Introduction to Web Scraping

The internet is an absolutely massive source of data — data that we can access using web scraping and Python!

In fact, web scraping is often the *only* way we can access data. There is a lot of information out there that isn't available in convenient CSV exports or easy-to-connect APIs and websites themselves are often valuable sources of data — consider, for example, the kinds of analysis you could do if you could download every post on a web forum.

To access those sorts of on-page datasets, we'll have to use web scraping.

Now let's go on for a formal definition of Web Scraping,

Web Scraping *(also termed Screen Scraping, Web Data Extraction, Web Harvesting etc.) is a technique employed to extract large amounts of data from websites whereby the data is extracted and saved to a local file in your computer or to a database in table (spreadsheet) format.*

Applications of Web Scraping

- ✓ Scraping stock prices into an app API
- ✓ Scraping data from YellowPages to generate leads
- ✓ Scraping data from a store locator to create a list of business locations
- ✓ Scraping product data from sites like Amazon or eBay for competitor analysis
- ✓ Scraping sports stats for betting or fantasy leagues
- ✓ Scraping site data before a website migration
- ✓ Scraping product details for comparison shopping
- ✓ Scraping financial data for market research and insights

Domain Level Applications

REAL TIME ANALYTICS

Real-time analytics is also used at Points of Sale in order to detect any kind of fraud. Real-time analytics depends on processing large quantities of data. Real-time analytics also works in a hassle-free manner if and only if large quantities of data can be processed quite quickly.
This is where web scraping comes in handy. Real-time analytics would not be possible if data could not be accessed, extracted and analysed quickly.

CONTENT MARKETING

When it comes to content marketing, web scraping is used for collating data from different sites such as Twitter, Tech Crunch etc. This data, then, can be used for creating engaging content. Engaging content, as you know, is the key to business growth and web traffic.

SEO MONITERING

Well, search engines tell us a lot about how the world of business moves. How content moves up and down in rankings is also a key to how one can thrive in this Internet age.

One can study the way content works on the Internet and derive insights and strategies.

However, manually it cannot be done. Therefore, there is a growing use of web scraping tools to scrape the data regarding what goes on behind the scenes in search engines. Web scraping can power your understanding of content in terms of SEO and provide actionable intelligence with respect to SEO.

Introduction to Beautiful Soup

Beautiful Soup (BS4) is a parsing library that can use different parsers. A parser is simply a program that can extract data from HTML and XML documents.

Beautiful Soup's default parser comes from Python's standard library. It's flexible and forgiving, but a little slow. The good news is that you can swap out its parser with a faster one if you need the speed.

One advantage of BS4 is its ability to automatically detect encodings. This allows it to gracefully handle HTML documents with special characters.

In addition, BS4 can help you navigate a parsed document and find what you need. This makes it quick and painless to build common applications.

There are 4 types of parser libraries,

- ✓ Python’s html.parser
- ✓ lxml’s html parser
- ✓ lxml’s XML parser
- ✓ html5lib

Functions of Beautiful Soup

A regular expression

If you pass in a regular expression object, Beautiful Soup will filter against that regular expression using its **find_all**() method. This code finds all the tags whose names start with the letter “b”; in this case, the <body> tag and the <b> tag:

```
import re
for tag in soup.find_all(re.compile("^b")):
    print(tag.name)
# body
# b
```

This code finds all the tags whose names contain the letter 't':

```
for tag in soup.find_all(re.compile("t")):
    print(tag.name)
# html
# title
```

If you set a tag's .string attribute to a new string, the tag's contents are replaced with that string:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'

soup = BeautifulSoup(markup)

tag = soup.a
tag.string = "New link text."
tag
# <a href="http://example.com/">New link text.</a>
```

extend()

Starting in Beautiful Soup 4.7.0, Tag also supports a method called .extend(), which works just like calling .extend() on a Python list:

```
soup = BeautifulSoup("<a>Soup</a>")
soup.a.extend(["'s", " ", "on"])

soup
# <html><head></head><body><a>Soup's on</a></body></html>
soup.a.contents
# [u'Soup', u''s', u' ', u'on']
```

NavigableString() and .new_tag()

If you need to add a string to a document, no problem–you can pass a Python string in to append(), or you can call the NavigableString constructor:

```
soup = BeautifulSoup("<b></b>")

tag = soup.b
tag.append("Hello")
new_string = NavigableString(" there")
tag.append(new_string)
tag
# <b>Hello there.</b>
tag.contents
# [u'Hello', u' there']
```

insert()

Tag.insert() is just like Tag.append(), except the new element doesn't necessarily go at the end of its parent's .contents. It'll be inserted at whatever numeric position you say. It works just like .insert() on a Python list:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.insert(1, "but did not endorse ")
tag
# <a href="http://example.com/">I linked to but did not endorse
<i>example.com</i></a>
tag.contents
# [u'I linked to ', u'but did not endorse', <i>example.com</i>]
```

insert_before() and insert_after()

```
soup = BeautifulSoup("<b>stop</b>")
tag = soup.new_tag("i")
tag.string = "Don't"
soup.b.string.insert_before(tag)
soup.b
# <b><i>Don't</i>stop</b>
```

