser drivers such as Google Chrome, Mozilla, and Opera. **WebDriver** allows a program or script to access the browser in a native way, just like a user would. For scraping-related tasks, we normally deal with WebDriver, and we will explore its features in the *Using Selenium WebDriver* section. For more details on WebDriver, please visit this link: `https://www.selenium.dev/documentation/webdriver/`.

- **The Selenium IDE**: This is a ready-to-use browser extension, also known as a **User Interface** (**UI**) of Selenium that records user actions in the browser. The IDE also provides a feature that plays back a recorded action, along with the commands deployed and the defined parameters with values. It also has debugging features such as setting breakpoints, addressing exceptions through the IDE, creating scripts that can run commands, and supporting the control-flow structure. This extension is available for Google Chrome and Mozilla Firefox. For more details on the Selenium IDE, please visit this link: `https://www.selenium.dev/documentation/ide/`.

- **Selenium Grid**: This allows running tests or doing automation across multiple machines and browsers and cross-platform testing. Grid also allows component-based configuration that is deployed across machines or platforms. Basically, Selenium Grid supports distributable testing. This helps reduce the testing time and identify performance issues in parallel for different systems. For more details on Selenium Grid, please visit this link: `https://www.selenium.dev/documentation/grid/`.

With this brief overview of and introduction to Selenium, we will now install and explore Selenium WebDriver using code and examples in the next section.

Using Selenium WebDriver

Selenium is used for browser automation, and one of its major components, WebDriver, is the core tool to access browsers. WebDriver implements code logic for selected browsers that is required during automation. It's also the core system that binds the Selenium framework with the browser and often gets called or referred to as **Selenium driver** or only **driver**. For more detailed information, visit this link: `https://www.selenium.dev/documentation/webdriver/getting_started/`.

Before going deep into the automation or using the framework, let's install the required libraries in the next section.

Setting things up

To explore browser automation using Python and Selenium WebDriver, first, we need to install the `selenium` library (a Python library), and browser-related drivers.

> **Important note**
>
> Selenium is a framework that contains various components such as WebDriver and others, whereas `selenium` is a Python library (`https://www.selenium.dev/documentation/webdriver/getting_started/install_library/`) that we use to code and maintain logic that binds WebDriver and the selected browser.

Let's first verify the setup of the `selenium` library:

```
import selenium
selenium.__version__      # 4.10.0
selenium.__spec__
ModuleSpec(name='selenium',…… submodule_search_locations=['C:\\
HOWScraping2E\\secondEd\\Lib\\site-packages\\selenium'])
```

This code block shows that the Python `selenium` library with version `4.10.0` has been successfully installed in our target environment (please refer to the *Setting things up* and *Creating a virtual environment* sections of *Chapter 2*).

After verifying the Python library, we will install the drivers (browser drivers) in the next section.

Installing drivers

Before installing the browser drivers, identifying system specifications is essential. For the purpose of this book, we will use *Windows OS* and choose *Google Chrome* as our browser driver.

> **Important note**
>
> Selenium supports multiple web browsers, such as Firefox, Internet Explorer, Safari, Chrome, and others. If you are using Firefox, please visit the `https://firefox-source-docs.mozilla.org/testing/geckodriver/Support.html` link for more details on setup.

Let's follow these few steps to set up drivers for Google Chrome:

1. Visit `https://www.selenium.dev/downloads/` and go to the **Platforms supported by Selenium** section.

2. Now go to the **Browsers | Chrome** option. The documentation link to `https://chromedriver.chromium.org/` should be available.

3. Click the link with the **Getting started with ChromeDriver on Desktop** text, which is available under the **ChromeDriver Documentation** option.

    Clicking the link will land you on the `https://chromedriver.chromium.org/getting-started` page.

4. In the **Setup** section, there is a **Downloads** link. Clicking the **Downloads** text will route you to the `https://chromedriver.chromium.org/downloads` URL.

5. Go to the `https://chromedriver.chromium.org/downloads` URL, and click the link with the **ChromeDriver 111.0.5563.64** text, as shown in *Figure 8.1*:

Figure 8.1: Choosing the stable release for ChromeDriver

6. Clicking the highlighted link in *Figure 8.1* will route you to `https://chromedriver.` `storage.googleapis.com/index.html?path=111.0.5563.64/`.

7. Clicking the link in *step 6* will load a few options, as shown in *Figure 8.2*:

Figure 8.2: OS-based ChromeDriver options

8. Depending on the OS (Windows OS in our case), choose `chromedriver` as shown in *Figure 8.2*, and download it.

9. After successfully downloading the zipped file (`chromedriver_win32.zip`), it can be unzipped or the contents (`chromediver.exe`) can be extracted to the project root folder or any appropriate location. The path for `chromedriver.exe` can be set in a variable and used to develop code.

> **Important note**
>
> There are multiple ways to install and access `chromedriver` using `selenium`. Please follow the links provided for more detailed information: `https://selenium-python.readthedocs.io/installation.html` and `https://www.selenium.dev/documentation/webdriver/getting_started/install_drivers/`. Similar steps can be followed if any different browser is to be tried or tested.

With the setup-related steps completed, in the next section, we will verify the setup with the help of a code example.

Verifying the setup

During the setup stage, we have to visit and process a few links, download the files, and so on. To verify the setup and confirm that the driver is running fine, let us deploy the code that loads the `https://www.python.org` URL in the browser (Chrome) with the help of `selenium` and `chromedriver`:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
chromedriver_path="C:\HOWScraping2E\driver\chromedriver.exe"    # path
service = Service(service=chromedriver_path)
driver = webdriver.Chrome(service=service)
# initiate an empty Chrome window
driver.get('https://www.python.org')  # loads URL in browser
driver.quit() # closes browser & terminates the session
```

In this code block, we first imported `webdriver` and then `Service`. `selenium.webdriver` consists of various tools and classes. `Service` is one of the classes from `webdriver.chrome.service`. `Service` is mainly associated with browser-based features and functions. The path (location) to `chromedriver.exe` is important and has to be provided to the service.

> **Important note**
>
> Earlier versions of Selenium could accept the path to `chromedriver.exe` as an `executable_`
> `path` property. The `executable_path` property is now deprecated. Now, `Chrome()` asks
> for a path as a `Service` object.

`driver = webdriver.Chrome(service=service)` creates a new session and displays an
empty browser window as shown in *Figure 8.3*:

Figure 8.3: Empty browser window of Chrome

There is always an alert message at the top, containing the text **Chrome is being controlled by automated
test software**, as shown in *Figure 8.3*. The text conveys that Selenium WebDriver is working and that
Chrome is being used by the automation-related software.

Normally, the steps so far can be considered as loading the web browser on a machine. The `driver.`
`get('https://www.python.org')` code loads the URL (as if the user typed the URL in
the browser). The `get()` method accepts the URL and passes it to the driver almost as an HTTP
request. Finally, the `driver.quit()` code closes the browser window loaded with `https://`
`www.python.org` and terminates the driver session.

In the next section, we will explore `selenium.webdriver` and the `selenium` library in more
detail and with examples.

Exploring Selenium

Once the Selenium WebDriver has been successfully installed and is working fine, there are plenty of
activities that can be done with the help of automation. We will explore some basic things and look
in depth at using automation as a solution to problems, along with code examples.

> **Important note**
>
> For more details and explanatory documentation, please visit `https://www.selenium.dev/`
> `documentation/webdriver/` and `https://selenium-python.readthedocs.`
> `io/getting-started.html`.

Basic exploring

In the following code example, we will get an understanding of a few basic activities using Selenium that are normally priority tasks:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
chromedriver_path="C:\HOWScraping2E\driver\chromedriver.exe"
service = Service(service=chromedriver_path)
driver = webdriver.Chrome(service=service)
# chromedriver
```

Here, an empty, new Chrome window gets loaded similar to the browser window shown in *Figure 8.3*.

We will now supply three different URLs and iterate them by obtaining some information:

```
urls = {
    'google':'https://www.google.com',
    'python':'https://www.python.org',
    'selenium':'https://www.selenium.dev'
}
```

Now, let's iterate items() in the urls dictionary:

```
for key,url in urls.items():      # iterate the urls items
    driver.get(url)
    driver.implicitly_wait(1)     # Wait 1 sec
    print(driver.title)           # HTML <title>
    print(driver.current_url)     # Current URL
```

driver.get(url) loads the URL in the browser. The implicitly_wait(1) method has been used as the sleep time, waiting for the allotted 1 second before any further action is taken. The driver.title code returns the <title> HTML value. driver.current_url reveals the redirect URL or the latest URL that is being loaded in the browser.

Selenium also supports features related to screenshots. get_screenshot_as_file() is one of the common methods used for this purpose:

```
print(driver.get_cookies())
driver.get_screenshot_as_file(key+".png")  # png files
print(driver.page_source)     # HTML page source
driver.implicitly_wait(3)     # wait 3 sec before page Refresh
driver.refresh()              # refresh the page
```

`driver.get_cookies()` lists all the cookie-related values that exist in JSON format.

Receiving, updating, and setting cookie-related values is quite common to bypass some security features. Using `selenium` to obtain cookie values and process them is one of the major tasks from the web-scraping perspective.

`driver.page_source` returns the page source of the page or the HTML source of the page. Receiving `page_source` is also one of the significant features of Selenium. In `page_source`, from `selenium`, we can find values that are dynamically formed or generated using JavaScript. `page_source` content is required and can be parsed using `lxml`, `pyQuery`, `bs4`, and other Python libraries. The `refresh()` method acts like a browser-based page refresh button.

Browser-based history traversing buttons are handled by the `driver` methods `back()` and `forward()`:

```
driver.back() # history Python.org
driver.forward() # go forward
```

`back()` takes you to the previous page, whereas `forward()` takes you to the next page.

In this example, we used the driver-related features and functions. In the next section, we will find and locate HTML elements and deal with dynamic values.

HTML forms and elements, and JavaScript

Form (HTML `<form>`) management or processing is one of the priority tasks for automation, testing, and overall **Quality Analysis** (**QA**). Using Selenium for such tasks is handy in terms of time (iterating through values, parallel processing).

Although we will be using an automation tool, we need to identify elements using *CSS Selector*, *XPath*, or some new methods available in Selenium. The identification of HTML elements (`id`, `name`), their positions, and values is required to process them. For the following example, we will be using HTML `<form>` from `https://phptravels.com/demo`.

During analysis, it seems that the **Instant Demo Request Form** form (as shown in *Figure 8.4*) has a total of five `<input>` elements and a `<button>` element to submit information.

Figure 8.4: Instant Demo Request Form

The form also implements basic security. It asks for the sum of two numeric values to be submitted. These two numeric values, when browsed through the page source or **DevTools**, cannot be identified except for their elements (span#numb1, span#numb2), as shown in *Figure 8.5*.

```html
<input type="text" name="first_name" class="first_name input mb1" placeholder="First Name">
<input type="text" name="last_name" class="last_name input mb1" placeholder="Last Name">
<input type="text" name="business_name" class="business_name input mb1" placeholder="Business Name">
<input type="email" name="email" class="email input mb1" placeholder="Email">

<div class="row fcr">
    <div class="col-lg-6 mb1">
        <button id="demo" class="btn w100">Submit</button>
    </div>
    <div class="col-lg-6">
        <!-- <div style="margin-bottom:10px" class="g-recaptcha" data-sitekey="6LdX3JoUAAAAAFCG5tm0MFJaCF3LKxUN4p
    <div class="df">
        <h2 class="cw mw100" style="margin-top:10px"><span id="numb1"></span> + <span id="numb2"></span> = </h2>
        <input id="number" type="text" placeholder="Result ?">
    </div>

    </div>
</div>
```

Figure 8.5: Dynamic content numb1, numb2 (to submit)

The HTML elements <span id="numb1"> and <span id="numb2"> are both empty. Their values are generated by JavaScript dynamically, as shown in the code in *Figure 8.6*:

```
<script type="daf78653d796298cd9e56337-text/javascript">
      numb1 = Math.floor((Math.random() * 10) + 1);
      numb2 = Math.floor((Math.random() * 10) + 1);

      document.getElementById("numb1").innerHTML = numb1;
      document.getElementById("numb2").innerHTML = numb2;
</script>
```

Figure 8.6: JavaScript supplying random values to HTML elements

The JavaScript `document.getElementById("numb1").innerHTML = numb1` code sets or assigns the `numb1` value (some random numeric value) to the HTML element with `id="numb1"` (`<span id="numb1"></span>`). For example, if `numb1` has the value 10, `<span id="numb1"></span>` will be `<span id="numb1">10</span>`. Similarly, `numb2` will also be assigned a numeric value:

```
num1 = driver.find_element(By.ID,"numb1").text
num2 = driver.find_element(By.ID,"numb2").text
result = int(num1)+int(num2)
```

This code extracts the `text` string value from the `<span>` elements with IDs `numb1` and `numb2`. These text values are then converted to integers using the Python `int()` method:

```
driver.find_element(By.ID,"number").send_keys(result)
```

The sum of the two integers, `result`, is entered or provided to `<input>` with `id="number"` using `send_keys()`.

> **Important note**
> `send_keys()` is one of the interactive types of command that can be applied to text fields and elements with content. For more detailed information, please visit `https://www.selenium.dev/documentation/webdriver/elements/interactions/`.

`selenium.webdriver` provides various types of element locators to identify HTML elements and attributes associated with them. These locators are provided as arguments to the driver methods:

- `find_element()`: Returns a single element
- `find_elements()`: Returns multiple or lists of elements

Importing or using the `By` class (`from selenium.webdriver.common.by import By`), various locators and attributes can be found. A few of them are as follows:

- `By.ID`: Find elements with the `id` attribute, used as `driver find_element (By.ID,"numb1")` for `<span id="numb1">`

- `By.XPATH`: Find elements by providing XPath expressions, for example, `driver.find_element(By.XPATH,"[id='demo']")`

- `By.NAME`: Find elements with the `name` attribute, for example, `driver.find_element (By.NAME,"first_name")`

- `By.TAG_NAME`: Find elements with a tag name, for example, `driver.find_element (By.TAG_NAME,"h2")`

- `By.CLASS_NAME`: Find elements with the `class` attribute, for example, `driver.find_element(By.CLASS_NAME,"email")`

- `By.CSS_SELECTOR`: Find elements using CSS selector expressions, for example, `driver.find_element(By.CSS_SELECTOR,".completed > h2")`

- `By.LINK_TEXT`: Find elements from the links available and those that match the complete string provided, for example, `driver.find_element(By.LINK_TEXT,"Childrens")` will match the anchor tag or `<a>` that has the `"Childrens"` text

- `By.PARTIAL_LINK_TEXT`: Find elements from the links available and those that match a part or portion of the string provided, for example, `driver.find_element(By.PARTIAL_LINK_TEXT,"click")` will match the anchor tag or `<a>` that contains the text (or portion of the text) `click`

For more information about **XPath** and **CSS Selector**, please refer to the *Introducing XPath and CSS Selector to process markup documents* section of *Chapter 3*. Along with locators, there are also many supporting methods and attributes. Some of the common ones and their examples are as follows:

- `text`: Extracts or copies the content. For example, the `num1 = driver.find_element(By.ID,"numb1").text` code will extract the content from elements with `id="numb1"`.

- `tag_name`: Returns the HTML tag name from the provided locator. For example, `driver.find_element(By.ID,"numb1").tag_name` will return the tag name that is being referenced by the `id="numb1"` locator.

- `click()`: Performs interactions on a web page such as mouse clicks. For example, `driver.find_element(By.ID,"demo").click()` will click the element with `id="demo"`.

- `clear()`: Cleans or clears out the text from editable elements. For example, `driver.find_element(By.NAME,'first_name').clear()` will clean the text or values if available with `<input name='first_name'>`.

- `get_attribute()`: Returns the attribute value if available. For example, `driver.find(By.NAME,'first_name').get_attribute('class')` will return the `class` attribute value from elements with the `name="first_name"` attribute.

- `get_screenshot_as_file()`: Captures a screenshot of the designated area of a web page or the default screen. For example, `driver.get_screenshot_as_file('filename.png')` will capture a screenshot in the code location with the filename `'filename.png'`.

- `is_enabled()`: Returns a Boolean value (`True` or `False`) for the provided locator based on the element property enabled or disabled (*disabled form elements are not usable or cannot interact*). For example, `driver.find_element(By.NAME,'first_name').is_enabled()` will return `True`. HTML elements can carry a `disabled` attribute, for example, `<input type="text" value="something" disabled />`.

> **Important note**
> For more detailed listings and information on WebDriver with examples, please visit `https://www.selenium.dev/documentation/webdriver/`.

This section provided us with a selective overview of Selenium WebDriver (`selenium.webdriver`), how to use it, and how to explore various methods and properties with a few examples. From the scraping perspective, we need to bind or manage the data and information available from Selenium and other parsing libraries to the logical or implemented context.

In the next section, we will be using Selenium to scrape sites and collect data.

Scraping using Selenium

Selenium is used for automation – primarily web testing – using various browsers and coding in different languages. Along with automation, the benefits or features provided are quite handy and can be utilized in tasks such as web scraping.

In this section, we will use and explore quite a few features from the `selenium` library for web scraping.

Example 1 – book information

In this example, we will collect some details from the books listed in the **Childrens** category at the URL `http://books.toscrape.com`, which are available in the fictional bookstore at the `https://toscrape.com` URL.

In particular, we are searching for the anchor element `<a>`, which contains the `bookstore` text (partial text or a portion of the text) after loading `mainUrl`. With element `<a>` being traced, the `href` attribute from `<a>` can be collected using the `get_attribute()` method for `link`. The `click()` method clicks the element that contains the `bookstore` text:

```
mainUrl= "https://toscrape.com/"
driver.get(mainUrl)  # load mainUrl
link = driver.find_element(By.PARTIAL_LINK_TEXT,
    "bookstore").get_attribute('href')

link  # http://books.toscrape.com
driver.find_element(By.PARTIAL_LINK_TEXT,
    "bookstore").click()
```

We now have the `http://books.toscrape.com` page loaded in the existing browser window. The `http://books.toscrape.com` page contains default listings and many categories. Here, we will be strict, looking for the exact `Childrens` text in the categories available or listed. We will collect the URL of the element with the `Childrens` text and apply `click()` to it. On the element found, `click()` impersonates a user action such as choosing or moving through the content and clicking the mouse button:

```
categoryURL = driver.find_element(By.LINK_TEXT,
    "Childrens").get_attribute('href')
print(f"Category URL: {categoryURL}")
Category URL: http://books.toscrape.com/catalogue/category/books/
childrens_11/index.html
driver.find_element(By.LINK_TEXT, "Childrens").click()
# load category page
```

With the page link stored as `categoryURL` being loaded, we will now collect the books available on the page. During this example preparation, there was pagination in the `Childrens` category. The `find_elements()` method finds multiple elements that match the locator provided.

Each book listing is found as `<li>` inside parent tag `<ol>` with the `row` class. Iterating over available `<li>` list elements, various values are collected, such as `articleLink`, `imageSrc`, `price`, and more. Different types of locators, such as `CSS_SELECTOR`, `TAG_NAME`, and `CLASS_NAME`, are deployed depending on their suitability:

```
listings = driver.find_elements(By.CSS_SELECTOR,
    "ol.row li")# multiple element
for listing in listings:
    # Iterate the listing available in the page
    article = listing.find_element(By.TAG_NAME,'article')
    image = article.find_element(By.CSS_SELECTOR,"a")
    articleLink = image.get_attribute('href')
```

```
    imageSrc = image.find_element(By.TAG_NAME,
        'img').get_attribute('src')
    …..
    price = article.find_element(By.CLASS_NAME,
        "price_color").text
```

The `articleLink` variable contains the `href` value that links to the detail page of the particular book. Some information is available only on the detail page, such as *UPC* and *Availability (stock status, quantity)*. An XPath locator is also used. For example, `//th[contains(text(),'UPC')]/following-sibling::td` points to the `<td>` element, which is available after `<th>..</th>`, containing the text `UPC`. Please refer to the *Introducing XPath and CSS selectors to process markup documents* section of *Chapter 3*:

```
if articleLink:
    listing.find_element(By.TAG_NAME,'img').click()
    upc = driver.find_element(By.XPATH,
        "//th[contains(text(),'UPC')]/
            following-sibling::td").text

    stockQty= driver.find_element(By.XPATH,
        "//th[contains(text(),'Availability')]/
            following-sibling::td").text
```

So far, the code will have collected information from each book available in the listings and loaded the detail page. Data collected from the listings and the detail page is cleaned, preprocessed using the Python `strip()` and `replace()` methods, and added to a temporary collector or `temp` list. Finally, `temp` is added to the main `dataSet` collector:

```
temp = [upc, title, price,
    rating.replace('star-rating','').strip(),
    stockQty.split('(')[0].strip(),
    stockQty.split('(')[1].replace('available', '')
    .replace( ')', '').strip(), articleLink, imageSrc]
dataSet.append(temp) # append temp list details to main dataSet.
```

> **Important note**
>
> `strip()` cleans or removes the extra whitespaces from the left and right of a string. `replace()` is used to replace a character with something else. The `append()` method of Python's `list()` is used to add/push elements to the existing Python `list()` object. Please refer to `https://docs.python.org/3/library/index.html` for more details.

With the required data being collected, `driver.back()` loads the listing page again (it provides immediate history, similar to the back button available in the browser window) and iterates over the next `<li>` list element available. Selenium WebDriver supports browser history related controls with the `back()` and `forward()` methods:

```
# Go back to history (From individual book detail page to listings
page)
driver.back()
```

As there are multiple pages in the `Childrens` category, the code iterates for the pages with a `pagination (True)` value and searches for the link with the `next` text. Python exception handling is used in this example to verify whether the element exists or not with the `next` text:

- If the element exists, the `page` count is incremented and another listing page is loaded

- If the element does not exist, it is caught with the `NoSuchElementException` exception, which disables the page loop by updating the `pagination` Boolean value to `False`:

```
try:
    # Check for Pagination with text 'next' in the
      Listing page
    driver.find_element(By.LINK_TEXT,'next').click()
    page = page+1
except NoSuchElementException:
    pagination = False
    print(f "Further Pagination is not possible,
        currently at {page}")
```

> **Important note**
>
> For more information on exception handling, please visit the following links: `https://docs.python.org/3/tutorial/errors.html` and `https://selenium-python.readthedocs.io/api.html`.

