EDUNET FOUNDATION - Self-Practice Exercise Notebook

LAB 12 - Implementing Web Scrapping Concepts in Python

Basics of Web Scrapping

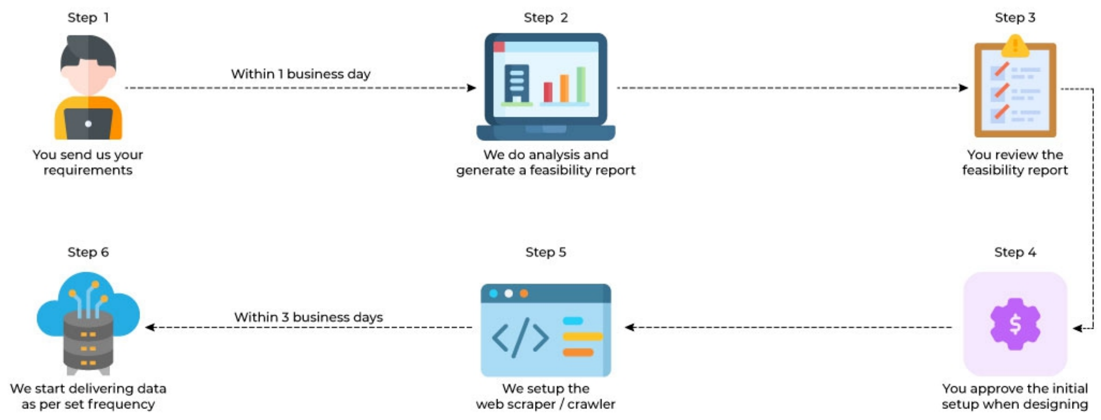
Web scraping is the process of collecting and parsing raw data from the Web, and the Python community has come up with some pretty powerful web scraping tools.

The Internet hosts perhaps the greatest source of information on the planet. Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from collecting and analyzing data from websites.

In this tutorial, you'll learn how to:

Parse website data using string methods and regular expressions Parse website data using an HTML parser Interact with forms and other website components

Introduction to Web Scraping Consider the following scenario: you need to pull an enormous volume of data from websites as rapidly as workable. How would you do it if you didn't go to each website and manually collect the data? Well, the answer is "web scraping". Scraping the web makes this process a lot easier and faster.


Step 1
You send us your requirements
Within 1 business day
Step 2
We do analysis and generate a feasibility report
Step 3
You review the feasibility report
Step 4
You approve the initial setup when designing
Step 5
We setup the web scraper / crawler
Within 3 business days
Step 6
We start delivering data as per set frequency

Overview:

What is a web scraping and how does it work with Python? Interestingly, Web scraping is a word that refers to the practice of extracting and processing vast amounts of data from the internet using a computer or algorithm. Scraping data from the web is a useful skill to have, whether you're a data scientist, engineer, or anyone who analyses enormous volumes of data. Let's get started.!

Table of content:

- Introduction
- Implementing web scraping using python

- Why is python used for web scraping?
- Step-by-Step process to Scrape Data From A Website

Become a Full Stack Data Scientist Transform into an expert and significantly impact the world of data science. How is Web Scraping Done?

Scrapy beautiful soup selenium Scraping E-commerce Website

Web scraping using beautiful soup and selenium

Web scraping/crawling using scrapy

Creating a Scrapy Spider class

Scraping many pages

About Myself

Conclusion

Introduction to Web Scraping Consider the following scenario: you need to pull an enormous volume of data from websites as rapidly as workable. How would you do it if you didn’t go to each website and manually collect the data? Well, the answer is “web scraping”. Scraping the web makes this process a lot easier and faster.

How Does Web Scraping works

Image 1

Web scraping is a technique for extracting vast amounts of data from websites. But why is it necessary to gain such vast amounts of data from websites? Look at the following web scraping applications to learn more:

Price comparison and competition monitoring – Web scraping tools can monitor this company’s catering products data at all times. Train and Test data for Machine Learning Projects – Web scraping aids in the collection of data for the testing and training of Machine Learning models. Academic Research. Gather hotel/restaurant reviews and ratings from sites such as TripAdvisor. Use websites like Booking.com and Hotels.com to scrape hotel room pricing and information. Scrape tweets from Twitter that are associated with a specific account or hashtag. We can scrape business contact information such as phone numbers and email addresses from yellow pages websites or Google Maps business listings to generate leads for marketing.