Advantages of Beautiful Soup

1.It is easy to learn and master. for example, if we want to extract all the links from the webpage. It can be simply done as follows —

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

for link in soup.find_all('a'): # It helps to find all anchor tag's
   print(link.get('href'))
```

In the above code, we are using the ***html.parser*** to parse the content of the ***html_doc.*** this is one of the strongest reason for developers to use Beautiful soup as a web scraping tool.

2. It has good comprehensive documentation which helps us to learn the things quickly.

3. It has good community support to figure out the issues that arise while we are working with this library.

BeautifulSoup Alternatives & Comparisons

Scrapy

It is the most popular web scraping framework in Python. An open source and collaborative framework for extracting the data you need from websites. In a fast, simple, yet extensible way.

Selenium

Selenium automates browsers. That's it! What you do with that power is entirely up to you. Primarily, it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should!) also be automated as well.

import.io

import.io is a free web-based platform that puts the power of the machine-readable web in your hands. Using our tools, you can create an API or crawl an entire website in a fraction of the time of traditional methods, no coding required.

ParseHub

Web Scraping and Data Extraction ParseHub is a free and powerful web scraping tool. With our advanced web scraper, extracting data is as easy as clicking on the data you need. ParseHub lets you turn any website into a spreadsheet or API

Portia

Portia is an open-source tool that lets you get data from websites. It facilitates and automates the process of data extraction. This visual web scraper works straight from your browser, so you don't need to download or install anything.

Each of the three tools presented has its advantages and disadvantages. We have clearly summarized these for you:

	Scrapy	Selenium	BeautifulSoup
Easy to learn	★★★	★	★★★
Readout dynamic content	★★	★★★	★
Realize complex applications	★★★	★	★★
Robustness against HTML errors	★★	★	★★★
Optimized for scraping performance	★★★	★	★
Pronounced ecosystem	★★★	★	★★

Introduction to Urllib

urllib.request is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the *urlopen* function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.
urllib.request supports fetching URLs for many “URL schemes” (identified by the string before the ":" in URL - for example "ftp" is the URL scheme of "ftp://python.org/") using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.
The urllib module in Python 3 allows you access websites via your program. This opens up as many doors for your programs as the internet opens up for you. urllib in Python 3 is slightly different than urllib2 in Python 2, but they are mostly the same. Through urllib, you can access websites, download data, parse data, modify your headers, and do any GET and POST requests you might need to do.

Functions of Urllib

urllib.request

Using urllib.request, with urlopen, allows you to open the specified URL. This is shown in the code snippet below:

```
from urllib.request import urlopen
myURL = urlopen("http://www.google.com/")
print(myURL.read())
```

Once the URL has been opened, the read() function is used to get the entire HTML code for the webpage.

urllib.parse

The code snippet below shows the usage of urllib.parse:

```
from urllib.parse import urlparse
parsedUrl = urlparse('https://www.educative.io/track/python-for-programmers')
print(parsedUrl)
```

The URL is split into its components such as the protocol ***scheme*** used, the network location ***netloc*** and the ***path*** to the webpage.

urllib.error

This module is used to catch exceptions encountered from url.request. These exceptions, or errors, are classified as follows:

1. **URL Error**, which is raised when the URL is incorrect, or when there is an internet connectivity issue.
2. **HTTP Error**, which is raised because of HTTP errors such as **404 (request not found)** and **403 (request forbidden)**. The following code snippet shows the usage of urlib.error:

```
from urllib.request import urlopen, HTTPError, URLError
try:
   myURL = urlopen("http://ww.educative.xyz/")
except HTTPError as e:
   print('HTTP Error code: ', e.code)
except URLError as e:
   print('URL Error: ', e.reason)
else:
   print('No Error.')
```

A request to open http://ww.educative.xyz/ is caught by the URLError exception; the URL is invalid. Experiment with the exceptions by opening different URL's.

Advantages of Urllib

- **requests** library is easy to use and fetch information
- It is extensively used for scraping data from website
- It is also used for web API request purpose
- Using request, we can GET, POST, PUT, and delete the data for the given URL
- It has an authentication module support
- It handles cookies and sessions very firmly.

Comparing versions of Urllib

urllib and **urllib2** are both Python modules that do URL request related stuff but offer different functionalities.

1) urllib2 can accept a Request object to set the headers for a URL request, urllib accepts only a URL.

2) urllib provides the **urlencode** method which is used for the generation of GET query strings, urllib2 doesn't have such a function. This is one of the reasons why urllib is often used along with urllib2.

Requests - Requests' is a simple, easy-to-use HTTP library written in Python.

1) Python Requests encodes the parameters automatically so you just pass them as simple arguments, unlike in the case of urllib, where you need to use the method **urllib.encode()** to encode the parameters before passing them.

2) It automatically decoded the response into Unicode.
3) Requests also has far more convenient error handling.If your authentication failed, urllib2 would raise a urllib2.URLError, while Requests would return a normal response object, as expected. All you have to see if the request was successful by boolean **response.ok**