Upon the completion of loops for pages and listings, the final data collected in `dataSet` is given planned column names listed in a list, `['Upc', 'Title', 'Price', 'Rating', 'Stock', 'Stock_Qty', 'Url', 'Image']`, and used to create a CSV and JSON file.

Example 2 – forms and searching

In *Example 2*, we will use Selenium to log in, log out, and go through available pages for collecting various quotes-based data from `http://quotes.toscrape.com/`. Similar to *Example 1*, we will locate and process the link that contains the `Login` text from `http://toscrape.com`.

The following code searches for an anchor element with the `Login` text and collects the `href` attribute. The `click()` action is applied to the `Login` element. This loads the `http://quotes.toscrape.com/login` page. WebDriver has a few attributes, such as `current_url`. This attribute returns the URL of the page loaded in the browser:

```
mainUrl=https://toscrape.com/
driver.get(mainUrl)  # loads toscrape.com
loginPage = driver.find_element(By.LINK_TEXT,
     "Login").get_attribute('href')
print(loginPage)
http://quotes.toscrape.com/login

driver.find_element(By.LINK_TEXT, "Login").click()

print(f"Before Login : {driver.current_url}")
http://quotes.toscrape.com/login
```

The element with the `Login` text is near the footer section of the page loaded, and we need to scroll down to view the element. If the browser window has a vertical scrollbar (depending on the dimensions of the window, if configured), the `driver.find_element(By.LINK_TEXT, "Login").click()` code will scroll down the page vertically till the element with the `Login` text is visible, and then click on it. This is a reduction of the window scrolling related code that had to be coded in versions before 4.1.0.

The `http://quotes.toscrape.com/login` page contains the HTML `<form>` element with a few inputs and a *submit* button. The form accepts any credentials, such as *username=test* and *password=test*. Since `<input>` types have to be filled with credentials, they are identified first, cleaned using `clear()`, and finally, the text value is entered using `send_keys()`. These actions automate scrolling or moving up to the element, clicking the element, and typing characters:

```
username = driver.find_element(By.ID, "username")
username.clear()  # cleans if there exist any characters
username.send_keys("test") # value entered
password = driver.find_element(By.ID, "password")
password.clear()
password.send_keys("test")
```

With `username` and `password` values provided to the respective input elements, form submission has to be done. Submit-related methods were available in versions before 4.0, but we can achieve the same thing by locating the button that submits form values and clicking it (using the `click()` method):

```
driver.find_element(By.CLASS_NAME,'btn').click()
# click submit button
quotesUrl = driver.current_url
print(f"After Login : {quotesUrl}") http://quotes.toscrape.com/
logoutUrl = driver.find_element(By.LINK_TEXT,
    "Logout").get_attribute('href')
print(f"Logout : {logoutUrl}") http://quotes.toscrape.com/logout
```

Upon submitting the login form, you will notice that there is a link with the `Logout` text and the `http://quotes.toscrape.com/logout` URL. This validates that the login-related step has been successfully completed.

> **Important note**
> Readers and developers will find the processing of the HTML form quite easy. The HTTP POST method is submitted with payload data (`username`, `password`, and `csrf_token`) if traced using **DevTools**.

There are multiple `quotes` `<div>` blocks on the page with the pagination element (`driver.find_element(By.CSS_SELECTOR,'li.next a').click()`). `quotes` identifies each individual block of the quote located using `find_elements()`. Pagination is managed using Boolean values and the required or planned data is extracted from each `quote` block. Finally, the data is added to the main `dataSet` collector:

```
quotes = driver.find_elements(By.CSS_SELECTOR,
    "div.row .quote")
for quote in quotes: # Iterate quotes available
    content =
        quote.find_element(By.CSS_SELECTOR,'.text').text
    author =
        quote.find_element(By.CLASS_NAME,"author").text
    authorLink =
        quote.find_element(By.PARTIAL_LINK_TEXT,"about")
            .get_attribute('href')
    authorGoodread =
        quote.find_element(By.PARTIAL_LINK_TEXT,"Goodread")
            .get_attribute('href')
    tags =
        quote.find_element(By.TAG_NAME,'meta')
            .get_attribute('content')
```

```
…….
# Add values to dataSet
dataSet.append([author, content, tags, tag_count,
    authorLink, authorGoodread])
```

With the completion of the pagination loop and quotes from each page, finally, the logout action is performed. After logout, the browser loads the `http://quotes.toscrape.com` page:

```
driver.get(logoutUrl)
print(f"Current URL: {driver.current_url}")
http://quotes.toscrape.com/
```

With data in hand or in the main `dataSet` collector, this data can be written as a CSV file (using the `'author','quote','tags','tag_count','author_url', 'goodread_url']` header) or a JSON file. Finally, the browser loaded by WebDriver is closed using `driver.quit()`.

The web-scraping examples in this section used various identifiers, locators, attributes, and methods from the `selenium` library. WebDriver is effective for automating tasks, either repetitive or non-repetitive. Web application-related testing is commonplace where Selenium is used, but we have explored it for scraping purposes. To explore Selenium further, please explore the content and links shared in this chapter.

Summary

The Selenium framework has many features and is widely used for application testing and browser automation. Exploring its features, we learned how to use WebDriver for scraping-related tasks. Python programming does have independent libraries to deal with HTML or web content, browser properties, networking, parsing, and more. Selenium can be used to process such features independently, and it is a major advantage that Selenium holds over various other Python libraries. The framework is also updated continuously, enriching the platform with testing, automation, and developer-friendly features.

In the next chapter, we will learn about regular expressions and using them to extract or scrape data.

Further reading

- *Selenium*:

  - `https://selenium-python.readthedocs.io/`

  - `https://developer.mozilla.org/en-US/docs/Web/HTML`

  - `https://www.lambdatest.com/blog/selenium-webdriver-tutorial-with-examples/`

- *Browser Automation*:

  - `https://www.browserstack.com/guide/what-is-browser-automation`
  - `https://community.dataquest.io/t/web-scraping-without-selenium/456297`

- *Automation Practice Site*:

  - `https://www.automationexercise.com/`
  - `https://practicetestautomation.com/practice-test-login/`

- *Chromium*: `https://chromedriver.chromium.org/documentation`

- *Puppeteer, Playwright*:

  - `https://pptr.dev/`
  - `https://playwright.dev/`

9

Using Regular Expressions and PDFs

So far, we have learned about and explored some of the core Python libraries in the context of web communication, content reading, and browser automation, for data finding and extraction.

Regular expressions (also referred to as **Regex**, **regex**, or **RegEx** – we will use **regex** throughout the rest of this chapter) are built using a predefined set of characters to form a pattern used for searching and similar activities. In *Chapters 3* and *4*, when carrying out web scraping, we tested and applied various available features, such as *CSS selectors*, *XPath*, and *PyQuery*, to find and locate specific types of activities. Regex helps us with pattern matching – we are knowingly or unknowingly using regex most of the time while working on documents or any textual content.

In a data-related context, it is very hard to avoid activities such as finding, searching, and matching. Regex provides us with a simple and elegant approach to dealing with such activities. Data is available in structured or unstructured formats in various forms and types of documents. **Portable Document Format** (**PDF**) is a secure, feature-enabled, nicely compressed document format that is easily transferred over the internet. PDF is different in comparison to other textual document formats. We will learn about data extraction from a PDF in this chapter.

In this chapter, we will learn about the following topics:

- Overview of regex
- Regex with Python
- Using regex to extract data
- Data extraction from a PDF

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required and we will be using *JupyterLab* for the Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections in *Chapter 2* to continue setting up and using the environment created. Refer to `https://pypdf2.readthedocs.io/en/3.0.0/user/installation.html` to install **PyPDF2**.

The Python libraries that are required for this chapter are as follows:

- `requests`

- `re`

- `pypdf2`

The code files for this chapter are available online in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter09`.

Overview of regex

There are plenty of cases when it's quite hard or even impossible to locate some web-based content or elements with XPath and CSS selectors. Fortunately, we can overcome such situations using a regex. A regex is an expression built using strings that is used to find or search content by identifying an existing pattern.

In web scraping and extraction-related activities, a regex is also used as a final or firsthand pattern-matching option. Patterns can be defined using various steps, often accompanied by special notations that represent predefined rules. A regex is like grouping and writing plain text, and many libraries and text-related features exist that use these expressions, providing us with handy, easy-to-use functions.

The latest code editors, document readers, and writing programs all provide facilities such as searching in files, multiple pages, and inside project folders, and using find and replace. To use these options, we need to input text or a regex; in any case, pattern matching or a regex will be used to search through the content.

A regex is usually used when other options haven't worked, or sometimes it's even the preferred method used. The scope can be extended to a few important features, such as the following:

- Splitting or breaking down the content into chunks, also known as exploding or slicing the contents into pieces

- Finding all or more than one match

- Substituting or replacing content based on a pattern

Listed here are some of the common use cases where regex is used extensively during data extraction:

- Validating email addresses

- Matching or collecting the zip code or postal code from addresses

- Matching the latitude and longitude from geo-addresses

- Finding the number of items listed on a page

- Verifying phone/mobile numbers

- Validating the date and time

- Extracting content using a generated (user-defined) pattern

Regex can be defined and run in most text-supporting applications or services. The quite simple, elegant-looking expressions can be used to deal with any type of use case. Its availability and scalability, depending on the content, make it a powerful text feature. Please visit `https://www.regular-expressions.info/` and `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions` for detailed information about regex.

> **Important note**
>
> The Unix/Linux-based **grep** (**global regular expression print**) command is very powerful and famous among Unix/Linux users and developers. `grep` is used to match and filter patterns and is even considered a pattern-matching engine. Please visit `https://www.gnu.org/software/grep/` for more detailed information.

In the next section, we will be exploring regex in more detail and using the `re` Python library in a practical example.

Regex with Python

Python programming is known for its simple, readable, reusable, and short code. Python is also popular because of its scientific computing and text computation features (**natural language processing** (**NLP**), **sentiment analysis** (**SA**), and many more). Regex is also one of the core powers of Python as `re` (the regex library) is provided as a system or built-in library that is available with Python installation.

Let's dive deep into `re` and have a look at some of the features using code. For this example, we will use the following famous quote from Sadhguru (available at `https://isha.sadhguru.org/us/en/wisdom/type/quotes`):

If you do not turn against yourself, the Human Potential is limitless.

We have defined a Python variable named `quote` that contains the preceding quote as its value:

```
quote="If you do not turn against yourself, the Human Potential is
limitless"
```

In the coming sections, we will find words that are at least three characters long or where the length of the string is three, using `re` methods that are common for data-related tasks.

re (search, match, and findall)

Python's `re` library provides various types of search-related methods. A few of the common ones that are generally used to search patterns within any code are listed here:

- `search()`: Looks for a match with a given pattern everywhere in the string and returns the first object that matches (`re.Match`). As seen in the preceding code, only `you` or the first match found in the `quote` string is returned.

- `match()`: This method looks for a match at the start of the string. In the case of the preceding code, it does not match anything. `re.match()`, on success, also returns various functions (`start`, `end`, `span`, `group`, `groups`, and more) and attributes (`lastindex` and `lastgroup`), which is informative in any case but more useful when using it with `re.finditer()`. Please visit https://docs.python.org/3/library/re.html?highlight=regular%20 expressions#functions for more detailed information.

- `findall()`: This method accepts a pattern to search for all the matches in the string. It's similar to iterating the `search()` method, from start to end.

In the upcoming subsections, we will be learning how to generate regex and apply it using examples.

Regex using a set of characters

In regex, we can look for a set of characters by using `[]`. As seen in the following code, `[a-z]` looks for any combination of words that are made up of characters from `a` to `z` :

```
re.search(r"([a-z]{3})",quote) # <re.Match object; span=(3, 6),
match='you'>
```

In the preceding code, to fix the length or check the occurrences, we use `{}`; so `{3}` in the expression `[a-z]{3}` is looking for a match of words that are made up of a maximum of three alphabetical characters (for example, *you*, *not*, and *the*).

Generally, regex patterns are provided as a string preceded by the `r` character. `r"[a-z]"` is a string but is also regex, and is also known as a raw string. In addition, we need to be careful about the character casing too (*lowercase or uppercase*) as regex patterns are case sensitive. In general cases, we

can apply `flags=re.IGNORECASE` to ignore the text case, as seen in the following example. We can also see that a combination of all three characters is being returned in the following code block:

```
re.findall("([a-z]{3})",quote)
# ['you', 'not', 'tur', 'aga', 'ins', 'you', 'rse', 'the', 'uma',
'ote', 'nti', 'lim', 'itl', 'ess']
re.findall(r"([a-z]{3})",quote, flags=re.IGNORECASE)
# ['you', 'not', 'tur', 'aga', 'ins', 'you', 'rse', 'the', 'Hum',
'Pot', 'ent', 'ial', 'lim', 'itl', 'ess']
```

Regex using escaped code

Apart from mentioning a set of characters, we can also use a sequence of characters. For example, \w represents any alphanumeric character and \w{3} represents any word with a total of three characters, as shown here:

```
re.findall(r"\w{3}",quote)
#['you', 'not', 'tur', 'aga', 'ins', 'you', 'rse', 'the', 'Hum',
'Pot', 'ent', 'ial', 'lim', 'itl', 'ess']
```

Similarly, \s matches whitespaces (space, tab, and newlines). The number of repetitions and occurrences can also be determined with the quantifiers listed here:

- `*`: Zero or more of the words in the expression.

- `+`: One or more than one of the words in the provided expression.

- `?`: Matches zero or one of the words in the expression (also known as a lazy quantifier).

- `{,}`: Matches unlimited characters. Occurrences work with a scope of {`minimum, maximum`}. Here are a few examples:

  - `{2,10}`: Matches from 2 to 10 characters

  - `{,5}`: Matches up to five characters

  - `{2,}`: A minimum of two characters

  - `{3}`: Match occurrences of every three character

Regex using concatenation

Another major benefit of regex is that it can be concatenated. Take the following examples:

- `\s+(\w{3})\s+`: Matches any words that are three characters in length and have at least one or more space before and at the end of the word

- `\s*(\w{3})\s*`: Matches any words that are three characters in length and have zero or multiple spaces before and at the end of the word

In the following code, lots of regex have been used (regarding the length, set, and occurrences of the characters):

```
re.findall(r"\s*(\w{3})\s*",quote)
# ['you', 'not', 'tur', 'aga', 'ins', 'you', 'rse', 'the', 'Hum',
'Pot', 'ent', 'ial', 'lim', 'itl', 'ess']
re.findall(r"\s+(\w{3})\s+",quote)          # ['you', 'not', 'the']
re.findall(r"\s+([a-z]{3})\s+",quote)         # ['you', 'not', 'the']
```

Here is a brief explanation of some of the regex used in the preceding code:

- Set of characters or [] (square brackets):

  - [A-Z]: Set of uppercase characters

  - [a-z]: Set of lowercase characters

  - [0-9]: Set of numeric characters from 0 to 9

> **Important note**
>
> The expression [a-zA-Z0-9] matches all alphanumeric characters. In addition, in regex, characters other than [A-Z0-9a-z] are written as escaped characters (\, for comma, \? for the question mark character, and \+ for the + character). The set of characters can also contain special or escape characters; for example, [0-9\+\-\.] means containing the numbers 0-9 and the +, -, and . characters or any combination of these numbers and characters.

- Round brackets () within the expression are used to hold a group of matching values. For example, r"([a-z]+)[0-9]{3}" holds values matching one or more characters from a to z.

- Escape code, sometimes also known as predefined shortcuts, is escaped with \. Here are some examples of escape code:

  - \s: Represents space (keyboard space bar) characters or whitespaces; for example, \s+ adds one or more spaces

  - \n: Newline character, used with multi-line content

  - \t: Tab character

  - \S: Non-whitespace

  - \w: Matches alphanumeric characters; is similar to [a-z0-9A-Z]

  - \W: Matches non-alphanumeric characters; is the opposite of \w

  - \b: Matches the word boundary

  - \B: Matches the non-word boundary

- \d: Matches a digit or [0-9]

- \D: Matches a non-digit

Important note

Escape characters or code are to be used carefully as they are case-sensitive. For example, \w and \W have the completely opposite effect.

Regex also supports the OR logical operation by using | (pipe) in the pattern. In the expression r"([A-Z]\w+ | [a-z]\w+)", it matches [A-Z]\w+ or [a-z]\w+, which results in returning all words in any case:

```
matches = re.findall(r"([A-Z]\w+|[a-z]\w+)",quote)
matches # ['If', 'you', 'do', 'not', 'turn', 'against', 'yourself',
'the', 'Human', 'Potential', 'is', 'limitless']
```

Besides the features of regex explored in this section, there are some more important methods and expression syntaxes. To explore this further, let us count the total words in the quote string in the next section.

In the next section, we will use re's split() method to create chunks based on matched expressions or provided criteria such as characters.

re.split

An example of re.split() working on an expression is provided in the following code. The Python split() string-based method uses space as the default splitting character. As seen in the following code, \s or space is used to split all words in the quote, as we mentioned in the *Overview of regex* section:

```
words = re.split("\s",quote)
words
# ['If', 'you', 'do', 'not', 'turn', 'against', 'yourself', 'the',
'Human', 'Potential', 'is',  'limitless']
print(f"Total words in quote: {len(words)}")
Total words in quote: 12
wordsA = re.split("\,",quote)
# ['If you do not turn against yourself', ' the Human Potential is
limitless']
print(f"Total words in wordsA: {len(wordsA)}")
Total words in wordsA: 2
```

Python's re also supports replacement or substitution using the sub() method. In the next section, we will use re.sub().

re.sub

`re.sub()` works exactly the same as find and replace or the string `replace()` method. In the following code block, the `r"(H[a-z]+)"` expression matches any word starting with `H` and a combination of one or more characters in the range `[a-z]` and is replaced with `HumanBeing`:

```
newQuote = re.sub(r"(H[a-z]+)",'HumanBeing',quote)
newQuote
# If you do not turn against yourself, the HumanBeing Potential is
limitless
```

In the following code, `\,` matches `,` (comma), which is replaced, using `sub()`, with a null or empty value:

```
newQuoteA = re.sub(r"\,",'',quote)
# If you do not turn against yourself the Human Potential is limitless
```

There's also a method that returns an object based on our pattern, called `compile()`. In the next section, we will be exploring `re.compile()` along with some important regex fundamentals.

re.compile

As the method name suggests, `re.compile()` compiles a regex and creates a pattern object. Using a pattern object is not compulsory but it makes the code more readable and scalable.

In the following code, we have a Python list called `languages`, containing the names of various programming languages:

```
languages = ["Javascript","Python","Go","Java","Kotlin",
"PHP","C#","Swift","R", "Ruby","C","C++","Matlab",
"TypeScript","Scala","SQL","HTML","CSS","NoSQL", "Rust", "Perl"]
vowel_start = r"^[AEIOU]"
vowel_end = r".*[aeiouAEIOU]$"
print(f" Expression {vowel_start}, Type:{type(vowel_start)}")
# Expression ^[AEIOU], Type:<class 'str'>
```

We have defined two regex patterns, as listed here:

- `vowel_start`: Checks whether the elements in `languages` start with a vowel (*case-sensitive*)
- `vowel_end`: Checks whether elements in `languages` end with vowels (*case-insensitive*)

It's also to be noted that although `vowel_start` and `vowel_end` are defined to hold regex, they are still string objects.

Listed here are a few regex characters, as found in `vowel_start` and `vowel_end`:

- `^` (caret): This is a positional character. It matches the start position or beginning of the string. For example, `(^\d+)` looks for any pattern that starts with a digit.

  When `^` is used inside a set of characters, it works as a *negation or exclusion*. For example, `[a-z]` matches characters from *a to z*, whereas `[^a-z]` matches anything except for characters from *a to z*.

- `$`: This is also a positional character, which matches the end position. For example, `(\d+$)` matches a string that ends in a digit. The `r".*[aeiouAEIOU]$"` expression looks for an end character that matches any of the following: `aeiouAEIOU`.

- `.` (dot or period): This matches any character but not a newline or `\n`. For example, `r".*[aeiouAEIOU]$"` matches any word that has any number of characters and ends in one of the characters `aeiouAEIOU` as the ending character.

Having defined the expressions, let's compile them. `re.compile(expression)` creates an object of the `re.Pattern` (`patternS` and `patternE`) type:

```
patternS = re.compile(vowel_start)
patternE = re.compile(vowel_end)
print(f"{patternS}, {type(patternS)}")
re.compile('^[AEIOU]'), <class 're.Pattern'>

print(dir(patternS))
['__class__',….', 'findall', 'finditer', 'flags', 'fullmatch',
'groupindex', 'groups', 'match', 'pattern', 'scanner', 'search',
'split', 'sub', 'subn']
```

The `compile()` method is provided with instructions in the form of regex. After compilation, we can use the compiled set of expressions where necessary, as seen in the preceding and following code blocks.

As seen in the following code, list iteration is performed to check the patterns using `re.match()`, which results in only three matches, that is, the `language` names that end in vowels:

```
for language in languages:
    if re.match(patternS, language):
        print(f"{language} starts with vowel character")
    if re.match(patternE, language):
        print(f"{language} ends with vowel character")
Output----
Go ends with vowel character
Java ends with vowel character
Scala ends with vowel character
```

Pattern objects possess various methods. A few of them have been used in the preceding code. There are plenty more that can be used, for which a complete study of regex is required. For more details, visit `https://www.regular-expressions.info/`.

Building regex is quite challenging. Any particular task can be handled with one or more expressions. There is no correct way to write expressions, but learning the basics of regex will be helpful.

In the next section, we will see multiple ways of handling entities that are written using regex, with `re` functions, and the logical steps to follow.

Regex flags

Regex flags are often considered additional top-ups or enhancements over basic expressions. There are various types of flags available, but the most common ones are as follows:

- `re.IGNORECASE`: This flag ignores the case-sensitive (lower or uppercase) issue, a short version can be used (`re.I`), and it can be used inline with (`?i`).

- `re.MULTILINE`: This flag helps to search and match a pattern in an expression that is distributed across multiple lines. It can be shortened to `re.M` or written as (`?m`) inline.