Implementing Web Scraping using Python Web scraping is a way to extract vast volumes of data from websites that are automated. Sometimes, the information on the web pages is not structured. In that case, Web scraping aids in the collection of unstructured data and its subsequent storage in a structured format. We can do scraping websites in a variety of methods, including using internet services, APIs, or building your programs in this article. We’ll look at how to use Python to implement web scraping.

Why is python used for web scraping?

1. Python includes many libraries, such as Numpy, Matplotlib, Pandas, and others, that provide methods and functions for a variety of uses. As a result, it’s suitable for web crawling and additional data manipulation.
2. Python is an easy language to program in. There are no semi-colons “;” or curly-braces “{}” required anywhere. So it is easier to use and less noisy.

3. Dynamically typed: You don't have to define data types for variables in Python; you can just use them wherever they're needed. This saves you time and speeds up your work.
4. Small code, long process: Web scraping is a technique for saving time. But what good is it if you waste more time writing code? You don't have to, though. We can write small codes in Python to accomplish large tasks. As a result, even while writing the code, you save time.
5. Python syntax is simple to learn because reading Python code is quite understandable compared to reading a statement in English. Python's indentation helps the user distinguish between distinct scopes/blocks in the code, making it expressive and easy to

How is Web Scraping using Python done?
We can do web scraping with Python using three different frameworks:

- Scrapy
- Beautiful Soup
- Selenium

In [1]:
```
!pip install beautifulsoup4
```

```
Requirement already satisfied: beautifulsoup4 in c:\users\rampa\anaconda3\li
b\site-packages (4.11.1)
Requirement already satisfied: soupsieve>1.2 in c:\users\rampa\anaconda3\lib
\site-packages (from beautifulsoup4) (2.3.2.post1)
```

In [5]:
```
import requests

# Making a GET request
r = requests.get('https://www.geeksforgeeks.org/python-programming-language/')

# check status code for response received
# success code - 200
print(r)

# print content of request
print(r.content)
```

```
-uploads/gfg_favicon.png type=image/x-icon><link rel=preconnect href=http
s://fonts.googleapis.com><link rel=preconnect href=https://fonts.gstatic.c
om crossorigin><meta name=theme-color content="#308D46"><meta name=image p
roperty="og:image" content="https://media.geeksforgeeks.org/wp-content/cdn
-uploads/gfg_200x200-min.png"><meta property="og:image:type" content="imag
e/png"><meta property="og:image:width" content="200"><meta property="og:im
age:height" content="200"><meta name=facebook-domain-verification content
="xo7t4ve2wn3ywfkjdvwbrk01pvdond"><script defer src=https://apis.google.co
m/js/platform.js></script><script async src=//cdnjs.cloudflare.com/ajax/li
bs/require.js/2.1.14/require.min.js></script><title>Python Tutorial | Lear
n Python Programming</title><link rel=profile href=http://gmpg.org/xfn/11>
<link rel=pingback href><script type=application/ld+json>{"@context":"htt
p://schema.org","@type":"Organization","name":"GeeksforGeeks","url":"http
s://www.geeksforgeeks.org/","logo":"https://media.geeksforgeeks.org/wp-con
tent/cdn-uploads/20200817185016/gfg_complete_logo_2x-min.png","descriptio
n":"A computer science portal for geeks. It contains well written, well th
ought and well explained computer science and programming articles, quizze
s and practice/competitive programming/company interview Questions.","foun
der":[{"@type":"Person","name":"Sandeep Jain","url":"https://in.linkedin.c
om/in/sandeep-jain-b3940815"}],"sameAs":["https://www.facebook.com/geeksfo
```

In [6]:
```
from urllib.request import urlopen
url = "http://olympus.realpython.org/profiles/aphrodite"
page = urlopen(url)
html_bytes = page.read()
html = html_bytes.decode("utf-8")
print(html)
```

```
<html>
<head>
<title>Profile: Aphrodite</title>
</head>
<body bgcolor="yellow">
<center>
<br><br>
<img src="/static/aphrodite.gif" />
<h2>Name: Aphrodite</h2>
<br><br>
Favorite animal: Dove
<br><br>
Favorite color: Red
<br><br>
Hometown: Mount Olympus
</center>
</body>
</html>
```

Extract Text From HTML With String Methods One way to extract information from a web page's HTML is to use string methods. For instance, you can use .find() to search through the text of the HTML for the tags and extract the title of the web page.</p> <p>To start, you'll extract the title of the web page that you requested in the previous example. If you know the index of the first character of the title and the index of the first character of the closing tag, then you can use a string slice to extract the title.