As seen in the following code block, the `sentence` Python variable is a multi-line string if declared with HTTP header-related values. There are also newline (`\n`) characters. Let's explore various ways of achieving the regex results, with and without flags:

```
sentence="""Accept:text/html,application/xhtml+xml,application/
xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/
signed-exchange;v=b3;q=0.7\n Accept-Encoding: gzip, deflate\nAccept-
Language: en-US,en;q=0.9\nCache-Control: max-age=0\nConnection: keep-
alive\nCookie: ci_session=%22session_id%22404495f061c71aca87121e\
nHost: anishchapagain.com\nIf-Modified-Since: Sun, 1 Apr 2023
11:22:33 GMT\nIf-None-Match: 64a-5f9724a6cdcf2-gzip\nUpgrade-
Insecure-Requests: 1\nUser-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0
Safari/537.36"""
```

Python `re` methods accept the `flags` parameter. As seen in the following code, in `re.findall("(if.*)",sentence, flags = re.IGNORECASE)`, `re.IGNORECASE` is passed to the `flags` argument:

```
re.findall("(if.*)",sentence) ['if,image/webp,image/
apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7', 'ified-Since:
Sun, 1 Apr 2023 11:22:33 GMT']
re.findall("(if.*)",sentence, flags = re.IGNORECASE) ['if,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
'If-Modified-Since: Sun, 1 Apr 2023 11:22:33 GMT','If-None-Match:
64a-5f9724a6cdcf2-gzip']
re.findall("(If.*)",sentence)
['If-Modified-Since: Sun, 1 Apr 2023 11:22:33 GMT', 'If-None-Match:
64a-5f9724a6cdcf2-gzip']
```

The difference in the output can be seen in the preceding code block when `re.IGNORECASE` is supplied.

Apart from the output, we can also modify or change the preceding code with some short forms; for example, `re.findall("(if.*)",sentence,flags = re.IGNORECASE)` can also be written as follows:

- `re.findall("(if.*)",sentence, re.IGNORECASE)`

- `re.findall("(if.*)",sentence, flags = re.I)`

- `re.findall("(if.*)",sentence, re.I)`

- `re.findall(r"(?i)(if.*)",sentence)` (this is inline flag notation; `(?i)` means the same as `re.I`)

In the following code, the `re.MULTILINE` flag is used in a multi-line string and outputs can be easily distinguished with and without the use of the `flag` regex:

```
re.findall (r"^Accept.*", sentence)
['Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7']
re.findall (r"^Accept.*", sentence, flags = re.MULTILINE)
['Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7', 'Accept-Encoding: gzip, deflate', 'Accept-
Language: en-US,en;q=0.9']
re.findall(r"(?m)^Accept.*",sentence) # inline flag (?m) multi-line
['Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7', 'Accept-Encoding: gzip, deflate', 'Accept-
Language: en-US,en;q=0.9']
```

The `re.findall (r"^Accept.*",sentence,flags=re.MULTILINE)` expression can also be written as follows:

- `re.findall(r"^Accept.*",sentence, re.MULTILINE)`

- `re.findall(r"^Accept.*",sentence, flags = re.M)`

- `re.findall(r"^Accept.*",sentence, re.M)`

- `re.findall(r"(?m)^Accept.*",sentence)` (inline flag)

Flags can also be applied together; for example, `flags=re.MULTILINE|re.IGNORECASE` will impose both flags, as shown here:

```
re.findall(r"^accept.*",sentence, re.MULTILINE|re.IGNORECASE)
re.findall(r"^accept.*",sentence, re.M|re.I)
re.findall(r"(?im)^accept.*",sentence) # (?im) the inline flag
together
```

The preceding code will result in the following output:

```
['Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.7',
'Accept-Encoding: gzip, deflate',
'Accept-Language: en-US,en;q=0.9']
```

As seen in the preceding code blocks, inline flags are mentioned prior to the expression and preceded with ?, for example, (?i) or (?im).

> **Important note**
>
> For more detailed information on writing expressions, flags, and regex in general, do explore these links: `https://www.regular-expressions.info/`, `https://regex101.com/` (you can use this link to practice and generate code too), `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions/Cheatsheet`, and `https://learnbyexample.github.io/python-regex-cheatsheet`.

In the next section, we will be using regex in web responses for data extraction.

Using regex to extract data

In the previous sections of this chapter, we explored various aspects of regex, with examples. Regex can be applied to all types of content – such as content analysis, extendibility, and time and resource (machine) analysis. This analysis is important to figure out which extraction-related options to choose, such as *XPath*, *CSS selectors*, and *PyQuery*.

> **Important note**
>
> It's often mentioned in the literature that regex should only be applied when the content is unstructured (for data extraction), but this is not the case. Regex can be used in any type of content (structured or unstructured).

To extract data, from a scraping point of view, we'll explore a few examples using regex and explore some of its functionality and properties.

Example 1 – Yamaha dealer information

In this example, we will be collecting information on motor dealers (dealers' geo-location, more precisely) from `https://yamaha-moto.cfaomotors.ng/en/dealership/yamaha-nigeria-cfao-motors`. The data to be collected is `Dealer_ID`, `Name (Dealer Name)`, `City (Location)`, `Latitude`, and `Longitude`.

To begin with, the page content or source is collected using the Python `requests` library:

```
url="https://yamaha-moto.cfaomotors.ng/en/dealership/yamaha-nigeria-
cfaomotors"
source = requests.get(url).text
```

In this case, because of the use of regex, we can find the desired data within the HTML tags (the page source of the `url` variable). The dealer-related content is found inside the `<script type="text/javascript">...</script>` tags in the page source. There were only a few dealers listed; so, as seen in the following code, using regex makes it easier to collect the dealer data. With pattern identification, `map_markers` is formed:

```
map_markers = re.findall(r"map\_markers.*\[\[(.*)\]\]\;",source)
```

The `map_markers` Python list returns a successful result with dealer data in a list using `re.findall()`:

```
AUSTINE BEST VENTURES LTD - Igbo-Ora",7.435186,3.2874431,"261","\/
media\/gamme\/marques\/graph\/8-map-pointer-yamaha-
moto.png"], ["BLESSED DEN-RACH DYNAMIC VENTURES LTD -
Enugu",6.9162632,7.5184719,"257","\/media\/gamme\/marques\/graph\/8-
map-pointer-yamaha-moto.png"],["OLUSOLA COMMERCIAL ENTERPRISES - Ib
adan",7.478631,3.9137284999999,"260","\/media\/gamme\/marques\/
graph\/8-map-pointer-yamaha-moto.png"],["S. AKINMADE NIGERIA LTD - Aku
re",7.2455287,5.1887157,"259","\/media\/gamme\/marques\/graph\/8-map-
pointer-yamaha-moto.png"
```

Each dealer's data was returned in an incomplete Python list structure or separated by `]`,. With the separation logic in hand, splitting or breaking down the `map_markers` data into pieces will be a suitable option. This is done using `re.split()`, as seen in the following code:

```
markers = re.split(r"\]\,", map_markers[0])
```

The `markers` Python list returns multiple dealers' data. It requires iteration to collect individual dealers' data.

As seen in the `map_markers` list in the preceding code, the text is not normal. It contains characters that need to be cleaned using `re.sub(r"\[|\"|\'",'',marker)`. Cleaned strings are further broken apart with `re.split(r"\,",marker)` for a list containing data, and finally added to `dataSet` or the container for the dealer data:

```
for marker in markers:
    marker = re.sub(r"\[|\"|\'",'',marker)  # cleaning
    details = re.split(r"\,",marker)  # break apart
    …..
    nameCity = re.split("\s+\-\s+",details[0])
    name = nameCity[0]
    city = nameCity[1]
    dataSet.append([id,name,city, lat,lng])
```

`dataSet` will result in the data looking as shown here:

```
[['261', 'AUSTINE BEST VENTURES LTD', 'Igbo-Ora', '7.435186',
'3.2874431'],
….
['259', 'S. AKINMADE NIGERIA LTD', 'Akure', '7.2455287', '5.1887157']]
```

With all the data collected, CSV and JSON files are created.

As seen in the preceding code blocks, re's `findall`, `sub`, and `split` methods are used in multiple places and at different times, and play a very important role in data extraction.

Example 2 – data from sitemap

In this example, we will extract data from sitemap links available at `https://www.zyte.com/post-sitemap.xml`.

As the sitemap file is in JSON format, it contains some system and user-defined tags. `source` collects the web response using Python `requests`:

```
url="https://www.zyte.com/post-sitemap.xml"
source = requests.get(url).text
```

The portion of the output from `source` or the content for a single link found in `sitemap.xml` looks as in the following code block – there are multiple links with similar output that are to be identified, extracted, and collected for extracting required data:

```
<url>
    <loc>https://www.zyte.com/blog/json-parsing-with-
        python/</loc>
    <lastmod>2023-04-06T14:42:05+00:00</lastmod>
</url>
```

The data we are interested in from the links is as follows:

- **Type**: The category, such as a blog, webinar, or uncategorized text extracted from `<loc>`

- **Topic**: Link text or title text available in links, which is displayed after the type, which we defined previously

- **Date**: The Y-M-D value found in `<lastmod>`

- **Time**: The H:M:S value in `<lastmod>`

Here, as there are identifiable XML tags (`<loc>` and `<lastmod>` for each link or `<url>`) in `source`, we will collect all available `<loc>` and `<lastmod>` details separately. `locs` collects

the data from all available `<loc>..</loc>` markups; similarly, `mods` lists the content from all `<lastmod>..</lastmod>` markups:

```
locs = re.findall(r"\<loc\>(.*)\<\/loc\>",source)
mods = re.findall(r"mod\>(.*)\<\/last",source)
```

`locFinals` is a Python list that contains individual elements from `locs` and `mods` as a tuple. The `zip(locs, mods)` Python function works as a merger that combines the argument passed to it and creates a tuple with the value from iterated `locs` and `mods`:

```
locFinals = list(zip(locs,mods))
print(locFinals[0])    # (locs[0], mods[0])
('https://www.zyte.com/blog/json-parsing-with-python/', '2023-04-
06T14:42:05+00:00')
```

`locFinals` has multiple elements. Iteration is initiated along with unpacking the tuple and cleaning plus splitting the data (using `re.sub()` and `re.split()`) into chunks to retrieve the desired data:

```
for locFinal in locFinals:
    loc, datetime = locFinal
# unpacking tuple (extracting tuple elements)
    loc = re.sub('^(http\:|https\:)','',loc)
# cleaning
    loc = re.sub('[\/]{2}','',loc)
    loc = re.sub('\/$','',loc)
    locSplit=re.split(r"\/",loc)
# breaking
    category=locSplit[1]
    categoryTopic=re.sub('\-',' ',locSplit[2])
    ymdTime = re.split("T", datetime)
# separating date and time
    ymd = ymdTime[0]
    time = re.sub("\+.*","",ymdTime[1])
# cleaning
```

The identified desired data is finally loaded into the `dataSet` data container:

```
    dataSet.append(["Zyte", category, categoryTopic, ymd,
        time])
print(dataSet)
[['Zyte', 'blog', 'json parsing with python', '2023-04-06',
'14:42:05'],….,
['Zyte', 'webinars', 'the right data fields for e commerce data
project', '2023-03-21',
'06:17:35'],……]
```

As seen in this example, `re` methods (`findall()`, `sub()`, and `split()`) and the `zip()` Python method play a major role. Regex-based extraction can be done effectively and easily by engaging Python `re` and in-built methods.

Example 3 – Godfrey's dealer

In this example, we will extract dealer details from `https://godfreysfeed.com/dealersandlocations`.

An HTTP request is initiated using the `requests` Python library, and the response is collected as `source`:

```
url="https://godfreysfeed.com/dealersandlocations"
source = requests.get(url).text
```

From `source`, we will be obtaining the data related to the desired columns, such as `Name`, `Address`, `City`, `State`, `Zip`, `Lat`, and `Lng`. Dealer-related data is only available in *JavaScript*. The data is also scattered across multiple lines, and it contains some HTML tags, as seen here:

```
var infoWindowContent = "<div style='overflow:hidden; width:200px'>";
infoWindowContent = infoWindowContent+ "<strong><span
style='color:#e5011c;'>American Cowboy Shop</span></strong><br>
<strong>513 D Murphy Hwy</strong><br><strong>Blairsville, GA</
strong><br> <strong>30512</strong><br><br>";
infoWindowContent+='</div>';
var infowindow = new google.maps.InfoWindow({maxWidth: 450});
infowindow.setContent(infoWindowContent);
var latLng = new google.maps.LatLng(34.8752421, -83.9716038);
```

The regex is defined to collect all coordinate data with the `latlngs` variable, and `contents` holds the address-carrying `infoWindowsContent` variable. Finally, `details` is created using Python's built-in `zip()`:

```
latlngs = re.findall(r"var\s+latLng.*\.LatLng\((.*)\)\;", source)
contents = re.findall(r"infoWindowContent\+\s*\"(.*)\"\;", source)
details = list(zip(contents, latlngs))    #zip
```

Now, we have identified `latlngs` and `contents` data in `details`. Iteration in `details` is required, which unpacks the `detail` tuple into a block of `fullAddress` and `coordinate`:

```
for detail in details:
    fullAddress, coordinate = detail
```

Further cleaning and splitting of `fullAddress` and `coordinate`-related content are required to get the exact data that we are seeking, as shown here:

```
fullAddress = re.sub('\<br\>','|',fullAddress)
# cleans <br>
fullAddress = re.sub('\|\|$','',fullAddress)
# cleans || at end
fullAddress = re.sub('(\<\/?\w+\>)','', fullAddress)
fullAddress = re.sub('(\<\/?.*\>)', '', fullAddress)
# cleans ending tags
fullAddress = re.split('\|',fullAddress)
name = fullAddress[0]
address = fullAddress[1]
city = re.split("\,",fullAddress[2])[0]
state = re.split("\,",fullAddress[2])[1].strip()
zipCode = fullAddress[3]
coordinates=re.split(r"\,",coordinate)
lat=coordinates[0]
lng=coordinates[1].strip()
```

The desired data is obtained and added to the `dataSet` data container:

```
dataSet.append([name, address, city, state, zipCode,
    lat, lng])
print(dataSet[1])
['American Cowboy Shop', '513 D Murphy Hwy','Blairsville', 'GA',
'30512','34.8752421', '-83.9716038']
```

As found again in this example, `re` methods (`findall()`, `sub()`, and `split()`) and the Python `zip()` function play a major role. Regex-based extraction seems short, targeting the exact content, and can be done effectively and easily by engaging Python `re` and using Python's in-built methods.

Regex is a powerful tool, applicable in all sorts of circumstances, and in the case of collecting data, it works as well as the parsing tools. Regex, being one of the oldest but most popular technologies, is still the favorite among developers because of its diverse applicability and usability. There are many Python libraries, browser-based extensions, CLIs, and query-driven applications that use regex as their underlying technology.

In the next section, we will be using Python's PDF-related library to extract data from PDF files.

Data extraction from a PDF

PDF is a rich (in terms of containing document features and formatting) document format that can be created, shared, and accessed on any supporting device. It is not an understatement to state that PDF files are everywhere, supported by all kinds of electronic devices and systems. It is also quite useful

to know that *Word documents, PowerPoint presentations, HTML, Jupyter notebooks, analysis reports from various applications*, and many more content types support exporting and saving files as PDF.

We often find various types of data (such as textual, tabular, and images) in a PDF file. In *Chapters 3* and *4*, we saw how to extract web-based content using Python. Here, we will be using the `PyPDF2` Python library to extract data from PDF files.

In the next sections, we will install and explore `PyPDF2` from a data extraction perspective.

The PyPDF2 library

`PyPDF2` (`https://pypdf2.readthedocs.io/en/3.0.0/index.html`) is a free, open source (`https://github.com/py-pdf/PyPDF2`) Python library to work with PDF files and is quite famous among Python developers and the community because of its various features.

Listed here are a few of its major functionalities:

- Text extraction
- Image extraction
- Metadata (document details) availability and extraction
- File conversion (PDF to Word, and more)
- Text modification (PDF files)
- Adding a watermark to existing or new PDF files
- Adding security, such as password-protecting PDF files
- Splitting a PDF file into pages
- Merging PDF files into one PDF
- Dealing with the page layout, cropping, transformation, and orientation (such as rotating the page)

It should also be noted that PDF files can be created using PDF applications or exported and saved from another document format. Though PDF files contain the proper formatting of the original document, their content is not like HTML or markup-based documents. PDF contents are available in raw format (with line breaks, spaces, and tabs), and the use of regex is almost compulsory to target and extract the desired information.

> **Important note**
> There are plenty of Python libraries that deal with PDF files, such as `pdfminer`, `pdfquery`, `xpdf`, `pdfrw`, `pikepdf`, and `pymupdf`. PyPDF2 is my favorite among those listed because of its various features. As explained at `https://pypdf2.readthedocs.io/en/3.0.0/user/installation.html`, PyPDF2 can be used for broad use cases, or can just be focused on a domain, such as *crypto (encrypting or decrypting) and image processing*.

In the next section, we will use `PyPDF2` to collect data.

Extraction using PyPDF2

`PyPDF2` is a complete package; there are various classes to deal with its different features. We will be dealing with the `PdfReader` class (`https://pypdf2.readthedocs.io/en/3.0.0/modules/PdfReader.html`) from `PyPDF2`, for extraction purposes.

As seen in the following code, for extraction, the `PdfReader` class needs to be imported from `PyPDF2`. The target file or file path (`pdffile`) is provided to `PdfReader`. `reader` is an object that carries methods and attributes from `PdfReader`:

```
from PyPDF2 import PdfReader
pdffile = "python_cheat_sheet_v1.pdf" # file path or name
reader = PdfReader(pdfFile)
# reader = PdfReader(open(pdffile,'rb'))
dir(reader)
['__class__',…,'_encryption',…,'isEncrypted', 'is_encrypted',
'metadata', 'namedDestinations', 'named_destinations', 'numPages',
'outline', 'outlines', 'pageLayout', 'pageMode', 'page_layout',
'page_mode', 'pages', 'pdf_header', 'read', 'readNextEndLine',
'readObjectHeader', 'read_next_end_line', 'read_object_header',
'resolved_objects', 'stream', 'strict', 'threads', 'trailer',…]
```

`PyPDF2` recently received an update that equipped it with new methods and attributes. There are also deprecated entities, which you might come across when exploring `PyPDF2`.

The number of pages in a PDF can be found by using `len(reader.pages)`, and individual page sources or content can be targeted with `page1 = reader.pages[0]` (here, index 0 refers to page 1, 1 to page 2, and so on). With the page number identified, content can be extracted using the `extract_text()` method:

```
page1 =  reader.pages[0]    #point to first page
page1_source = page1.extract_text()   #extract from first page
len(page1_source)          # 964
page1_source[:100]  # Python is a beautiful language. It's easy to
learn and fun, and its syntax is simple yet elegant. Pyt
page1_source
"Python is a beautiful language. It's easy to learn and fun, and
its syntax is simpleyet elegant. …..……. Python Cheat Sheet\n1.
PrimitivesNumbers"
```

The preceding code loads the PDF file content or pages using `PyPDF2`. In the examples discussed next, we will be applying regex to extract the desired content from the PDF.

Example 1 – string extraction

Here is the first paragraph from the `python_cheat_sheet_v1.pdf` PDF file that we want to extract (the PDF file can be found in this chapter's GitHub repository):

Python Cheat Sheet

Python is a beautiful language. It's easy to learn and fun, and its syntax is simple yet elegant. Python is a popular choice for beginners, yet still powerful enough to to back some of the world's most popular products and applications from companies like NASA, Google, Mozilla, Cisco, Microsoft, and Instagram, among others. Whatever the goal, Python's design makes the programming experience feel almost as natural as writing in English.

Figure 9.1: First paragraph of text from the python_cheat_sheet_v1.pdf file

To extract the first paragraph, as seen in *Figure 9.1*, the regex should be applied to `page1_source`, and the content of the first paragraph is returned using Python's `list()`, in this case, with an index of 0 – `python_definition[0]`:

```
python_definition = re.findall(r"^(.*?)Check\s*out", page1_source)
python_definition[0]
"Python is a beautiful language. It's easy to learn and fun, and its
syntax is simpleyet elegant. Python is a popular choice for beginners,
yet still powerful enough toto back …….. Whatever the goal, Python's
design makes the programming experiencefeel almost as natural as
writing in English."
```

In this example, we have learned how to identify content and apply regex to collect the desired content.

Example 2 – tabular content

In this example, we will extract the tabular data found on page 3, as seen in *Figure 9.2*, of the `GeoBase_NHNC1_Data_Model_UML_EN.pdf` file (the PDF file can be found in this chapter's GitHub repository), and create a CSV file:

Date	Version	Description
September 2002	Draft 01	First draft for discussion with Nova Scotia
January 2003	Draft	Second draft after discussion with Nova Scotia and major review of the hydro network model through: • Proposal of options
March 2003	Alpha	Draft version after discussion and decisions made concerning NHNC1 scope and content with Nova Scotia and British Columbia
July 2003	Draft 02	Draft version after discussion and decisions made concerning the detailed NHNC1 model and content with Nova Scotia, British Columbia, and the Yukon. Meeting in Victoria, May 2003.
December 2003	Draft	Integration of UML model for both LRS and Segmented views of the NHNC1.
February 2004	Draft	English review; Remove the UML model for the Segmented view.
May 2004	Draft	Update from the March Workshop comments.
August 2004	2004.1	The object metadata attribute « date » is renamed « validity_date » in section 3.1.6.2 and at all other parts referring to object metadata. A new Feature type value is added for inland water (see section 3.1.5)

Figure 9.2: Extracting tabular data from a PDF file

To begin with, let us extract the content from page 3 of the PDF file:

```
from PyPDF2 import PdfReader
import re
pdfFile="GeoBase_NHNC1_Data_Model_UML_EN.pdf"
reader = PdfReader(pdfFile)
page3 = reader.pages[2]
page3_source = page3.extract_text()
page3_source
'….\nDate  Version  Description  \nSeptember 2002  Draft 01  First
draft for discussion with Nova Scotia  \nJanuary 2003  Draft  Second
draft after discussion with Nova Scotia and major review of the \
nhydro network model through:  \n\uf0b7 Proposal of options  \nMarch
2003  Alpha  Draft version after discussion and decisi ……\nFUTURE
WORK  \nKey word  Description  \n  \n '
```

The page3 variable (reader.pages[2]) points to page 3. page3_source holds the content of page3 after deploying the extract_text() function. In the preceding code, the content of page3_source can be seen, which is in raw format with line breaks.

With the page source (`page3_source`) in hand, to match or extract the raw tabular content from `page3_source`, the help of regex is required. As found in `page3_source`, each row of the table begins with a line break, `\n`, the *month name*, followed by the *four-digit year value*. `page3_split` collects the chunks of tabular data from regex – `r"\n(January|August|May|December|February|July|March|September)\s+([0-9]{4})"`:

```
page3_split = re.split(r"\n(January|August|May|December|February|July|
March|September)\s+([0-9]{4})", page3_source)
['National Hydro Network, Data Model – Edition 1.0  2004 -06 \
nGeoBase®  iii REVISION HISTORY  \nDate  Version  Description  ',
'September', '2002', '  Draft 01  First draft for discussion with Nova
Scotia  ', 'January', '2003', '  Draft  Second draft after discussion
with Nova Scotia and major review of the \nhydro network model
through:  \n\uf0b7 Proposal of options  ',……..]
```

Throughout the new variables and values, cleaning is necessary, and this is done using `strip()` and `re.sub()`. As the data available was in tabular form, a few activities, such as cleaning, splitting, and substituting, were also required in this example.

Further, to combine the chunks from `page3_split` and add the values to the final data container, `dataSet`, iteration is performed. Iteration is targeted at each of the three elements from `page3_split`, and they are accessed with `page3_split[i]`, `page3_split[i+1]`, `page3_split[i+2]`:

```
dataSet = []    # data container
for i in range(0, len(page3_split), 3):
# loop through each 3 elements
    x = page3_split[i].strip()    # cleaning Month
    y = page3_split[i+1].strip() # cleaning Year
    z = re.sub(r"\n",'',page3_split[i+2]).strip()
    if re.match(r"December", x):           # case
        zv = re.split(r"(\s{1})",z) # single space
    else:
        zv = re.split(r"(\s{2})",z) # multiple spaces
    ver = zv[0].strip() # version
    del zv[0]    # remove unwanted element
    desc = "".join(zv).strip()    # description
    if re.search(r"FUTURE",desc):  # case
        desc = re.sub(r"(FUTURE.*)$",'',desc).strip()
```

After retrieving and cleaning the data, `dataSet` is appended with a Python `list()` of values equivalent to a row of a table:

```
dataSet.append([x+" "+y, ver, desc])
dataSet
[['September 2002', 'Draft 01', 'First draft for discussion with Nova
```

```
Scotia'],….,
['May 2004', 'Draft', 'Update from the March Workshop comments.'],….]
```

Data available in `dataSet` is saved as a CSV file, which looks like *Figure 9.3*:

	Date	Version	Description
1	September 2002	Draft 01	First draft for discussion with Nova Scotia
2	January 2003	Draft	Second draft after discussion with Nova Scotia and major review of the hydro net…
3	March 2003	Alpha	Draft version after discussion and decisions made concerning NHNC1 scope and …
4	July 2003	Draft 02	Draft version after discussion and decisions made concerning the detailed NHNC…
5	December 2003	Draft	Integration of UML model for both LRS and Segmented views of the NHNC1.
6	February 2004	Draft	English review; Remove the UML model for the Segmented view.
7	May 2004	Draft	Update from the March Workshop comments.
8	August 2004	2004.1	The object metadata a ttribute « date » is renamed « validity_date » in section 3…

Figure 9.3: Tabular data as a CSV after PDF extraction

It also has to be noted that the CSV file does not possess the exact formatting of the raw table, as seen in *Figure 9.2*.

When dealing with PDF extraction in this chapter, we have come across using various functions and techniques. Using regex was one such option, but the decision of what option to use will differ from developer to developer. Regex features such as searching, splitting, substituting, finding, and iterating were quite useful in the examples we explored.

Summary

Regex and its features are popular for use with not only unstructured but also structured data. Regex provides us with many options, and the code will definitely differ from one use case to another. An advantage of regex is that it can be applied in various cases; there might be a few more steps to deal with, but we can focus on the target using regex. The process of PDF extraction is still evolving. While there are other approaches that can be taken, regex is also one of the most important components of data-related tasks.

The topics covered in this chapter helped you to gain a practical perspective on using regex as required. Regex plays an irreplaceable role in the data extraction activity, unaffected by content structures and document types.

In the next chapter, we will be learning about data mining and data visualization.

Further reading

- *Regular expressions*: `https://regexone.com/, https://www.regular-expressions.info/tutorial.html`

- *Regular expressions – Python*:

    - `https://docs.python.org/3/library/re.html`

    - `https://www.w3schools.com/python/python_regex.asp`

- *Online regex tester and debugger*: `https://regex101.com/`

- *PDF*: `https://www.adobe.com/acrobat/about-adobe-pdf.html`

- *PDF – Python*:

    - `https://pypdf2.readthedocs.io/en/3.0.0/`

    - `https://nanonets.com/blog/pypdf2-library-working-with-pdf-files-in-python/`

- *Scrape PDF*:

    - `https://towardsdatascience.com/scrape-data-from-pdf-files-using-python-fe2dc96b1e68`

    - `https://docparser.com/industries/pdf-scraper-data-providers/`

Part 4: Advanced Data-Related Concepts

In this part, you will learn advanced concepts that can be practiced and performed after collecting high-quality scraped data. You will learn how to read the collected data from various sources and perform activities such as analysis and visualization that help to generate information and patterns from data. This information will help you in decision-making and so on. You will also learn about machine learning and deploy your data models to generate information to analyze sentiments from text, conduct predictions, and so on.

This part contains the following chapters:

- *Chapter 10*, *Data Mining, Analysis, and Visualization*
- *Chapter 11*, *Machine Learning and Web Scraping*

10

Data Mining, Analysis, and Visualization

So far, we have learned about some of the core Python libraries and techniques regarding HTTP/HTTPS communication, reading content, browser automation, and more from a data extraction perspective.

Data is the new oil (we all agree about this), but solely obtaining or collecting data does not provide any significant value. Collected data is stored in files (JSON, CSV, and XML), databases, and more. Stored data needs to be identified, searched, arranged, cleaned, transformed, explored, or modeled using algorithms and can sometimes be used by many services and applications before there's any profit from the information from it.

Various technologies and concepts are involved in identifying and collecting data and processing it in order to extract some value. Data analysis implements and executes logic and algorithms using data-related applications and tools to generate valuable information. Visualization, on the other hand, displays the data in a more presentable format using plots, charts, and much more.

In this chapter, we will discuss topics that are applicable to collecting data. We will also discuss how to use concepts and techniques that will help us to gain key information and valuable insights. These concepts and topics are in high demand in domains such as **data science**, **data analysis**, **knowledge discovery** (**KD**), **artificial intelligence** (**AI**), and **machine learning** (**ML**).

To that end, we will cover the following topics in this chapter:

- Data mining
- Handling collected data
- Data analysis and visualization

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required, and we will be using *JupyterLab* for Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections of *Chapter 2* to continue with setting up and using the environment we have created. Refer to the following links to install and upgrade the required libraries:

- `pandas`:`https://pandas.pydata.org/docs/getting_started/install.html`
- `ydata_profiling` :`https://ydata-profiling.ydata.ai/docs/master/pages/getting_started/installation.html`
- `plotly`:`https://plotly.com/`
- `wordcloud`:`https://pypi.org/project/wordcloud/`

The Python libraries that are required for this chapter are as follows:

- `csv`
- `json`
- `sqlite3`

The code files for this chapter are available online in the book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter10`.

Introduction to data mining

The term "mining" normally means the extraction or the process of extracting something. Data mining is the process of extracting data or discovering information from data. Data mining is a growing and ever-developing concept that discovers hidden, unexpected, and other various forms of information from datasets or databases, which helps in KD and decision-making.

In terms of data, mining is used as a form of analysis to discover patterns, hidden facts, and more. When knowledge is discovered using mining techniques, this is known as **knowledge discovery in databases** (**KDD** or *knowledge discovery and data mining*). There are plenty of terms used to describe data mining, such as KDD, information harvesting, pattern discovery from databases, and many more; although the final results are the same, these terms differ in the steps and processing architecture.

> **Important note**
>
> KDD is an almost cyclical process that has data mining as one of its major components. Various steps are used by KDD, such as data selection, cleaning, enrichment (preprocessing), coding and transformation, data mining, and reporting (interpretation and evaluation). The output from each step is processed as feedback or input for the next step until the required information or knowledge is obtained.

Data analysis and data mining are subsets of data science and are often compared and treated as similar processes. Data mining is based on scientific and mathematical methods, whereas data analysis is done using analytical models and intelligence systems.

Data mining is also considered a part of the data analysis process, but mining is normally done on structured data, whereas analysis can be done on structured, unstructured, and semi-structured data. Similar to KD and data analysis, "data warehouse" is a term that we often encounter while discussing data mining.

A data warehouse is central storage for data of various types, such as recent, historical, and subject-oriented. Data in a data warehouse is used for decision support systems or queries. Hence, the availability and meaningful retrieval of data from data warehouses for the mining process significantly affects their usefulness. Although the use of data warehouses is common in data science, there is no exact or strict relationship between data mining and data warehouses. However, in the implementation phases related to mining, they prove to be valuable.

> **Important note**
>
> Query processing tools and data mining tools are also often used side by side. Data mining does not replace queries or tools related to querying; rather, it tops up the possible add-ons. In a normal scenario, using **structured query language** (**SQL**) is much harder than mining to find hidden knowledge from databases. In general practice, if we know what we are looking for, we will use SQL. Otherwise, we use data mining.

Globally, there is a growing demand for information in various types and forms from systems and frameworks related to data mining. AI and ML are also largely used to facilitate the correct procedures, algorithms, and time-based information extraction. Data mining used to largely process only text content. Now, there are various types of content to be mined, such as images, audio, video, web content, and social media.

Data mining is used to collect and discover new patterns and knowledge that describe data. We can use this along with some intelligent automated systems to predict, discover, and analyze data patterns, deal with large volumes of various types of data, and reduce the amount of time it takes to handle data.

Normally, the goal of data mining is to extract hidden and unidentified information and knowledge from data. Tasks related to data mining are either predictive (prediction, analytics) or descriptive (description, modeling). There is a large number of techniques in both predictive and descriptive processes. In this section, we have chosen only a few techniques, which are important and in high demand.

Predictive data mining

Predictive data mining primarily predicts unknown or future values. It uses statistical analysis to turn data into valuable information. Techniques that analyze data and predict output fall into this category and are as follows:

- **Classification**: This is one of the most common mining techniques, and it classifies data into various categories based on some predictor variables, or predictors. Classification divides a dataset into subsets by applying algorithms. Classification generally classifies data into predefined categories.

- **Regression**: This technique is used to predict, forecast, and analyze the relationships between variables. Some variables might be dependent on others and are called dependent variables. This technique helps in the analysis or prediction of dependent variable values when independent variables change. Normally, various regression methods (such as linear and multi-linear) establish a relationship or link between all types of variables.

- **Prediction**: This technique analyzes past events and predicts future values. It is used to predict relationships between independent and dependent variables, and within these variables as well. It uses past or raw information from other techniques, such as classification and clustering.

Descriptive data mining

Descriptive data mining finds interpretable or detectable patterns. Also, it is considered to be the preliminary stage of data processing. It analyzes past data and uses some of the techniques listed here:

- **Clustering**: This technique groups items that are similar or possess similar characteristics. It divides data into clusters with similar content. It also tries to make the clusters as different as possible. Clustering and classification are similar, but they differ in that clustering finds and stores data clusters based on the data's characteristics and not based on predefined characteristics.

- **Summarization**: This technique provides a concise representation of a dataset, including generating reports and visualization. Summarization stores a descriptive summary that is easy to understand.

- **Association rules**: This technique associates a number of variables in datasets. Association rules state a statistical correlation between certain attributes in a dataset. It helps with identifying relations (or associations) between two or more variables, which can help with finding hidden observations or patterns. This technique is commonly employed in market analysis.

> **Important note**
> Market analysis is a modeling technique used by retailers to find associations between items. For example, if a customer is buying item A, what other items might be bought?

Data mining is an important component in today's organizations that uses analytics or similar information for decision-making. Companies or organizations can predict and collect information based on usage patterns of their clients or customers, classify them, and generate information that will be helpful for management and can be used in research, to predict client or customer behavior, to detect fraud, in **business intelligence** (**BI**), to form marketing strategies, in brand management, for ML models, and much more.

There are also some steps in data mining that need to be taken seriously and consistently, to prevent invalid or incorrect interpretations and outcomes. The quality (clean, preprocessed, filled, enriched, duplicates removed) and volume (daily, per transaction, time series) of data drives the results.

In the next section, we will be exploring how data is collected, stored, and read, and how the various file formats or databases are chosen.

Handling collected data

The availability of data is the main concern before attempting to process it for information and pattern detection. Handling collected data normally refers to gathering data in files and databases in some format and using it effectively and efficiently.

There are many tools and applications that handle data. Choosing the right tool or way of storing and using data shows your professionalism as a developer.

In the following sections, you will be learning about concepts related to handling files and dealing with types of files (JSON and CSV) that are in huge demand in the market and are associated with a large number of IT-driven applications.

Basic file handling

File handling is the core or basic technique for storing and reading data from files. This technique of handling and managing data is used a lot in various programming languages. File handling does not require additional software or tools unless some application extensions are used. Formatting is a major issue; we are pretty much dealing with the files' contents in a very raw format.

In Python, as seen in the following code, an inbuilt function called `open()` is used to read in a file, and the `close()` function removes the file from memory and is always used at the end of all file-related activities, such as reading and writing. The `open()` method is provided with some optional arguments, such as `mode` and `encoding`:

```
content = open(file_with_path, mode, encoding)
# encoding is optional
try:
    …… #file reading writing seeking positions
finally:
    content.close()
```

The `encoding` argument is optional. By default, it uses the value `utf-8`. There are various modes in which files can be managed or operated:

- `r`: Read mode; this reads the file. It is also the default mode if no other mode is provided with `open()`. `r+` reveals combined mode both for reading and writing and `rb` opens the file in read mode and binary mode. The full filename or location of the file should be available if you want to use this mode. There are methods for read mode such as the following:

  - `read()`: Reads the complete file content and returns a string, for example, `variable = content.read()` or `variable = content.read(200) # reads first 200 characters from content`

  - `readline()`: Reads a single line from the content, for example, `line_1 = content.readline()`

- `w`: Writes in the file. `w+` is used for writing and reading, whereas `wb` is used to write binary values. The `write()` method writes the provided argument to the file, for example, `content.write(string)`.

- `a`: Append mode is used to write content at the end of the loaded file. If the file is not found or the wrong filename or file path is provided, then it creates a new file, for example, `f = open("file.txt", "a")` and `f.write("this will be added to last line of file.txt")`

`rb+` and `wb+` are often encountered with binary file format. There are also a few methods that work on content inside the file:

- `tell()`: Returns the current position of the file object in the file, for example, `content.tell()`.

- `seek()`: Adds or changes the file object position to the provided offset value. Generally, `seek(offset, whence)` accepts two of the `offset`, `whence`, or `from` arguments.

For example, `content.seek(199)` points to the two-hundredth character in the file and `content.seek (100,199)` points to the one-hundredth character after the first 200 characters.

Python also supports a statement called `with`, which is used for managing resources such as files and database connections. The `with` statement is also used for making cleaner, more readable code, and it also helps with exception handling. There's no need to use methods such as `close()` because `with` handles the code scope automatically. Here's an example:

```
with open(fileNamePath, "r+") as file:

content = file.read()
…..
```

As shown in the preceding code template, `with` is used to open the file referenced with `fileNamePath` using `r+` mode. After loading the file, the file's contents are read using the `read()` function.

In this section, we have covered some important techniques and methods that are used for file handling.

In the next sections, we will explore some structured types of files and look at how to read and write to them.

JSON

JavaScript object notation (**JSON**) is a structured file format. From web APIs to local file content, the utilization of JSON is on the rise due to its string-based features and resemblance to Python. JSON contents are made from a mixture of *dict()* and *list()*. Python's built-in `json` library can be used to manage JSON files. There are various ways to read files; we can use Python file-handling techniques, as shown in the following code:

```
import json
with open("book_details.json", "r") as file:
    books = json.loads(file.read())
    print(books[0]['Title'])
    Birdsong: A Story in Pictures

    print(books[0])
    {'Upc': '9528d0948525bf5f', 'Title': 'Birdsong: A Story
    in Pictures', 'Price': '£54.64',
    'Rating':'http://books.toscrape.com/media/cache/af/6e/
    af6e2.jpg'}
```

The `json` method loads content using the `loads()` method and the content is returned by the `file` object using the `read()` method. Once the JSON content is loaded, it can be accessed and used as a Python dictionary by using keys and indexes.

In the following sections, we'll be dealing with basic read and write activities with JSON files.

Reading JSON files

Today's web content is more and more JSON-driven, and JSON has been an effective part of storing and analyzing data. The `pandas` Python library is used to analyze data, and it also provides a method called `read_json()` for reading JSON files. It converts JSON content into `DataFrame` format with a row and column (almost tabular) structure. There is also an attribute named `shape`, which returns the shape (rows, columns) of the `DataFrame`, as shown in the following code block:

```
import pandas as pd
bookJson = pd.read_json("book_details.json")
print(type(bookJson))
pandas.core.frame.DataFrame
print(bookJson.shape)
(29,8)
```

After obtaining a `DataFrame`, we can read columns by chaining columns with a period, such as `bookJson.Title`. Indexes can also be provided to a `DataFrame` using `iloc` from pandas; for example, `bookJson.iloc[2]` will return data from the third index in the `DataFrame`, as shown here:

```
bookJson.Title
0     Birdsong: A Story in Pictures
.........
28    Charlie and the Chocolate Factory (Charlie Buc...
Name: Title, dtype: object

bookJson.iloc[2]                   # read index 3
Upc                        b5ea0b5dabed25a8
Title                 The Secret of Dreadwillow Carse
Price                 £56.13
Rating                One
Stock                 In stock
Stock_Qty             16
Url       http://books.toscrape.com/catalogue/the-secret...
Image     http://books.toscrape.com/media/cache/c4/a2/c4...
Name: 2, dtype: object
```

In this section, we used pandas methods to read and load JSON file contents. In the next section, we will be using the native Python library `json` to create JSON files.

Writing JSON files

There's another method provided by pandas for creating JSON files, `to_json()`. We will be using the `json` library and its `dump()` method to create a JSON file. As shown in the following code, we have an empty list that gets loaded with two sample dictionaries. The `demo.json` file is opened using write (w) mode, and a `dataSet` list with values passed to `json.dump()`. The `dump()` method converts a Python object into a JSON string and provides the object to the file:

```
dataSet = []
dataSet.append({'hotel':'Hotel1','reviews':{'country':'UK',
    'reviewcount':21,'rating':9.9, 'positive':9,
        'negative':0, 'stayDate':'2022-12-26'}})
dataSet.append({'hotel':'Hotel2','reviews':{'country':'US',
    'reviewcount':11,'rating':7.9, 'positive':8,
        'negative':0,'stayDate':'2023-01-02'}})

import json
with open("demo.json", "w") as file:
    json.dump(dataSet, file, indent=4, sort_keys=False)
```

Two dictionary or `dict()` objects are appended to the `dataSet` collection, which are then written to the `demo.json` file.

The output of the `demo.json` file is shown in *Figure 10.1*:

Figure 10.1: JSON output (Jupyter notebook)

As we have seen in this section, the easiest and most effective way of reading and creating files is by using the JSON file format. In the next section, we will deal with CSV files.

CSV

Comma-separated values (CSV) is another format of structured data. CSV files are somewhat compact and voluminous in nature. Each entity in a file is either a column or row record, and they are separated by a delimiter, generally a comma (,). We can have different delimiters or separators, but need to be careful when reading and processing such files.

Converting CSV files into tables is common in practice. CSV stores data in row and column format, almost in a tabular structure. CSV headers, or the first row, contain the titles of the columns. From the second row, data is listed line by line. Python provides a built-in library, `csv`, that has certain methods that make writing CSV files almost like basic file handling.

As shown in the following code, the `with` statement is provided with `w+`, `encoding`, and `newline` arguments:

```python
import csv
def writeto_csv(data, filename, columns):
    with open(filename, 'w+', newline="", encoding="UTF-8")
    as file:
        writer = csv.DictWriter(file, fieldnames=columns)
        # Column header

        writer.writeheader()                # writes header
        writer = csv.writer(file)
        for element in data:
            writer.writerows([element])
            # writes each list element as row
```

There is a class called `DictWriter` in the `csv` library that has a few dedicated methods for writing CSV headers (`writeheader()`) and rows (`writerows()`). Throughout this book, we are using the `writeto_csv()` function, which receives the three arguments listed here:

- `data`: A list of elements that will be added as CSV rows

- `filename`: The filename that will be created

- `columns`: A list of the column names or headers for the CSV file

With some sample data provided in the following `holidays` list, a call to the defined `writeto_csv()` function has been made, which will create a file called `holidays.csv` with `Date` and `Name` columns, and rows from the `holidays` list:

```python
holidays = [['2023-01-02', "New Year's Day"],
['2023-01-16', 'Martin Luther King, Jr. Day'], .........,
['2023-11-10', 'Veterans Day'], ['2023-11-23',
'Thanksgiving Day'], ['2023-12-25', 'Christmas Day']]

writeto_csv(holidays, 'holidays.csv', ['Date', 'Name'])
# function call
```

The raw `holidays.csv` file looks as shown in *Figure 10.2*:

```
 1    Date,Name
 2    2023-01-02,New Year's Day
 3    2023-01-16,"Martin Luther King, Jr. Day"
 4    2023-02-20,Washington's Birthday
 5    2023-04-07,Good Friday
 6    2023-04-07,Good Friday
 7    2023-05-29,Memorial Day
 8    2023-06-19,Juneteenth
 9    2023-07-04,Independence Day
10    2023-09-04,Labour Day
11    2023-10-09,Columbus Day
12    2023-11-10,Veterans Day
13    2023-11-23,Thanksgiving Day
14    2023-12-25,Christmas Day
15
```

Figure 10.2: CSV file without formatting

Using editors such as Jupyter Notebook, `holidays.csv` will look as shown in *Figure 10.3*:

	Date	Name
1	2023-01-02	New Year's Day
2	2023-01-16	Martin Luther King, Jr. Day
3	2023-02-20	Washington's Birthday
4	2023-04-07	Good Friday
5	2023-04-07	Good Friday
6	2023-05-29	Memorial Day
7	2023-06-19	Juneteenth
8	2023-07-04	Independence Day
9	2023-09-04	Labour Day
10	2023-10-09	Columbus Day
11	2023-11-10	Veterans Day
12	2023-11-23	Thanksgiving Day
13	2023-12-25	Christmas Day

Figure 10.3: CSV file with formatting

The pandas library provides methods for reading and writing DataFrames to CSV files. The `books.csv` file is being read into a DataFrame using pandas' `read_csv()` function. There are more than four columns in the `books.csv` file, as shown here:

```
category, image, no_review, price, rating, stock, title, upc, url
```

But we are reading only four selected columns with the help of the `usecols` argument (in many situations, memory overload can be controlled or managed by using required values only):

```
import pandas as pd
books = pd.read_csv("books.csv", usecols = ["category",
    "price", "rating", "title"])
type(books)    # pandas.core.frame.DataFrame
books.shape    # (1000, 4)
```

As shown in *Figure 10.4*, the `books` DataFrame with four columns and 1,000 rows is being displayed:

	category	price	rating	title
0	Poetry	£51.77	Three	A Light in the Attic
1	Historical Fiction	£53.74	One	Tipping the Velvet
2	Poetry	£52.15	One	The Black Maria
3	Young Adult	£22.65	One	The Requiem Red
4	Fiction	£50.10	One	Soumission
...	...	...	...	...
995	Sequential Art	£57.06	Four	Ajin: Demi-Human, Volume 1 (Ajin: Demi-Human #1)
996	Classics	£55.53	One	Alice in Wonderland (Alice's Adventures in Won...
997	Travel	£26.08	Five	1,000 Places to See Before You Die
998	Mystery	£53.98	One	1st to Die (Women's Murder Club #1)
999	Historical Fiction	£16.97	Five	A Spy's Devotion (The Regency Spies of London #1)

1000 rows × 4 columns

Figure 10.4: DataFrame with selected columns from CSV

pandas also supports writing DataFrames to CSV files using `to_csv()`. For example, `books.to_csv('books_4_col.csv')` will create a CSV file with data from the `books` DataFrame.