Because .find() returns the index of the first occurrence of a substring, you can get the index of the opening tag by passing the string "<title>" to .find():</p>

In [7]:
```
title_index = html.find("<title>")
title_index
```

Out[7]:
```
14
```

You don't want the index of the tag, though. You want the index of the title itself. To get the index of the first letter in the title, you can add the length of the string "<title>" to title_index: </p>

In [8]:
```
start_index = title_index + len("<title>")
start_index
```

Out[8]:
```
21
```

Now get the index of the closing tag by passing the string "" to .find():

```
In [9]: end_index = html.find("</title>")
        end_index
```

```
Out[9]: 39
```

Finally, you can extract the title by slicing the html string:

```
In [10]: title = html[start_index:end_index]
         title
```

```
Out[10]: 'Profile: Aphrodite'
```

Real-world HTML can be much more complicated and far less predictable than the HTML on the Aphrodite profile page. Here’s another profile page with some messier HTML that you can scrape:

```
In [11]: url = "http://olympus.realpython.org/profiles/poseidon"
```

```
In [12]: url = "http://olympus.realpython.org/profiles/poseidon"
         page = urlopen(url)
         html = page.read().decode("utf-8")
         start_index = html.find("<title>") + len("<title>")
         end_index = html.find("</title>")
         title = html[start_index:end_index]
         title
```

```
Out[12]: '\n<head>\n<title >Profile: Poseidon'
```

Get to Know Regular Expressions Regular expressions—or regexes for short—are patterns that you can use to search for text within a string. Python supports regular expressions through the standard library’s re module.

To work with regular expressions, the first thing that you need to do is import the re module:

```
In [10]: import re
```

Regular expressions use special characters called metacharacters to denote different patterns. For instance, the asterisk character (*) stands for zero or more instances of whatever comes just before the asterisk.

In the following example, you use .findall() to find any text within a string that matches a given regular expression:

```
In [11]: re.findall("ab*c", "ac")
         ['ac']
```

```
Out[11]: ['ac']
```

The first argument of re.findall() is the regular expression that you want to match, and the second argument is the string to test. In the above example, you search for the pattern "ab*c" in the string "ac".

The regular expression "ab*c" matches any part of the string that begins with "a", ends with "c", and has zero or more instances of "b" between the two. re.findall() returns a list of all matches. The string "ac" matches this pattern, so it's returned in the list.

Here's the same pattern applied to different strings:

```
In [12]: re.findall("ab*c", "abcd")

Out[12]: ['abc']

In [13]: re.findall("ab*c", "acc")

Out[13]: ['ac']

In [14]: re.findall("ab*c", "abcac")

Out[14]: ['abc', 'ac']

In [15]: re.findall("ab*c", "abdc")

Out[15]: []
```

Often, you use re.search() to search for a particular pattern inside a string. This function is somewhat more complicated than re.findall() because it returns an object called MatchObject that stores different groups of data. This is because there might be matches inside other matches, and re.search() returns every possible result.

The details of MatchObject are irrelevant here. For now, just know that calling .group() on MatchObject will return the first and most inclusive result, which in most cases is just what you want:

```
In [16]: match_results = re.search("ab*c", "ABC", re.IGNORECASE)
         match_results.group()

Out[16]: 'ABC'
```

There's one more function in the re module that's useful for parsing out text. re.sub(), which is short for substitute, allows you to replace the text in a string that matches a regular expression with new text. It behaves sort of like the .replace() string method.

The arguments passed to re.sub() are the regular expression, followed by the replacement text, followed by the string. Here's an example:

In [17]:
```
string = "Everything is <replaced> if it's in <tags>."
string = re.sub("<.*>", "ELEPHANTS", string)
string
```

Out[17]:
```
'Everything is ELEPHANTS.'
```

Perhaps that wasn’t quite what you expected to happen.

re.sub() uses the regular expression "<.*>" to find and replace everything between the first < and the last >, which spans from the beginning of to the end of . This is because Python’s regular expressions are greedy, meaning they try to find the longest possible match when characters like * are used.