> **Important note**
>
> pandas supports reading and creating various file types, such as JSON, CSV, XLSX (Excel), HTML, XML, Parquet, and ORC. Please refer to `https://pandas.pydata.org/docs/reference/io.html` for more details.

In the next section, we will discuss the lightweight database **SQLite**.

SQLite

Databases store data in the form of tables, or in a row and column structure. Data collected from various tables can be used, linked, and queried using the features of **relational database management systems** (**RDBMSs**). We can manage data inside databases using **structured query language** (**SQL**) or by embedding or using programming languages. When there are security concerns and SQL is required, databases with tables are generally preferred to file handling.

SQLite is one of the most commonly used database engines with Python, `https://docs.python.org/3/library/sqlite3.html`. It is considered fast and easy to handle and can be used with or without applications. It creates a file with the `.db` extension in a given location, and can also access or read the database file as required.

`sqlite3` is a standard Python library. SQLite supports column types such as `integer`, `float`, and `text`. For example, take a look at the following code block:

```
import sqlite3 as db
connection = db.connect('sample.db')
connection.execute('create table holiday(ID int,
    Title text, Date text)')
```

Some of the main details of the preceding code are as follows:

- The `connect()` method connects the database with the code file and manages disk activities.

- Every process in SQLite should begin by establishing the connection with the database using `connect()`. The `connection` object of the `sqlite3.Connection` class is used to deal with database activities.

- `connection.execute()` creates an `sqlite3.Cursor` object, which is used to process SQL queries. `cursor` objects are used to execute SQL statements and deal with rows of data inside tables.

As per the preceding code block, the `sample.db` file will be created in the stated location, and it contains a table named `holiday`, created using a SQL statement. For more information on SQL, please visit `https://www.sqltutorial.org`.

The `commit()` method is used as a saving option. A call to `commit()` secures the availability or execution of the SQL statement in the database:

```
connection.execute('insert into holiday(ID, Title, Date) values
(1,"New Year Day", "2023-01-01")')
......
connection.commit()
```

After inserting rows into the table, results can be obtained using SQL queries. Also, methods such as `fetchone()` and `fetchall()` can be used to retrieve single rows and all rows respectively:

```
results = connection.execute('select * from holiday')
for row in results:
    print(row)
    (1, 'New Year Day', '2023-01-01') (2,……….
result = cursor.execute('select * from holiday')
result.fetchone()  # (1, 'New Year Day', '2023-01-01')
```

Converting a `sqlite3` table into a CSV file is simple using pandas. pandas provides a method called `read_sql()` that accepts a SQL query that can be used with the `sqlite3` connection object, `dbConnect`. The `to_csv()` method from pandas is used to create a CSV file. The `index` argument can also be removed while creating a file:

```
dbConnect = db.connect('sample.db')
holidayData = pd.read_sql('select * from holiday'
    ,dbConnect)

holidayData.to_csv('holiday_sqlite.csv')
holidayData.to_csv('holiday_sqlite_noindex.csv',
    index=False)
```

SQLite is also used professionally for learning and practicing DBMS and RDBMS concepts. It can also be used to build the confidence of practitioners to handle DBMSs such as MySQL and MongoDB.

> **Important note**
>
> Although there's no in-built security provided with SQLite, encrypted data can also be stored in tables. This somewhat prevents access-level security vulnerability. Please visit these links for more details on Python database related libraries: `https://www.sqlite.org/index.html` (sqlite3), `https://pymongo.readthedocs.io/en/stable/` (pymongo), and `https://pymysql.readthedocs.io` (pymysql).

In the next section, we will be exploring analysis and visualization using the pandas and plotly Python libraries.

Data analysis and visualization

Python programming is popular because of its easy usage and the availability of libraries for scientific computing, text computation, data analysis, machine learning, and much more. Data analysis is a systematic process. Unknown facts, hidden patterns, summary data, and a lot of other information

can be obtained using data analysis. Data analysis is also treated as a subset of data science, and it has been booming with the use of Python and its features.

In this section, we will be analyzing some datasets, exploring some of the important features of pandas, and visualizing the results using plotly.

Analyzing data generally involves a few basic steps:

1. **Identify**: Identify the source of data or the origin of data, such as a website, PDF file, or image.
2. **Collect**: Collect the identified data using scraping or other techniques. Storing data is also important here.
3. **Clean**: Preprocess and clean the collected data. Clean data is easier to process with statistical tools and is more likely to return accurate results.
4. **Analyze**: Clean and collected data is processed with logical techniques, statistics, and qualitative and quantitative approaches. Obtained results are collected, stored, and visualized.
5. **Interpret**: Visualized results are interpreted and a summary of the information is explored.

There are various ways to analyze data. We will be using a reporting tool to explore the data we have in the next section.

Exploratory Data Analysis using ydata_profiling

Exploratory Data Analysis (**EDA**) is a compulsory activity that has to be carried out on data that will be used for analysis. Performing and studying EDA reports can provide more information about the data, and can also provide a small amount of analyzed information. For simple datasets with clean, high-quality data, it might not be necessary to conduct further analysis by writing code, after doing EDA.

In Python, EDA steps and processes used to involve writing code using various Python analysis libraries and graphing tools. There has been a lot of development in EDA; we now have a Python library called `ydata_profiling` that automatically creates a report based on the dataset provided. `ydata_profiling` is easy to use and helps with generating reports (HTML, embedded notebooks, and more) with just a few lines of code. Some of the benefits of using these EDA libraries are listed here:

*   Only a few lines of code need to be used
*   Ready-to-use generated reports
*   Provides detailed statistics on variables (`count`, `duplicate`, `missing`, `types`, `distinct`, correlation with other variables, and many more)
*   Provides filterable graphs for correlations, interactions, missing values, and more

As seen in the following code, the `book_details.json` dataset is being loaded to a pandas DataFrame using `read_json()`. The `ProfileReport` class of `ydata_profiling` accepts the `books` DataFrame and creates a report with the provided `title` attribute.

Here, the report generated has been exported to an HTML file named `book_details_rawdata.html`:

```python
import pandas as pd
from ydata_profiling import ProfileReport
books = pd.read_json("book_details.json")# input
books_profile = ProfileReport(books,
    title="Book Details - Raw Data , Report")

books_profile.to_file("book_details_rawdata.html")# output
```

The generated `book_details_rawdata.html` file will have various menu options, as shown in *Figure 10.5*. This report file can also be opened in a new browser tab:

Figure 10.5: EDA – HTML report

Moving through the report menus, we can also find detailed statistical descriptions for data variables, as shown in *Figure 10.6*. Depending on the type (*numerical, categorical, and others*) of the variable, detailed analysis reports will vary.

| Statistics | Histogram | Common values | Extreme values |

Quantile statistics

Minimum	1
5-th percentile	1.4
Q1	3
median	7
Q3	14
95-th percentile	17.2
Maximum	19
Range	18
Interquartile range (IQR)	11

Descriptive statistics

Standard deviation	5.8575033
Coefficient of variation (CV)	0.74177989
Kurtosis	-1.3381785
Mean	7.8965517
Median Absolute Deviation (MAD)	4
Skewness	0.52015565
Sum	229
Variance	34.310345
Monotonicity	Decreasing

Figure 10.6: Statistical description in an EDA report

As mentioned in the introduction to this section, the cleanliness and accuracy of the analysis are based on the data we are using. For this chapter, book data scraped from `http://books.toscrape.com/` has been used. Two EDA reports have been made and are available in this chapter's GitHub repository: one uses the raw dataset, and another uses a cleaned data file.

> **Important note**
>
> `ydata_profiling` is an updated version of the existing `pandas_profiling` library. As mentioned in `https://pypi.org/project/pandas-profiling/`, `pandas_profiling` will be outdated soon. There is also another library named `dataprep` from `https://dataprep.ai/`. `dataprep` provides more customizable and granular reports than `pandas_profiling`.

Despite their ease of use and the lack of long code, output in EDA reports is limited and might not be suitable for in-depth analysis of data variables. In such cases, coding and performing manual analysis is compulsory, which we will be doing in the next section.

pandas and plotly

`pandas` is one of the most widely used data analysis libraries. Performance, data structure (series, DataFrames), ease of use, convertibility, input-output compatibility, availability of features, and many other things keep pandas at the top of its category. For detailed information on pandas, please visit `https://pandas.pydata.org/`.

`plotly`, on the other hand, is a graphing library (`https://plotly.com/python/`). It contains some advanced imaging features such as downloading images in `.png` format, zooming in and out, auto-scaling, pan clicking and dragging, resetting axes, and more. These plotly features are available in the top right-hand corner of visualizations without using code, as shown in *Figure 10.7*:

Figure 10.7: Plotly image options

pandas can be used with various graphing libraries, such as matplotlib and seaborn. The plot() method from pandas can be used directly on top of data. pandas also provides options to choose graphing libraries and apply them as required in code or notebooks. For example, pd.options. plotting.backend = "plotly" sets pandas' plotting library to plotly:

```
import pandas as pd
import numpy as np
import plotly.express as px
pd.options.plotting.backend = "plotly"
```

Since *Jupyter notebooks* are used in code examples, we will find the preceding code at the top of the notebook. This code can also be called or imported in the required places.

Let's explore some basic usage of pandas and analyze data through some examples.

Example 1 – book analysis

The code for this example is available at data_analysis_book.ipynb.

The read_json() method reads the JSON file and creates a DataFrame called books. The shape attribute returns a tuple object where the first value is the number of rows and the last value is the number of columns. describe() displays statistical details about the numerical columns. The include="all" argument in describe() provides the statistical detail of all available columns or those returned from books.columns:

```
books = pd.read_json("book_details.json")
books.shape          # (29, 8)
books.describe()
books.describe(include="all")
books.info

books.columns
Index(['Upc', 'Title', 'Price', 'Rating', 'Stock', 'Stock_Qty', 'Url',
'Image'], dtype='object')
```

Column names can be used as attributes, such as books.Title, or in indexed format, such as books['Title']. Python supports indexing in [START: STOP: STEP] syntax. In books. Title[::3], every third book title is provided:

```
books.Title[::3]    # no start an stop, only step
0        Birdsong: A Story in Pictures
```

```
3          The White Cat and the Monk: A Retelling of the...
.......
24                                      Counting Thyme
27                                      Matilda
Name: Title, dtype: object
```

There are various methods and attributes in pandas for accessing row data. The most common one is index and location, or `iloc[]`:

```
books.iloc[2]              # returns row from index 3 (0,1,2)
Upc                                      b5ea0b5dabed25a8
Title                      The Secret of Dreadwillow Carse
.........
Url        http://books.toscrape.com/catalogue/the-secret...
Image      http://books.toscrape.com/media/cache/c4/a2/c4...
Name: 2, dtype: object
```

Checking unique values and counting the total number of unique values is quite applicable during analysis. Here, `books` has 29 rows, but there are only 5 unique values (`nuinque()` returns the number of unique elements). This suggests that duplicate values exist in the `Rating` column. `Rating` values are strings, which might cause problems if any statistical computation is required. Therefore, cleaning is required for the `Rating` column:

```
books["Rating"].unique()
# array(['Three','One','Four','Five', 'Two'], dtype=object)
books["Rating"].nunique() # 5
```

One of the important aspects of using pandas is *data filtering*. Various methods can be used for filtering, such as `filter()`, `where()`, and `query()`. Also, as shown in the preceding code, logical operations (such as and and or) can be used for filtering. `[['Title', 'Price', 'Stock_Qty']]`, after the `query()` operation, collects and displays results from only the three columns mentioned:

```
books.query("Stock_Qty >= 15 and Title.str.contains('Bear')")
[['Title', 'Price', 'Stock_Qty']]
```

Similar to the `str.contains()` method, there are also `startswith()` and `endswith()` methods. These methods play crucial roles when dealing with string types, and again when filtering techniques are applied.

As shown in the preceding code, the `value_counts()` method counts the total number of occurrences of the provided value for the selected columns:

```
ratingCount = books["Rating"].value_counts()
```

The `ratingCount` variable contains the counts related to book rating and plots the bar chart, as shown in the following code:

```
ratingCount.plot.bar(title="Rating Count", labels=dict(index="Rating",
value="Count", variable="Detail"))
```

plotly support various types of chart, such as bar, line, barh, and pie.

Figure 10.8: Using a plotly bar chart

The following code plots a default plot, which is a line chart, with the labels provided:

```
ratingCount.plot(title="Rating Count", template="simple_white",
labels=dict(index="Rating", value="Rating_Count", variable="Legends"))
```

If no type is mentioned, `plotly` will draw a line chart, as shown in *Figure 10.9*:

Figure 10.9: Line chart (ratingCount)

The pandas `copy()` method creates a duplicate DataFrame. Duplication is necessary in this case because we are going to clean some column values. The `inplace=True` argument is quite sensitive because it makes changes to the `bookDF` DataFrame:

```
bookDF = books.copy()  # Duplicating a DataFrame
bookDF["Rating"].replace(["One","Two","Three","Four",
    "Five"],[1,2,3,4,5],inplace=True)

bookDF["Price"] = bookDF["Price"].str.replace("£","")
bookDF["Price"] = bookDF.Price.astype(float)
```

`bookDF` is clean and now has `Price`, which is a float, and `Rating`, which is an integer. The `groupby()` method groups or combines the `Price` values for each `Rating` and sums the `Price` values using `np.sum`:

```
price_groupby=bookDF.groupby("Rating")["Price"].agg(np.sum)
price_groupby.plot.bar()
```

The result of the preceding code has been plotted as a bar chart, as shown in *Figure 10.10*:

Figure 10.10: Aggregated price for each rating

As shown in *Figure 10.10*, we can determine the following based on the `Rating` number (from client feedback) values:

- Books with `Rating` values of 3 and 1 are the top two sellers
- Books with a `Rating` value of 5 are the lowest sellers

Finally, the DataFrame with clean data, `bookDF`, was written to a new CSV file using `to_csv()`. In addition, EDA was conducted using the `book_details_clean.csv` file:

```
bookDF.to_csv('book_details_clean.csv', index=False)
```

> **Important note**
> EDA HTML report files with both uncleaned and cleaned data are available as reports. It is recommended to view both reports and see the difference in details they carry.

The contents of the `book_details_clean.csv` file were cleaned and processed and then used in the code. In the next example, we will do some more exploration of charts and pandas activities.

Example 2 – quote analysis

The code for this example is available at `data_analysis_quote.ipynb`.

In *Example 1 – book analysis*, a DataFrame was created by reading a JSON file. In this case, we have data in a CSV file, which is read using `read_csv()`. The `usecols` argument can be assigned to choose columns that are to be placed in the new DataFrame:

```
quotes = pd.read_csv("quote_details.csv",
usecols=["author","quote","tags", "tag_count"])
```

Tags are separated by commas (`,`). `tag_count` counts the words inside `tags`.

The following code, with the condition that `tag_count` is greater than one, resulted in a new DataFrame, `newQuoteDF`. Only two columns, `tags` and `author`, are selected:

```
newQuoteDF = quotes[quotes["tag_count"]>1][['tags', 'author']]
newQuoteDF
```

The new DataFrame, `newQuoteDF`, and the selected two columns (`tags` and `author`) can be seen in *Figure 10.11*:

	tags	author
0	change,deep-thoughts,thinking,world	Albert Einstein
1	abilities,choices	J.K. Rowling
2	inspirational,life,live,miracle,miracles	Albert Einstein
3	aliteracy,books,classic,humor	Jane Austen

Figure 10.11: Sample newQuoteDF with tag_count > 1

For proper analysis, we need to take each word from `tags` and link them with the author. For example, the `change,deep-thoughts,thinking,world -- Albert Einstein` row needs to be changed to `['change', 'Albert Einstein'], ['deep-thoughts', 'Albert Einstein'], ['thinking', 'Albert Einstein'], ['world', 'Albert Einstein']`. To achieve this, row-wise iteration is required, which is used to split the tags and also create a new DataFrame, `tagsDF`. The chart with the values obtained from the `value_counts()` method plotted on `Tag of DataFrame tagsDF` is shown in *Figure 10.12*:

```
tmpData = []
for index, row in newQuoteDF.iterrows(): # tags, author
    tagList = row.tags.split(",")
    for tag in tagList:
        tmpData.append([tag, row.author])

tagsDF = pd.DataFrame(tmpData, columns=['Tag', 'Author'])
tagsDF.Tag.value_counts().plot()
```

As shown in *Figure 10.12*, the tags with the text `life` and `love` seem to be high in count:

Figure 10.12: Count of tags

As shown in the following query, there are a total of 11 tags that start with the `l` character:

```
tagsDF.query("Tag.str.startswith('l')")['Tag'].nunique()    # 11
```

As shown in *Figure 10.13*, it also looks like there are many tags that have a length that is less than or equal to 10 – almost 80% of the total tag count:

```
tagsDF.query('Tag.str.len() > 10').count()

Tag       32
Author    32
dtype: int64
```

```
tagsDF.query('Tag.str.len() <= 10').count()

Tag       164
Author    164
dtype: int64
```

Figure 10.13: Count of tag lengths

In *Figure 10.14*, using the `describe()` method, we can see that out of 196 total rows, there are 120 unique tags, and there are quotes by 38 authors. The highest-frequency tag is `life` with `12` occurrences. Similarly, `Albert Einstein` is the highest-frequency author with `22` occurrences:

```
tagsDF.describe()
```

	Tag	Author
count	196	196
unique	120	38
top	life	Albert Einstein
freq	12	22

Figure 10.14: Statistical detail for tagsDF

The information shown in *Figure 10.14* can also be found using `value_counts()`:

```
authorCount = tagsDF.Author.value_counts(ascending=True)
authorCount.plot.area()
```

`Author.value_counts()` in `tagsDF` is plotted in *Figure 10.15*. The results are placed in ascending order, hence the steep gradient in this figure:

Figure 10.15: Area plot

The following code executes a filter action on `authorDF1.Count` (occurrence frequency) to be displayed in a pie chart:

```
fig1 = px.pie(authorDF1[authorDF1.Count>5], values='Count',
names="Author", title='Author with count >5')
fig1.show()
```

Figure 10.16 conveys similar information as *Figure 10.15*, but using a pie chart. Out of 38 authors, those with more than 5 occurrences are mentioned, and **Albert Einstein** makes up **16.8%** of the chart:

Figure 10.16: Pie chart of authors

In the preceding example, we were able to find mathematical information for the most occurring texts. Handling raw data in pandas and moving it to a new DataFrame for analysis was also one of the objectives.

Also, when you want to visualize string data, `wordcloud` is a useful library. In the following code, the `WordCloud` class from the Python `wordcloud` library is imported:

```
from wordcloud import WordCloud   # accepts string
```

The `WordCloud` class, with few properties, such as `max_words` and `background_color`, is initiated and provided with content of the `Tag` column from `tagsDF`, which results in the screenshot shown in *Figure 10.17*:

```
wordcloud2 = WordCloud(background_color="white",max_words=500).generate(' '.join(tagsDF.Tag))
px.imshow(wordcloud2)
```


Figure 10.17: WordCloud generated using tags

Conducting EDA, cleaning data, storing data in files and DataFrames, analyzing datasets, creating visualizations, and generating meaningful or hidden patterns and information were the main takeaways from this section. There is more to explore using pandas; please visit `https://pandas.pydata.org/` for more details.

Summary

Generating and gathering information using different analysis techniques and using it for decision-making is a growing field. Fields such as **business intelligence** (**BI**), AI, and ML require, and use, various data analysis techniques. Python programming provides a great infrastructure for the processes of data collection, data processing, information abstraction, and knowledge discovery. Libraries such as `pandas`, `NumPy`, `csv`, `json`, and `plotly` are the core Python libraries of the overall systematic process.

A practical introduction to the concepts related to data mining, data analysis, EDA, and data visualization was the main agenda of this chapter.

In the next chapter, we will be learning about machine learning and web scraping.

Further reading

- *Data Mining*:

  - `https://www.ibm.com/topics/data-mining`

  - `https://www.springboard.com/blog/data-science/data-mining-python-tutorial/`

  - `https://www.investopedia.com/terms/d/datamining.asp`

- *Data Analysis*:

  - `https://monkeylearn.com/data-analysis/`

  - `https://www.datapine.com/blog/data-analysis-methods-and-techniques/`

  - `https://www.investopedia.com/terms/d/data-analytics.asp`

- *Exploratory Data Analysis*:

  - `https://ydata-profiling.ydata.ai/docs/master/index.html`

  - `https://www.digitalocean.com/community/tutorials/exploratory-data-analysis-python`

  - `https://www.kaggle.com/code/imoore/intro-to-exploratory-data-analysis-eda-in-python`

- *Visualization*:

  - `https://plotly.com/python/`

  - `https://www.ibm.com/topics/data-visualization`

  - `https://matplotlib.org/stable/tutorials/introductory/pyplot.html`

  - `https://gilberttanner.com/blog/introduction-to-data-visualization-inpython/`

11

Machine Learning and Web Scraping

So far, we have learned about data extraction, data storage, and acquiring and analyzing information from data by using a number of Python libraries. This chapter will provide you with introductory information on **Machine Learning** (**ML**) with a few examples.

Web scraping involves studying a website, identifying collectible data elements, and planning and processing a script to extract and collect data in datasets or files. This collected data will be cleaned and processed further to generate information or valuable insights. ML is a branch of **Artificial Intelligence** (**AI**) and generally deals with statistical and mathematical processes. ML is used to develop, train, and evaluate algorithms that can be automated, keep learning from the outputs, and minimize human intervention.

ML uses data to learn, predict, classify, and test situations, and for many other functions. Data is collected using web scraping techniques, so there is a correlation between ML (its performance) and scraped data. Web scraping is primarily linked with ML because quality data is provided to ML algorithms as input. ML is also considered for use for any tasks that require lots of interpretation, iterations, handling large volumes of data (in GB, TB, or PB), complex datasets, and any tasks that ML can reduce the processing time of, and reduce errors, when compared to humans.

In this chapter, we will learn about the following topics:

- Introduction to ML
- ML using scikit-learn
- Sentiment analysis

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) will be required and we will be using *JupyterLab* for Python code.

Please refer to the *Setting things up* and *Creating a virtual environment* sections of *Chapter 2* to continue with setting up and using the environment created.

The Python libraries that are required for this chapter are as follows:

- scikit-learn (visit `https://scikit-learn.org/stable/install.html` for installation)
- `textblob` (visit `https://textblob.readthedocs.io/en/dev/install.html` for installation)
- `vaderSentiment`
- `plotly`
- `numpy`
- `pandas`
- `matplotlib`

The code files for this chapter are available online in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/tree/main/Chapter11`.

Introduction to ML

Data collection, analysis, and the mining of data to extract information are major agendas of many data-related systems. Processing, analyzing, and executing mining-related functions requires processing time, evaluation, and interpretation to reach the desired state. Using ML, systems can be trained on relevant or sample data and ML can be further used to evaluate and interpret other data or datasets for the final output.

ML-based processing is implemented similarly to and can be compared to data mining and predictive modeling, for example, classifying emails in an inbox as spam and not spam. Spam detection is a kind of decision-making to classify emails according to their content. A system or spam-detecting algorithm is trained on inputs or datasets and can distinguish emails as spam or not.

ML predictions and decision-making models are dependent on data. ML models can be built on top of, and also use, several algorithms, which allows the system to provide the nearest possible predictions and accurate results. Models trained on a specific dataset can also be applied to new datasets, incorporating the new data and learning from additional collected data or real-time data obtained from scrapers or crawlers. There are ML systems that have theses abilities and assist applications in achieving desirable outputs, such as those listed here:

- Stock price prediction
- Classifying movies

- Sentiment analysis (**Natural Language Processing** (**NLP**))

- Disease prediction

- Weather forecasting

- Customer support and translations

- Recommendation tools

- Fraud detection systems and cybersecurity

- Text to speech

- Image recognition

- Machine or self-driving vehicles

- Scheduled analysis and reporting

ML applicability is boundless; it can be used in the fields of research and study, health and medicine, business and marketing, and more. There are some common challenges in ML (from a data perspective):

- Quality of data

- Unbiased data

- Limited availability of data

ML models are trained on data. Raw data, most of the time, is unstructured, incomplete, and may also contain some flaws. The following sections will provide basic information on ML.

In the next section, Python libraries that are quite often used in mathematical and ML-related scopes are introduced.

ML and Python programming

Python is also known as a programming language for scientific computing. The popularity of Python, its tools, and available libraries with proper documentation, along with its usability, maintainability, and versatility, make Python the number one choice for ML and AI-related development.

In the upcoming sections, we will be introducing a few popular Python libraries. With a view to showing how Python is used in fields such as ML and AI, the upcoming sections mention important libraries that are very useful for starting an ML and AI journey using Python programming.

In the next section, we will identify certain types of data, as the type of data inputted ML algorithms is important to consider.

Types of data

Data is a raw material that can be used to generate desirable output. Data can also be categorized. ML algorithms are also classified according to the types of data that are available in certain formats to them as input. There are not just basic data types such as numbers (integer, decimal, or complex) but also Booleans, strings, and more.

It's often the case as we progress through ML projects that we encounter the topic of data processing and data preprocessing. This aspect of data in ML is about preparing, cleaning, and organizing datasets, and even converting datasets into a certain format that is easily identifiable and ready to progress with building and training ML models. The process ensures the accuracy of the input and cuts down on the time that might have been consumed unnecessarily. In *Chapter 10*, similar topics were introduced with examples.

Here, we are talking about data categories – though most AI and ML libraries and frameworks accept various types of data, providing them with the main type or the required type will enhance and ensure the desired outcome. Listed here are a few types of data:

- **Quantitative**: Data that has numerical values – numbers that can be counted and measured. It plays an important role in statistical and data analysis projects' objective outcomes. Graphs such as scatter plots and line graphs are mostly used for visualization. Quantitative data can be of the following types:

    - **Discrete**: Includes countable whole numbers or concrete numbers. For example, the total number of players in a squad, female students in a class, or children in a house.

    - **Continuous**: Data might be complex, varying in nature over time, or infinite, and can be measured not counted. For example, height and weight, banks' interest rates, shoe size, or the temperature of the body.

- **Qualitative**: Categories or groups of data that cannot be expressed in a numerical format. It is used most of the time for subjective outcomes, as this type of data expresses the qualities and characteristics of the data. Graphs such as bar charts, horizontal bar charts, and pie charts are mostly used to visualize such data. Qualitative data can be of the following types:

    - **Structured**: This data can be numbers but is not applicable to mathematical operations:

        - **Nominal**: Values in the form of labels where no order or ranking exists, for example, gender, religion, or country

        - **Binary**: Only two options are available, for example, yes or no, true or false, pass or fail, or good or bad

        - **Ordinal**: Countable but not measurable, with some associated order, for example, blood group, academic grades, ranking, or movie ratings

    - **Unstructured**: There's no proper format, for example, images or sounds

Qualitative data collected from interviews and surveys is mostly unstructured, descriptive, and subjective in nature, which requires lots of interaction and interpretation. It is also difficult to collect (scrape) and store. Data lakes and non-relational **Database Management Systems** (**DBMSs**) are often prescribed to store such data. Coding with qualitative data is a challenge as it requires many rounds to process and understand the data.

On the other hand, quantitative data is mostly structured, easily accessible, and storable in data warehouses and **Relational Database Management Systems** (**RDBMSs**). It is also easy to code, search, and analyze.

> **Important note**
>
> From an analysis point of view, qualitative data is susceptible to many risk factors, if it is not processed and prepared in the correct way. There are ML and analysis libraries in Python that assist in converting and representing qualitative data as quantitative (using *deductive coding*, *inductive coding*, and various analysis methods such as *content analysis*, *thematic analysis*, *grounded analysis*, and *text vectorization*).

In the next section, we will list Python libraries with scientific and statistical significance.

Python for statistical and numerical computation

The development of ML-related algorithms and even models uses statistical and mathematical concepts. Python provides various in-built, installable, and open source modules for mathematical and statistical purposes.

Listed here are a few important libraries from a statistical and mathematical perspective:

- numpy: This Python library for numeric and scientific computation also deals with *n-dimension matrices, arrays, linear algebra*, and more. Please visit `https://www.numpy.org` for detailed information.

- statsmodel: A library for *statistical computation, statistical modeling, testing hypotheses, data exploration, estimations*, and more. Please visit `https://www.statsmodels.org` for detailed information.

- scipy: A Python library for scientific computation. scipy is used for *statistics, linear algebra, optimizing algorithms, n-dimension interpolations*, and more. Please visit `https://scipy.org` for detailed information.

The libraries that we just listed are quite large – ever-growing – with in-depth coverage of features and functionalities in their respective profiles. There might also be some overlapping of certain common features among the modules, but developers will find such overlapping and similarities between libraries and will opt for the one that best suits their implementation.

There are plenty of Python packages and projects that are built on top of certain important features of existing libraries, such as `numpy` and `scipy`, and standard libraries, such as `math`, `cmath`, `random`, and `statistics`. Please visit `https://docs.python.org/3/library` for more information on Python standard libraries and `https://pypi.org` for more information on third-party and open source libraries.

> **Important note**
>
> `https://pypi.org` (the **Python Package Index**) is a repository where we can find links and often complete details about packages, including their home pages, bug fixes, and more. Developers can also submit their projects to be released to global Python audiences and developers.

In the next section, we will learn about some of the Python libraries that are used specifically in ML projects.

Python libraries for ML

In this section, we will explore a few libraries that are quite popular in Python programming, especially in the field of ML and AI. It is clear and understood that statistics and mathematical procedures are compulsory in ML. There are dozens of Python libraries and frameworks we can find dedicated to ML. The number of such libraries is growing rapidly in terms of a few important aspects:

- Applicability to large volumes of data
- Able to handle different types of data
- Deployable and manageable in various systems and environments
- Adaptable with mining, analysis, visualization, multi-layer processing, and much more

It is very important that frameworks and libraries in the field of ML and AI are open to changes (studies and environments). There are plenty of examples of ML-based implementations and scopes. Listed here are a few selected Python modules that are making, and have made, a distinguished practical impact in the field of ML and AI using Python programming:

- **PyTorch**: An open source deep learning library and ML framework based on the `torch` library and used for both research and production. Fast execution, the capability to process very large datasets, cloud-based accessibility and support, and better debugging capabilities than competitors' libraries make PyTorch more significant. It is also popular in the field of computer vision and NLP. Please visit `https://pytorch.org` for more information.

- **TensorFlow**: A free, open source library from Google for AI and ML, it is mostly used for deep neural networks. Its scalability, flexibility, processes related to the development, training, and re-training of ML models, high-speed processing capabilities, ease of deployment, and applicability on top of various platforms make TensorFlow one of the best among its competitors. Please visit `https://www.tensorflow.org` for more details.

- scikit-learn **or sklearn**: One of the most popular, highly appreciated free ML libraries, built on top of `scipy`, `numpy`, and `matplotlib`, `sklearn` is suitable for all categories of developers, as it handles and facilitates various types of ML algorithms (*supervised and unsupervised*). It is often recommended as a first choice for developers to learn about and get used to ML. Please visit `https://scikit-learn.org/stable/` for detailed information.

The ML frameworks and libraries we just listed are a few of the best. There are plenty of ML-related libraries, such as SpaCy, Theano, Transformers, and Keras, that can be found on the internet. Some such libraries are dedicated to particular ML- and AI-related issues, such as *NLP* (NLP is considered a subset of ML, dedicated to text and languages), *text analysis*, *computer vision*, *image classification*, and *speech processing*.

Though we have plenty of choices of libraries to use for ML, features such as data analysis, visualizations, and **Exploratory Data Analysis** (**EDA**). Certain ML libraries use their in-built methods and attributes for this, whereas a few work jointly with dedicated libraries, such as the following:

- `pandas`: For data **input/output** (**I/O**), analysis, and manipulation
- `matplotlib`: For visualizing dataset and model outcomes
- Libraries such as `seaborn`, `plotly`, and `bokeh` are also used for visualization purposes

Choosing libraries that are appropriate for the task is a challenge in itself. In this section, only some of the common libraries were introduced. It's a developer's core duty and responsibility to analyze scenarios, types of data, and volumes of data to work with and choose the appropriate libraries.

In the next section, we will learn about various types of ML algorithms.

Types of ML

ML is a growing and evolving field of AI and computer science. With statistical and computational capabilities, on top of raw or sample data, ML libraries use plenty of algorithms to build a model and to provide or assist us with information. Data mining techniques also use various ML algorithms to discover patterns in data.

There are plenty of ML algorithms (learning algorithms), and they are categorized into three major categories, as listed here, based on their training process:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

In the next sections, we will be exploring each type of ML algorithm in some detail.

Supervised learning

In this type of learning, machines are provided with or trained using training data (labeled data), and finally, they predict or classify the output based on the training data. The output is based on the input data supplied and the algorithm learns from the training data. A quality dataset (clean, formatted, preprocessed, and EDA conducted) must be available if we want the nearest possible output or prediction. Here, the output variable is dependent on the input variable.

Listed here are a few steps explaining how the supervised learning technique works:

1. An input dataset, also known as a labeled dataset, is selected.
2. The dataset is chunked into two sets, training and testing (normally 80% for training and 20% for testing). Records or rows are selected either randomly or sequentially.
3. An input variable (or variables) is determined or picked.
4. An applicable learning algorithm is chosen (such as linear regression).
5. The selected algorithm is executed on the training set, whereas the efficiency of the model is evaluated with the testing set.
6. If the output or predicted result is correct, then the model is accurate. If not, *steps 1* to *5* are repeated with a different algorithm.
7. New sets of data are provided to the trained model and outputs are received.

> **Important note**
>
> If the output does not result in the correct classification or is not near the calculated value, the input dataset is thoroughly analyzed again using EDA features and processed further, and training is conducted. In the case of regression, the effectiveness of the regression predictive model can be measured by calculating the **Root Mean Square Error** (**RMSE**).

A supervised model uses prior observations, data, and inputs to classify or predict the new output. This type of algorithm is useful in daily, basic activities. It will not work if the data is different for training and testing sets. For complex, inaccurate input data, these models will require lots of computation and might not reveal accurate results.

Supervised learning can be used in data filtering, house price prediction, sentiment analysis, spam detection, classifying images, fraud detection, and more.

Based on the applicable problems, supervised learning algorithms are basically classified into two types – classification and regression.

Classification

Grouping, identifying, or categorizing a data model based on the attributes is known as classification. The classification technique normally involves an arrangement based on similarities or dissimilarities and is also known as the **sorting** technique. In this technique, categories or classification labels, also known as classes, are pre-determined or set in advance. The classification model identifies which category a dependent variable belongs to, based on the single or multiple independent variables.

Some of the common classification algorithms are listed here:

- **Linear classifier**: The simplest form of classification algorithm, it classifies a set of data based on a simple linear function, assuming data is distinguishable linearly by plotting a straight line on a graph.

- **Support Vector Machine** (**SVM**): A classifier that separates data into multiple sets, based on boundaries (hyperplanes), by detecting overlapping data. If the dimensions are greater than the number of samples, SVM can be used. The SVM algorithm creates points that are closest to the line from different dimensions called **Support Vectors** (**SVs**). The SVM algorithm results in high accuracy, with a dataset that has less noise (overlapping). Outlier detection is one of the major advantages of the SVM algorithm. The SVM algorithm is not really suitable for large datasets, as the training complexity is high, and the model makes decisions on top of complex boundaries.

- **K-Nearest Neighbors** (**kNN**, **k-NN**, or **KNN**): KNN is known for its high accuracy. It stores available data, generates new data points based on the similarity, and finds the nearest neighbor. It calculates the *Euclidean distance* (between two points) on *K* number of neighbors. KNN is often used on top of large datasets. It is also a bit slow, as it has to generate and calculate distances between new data points.

- **Decision tree**: This algorithm uses a set of rules to make decisions. Rules are set in the form of a tree structure that contains a *root (issue)*, *branches (decision rules or decision nodes)*, and *leaves (leaf nodes)* representing the outcome. A tree branch node can have multiple branches. Training datasets are broken down into many subsets by applying rules with categorical and quantitative data. Data in the tree must be numerical. It's comparable to the thinking ability of humans and deeply nested `if-elif` statements in coding. A decision tree with a large number of branches is complex and time-consuming.

- **Random forest**: This is a random collection of *decision trees* (a forest), where multiple results are combined into a single value treated as output. Random forest is also known as **ensemble learning**. It reduces overfitting, as the result is often an average value of decision trees. Because of its suitability for noisy datasets, scalability, low training time, and highly accurate results, it is used on large datasets.

Classification learning can be applied in various situations; for example, images of clothes can be classified by color name, emails in an inbox can be classified as spam or not spam, or common diseases can be classified by studying symptoms.

Regression

Regression estimates the relationship between variables. It is also known as a **predictive algorithm**. A regression model attempts to determine the strength of the relationship between dependent variables and one or more independent variables. The output variable in regression is generally a quantitative value. Normally, regression techniques are applied to make predictions from a given set of inputs called **predictors**. A strong statistical background is necessary to handle regression algorithms.

Listed here are a few common types of regression algorithms:

- **Linear regression**: Also known as **Simple Linear Regression** (**SLR**), this is used to predict one (dependent variable) value based on the value of one independent variable. It finds the linear correlation between variables, using the best-fitting straight line. It is best for small datasets and results in continuous output.

- **Multilinear regression**: Also known as **Multiple Linear Regression** (**MLR**) is a type of linear regression model. It models linear and non-linear relationships with more than one independent variable and one dependent variable. The output reflects the influence or relationship between multiple independent and single dependent variables.

- **Logistic regression**: This is also known as a *statistical regression* model or probabilistic algorithm, which helps in analyzing datasets. This algorithm is used to find the best-fitting model that calculates the probability of binary types (such as yes and no). It also describes the relationship between diploid (the involvement of two possible outcomes) or dependent variables and sets of predictors. In terms of the number of variables, the algorithm describes the relationship between multiple independent and single dependent variables. This regression technique results in a discrete output and is used mostly with classification-related problems.

- **Polynomial regression**: Polynomial regression describes *non-linear relationships*. Generally, if there's no possibility of a linear relationship between dependent and independent variables, it might be a case of polynomial regression. Polynomial regression is also considered a special case of MLR. With a trained model, a non-linear equation is created and used for prediction.

There are many more forms of regression learning. Measuring the effectiveness of algorithms is also quite challenging. A few techniques, such as **Ordinary Least Squares** (**OLS**) and **R-Squared** (**R2**), are used to minimize the possibility of errors. OLS estimates the coefficients of linear regression. It minimizes the sum of squared residuals between actual and predicted values. R2, with values between 0 and 1, calculates the variation percentage of the dependent variable. The greater the value of R2, the better the model is.

Regression-based learning can be used in cases such as weather forecasting, sale price prediction, house price prediction, and stock markets.

Important note

There are a few supervised ML algorithms that can be used for both regression and classification. Logistic regression is most often used for classification-related problems. SVM, being a deep learning model, is used for both classification and regression. Similarly, algorithms such as random forest, kNN, and decision tree are also used for both types of supervised learning.

Choosing the best learning model depends on the type of data, the volume of data, the correlation between columns of data, processing time, expected output, the inclusion of variation in data, and many more factors.

With data collected using web scraping, a supervised form of ML is the most used format for analysis and decision-making. In the next section, we will explore unsupervised ML.

Unsupervised learning

Unsupervised learning is effective in discovering trends and finding hidden patterns in data. In unsupervised learning, the algorithms are left to detect, explore, and learn on their own. There is nothing to supervise the learning process. This type of learning is beneficial in the field of data mining, and also when we are not sure of what we are looking for.

In comparison to supervised learning, unsupervised learning tasks are complex, with many challenges. In unsupervised learning, time to prepare data and perform preliminary analysis is not required. Hidden or unknown patterns of data are exposed, which is not possible in supervised learning. Results from unsupervised learning might be low in accuracy, in comparison to supervised learning.

Unsupervised learning is mostly divided into two parts:

- Clustering
- Association

In the next section, types of unsupervised learning are introduced.

Clustering

Clustering is a process of grouping, sorting, categorizing, and classifying data or objects. Classification models in supervised learning also group and categorize input objects. Classification models are provided with predefined categories or group names, whereas in clustering the groups or categories are generated after processing the data.

Groups or clusters of data generated will contain data with similar characteristics and types. The flexibility of clustering is the most in-demand feature. Learning models, once trained, can be used in various other scenarios.

K-means clustering, one of the most widely used clustering algorithms, demands the value of k. Here, k refers to the number of clusters that are to be created or the data that is to be divided and put into groups. It identifies a seed point as a centroid (mean point) in the current cluster and collects data in a cluster that is nearest to the centroid.

Similar to kNN, distance measurement (*Euclidean*) is conducted between pieces of data or data points, and is used to create or separate clusters that are similar and dissimilar. kNN and k-means clustering are often categorized as search-based (learning) algorithms. The data points use themselves to find other data points.

Association

Association helps to discover new patterns, find relationships in data, and produce information from large datasets. It identifies the frequency of patterns in a dataset. In addition, it maps the data based on its dependencies.

Association, also known as **Association Rule Mining** (**ARM**), tries to identify associations among dataset variables. One of the common algorithms used for ARM is **Apriori**, which discovers the most frequent data items in datasets and identifies associations and correlations among them. ARM is normally processed in large datasets, but is likely to carry lots of noise as there is the possibility of the number of associations being infinite. Most of the time, binary attributes such as yes and no are used to define association rules.

Please refer to *Chapter 10* for more detailed information. In the next section, we will learn about the basics of the reinforcement type of learning.

Reinforcement learning

This type of learning algorithm uses and recognizes its environment and learns the behaviors by accessing data patterns or trains its learning model based on trial and error (also known as the reward and punishment model). Reinforcement learning is about making decisions or learning actions (based on the situation) and their feedback, which results in the maximum reward.

Reinforcement learning algorithms learn from a chain of action that is almost similar to decision tree processing. For example, situation-based input results in output that becomes input for the next cycle and so on, until a decision is made or obtained.

Reinforcement Agents (**RAs**), also called AI-driven systems, are self-trained and make the decision to resolve an assigned task based on the training data. To analyze or recognize situations, external agents and systems such as software (logs), system settings, browser-based extensions, cameras in machines, and more are used to find the most information possible about the situation.

So far in this chapter, information regarding ML-related features has been explored. Now, we will build an ML model and use practical concepts of some ML types that can be applied for predictions and analyzing sentiments.

In the next section, we will explore practical ML using Python programming.