Alternatively, you can use the non-greedy matching pattern *?, which works the same way as * except that it matches the shortest possible string of text:

In [18]:
```
string = "Everything is <replaced> if it's in <tags>."
string = re.sub("<.*?>", "ELEPHANTS", string)
string
```

Out[18]:
```
"Everything is ELEPHANTS if it's in ELEPHANTS."
```

This time, re.sub() finds two matches, and , and substitutes the string "ELEPHANTS" for both matches.

Extract Text From HTML With Regular Expressions

Equipped with all this knowledge, now try to parse out the title from another profile page, which includes this rather carelessly written line of HTML:

The .find() method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code quickly and efficiently:

In [22]:
```

import re
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")

pattern = "<title.*?>.*?</title.*?>"
match_results = re.search(pattern, html, re.IGNORECASE)
title = match_results.group()
title = re.sub("<.*?>", "", title) # Remove HTML tags

print(title)
```

```
Profile: Dionysus
```

Install Beautiful Soup

In [23]:
```
!pip install beautifulsoup4
```

```
Requirement already satisfied: beautifulsoup4 in c:\users\prs\appdata\local
\programs\python\python311\lib\site-packages (4.12.2)
Requirement already satisfied: soupsieve>1.2 in c:\users\prs\appdata\local\p
rograms\python\python311\lib\site-packages (from beautifulsoup4) (2.4.1)
```

Create a BeautifulSoup Object

Type the following program into a new editor window:

In [24]:
```
# beauty_soup.py

from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")
soup = BeautifulSoup(html, "html.parser")
```

This program does three things:

- Opens the URL http://olympus.realpython.org/profiles/dionysus (http://olympus.realpython.org/profiles/dionysus) by using urlopen() from the urllib.request module
- Reads the HTML from the page as a string and assigns it to the html variable
- Creates a BeautifulSoup object and assigns it to the soup variable

Use a BeautifulSoup Object

Save and run the above program. When it's finished running, you can use the soup variable in the interactive window to parse the content of html in various ways.

For example, BeautifulSoup objects have a .get_text() method that you can use to extract all the text from the document and automatically remove any HTML tags.

Type the following code into IDLE's interactive window or at the end of the code in your editor:

In [25]:
```
print(soup.get_text())
```

```


Profile: Dionysus





Name: Dionysus

Hometown: Mount Olympus

Favorite animal: Leopard

Favorite Color: Wine

```

There are a lot of blank lines in this output. These are the result of newline characters in the HTML document's text. You can remove them with the .replace() string method if you need to.

Often, you need to get only specific text from an HTML document. Using Beautiful Soup first to extract the text and then using the .find() string method is sometimes easier than working with regular expressions.

However, other times the HTML tags themselves are the elements that point out the data you want to retrieve. For instance, perhaps you want to retrieve the URLs for all the images on the page. These links are contained in the src attribute of
HTML tags.

In this case, you can use find_all() to return a list of all instances of that particular tag:

In [26]:
```
soup.find_all("img")
```

Out[26]:
```
[<img src="/static/dionysus.jpg"/>, <img src="/static/grapes.png"/>]
```

This returns a list of all
tags in the HTML document. The objects in the list look like they might be strings representing the tags, but they're actually instances of the Tag object provided by Beautiful Soup. Tag objects provide a simple interface for working with the information they contain.

You can explore this a little by first unpacking the Tag objects from the list:

In [27]:
```
image1, image2 = soup.find_all("img")
```

Each Tag object has a .name property that returns a string containing the HTML tag type:

In [28]:
```
image1.name
```

Out[28]:
```
'img'
```

You can access the HTML attributes of the Tag object by putting their names between square brackets, just as if the attributes were keys in a dictionary.

To get the source of the images in the Dionysus profile page, you access the src attribute using the dictionary notation mentioned above:

In [29]:
```
image1["src"]
```

Out[29]:
```
'/static/dionysus.jpg'
```

In [30]:
```
image2["src"]
```

Out[30]:
```
'/static/grapes.png'
```

Certain tags in HTML documents can be accessed by properties of the Tag object. For example, to get the tag in a document, you can use the .title property:</p>

In [31]:
```
soup.title
```

Out[31]:
```
<title>Profile: Dionysus</title>
```

You can also retrieve just the string between the title tags with the .string property of the Tag object:

In [32]:
```
soup.title.string
```

Out[32]:
```
'Profile: Dionysus'
```

In [ ]:
```

```