ML using scikit-learn

To develop a model, we need datasets. Web scraping is again the perfect technique to collect the desired data and store it in the relevant format. There are plenty of ML-related libraries and frameworks available in Python, and they are growing in number. scikit-learn is a Python library that addresses and helps to deal with the majority of supervised ML features.

scikit-learn is also known and used as `sklearn`. It is built upon `numpy`, `scipy`, and `matplotlib`. The library provides a large number of features related to ML aspects such as *classification*, *clustering*, *regression*, and *preprocessing*. We will explore beginner and intermediate concepts of the *supervised learning* type with *regression* using scikit-learn. You can also explore the `sklearn` user guide available at `https://scikit-learn.org/stable/user_guide.html`.

We have covered a lot of information about **regression** in previous sections of this chapter. Regression is a supervised learning technique that is used to make predictions based on labeled data. In the next sections, SLR is explored with a few examples.

Simple linear regression

SLR is used to estimate the relationship between an independent variable and a single dependent variable. It is usually used as the default regression technique and is used to predict the value of one variable based on the value of another variable, which is also known as the predictor variable or input variable for SLR.

We will now discuss certain SLR concepts using code. The code files can be found here: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter11/Chapter11_regression.ipynb`.

The data (`Fish.csv`) used for the examples has been collected from **Kaggle** (`https://www.kaggle.com/datasets/aungpyaeap/fish-market/download`) and has been read to a `DataFrame` named `input` using `pandas`. For charts and graphs, `plotly` will be used.

As seen in the following code, we import regular libraries such as `pandas`, `numpy`, and `plotly`. The `Fish.csv` data file has been read using the `pandas` method `read_csv()`, which will create a `DataFrame` type of object named `input`:

```
import pandas as pd
import numpy as np
import plotly.express as px
pd.options.plotting.backend = "plotly" # use plotly as plotting option
input = pd.read_csv("Fish.csv")        # load data
```

`input` contains data about a few fish species, including measurement-related, as listed here:

- `Species`: Species name

- `Length1`: Vertical length in cm

- `Length2`: Diagonal length in cm

- `Length3`: Horizontal length in cm

- `Weight`: Weight of fish in g

- `Width`: Diagonal width in cm

The `pandas` method `info()`, as shown in *Figure 11.1*, displays the column names, `Dtype` (data type, such as object or float), and `null` or `non-null` count-like information from the `DataFrame` input:

```
input.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 159 entries, 0 to 158
Data columns (total 7 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Species  159 non-null    object
 1   Weight   159 non-null    float64
 2   Length1  159 non-null    float64
 3   Length2  159 non-null    float64
 4   Length3  159 non-null    float64
 5   Height   159 non-null    float64
 6   Width    159 non-null    float64
dtypes: float64(6), object(1)
memory usage: 8.8+ KB
```

Figure 11.1: Dataset info Fish.csv

As seen in *Figure 11.1*, there is a total of seven columns in the dataset. Six of them are of the `float` type, and in total there are 159 rows of records. The dataset looks clean, as there are no null values.

Studying the dataset is compulsory to keep an eye on preprocessing activity such as filling in null values, dropping duplicates, converting data types, and sorting. These activities are compulsory as, during regression processing, we need to find independent and dependent data as variables, and can't compromise on the output (prediction) quality.

The `input.describe()` code works only with statistical value-carrying columns. With the `include='all'` parameter, even string and other objects can be included, as seen in *Figure 11.2*. We can see that, among the seven unique species, Perch is the one that occurs most; it occurs *56 times out of 159.*

```
input.describe(include='all')
```

	Species	Weight	Length1	Length2	Length3	Height	Width
count	159	159.000000	159.000000	159.000000	159.000000	159.000000	159.000000
unique	7	NaN	NaN	NaN	NaN	NaN	NaN
top	Perch	NaN	NaN	NaN	NaN	NaN	NaN
freq	56	NaN	NaN	NaN	NaN	NaN	NaN
mean	NaN	398.326415	26.247170	28.415723	31.227044	8.970994	4.417486
std	NaN	357.978317	9.996441	10.716328	11.610246	4.286208	1.685804
min	NaN	0.000000	7.500000	8.400000	8.800000	1.728400	1.047600
25%	NaN	120.000000	19.050000	21.000000	23.150000	5.944800	3.385650
50%	NaN	273.000000	25.200000	27.300000	29.400000	7.786000	4.248500
75%	NaN	650.000000	32.700000	35.500000	39.650000	12.365900	5.584500
max	NaN	1650.000000	59.000000	63.400000	68.000000	18.957000	8.142000

Figure 11.2: Describing details of dataset columns

Individual counts, or the value_counts() method of the species, are plotted using the following code:

```
species_count = input.Species.value_counts()
species_count.plot(title="Species Count", labels=dict(index="Species_
Name", value="Species_Count", variable="Fish"))
```

As seen in *Figure 11.3*, Species_Count, or the number of occurrences of certain species, is counted from the DataFrame input and plotted with the species name:

Figure 11.3: Plotting species count

While analyzing the dataset, there seem to be many correlated factors. Therefore, we will predict `Weight` by training a model on values of `Length3`, only for the targeted species named `Perch`:

```
perch = input[input.Species=="Perch"]
```

A Plotly scatter plot supports the regression trends using the `trendline` argument (https://plotly.com/python/linear-fits/#linear-fit-trendlines-with-plotly-express), as implemented in the following code. This code uses `ols` as the `trendline` argument:

```
fig = px.scatter(perch, x="Length3", y="Weight", trendline="ols")
fig.show()
```

As seen in *Figure 11.4*, we can find the *linear equation* along with the calculated value of *R2*, the *coefficient*, *intercept*, and the *predicted weight for the chosen* `Length3`. The `ols` argument for `trendline` calculates this for us, and we can interpret `Length3` and the weight of `Perch` in the chart displayed in *Figure 11.4*:

Figure 11.4: OLS trendline, Length3, and weight of Perch

From *Figure 11.4*, we can obtain the following listed properties of the SLR equation:

- **R2**: 0.920652 (accuracy score – 1 is the highest)
- **Linear equation**: Weight = 35.0009 * Length3 + -652.787
- **Intercept**: -652.787
- **Coefficient**: 35.0009

> **Important note**
> Values such as R2 and coefficient help us to validate the model. It should also be noted that we have not used `sklearn` yet.

To predict `Weight` using the `Length3` value (of the species named `Perch`) using `sklearn`, we need to use the following process:

1. Identify dependent and independent variables.
2. Split the data into training and testing sets.
3. Create a linear regression model.
4. Fit the linear regression model with training data.
5. Finally, make predictions on the test data.

As seen in the following code, `sklearn` has various modules, such as `model_selection`, `linear_model`, and `metrics`:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_
absolute_error
```

`train_test_split()` is provided with `Length3` and `Weight` values. This will create training and testing sets. A training set will occupy 80% of the values as set by `train_size`, and the remaining 20% will be taken by the testing set. There's also an extra parameter, `shuffle`, being used, which randomizes the order of records available:

```
X_train, X_test, y_train, y_test = train_test_split(perch.Length3,
perch.Weight, train_size=0.8, shuffle=True)
```

The model named `lr` is created by fitting it with the training data. The `fit()` method is provided with training data:

```
lr = LinearRegression().fit(X_train, y_train)
type(lr)
sklearn.linear_model._base.LinearRegression

dir(lr)
['__abstractmethods__', …., '_more_tags', '_parameter_constraints',…
.,'coef_', 'copy_X', 'feature_names_in_','fit', 'fit_intercept',
'get_params', 'intercept_', 'n_features_in_', 'n_jobs', 'positive',
'predict', 'rank_', 'score', 'set_params', 'singular_']
```

As seen in the preceding code, with the `lr` model having been created, we can explore some of the important `lr` attributes as follows:

```
lr.intercept_      # array([-634.55508041])
lr.coef_           # array([[34.68344575]])
lr.rank_           # 1
lr.score(X_train, y_train)         # 0.9277688566667214
```

In the preceding code, we are going to predict weight based on the data available or `X_test`. The `weight_pred` prediction array can now be made using the `lr` model, the `predict()` method, and `X_test` with 20% of the records used to predict `Weight`:

```
weight_pred = lr.predict(X_test)
weight_pred
array([[461.4418054 ], [180.5058948 ],[801.33957378],[659.13744619
], [544.68207521], [801.33957378], [360.85981271], [204.78430682],
 [731.97268228],   [284.55623206], [232.53106343],[274.15119833]])
```

Similar to `X_test` values, we can provide the model predictor with a new value of `Length3` and get the predicted result (`Weight`):

```
lr.predict(np.array([[20.5]])) [0] [0]
# 76.45555753771407
lr.predict(np.array([[29.5]]))
# array([[388.60656932]])
lr.predict([[29.9,]])
# array([[402.47994762]])
```

By having the prediction model ready, we can also use it to calculate the performance and evaluate the model itself using methods from `sklearn.metrices. r2_score(0.86)` is also interpreted as the accuracy score, which floats between 0 and 1 – the higher the score, the better the accuracy. This shows that the model is acceptable to predict outcomes:

```
mean_squared_error(y_test, weight_pred)
# 9910.93927217145
mean_absolute_error(y_test, weight_pred)
# 87.86230207025248
mean_squared_error(y_test, weight_pred, squared=False)
# 99.5537004443
np.sqrt(mean_squared_error(y_test, weight_pred))
# 99.5537004443
r2_score(y_test, weight_pred)
# 0.8681631783134378
```

> **Important note**
>
> The **Mean Squared Error** (**MSE**) value of 9910.9392 is the mean of the sum of the square of residuals. The RMSE value of 99.5537 is the square root of the MSE. The **Mean Absolute Error** (**MAE**) value of 87.8623 is the mean of the sum of the absolute values of residuals. In general cases, the MSE is always greater than the MAE.

We have explored SLR techniques in this section with code examples, as well as identifying various attributes from `sklearn`. In the next section, we will be using MLR.

Multiple linear regression

MLR is used when there is a correlation between multiple independent variables and a single dependent variable. In this case, we want to check the correlation of values from the `Width` and `Height` columns with `Weight` for the `Perch` species in the dataset used in SLR in the *Simple linear regression* section.

We will now discuss certain MLR concepts using code, and here is where the code files are available: `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter11/Chapter11_multi_regression.ipynb`.

`inDepd` contains independent variables collected from `Perch`, whereas `depd` is the `Weight` dependent variable:

```
inDepd = perch[['Width', 'Height']]# independent variables
depd = perch['Weight']            # dependent variable
```

Again, 80% of the values are used for training and 20% for testing. Training and testing sets are created using the `train_test_split()` method of the `sklearn.model_selection` model:

```
X_train, X_test, y_train, y_test = train_test_split(inDepd, depd,
train_size=0.8, shuffle=True)
```

The `mlr` model is created and fitted or trained using the training dataset:

```
mlr = LinearRegression().fit(X_train, Y_train)
mlr.score(X_train, y_train)
0.9560246146854837
```

`X_train` contains multiple pieces of data to train the model, as seen in *Figure 11.5*:

	Width	Height
113	6.7408	10.6091
86	3.4075	6.1100
96	3.7230	7.2930
75	2.6316	4.5924
106	4.2042	7.8204
101	4.1440	7.1680
74	2.4320	3.8240
76	2.9415	4.5880
119	7.1064	11.9286

X_train[::5]

Figure 11.5: X_train with multiple independent variables

After training, the `mlr` model is used to predict the data from `X_test`. It is also to be noted that `r2_score` is `0.81799`, which is acceptable in MLR cases:

```
weight_pred_mlr = mlr.predict(X_test)
weight_pred_mlr
array([643.07827704,96.06295718,228.07992289,76.27638592,136.430909,
143.8607218,163.98405844, -300.15848523,
446.23998044,  291.21291802,  944.3976018, 243.67945607])

r2_score(y_test, weight_pred_mlr)
0.817999125036539
```

Inputs for `Width` and `Height` can also be provided directly to `mlr.predict()`. For example, see the following code block:

```
mlr.predict(np.array([[3.45, 6.05]]))
array([154.53150904])

mlr.predict(np.array([[7.10, 11.22]]))
array([807.225824])
```

> **Important note**
>
> As seen in the code implementation in this section, scikit-learn possesses a large set of assets for various ML-related concepts. If users are seeking a large number of statistical and quantitative values from their ML models, then `statsmodel` and `scipy` are more applicable, because of the short code.

In the next section, we will use an NLP-related technique.

Sentiment analysis

NLP is a subset of ML, and sentiment analysis is a subset of NLP. Using sentiment analysis, we can detect whether the sentiment of a text is positive, neutral, or negative. Sentiment analysis is done on text, reviews, feedback, and more.

This analysis helps to evaluate client or customer reviews and feedback and whether they are intended as positive or negative. The detected sentiment allows better marketing, and companies, e-commerce sites, banks, travel offices, and many more organizations can monitor and manage their strategies, marketing policy, product branding, customer needs, and more.

We will be using code and examples in the following sections that are available at `https://github.com/PacktPublishing/Hands-On-Web-Scraping-with-Python-Second-Edition/blob/main/Chapter11/Chapter11_textblob_vader.ipynb`.

The `nltk` Python library is one of the pioneers of NLP. There are plenty of ways that NLP can be used, such as classification. In the current case, we will be using the `textblob` and `vaderSentiment` libraries:

- `textblob`: This is a text-processing library, built on top of `nltk`. It can be used for various NLP aspects, such as POS tagging, sentiment analysis, lemmatizing, tokenizing, and text classification. To analyze the sentiment of text, `textblob` calculates and returns `polarity` and `subjectivity` as sentiment results:

  - `polarity`: Indicates the positive or negative sentiment of the text. The values range from -1 (very negative) to 1 (very positive), with 0 being neutral.

  - `subjectivity`: Indicates personal feeling. The values range from 0 (objective) to 1 (subjective).

- `vaderSentiment`: This is a social media sentiment analyzer. Similar to `TextBlob`, **Valence Aware Dictionary and Sentiment Reasoner** (**VADER**) is a rule-based sentiment analyzer. VADER has been trained with a large collection of social media texts. `vaderSentiment` returns these four elements:

  - `compound`: This is the valence score of words in the lexicon. Values range from -1 (extremely negative) to 1 (very positive).

  - `pos`: A positive value with a compound score of >=0.05.

  - `neg`: A negative value with a compound score of <=-0.05.

  - `neu`: A neutral sentiment with a compound score of >=-0.05 and a compound score of <0.05.

Sentiment analysis is done upon the `quote_details.csv` data, which was scraped from the `http://quotes.toscrape.com/` URL in *Chapter 8*. The column named `quote` is only required in the current case. It's the only string or text element that will be used with `textblob` and `vaderSentiment` to analyze sentiment. Individual columns or groups of selected columns can only be read from a file by using the `usecols` parameter in `pd.read_csv()`:

```
import pandas as pd
quotes = pd.read_csv("quote_details.csv",usecols=["quote"])
quotes['quote'][0]
'The world as we have created it is a process of our thinking. It
cannot be changed without changing our thinking.'
```

With the content read to be explored as the `quotes` DataFrame, in the following examples, we will analyze the sentiment of the contents using two quite popular Python libraries.

Example 1 – using textblob

The `textblob` library provides a class named `TextBlob`, which holds various text-processing-related attributes and methods, such as `tags`, `noun_phrases`, `words`, and `word_counts`:

```
from textblob import TextBlob
text = TextBlob(quotes['quote'][1])
type(text)                 # textblob.blob.TextBlob
print(dir(text))
['__add__',……,'ends_with', 'endswith', 'find', 'format', 'index',
'join', 'json', 'lower', 'ngrams', 'noun_phrases', 'np_counts', 'np_
extractor', 'parse', 'parser', 'polarity', 'pos_tagger', 'pos_tags',
'raw', 'raw_sentences', 'replace', 'rfind', 'rindex', 'sentences',
'sentiment', 'sentiment_assessments', 'serialized', 'split', 'starts_
with', 'startswith', 'string', 'strip', 'stripped', 'subjectivity',
'tags', 'title', 'to_json', 'tokenize', 'tokenizer', 'tokens',
'translate', 'translator', 'upper', 'word_counts', 'words']
```

To analyze the sentiment of text, `TextBlob` can be provided with a string or text and `sentiment` access attribute. The `sentiment` attribute returns a `Sentiment` object with calculated values for `polarity` and `subjectivity`. An object element – say, `polarity` – can be accessed as `text.sentiment.polarity`:

```
text = TextBlob(quotes['quote'][1])
text.sentiment  # Sentiment(polarity=0.3, subjectivity=0.75)
text.sentiment.polarity    # 0.3
```

There is a total of 100 quotes available in the dataset. Therefore, iteration has been used to calculate the sentiment of each record and collect them in Python lists named `polarity` and `subjectivity`:

```
polarity, subjectivity=[],[]
for quote in quotes["quote"]:
    text = TextBlob(quote)
    print(f" Text {text.sentiment}")
    polarity.append(text.sentiment.polarity)
    subjectivity.append(text.sentiment.subjectivity)
```

Finally, the collected `polarity` and `subjectivity` lists are added to `quotes`:

```
quotes['polarity']=polarity
quotes['subjectivity']=subjectivity
```

quotes, as seen in *Figure 11.6*, now has enough information about `sentiment`, which can be analyzed:

Figure 11.6: DataFrame quotes with polarity and subjectivity

The details related to sentiment are obtained from the chosen dataset using `textblob`. While we use ready-to-use libraries for procedures like in this example, the result might not be more insightful compared to the result we can obtain using the dataset.

So, `vaderSentiment` will be used on the same dataset (`quotes`) in the next section, which will help generate more information and help in comparing the results obtained with `textblob`.

Example 2 – using vaderSentiment

The `SentimentIntensityAnalyzer` class of `vaderSentiment` has the `polarity_score()` method, which accepts a string and returns four keys – `pos`, `neg`, `neu`, and `compound`. Each returned key's value is collected and added to the original *DataFrame*, `quotes`:

```
positive, negative, neutral, compound=[],[],[],[]
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
for quote in quotes["quote"]:
    text = analyzer.polarity_scores(quote)
    positive.append(text['pos'])
    negative.append(text['neg'])
    neutral.append(text['neu'])
    compound.append(text['compound'])
quotes['positive']=positive
quotes['negative']=negative
quotes['neutral']=neutral
quotes['compound']=compound
```

As seen in *Figure 11.7*, there's only one record that satisfies the query, `positiveQuotes`:

```
positiveQuotes = quotes.query("positive >= 0.5 and polarity >= 0.5")

positiveQuotes[['quote','polarity','positive']]
```

	quote	polarity	positive
18	Good friends, good books, and a sleepy conscie...	0.766667	0.577

Figure 11.7: Quote with maximum positive sentiment

The `negativeQuotes = quotes.query("negative < 0.5 and polarity < 0.5")` query returns 90 records.

Figure 11.8 displays the statistical details of all calculated and added column values. The highest or maximum rated `positive`, `negative`, and `polarity` values are `0.57`, `0.42`, and `0.766`, respectively.

```
quotes.describe()
```

	polarity	subjectivity	positive	negative	neutral	compound
count	100.000000	100.000000	100.000000	100.00000	100.00000	100.000000
mean	0.107863	0.446212	0.161550	0.09602	0.74248	0.188439
std	0.254294	0.319113	0.144779	0.11379	0.17783	0.494445
min	-0.800000	0.000000	0.000000	0.00000	0.37300	-0.918900
25%	0.000000	0.150000	0.000000	0.00000	0.60750	-0.099500
50%	0.050481	0.496875	0.158500	0.06700	0.73000	0.042650
75%	0.261719	0.637500	0.260750	0.17800	0.87925	0.688150
max	0.766667	1.000000	0.577000	0.42500	1.00000	0.995100

Figure 11.8: Quotes with TextBlob and vaderSentiment values

Figure 11.9 shows the plotting of `quotes[['positive', 'negative']]`:

Figure 11.9: Plotting of positive and negative values

With some data and analysis done in this section, it might be fair to suggest that most of the records found in `quotes` are neutral in nature.

> **Important note**
>
> Sentiment analysis using `nltk` involves more steps in comparison to `textblob` and `vaderSentiment`. In addition, it has by far the largest collection of methods and attributes. Please visit `https://www.nltk.org/` for more details.

In this chapter, we learned about a few basics regarding ML and how ML, with its various techniques, can be helpful and play a significant role in various situations when quality data scraped from the web is provided.

Summary

Python programming makes a huge contribution in AI- and ML-related domains. In this chapter, we have had only a glimpse of that. Quality data plays a very important role in ML. Whether collecting data via web scraping and storing it or providing scraped data on the fly to an ML model, prepared data is in demand. The better the quality of the data – and the more precise the data is – that we provide to ML algorithms, and for plotting charts, the more accurate results, visualizations, and descriptive plots we can expect.

We have now learned about ML concepts and various aspects of ML by exploring them. We have also learned how to implement ML models and collect the results, if required, from various processes. To summarize, we now have an overview of how to use scikit-learn and conduct sentiment analysis. ML is data-driven and quality data is a basic requirement for ML models to provide accuracy.

In the next chapter, we will learn about a few further steps and topics that are beneficial and applicable to web-scraped data and can be used in the web scraping process.

Further reading

- **Machine learning**:

    - `https://www.ibm.com/topics/machine-learning`

    - `https://www.mygreatlearning.com/blog/what-is-machine-learning/`

    - `https://www.run.ai/guides/machine-learning-engineering/machine-learning-automation`

- **Machine learning – Python**:

    - `https://scikit-learn.org/`

    - `https://www.w3schools.com/python/python_ml_getting_started.asp`

    - `https://realpython.com/tutorials/machine-learning/`

    - `https://machinelearningmastery.com/machine-learning-in-python-step-by-step/`

- **Regression analysis**:

    - `https://www.alchemer.com/resources/blog/regression-analysis/`

    - `https://www.investopedia.com/terms/r/regression.asp`

- **Natural language processing**:

    - `https://www.nltk.org/`

    - `https://www.ibm.com/topics/natural-language-processing`

    - `https://hbr.org/2022/04/the-power-of-natural-language-processing`

- **Sentiment analysis**:

    - `https://monkeylearn.com/sentiment-analysis/`

    - `https://huggingface.co/blog/sentiment-analysis-python`

    - `https://aws.amazon.com/what-is/sentiment-analysis/`

Part 5: Conclusion

In this part, you will be provided with a few select advanced concepts that can be implemented alongside or on top of scraped data or datasets, for advanced processing and features. You will also learn about the jobs and career-related scopes related to web scraping and data science that are in high demand globally.

This part contains the following chapter:

- *Chapter 12, After Scraping – Next Steps and Data Analysis*

<div align="right">

12

</div>

After Scraping – Next Steps and Data Analysis

So far, we have learned how to scrape the web, analyze the data collected, and apply **machine learning** (**ML**) algorithms using Python programming.

This chapter will provide a basic introduction to some emerging concepts and technologies that are becoming crucial in the field. Being able to draw insights and knowledge from data is in high demand, and even supply-related scopes have been growing exponentially. From the fields of research to business, the importance of data has grown rapidly. **Artificial intelligence** (**AI**)-based systems, powered by ML logic, will continue to consume and generate more data.

In this chapter, we will learn about the following topics:

- What happens after scraping?
- Web requests
- Data processing
- Jobs and careers

Technical requirements

A web browser (*Google Chrome* or *Mozilla Firefox*) is required to explore the provided resources.

What happens after scraping?

Web scraping is a method of collecting and supplying quality data, and applying various techniques that depend on data. Any faculty or domain that involves data provides the opportunity to learn and grow as data collected from scraping is used for many other tasks such as analysis, reporting, dataset creation, and mining. Web-scraping-related techniques have always been dynamic and challenging, alongside the growth of web-based technologies.

Systems based on data, returning raw data, processed data, and visualization plots, in the form of images and videos, are growing by involving global audiences in multiple forms. **Information technology (IT)**-driven systems are core applications used in all industries, and Python programming has played a major role in the development and processing of data-related systems.

Earlier chapters of this book presented various steps that consolidated a few main concepts and actions, as listed here:

- Demand for data
- Collection of data
- Data analysis (information generation)
- Use of data (ML/AI)

A large number of frameworks and programming languages are being developed. Applications related to data, information, and knowledge management are most used in business- and research-based fields.

While we have discussed various features and techniques of web scraping and collecting data, as entities are constantly being created and evolving, being able to adapt our approaches and techniques is important. Some of the factors, such as the effectiveness of the system, consumption of time, content volume, and availability of applicable technology, affect the overall web scraping process, and even the steps carried out after data has been collected.

In the next section, we will introduce some advanced technologies and Python libraries that deal with web-related requests.

Web requests

Throughout the chapters of this book, the `requests` Python library has been used to establish communication between the code and the web. Plenty of Python libraries can be found at `https://pypi.org/` if we search for ones similar to `requests`.

The following subsections list some Python libraries and technologies and provide brief introductions to them.

pycurl

The `pycurl` Python library (`http://pycurl.io/`) is a wrapper on top of the popular `libcurl` library. `libcurl` is one of the earliest Python libraries that was used to communicate with websites on the internet, based on the **curl** tool (also known as **cURL**).

`curl` (`https://curl.se/`) is a command-line tool that is used to connect and transfer data over the web. `curl` is the basis of network communication; it's a core implementation that is used with the help of a wrapper across different **operating systems (OSs)**, browsers, and machines that communicate

with the internet. The `curl` command is machine-independent. Programming languages such as *Python*, *PHP*, *C*, *R*, *Java*, and *Rust* convert the raw `curl` commands into their preferred format, which can be used in their code or by their libraries.

`pycurl`, developed in the *C* programming language, provides `curl`-based facilities to Python code. The `requests` library is often considered easy to use, in comparison to other libraries. `pycurl` provides almost all of the same features and functionality as `curl`, including its attributes and methods.

Listed here are some of the major advantages that `pycurl` possesses over `requests`:

- Supports a large number of communication protocols (*HTTPS*, *POP3*, *FTP*, *LDAP*, *IMAP*, and many more)

- Faster in comparison to `requests` and other similar Python libraries

Modern browsers even provide the raw `curl` command, which can be found using browser-based developer tools, as mentioned earlier in *Chapter 2*. These `curl` commands can be copied for use with the following:

- **Command-line interface** (**CLI**) applications (Bash shells or scripts) or terminals

- Web-based terminals and API tools such as **Postman** (`https://www.postman.com/`)

- Python libraries such as `requests` (`https://curlconverter.com/python/`) with headers

- Converting a command into other programming languages

Figure 12.1 shows the `curl` command to load `https://curl.se/` copied from the browser-based developer tools using the **Copy as cURL (bash)** option.

```
curl 'https://curl.se/' \
  -H 'authority: curl.se' \
  -H 'accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,
application/signed-exchange;v=b3;q=0.7' \
  -H 'accept-language: en-US,en;q=0.9' \
  -H 'cache-control: max-age=0' \
  -H 'if-modified-since: Thu, 01 Jun 2023 10:05:02 GMT' \
  -H 'if-none-match: "2153-5fd0e91e9a8fc-gzip"' \
  -H 'sec-ch-ua: "Not.A/Brand";v="8", "Chromium";v="114", "Google Chrome";v="114"' \
  -H 'sec-ch-ua-mobile: ?0' \
  -H 'sec-ch-ua-platform: "Windows"' \
  -H 'sec-fetch-dest: document' \
  -H 'sec-fetch-mode: navigate' \
  -H 'sec-fetch-site: none' \
  -H 'sec-fetch-user: ?1' \
  -H 'upgrade-insecure-requests: 1' \
  -H 'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/114.0.0.0 Safari/537.36' \
  --compressed
```

Figure 12.1: curl command

> **Important note**
>
> Using the **Copy as cURL (bash)** option is almost like copying *request headers* from the browser developer tools. Browsers extract, prettify, and present such information with some formatting. **Postman** also allows importing `curl` commands and presenting them in an *HTML <form>*-based layout (editable text boxes and more), where developers can make their choices and proceed with web requests, as well as carry out testing in parallel.

The `curl` command obtained through the browser, as seen in *Figure 12.1*, can be copied to `https://curlconverter.com/python` and converted into Python code ready for libraries such as `requests` and `http.client`. The following code is the converted/generated Python code for `requests`:

```python
import requests
headers = {
    'authority': 'curl.se',
    'accept': 'text/html,application/xhtml+xml,
            application/xml;q=0.9, image/avif,
            image/webp, image/apng,*/*;q=0.8,
            application/signed-exchange;v=b3;q=0.7',
    'accept-language': 'en-US,en;q=0.9',
    'cache-control': 'max-age=0',
    'if-modified-since': 'Thu, 01 Jun 2023 10:05:02 GMT',
    'if-none-match': '"2153-5fd0e91e9a8fc-gzip"',
    'sec-ch-ua': '"Not.A/Brand";v="8", "Chromium";v="114",
            "Google Chrome"; v="114"',
    'sec-ch-ua-mobile': '?0',
    'sec-ch-ua-platform': '"Windows"',
    'sec-fetch-dest': 'document',
    'sec-fetch-mode': 'navigate',
    'sec-fetch-site': 'none',
    'sec-fetch-user': '?1',
    'upgrade-insecure-requests': '1',
    'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64;
                x64) AppleWebKit/537.36  (KHTML, like Gecko)
                Chrome/114.0.0.0 Safari/537.36',
}
response = requests.get('https://curl.se/',
headers=headers)
```

Using `curl` commands, even using `pycurl`, will help us significantly in learning how to use core technologies related to web-based communication. As technical development-related processes are being routed to AI-driven and ML-based systems, understanding these core concepts will make a difference.

In the next section, we will learn about an add-on feature (middleware feature) that can be used in web requests.

Proxies

A proxy or proxy web server (also known as middleware) is one of the most important and core technologies in network communication, especially web scraping. Plenty of web security-related challenges arise in network communication during scraping, and using a proxy helps us to overcome them.

Using a proxy when scraping usually hides the original machine's IP. Be aware, however, that some providers will not always hide the original IP. A proxy communicates with the target site using some other secure IPs that are available through proxy service providers. A proxy works as middleware and helps to bypass web security on physical machines. One of the most used proxy types in web scraping is rotating **residential proxies**, which change the IP address on every new request.

The internet is open content. Web servers impose a large number of security measures to keep their content safe. Many of you will have encountered some content that was blocked to view or access, which can be overcome using proxies. In some way or another, either on public or on-premises networks, proxies provide great security and filtering capabilities.

When dealing with web scraping for long-form content, such as daily blogs, reviews, stock information, and ratings, there are various recurring scraping tasks for fixed sites, as well as scheduled scraping options, including hourly intervals. However, it is important to consider the possibility of encountering web security measures that could potentially lead to blocking, such as the following:

- IP blacklisting
- Facing CAPTCHAs on every page
- The inability to use filtering options
- Redirecting to pages that are not targeted (such as those hosted in a different country or in a different language from that of the user)
- Redirecting to the first page only
- Redirecting to the main page

These kinds of things can lead to wasted time, financial loss, loss of disk and web space, collecting irrelevant data, reduced quality of data, and much more. To resolve this, developers need to use and implement web-based penetration testing, such as using proxies; selecting rotating IPs for a selected country, a random combination of proxies with **User-Agent**, or randomized or standard *request headers*; or executing `curl` commands in the terminal to verify content.

Listed here are some of the situations in which proxies can be helpful:

- Overcoming territory (border-based) restrictions
- Preventing blacklisting

- Rotating the IP address

- Performance enhancement

- Bypassing CAPTCHAs, malware, spam, and more

- Overcoming blocked access

- Implementing privacy features

With numerous advantages of using proxies, there are two significant disadvantages too, as listed here:

- Extra financial requirements (buying proxies)

- Extra configuration (on top of the existing network and code)

Proxies are often found and prescribed for use with scraping or crawling tasks.

As seen in the following sample code template, the Python `random` library's `shuffle` method is being used to randomize values from `user_agents` and `proxyList`. The following code will not run until and unless we acquire a paid subscription from the website for the valid `api-key`, which results in multiple collections of *IPs with ports* for `http` and `https`:

```
import requests
import random
user_agents = ['Mozilla/5.0 (iPhone; CPU iPhone OS 16_5 like Mac
OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) CriOS/114.0.5735.99
Mobile/15E148 Safari/604.1', 'Mozilla/5.0 (iPod; CPU iPhone OS
16_5 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko)
CriOS/114.0.5735.99 Mobile/15E148 Safari/604.1', 'Mozilla/5.0
(X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/114.0.0.0 Safari/537.36', 'Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0
Safari/537.36',]
# proxyList = dict(requests.get("https://api.demo-web.com?api-
key=XXXX-XX-XXXX-XXX&zone=US&type=multiple&timeloc=12h&channel=chr_
moz").content)
proxyList = {'http':['xxx.xxx.xxx.xxx:xx', 'xxx.xxx.xxx.xxx:xxx','xxx.
xxx.xxx.xxx'],
'https':['xxx.xxx.xxx.xxx:xx','xxx.xxx.xxx.xxx:xxx', 'xxx.xxx.xxx.
xxx'],}
def rand_output(input):
    return random.shuffle(input)
response = requests.get('https://www..…com',
headers={'User-Agent': rand_output(user_agents)},
proxies = rand_output(proxyList['https'])).text
```

`proxyList` contains a list of IPs that have visited the service provider pages. IPs can also be collected using API requests to proxy providers. The `rand_ouput` function receives a Python `list()` object

and returns the `shuffle` list. `requests` supports the `headers` and `proxies` arguments, and more, where desired values can be provided.

Most cloud-based platforms and web hosting organizations provide an option if required to use their in-house proxy services or even allow using third-party proxy settings through some additional configuration. Many proxy providers use a web API and allow users to access proxies using some authentication system (an API key). Users can search `https://www.google.com/search?q=proxy+providers` on Google to find the best proxy solution and use or choose the appropriate solution.

> **Important note**
>
> There are plenty of browser-based extensions that provide **virtual private network** (**VPN**) access, rotating IPs, and more. Python libraries also exist for proxies, user agents, generating security-related tokens, and more. These extensions and libraries might cause system vulnerabilities, or work temporarily or permanently, but it's the responsibility of the users to do the proper research before deciding whether to use them.

In the next section, we will introduce some new tools and applications that are useful in data processing.

Data processing

Data processing, in the context of web scraping, refers to storing, handling, managing, and analyzing the data that is generated from scraping. In previous chapters of the book, we focused on the concept of effective and efficient scraping with code examples.

As the demand for data is growing, technologies are also evolving and adapting to new changes. Currently, as there has been a boom in AI/ML-based systems, there is competition to provide easy and quick solutions to problems without compromising on quality.

In the coming sections, we will introduce some technologies that help with data processing.

PySpark

The Python library for **Apache Spark**, `pyspark` (`https://spark.apache.org/`), is used to process and analyze data, especially of a large volume. **Spark** is a framework that is used to handle big data (data with variety, volume, and velocity) and is more effective than **Hadoop** (`https://hadoop.apache.org/`), a framework for parallel processing, scheduling, and resource management.

Apache Spark is famous for its distributed processing system, data processing, effective clustering of the environment, CLI applications, and parallel processing for large volumes of data. It is based on Hadoop's **MapReduce**, with additional optimization for fast computing, and is also available for various programming languages, such as *C#*, *Java*, and *Python*.

`pyspark` is known for its lazy computation and analyzing big data and is supported by Spark functionalities such as the following:

- In-memory processing

- Using **Structured Query Language** (**SQL**) on DataFrames

- Using the `pandas` API

- The inbuilt `MLlib` ML library

- Computational speed

- Fault tolerance

PySpark also uses the Python `pandas` API, so most of the activities that can be done using `pandas` are applicable using `PySpark`, with the additional added benefits and facilities of Apache Spark. Generally, for large datasets, `PySpark` is very effective. `PySpark`'s features can also be used independently, such as only for data analysis, using ML, distributing datasets into chunks, and merging selected chunks of a dataset.

PySpark, like `pandas` and other Python libraries, can be used for visualization, data cleaning, data formatting, data sampling, and more. Spark DataFrames possess similar attributes to those in `pandas`, and have many advanced functionalities implementing various features of the Spark framework. Please visit `https://spark.apache.org/docs/latest/quick-start.html` and `https://sparkbyexamples.com/` for more details.

In the next section, we will look into a Python library that is rising in popularity due to its processing speed.

polars

The `polars` (`https://www.pola.rs/`) Python library is a data analysis library considered to be one of the fastest DataFrame libraries. Built using the **Rust** programming language, `polars` is designed for faster and more efficient computing. `polars` has been found to be more effective compared to Python's `pandas` while working with large datasets.

Code written in `polars` is a bit longer and not quite as readable as in `pandas`. Listed here are a few benefits of `polars`, which make it favorable over other data analysis libraries:

- **Graphics processing unit** (**GPU**) integration

- Lazy evaluation (executing only when required)

- Parallel processing of operations

- **Online analytical processing** (**OLAP**) query engine implemented in Rust

- **Single instruction, multiple data** (**SIMD**)-based vectorization (processing of multiple datasets with a single instruction) implemented by modern processors

`polars` uses **Apache Arrow**, or **Arrow** (`https://arrow.apache.org/`), arrays, whereas `pandas` uses **NumPy** arrays. Arrow uses a language-independent columnar memory format for data.

Arrow recommends **Apache Parquet** (`https://parquet.apache.org/`), a file format that implements data compression and encoding schemes. The use of Parquet is growing rapidly, as it is fast in comparison to CSV and JSON, and supports storing data in a columnar format. One of the major benefits of Parquet files is in data loading; they allow columns to be accessed without having to load the whole data structure. Arrow is also compatible for use or implementation with `pandas`, `NumPy`, `PySpark`, and more.

> **Important note**
>
> `PyArrow` is a Python library that implements Arrow. Arrow can be integrated with built-in Python objects, such as `NumPy` and `pandas`. Arrow manages data and data types in the same array, and the data in arrays is grouped to create *tables*. `PyArrow` implements data as a *table*. Each array represents data in columns of a *table*. `PyArrow` is also integrated into `pandas` through the `ExtensionArray` interface. It also enables the reading and writing of a *table* to a DataFrame and vice versa.

In the next section, we will talk about the job market and careers that can be pursued with the context and information we have discussed and explored throughout the chapters of the book.

Jobs and careers

In this book so far, we have covered various topics on Python programming and web scraping. We have learned different techniques, particularly focused on data, including searching, acquiring, mining, transforming, analyzing, and visualizing.

The availability of job opportunities that allow individuals to implement the learned skills while also acquiring additional and up-to-date ones serves as a strong motivation to continue to learn and develop their skills. Data-related careers offer attractive prospects globally, with excellent salaries offered.

In this section, we will provide a list of job titles related to web scraping and Python programming for your reference. This is especially relevant for those interested in developing data-related skills (such as data science and AI/ML), considering the current demand in the field.

Here are some job titles sourced from various job sites across the globe:

- Senior Python programmer
- Expert Python programmer
- Web scraper
- Data extractor
- Data researcher

- Historical data researcher

- Geo-data researcher

- Data scientist

- Data science practitioner

- Data analyst

- Data security

- Web security

- Data visualizer

- Data freelancer

- Data and big data engineer

- MIS expert

- Report developer (data-driven)

- Information analyzer

- Information practitioner

- ML engineer

- ML practitioner

- NLP practitioner

- Sentiment analyst

These job titles are just a few examples, and you can also consider exploring freelancing opportunities. It is certain that you will encounter data and its related processing activities in various fields associated with data science and management. Whenever tasks involve activities such as analysis, information generation, information processing, and reporting, the skills we have acquired throughout the chapters of this book will come into play.

Summary

In this chapter, we explored some of the hot technologies on the market and in the field of data science. Technological frameworks and tools are always evolving. Therefore, it is the developer's duty to keep up with the latest updates in technologies.

Web scraping or data extraction is one of the core fields of data science, though data processing and analyzing tasks closely follow. Collecting, gathering, and storing data from target websites using scraping techniques used to be core aspects of web scraping. However, as there have been various breakthroughs in systems that interact with data, providing quality data that can be directly implemented

in the systems, stored in formats required by the customer, applicable for real-time processing, and more are also to be considered.

Data or datasets are made available using APIs, through sites dedicated to providing tools for research, and more. Web scraping comes into play when we require specific data from a target website in the desired format. Data received in this way can be collected and merged with or appended to other data or datasets, to create a dataset.

With the growing use of online applications and frameworks, whether it's a small dataset or a dataset with a large volume, or on-the-fly-collected records or rows resulting from scraping scripts, they need to be accessible and performance-driven while interacting with data-related systems to verify the demand of web scraping using Python programming.

We have reached the end of the book. Web scraping is a core task to be carried out in data science and data-driven systems. It has always been dynamic, progressive, and assistive to the generation of data, information, and knowledge. Topics such as AI, ML, data mining, NLP, and Python programming are always associated in some way or the other with web scraping. Web scraping-related topics are worth exploring from a career and knowledge perspective.

Further reading

- *Data science*:

    - `https://aws.amazon.com/what-is/data-science/`

    - `https://www.ibm.com/topics/data-science`

- *Apache Arrow*: `https://arrow.apache.org/docs/python/`

- *Apache Spark*: `https://spark.apache.org/docs/3.3.1/api/python/index.html`

- *Data science – career*:

    - `https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century`

    - `https://www.coursera.org/articles/data-science-career`

    - `https://www.discoverdatascience.org/career-information/`

    - `https://www.montecarlodata.com/blog-the-future-of-big-data-analytics-and-data-science/`

    - `https://emeritus.org/in/learn/scope-of-data-analytics-in-the-future/`

    - `https://www.knowledgehut.com/blog/data-science/data-scientist-future`

- *Python concurrency*:

  - `https://docs.python.org/3/library/concurrent.futures.html`

  - `https://vegibit.com/how-to-perform-parallel-programming-with-pythons-concurrent-futures-library/`

- *AI and data*:

  - `https://www.mygreatlearning.com/blog/difference-data-science-machine-learning-ai/`

  - `https://www.forbes.com/sites/nishatalagala/2022/11/10/ai-and-data-sciencewhat-is-the-difference/?sh=81569d64b442`

  - `https://hevodata.com/learn/artificial-intelligence-in-data-science/`

- *Big data*: `https://www.oracle.com/big-data/what-is-big-data/`

Index

‹packt›

Other Books You May Enjoy

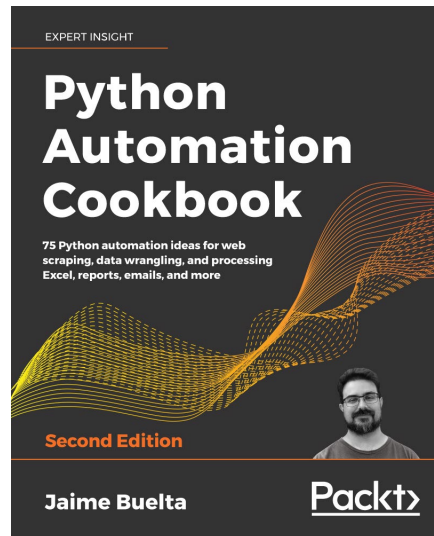If you enjoyed this book, you may be interested in these other books by Packt:

Python Web Scraping Cookbook

Michael Heydt

ISBN: 978-1-78728-521-7

- Use a variety of tools to scrape any website and data, including BeautifulSoup, Scrapy, Selenium and many more

- Master expression languages, such as XPath and CSS, and regular expressions to extract web data

- Deal with scraping traps such as hidden form fields, throttling, pagination, and different status codes

- Build robust scraping pipelines with SQS and RabbitMQ

- Scrape assets like image media and learn what to do when Scraper fails to run

- Explore ETL techniques of building a customized crawler, parser, and convert structured and unstructured data from websites

- Deploy and run your scraper as a service in AWS Elastic Container Service

Python Automation Cookbook – Second Edition

Jaime Buelta

ISBN: 978-1-80020-708-0

- Learn data wrangling with Python and Pandas for your data science and AI projects

- Automate tasks such as text classification, email filtering, and web scraping with Python

- Use Matplotlib to generate a variety of stunning graphs, charts, and maps

- Automate a range of report generation tasks, from sending SMS and email campaigns to creating templates, adding images in Word, and even encrypting PDFs

- Master web scraping and web crawling of popular file formats and directories with tools like Beautiful Soup

- Build cool projects such as a Telegram bot for your marketing campaign, a reader from a news RSS feed, and a machine learning model to classify emails to the correct department based on their content

- Create fire-and-forget automation tasks by writing cron jobs, log files, and regexes with Python scripting

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Hi!

I'm Anish Chapagain, the author of Hands-On Web Scraping with Python Second Edition. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in Web Scraping with Python.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on Hands-On Web Scraping with Python Second Edition here.

Go to the link below or scan the QR code to leave your review:

```
https://packt.link/r/1837636214
```

Your review will help me to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Anish Chapagain

Anish Chapagain

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781837636211

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly