BeautifulSoup

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

In this tutorial, we will show you, how to perform web scraping in Python using Beautiful Soup 4 for getting data out of HTML, XML and other markup languages. In this we will try to scrap webpage from various different websites (including IMDB). We will cover beautiful soup 4, python basic tools for efficiently and clearly navigating, searching and parsing HTML web page. We have tried to cover almost all the functionalities of Beautiful Soup 4 in this tutorial. You can combine multiple functionalities introduced in this tutorial into one bigger program to capture multiple meaningful data from the website into some other sub-program as input.

Audience

This tutorial is basically designed to guide you in scarping a web page. Basic requirement of all this is to get meaningful data out of huge unorganized set of data. The target audience of this tutorial can be anyone of:

- Anyone who wants to know – how to scrap webpage in python using BeautifulSoup 4.
- Any data science developer/enthusiasts or anyone, how wants to use this scraped (meaningful) data to different python data science libraries to make better decision.

Prerequisites

Though there is NO mandatory requirement to have for this tutorial. However, if you have any or all (supercool) prior knowledge on any below mentioned technologies that will be an added advantage:

- Knowledge of any web related technologies (HTML/CSS/Document object Model etc.).
- Python Language (as it is the python package).
- Developers who have any prior knowledge of scraping in any language.
- Basic understanding of HTML tree structure.

Copyright & Disclaimer

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

1. Beautiful Soup — Overview

In today's world, we have tons of unstructured data/information (mostly web data) available freely. Sometimes the freely available data is easy to read and sometimes not. No matter how your data is available, web scraping is very useful tool to transform unstructured data into structured data that is easier to read & analyze. In other words, one way to collect, organize and analyze this enormous amount of data is through web scraping. So let us first understand what is web-scraping.

What is web-scraping?

Scraping is simply a process of extracting (from various means), copying and screening of data.

When we do scraping or extracting data or feeds from the web (like from web-pages or websites), it is termed as web-scraping.

So, web scraping which is also known as web data extraction or web harvesting is the extraction of data from web. In short, web scraping provides a way to the developers to collect and analyze data from the internet.

Why Web-scraping?

Web-scraping provides one of the great tools to automate most of the things a human does while browsing. Web-scraping is used in an enterprise in a variety of ways:

Data for Research

Smart analyst (like researcher or journalist) uses web scrapper instead of manually collecting and cleaning data from the websites.

Products prices & popularity comparison

Currently there are couple of services which use web scrappers to collect data from numerous online sites and use it to compare products popularity and prices.

SEO Monitoring

There are numerous SEO tools such as Ahrefs, Seobility, SEMrush, etc., which are used for competitive analysis and for pulling data from your client's websites.

Search engines

There are some big IT companies whose business solely depends on web scraping.

Sales and Marketing

The data gathered through web scraping can be used by marketers to analyze different niches and competitors or by the sales specialist for selling content marketing or social media promotion services.

Why Python for Web Scraping?

Python is one of the most popular languages for web scraping as it can handle most of the web crawling related tasks very easily.

Below are some of the points on why to choose python for web scraping:

Ease of Use

As most of the developers agree that python is very easy to code. We don't have to use any curly braces "{ }" or semi-colons ";" anywhere, which makes it more readable and easy-to-use while developing web scrapers.

Huge Library Support

Python provides huge set of libraries for different requirements, so it is appropriate for web scraping as well as for data visualization, machine learning, etc.

Easily Explicable Syntax

Python is a very readable programming language as python syntax are easy to understand. Python is very expressive and code indentation helps the users to differentiate different blocks or scoopes in the code.

Dynamically-typed language

Python is a dynamically-typed language, which means the data assigned to a variable tells, what type of variable it is. It saves lot of time and makes work faster.

Huge Community

Python community is huge which helps you wherever you stuck while writing code.

Introduction to Beautiful Soup

The Beautiful Soup is a python library which is named after a Lewis Carroll poem of the same name in "Alice's Adventures in the Wonderland". Beautiful Soup is a python package and as the name suggests, parses the unwanted data and helps to organize and format the messy web data by fixing bad HTML and present to us in an easily-traversible XML structures.

In short, Beautiful Soup is a python package which allows us to pull data out of HTML and XML documents.

2. Beautiful Soup — Installation

As BeautifulSoup is not a standard python library, we need to install it first. We are going to install the BeautifulSoup 4 library (also known as BS4), which is the latest one.

To isolate our working environment so as not to disturb the existing setup, let us first create a virtual environment.

Creating a virtual environment (optional)

A virtual environment allows us to create an isolated working copy of python for a specific project without affecting the outside setup.

Best way to install any python package machine is using pip, however, if pip is not installed already (you can check it using – “pip –version” in your command or shell prompt), you can install by giving below command:

Linux environment

```
$sudo apt-get install python-pip
```

Windows environment

To install pip in windows, do the following:

- Download the get-pip.py from https://bootstrap.pypa.io/get-pip.py or from the github to your computer.
- Open the command prompt and navigate to the folder containing get-pip.py file.
- Run the following command:

```
>python get-pip.py
```

That’s it, pip is now installed in your windows machine.

You can verify your pip installed by running below command:

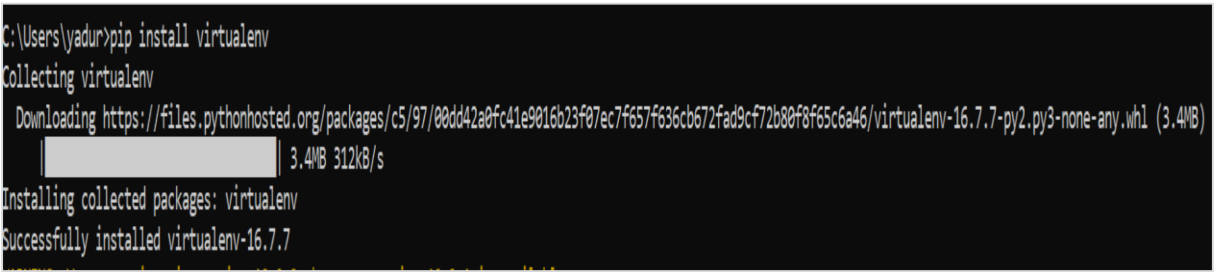
```
>pip --version

pip 19.2.3 from c:\users\yadur\appdata\local\programs\python\python37\lib\site-packages\pip (python 3.7)
```

Installing virtual environment

Run the below command in your command prompt:

```
>pip install virtualenv
```

After running, you will see the below screenshot:

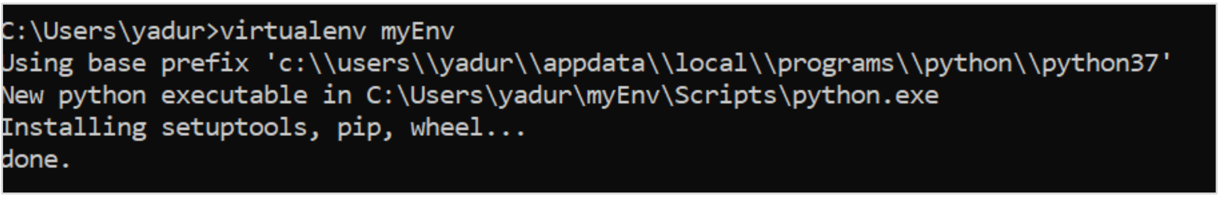


```
C:\Users\yadur>pip install virtualenv
Collecting virtualenv
  Downloading https://files.pythonhosted.org/packages/c5/97/00dd42a0fc41e9016b23f07ec7f657f636cb672fad9cf72b80f8f65c6a46/virtualenv-16.7.7-py2.py3-none-any.whl (3.4MB)
     |████████████████████████████████| 3.4MB 312kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-16.7.7
```

Below command will create a virtual environment ("myEnv") in your current directory:

```
>virtualenv myEnv
```

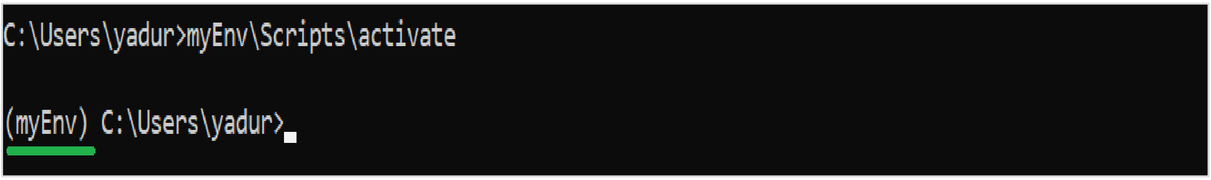
Screenshot



```
C:\Users\yadur>virtualenv myEnv
Using base prefix 'c:\\users\\yadur\\appdata\\local\\programs\\python\\python37'
New python executable in C:\Users\yadur\myEnv\Scripts\python.exe
Installing setuptools, pip, wheel...
done.
```

To activate your virtual environment, run the following command:

```
>myEnv\Scripts\activate
```



```
C:\Users\yadur>myEnv\Scripts\activate

(myEnv) C:\Users\yadur>
```

In the above screenshot, you can see we have "myEnv" as prefix which tells us that we are under virtual environment "myEnv".

To come out of virtual environment, run deactivate.

```
(myEnv) C:\Users\yadur>deactivate
C:\Users\yadur>
```

As our virtual environment is ready, now let us install beautifulsoup.

Installing BeautifulSoup

As BeautifulSoup is not a standard library, we need to install it. We are going to use the BeautifulSoup 4 package (known as bs4).

Linux Machine

To install bs4 on Debian or Ubuntu linux using system package manager, run the below command:

```
$sudo apt-get install python-bs4 (for python 2.x)
```

```
$sudo apt-get install python3-bs4 (for python 3.x)
```

You can install bs4 using easy_install or pip (in case you find problem in installing using system packager).

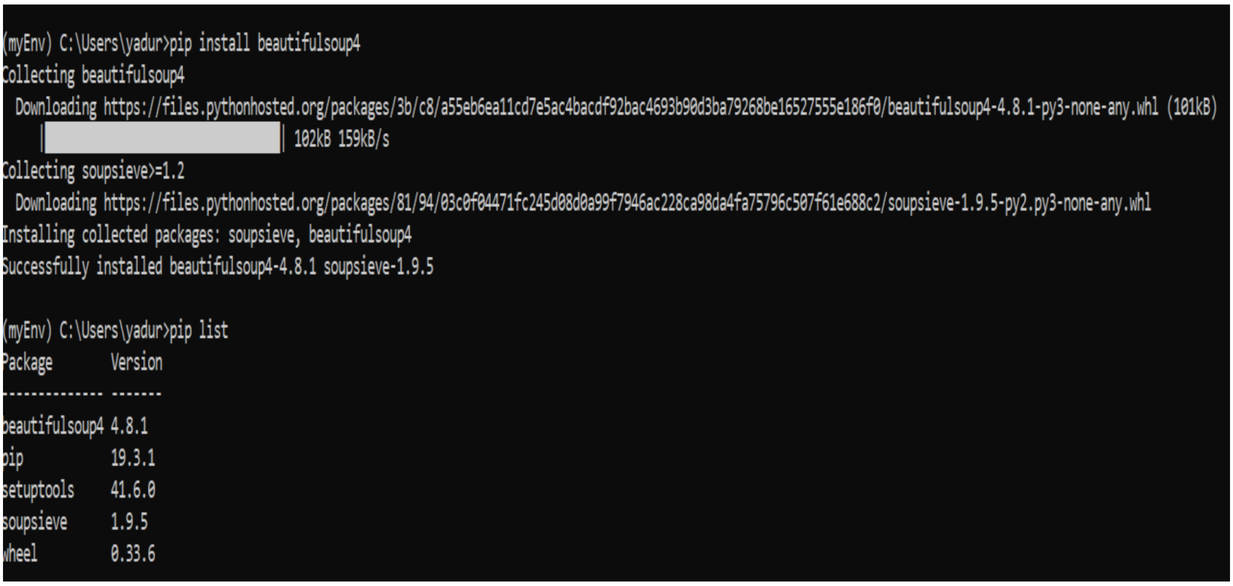
```
$easy_install beautifulsoup4


$pip install beautifulsoup4
```

(You may need to use easy_install3 or pip3 respectively if you're using python3)

Windows Machine

To install beautifulsoup4 in windows is very simple, especially if you have pip already installed.

```
>pip install beautifulsoup4
```



```
(myEnv) C:\Users\yadur>pip install beautifulsoup4
Collecting beautifulsoup4
  Downloading https://files.pythonhosted.org/packages/3b/c8/a55eb6ea11cd7e5ac4bacdf92bac4693b90d3ba79268be16527555e186f0/beautifulsoup4-4.8.1-py3-none-any.whl (101kB)
    |████████████████████████████████| 102kB 159kB/s
Collecting soupsieve>=1.2
  Downloading https://files.pythonhosted.org/packages/81/94/03c0f04471fc245d08d0a99f7946ac228ca98da4fa75796c507f61e688c2/soupsieve-1.9.5-py2.py3-none-any.whl
Installing collected packages: soupsieve, beautifulsoup4
Successfully installed beautifulsoup4-4.8.1 soupsieve-1.9.5

(myEnv) C:\Users\yadur>pip list
Package        Version
-------------- -------
beautifulsoup4 4.8.1
pip            19.3.1
setuptools     41.6.0
soupsieve      1.9.5
wheel          0.33.6
```

So now beautifulsoup4 is installed in our machine. Let us talk about some problems encountered after installation.

Problems after installation

On windows machine you might encounter, wrong version being installed error mainly through:

- error: ***ImportError "No module named HTMLParser"***, then you must be running python 2 version of the code under Python 3.

- error: ***ImportError "No module named html.parser"*** error, then you must be running Python 3 version of the code under Python 2.

Best way to get out of above two situations is to re-install the BeautifulSoup again, completely removing existing installation.

If you get the ***SyntaxError "Invalid syntax"*** on the line ROOT_TAG_NAME = u'[document]', then you need to convert the python 2 code to python 3, just by either installing the package:

```
$ python3 setup.py install
```

or by manually running python's 2 to 3 conversion script on the bs4 directory:

```
$ 2to3-3.2 -w bs4
```

Installing a Parser

By default, Beautiful Soup supports the HTML parser included in Python's standard library, however it also supports many external third party python parsers like lxml parser or html5lib parser.

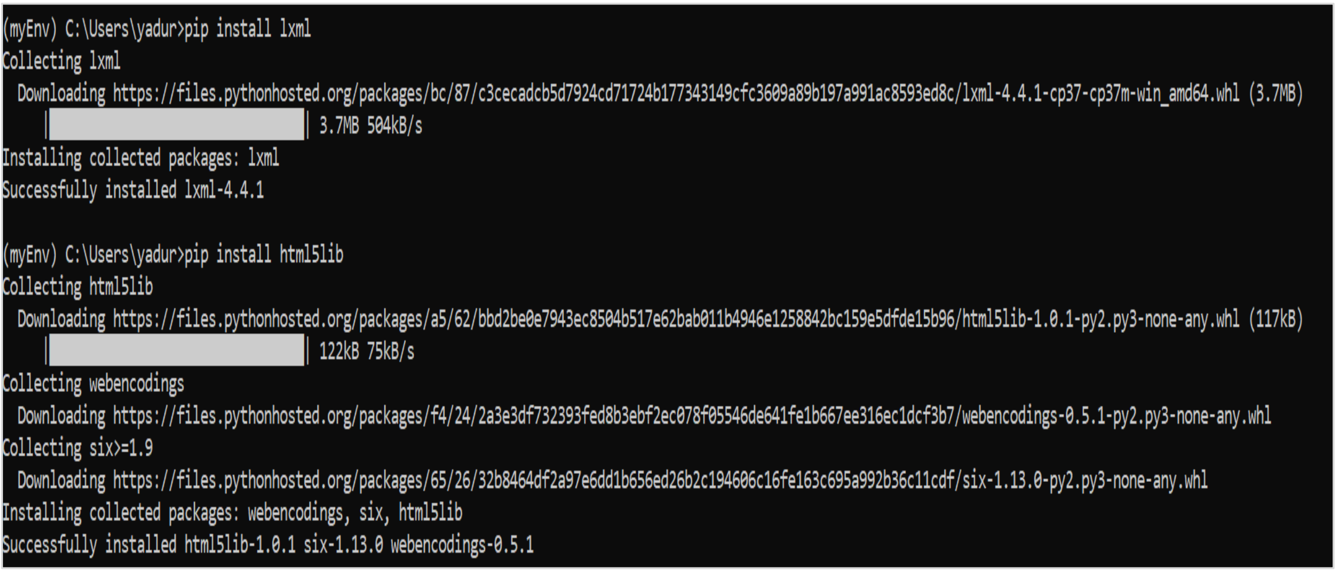
To install lxml or html5lib parser, use the command:

Linux Machine

```
$apt-get install python-lxml


$apt-get insall python-html5lib
```

Windows Machine

```
$pip install lxml


$pip install html5lib
```



```
(myEnv) C:\Users\yadur>pip install lxml
Collecting lxml
  Downloading https://files.pythonhosted.org/packages/bc/87/c3cecadcb5d7924cd71724b177343149cfc3609a89b197a991ac8593ed8c/lxml-4.4.1-cp37-cp37m-win_amd64.whl (3.7MB)
     |████████████████████████████████| 3.7MB 504kB/s
Installing collected packages: lxml
Successfully installed lxml-4.4.1

(myEnv) C:\Users\yadur>pip install html5lib
Collecting html5lib
  Downloading https://files.pythonhosted.org/packages/a5/62/bbd2be0e7943ec8504b517e62bab011b4946e1258842bc159e5dfde15b96/html5lib-1.0.1-py2.py3-none-any.whl (117kB)
     |████████████████████████████████| 122kB 75kB/s
Collecting webencodings
  Downloading https://files.pythonhosted.org/packages/f4/24/2a3e3df732393fed8b3ebf2ec078f05546de641fe1b667ee316ec1dcf3b7/webencodings-0.5.1-py2.py3-none-any.whl
Collecting six>=1.9
  Downloading https://files.pythonhosted.org/packages/65/26/32b8464df2a97e6dd1b656ed26b2c194606c16fe163c695a992b36c11cdf/six-1.13.0-py2.py3-none-any.whl
Installing collected packages: webencodings, six, html5lib
Successfully installed html5lib-1.0.1 six-1.13.0 webencodings-0.5.1
```

Generally, users use lxml for speed and it is recommended to use lxml or html5lib parser if you are using older version of python 2 (before 2.7.3 version) or python 3 (before 3.2.2) as python's built-in HTML parser is not very good in handling older version.

Running BeautifulSoup

It is time to test our Beautiful Soup package in one of the html pages (taking web page – https://www.tutorialspoint.com/index.htm, you can choose any-other web page you want) and extract some information from it.

In the below code, we are trying to extract the title from the webpage:

```
from bs4 import BeautifulSoup

import requests




url = "https://www.tutorialspoint.com/index.htm"

req = requests.get(url)


soup = BeautifulSoup(req.text, "html.parser")


print(soup.title)
```

Output

```
<title>H2O, Colab, Theano, Flutter, KNime, Mean.js, Weka, Solidity, Org.Json,
AWS QuickSight, JSON.Simple, Jackson Annotations, Passay, Boon, MuleSoft,
Nagios, Matplotlib, Java NIO, PyTorch, SLF4J, Parallax Scrolling, Java
Cryptography</title>
```

One common task is to extract all the URLs within a webpage. For that we just need to add the below line of code:

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

Output

```
https://www.tutorialspoint.com/index.htm
https://www.tutorialspoint.com/about/about_careers.htm
https://www.tutorialspoint.com/questions/index.php
https://www.tutorialspoint.com/online_dev_tools.htm
https://www.tutorialspoint.com/codingground.htm
https://www.tutorialspoint.com/current_affairs.htm
https://www.tutorialspoint.com/upsc_ias_exams.htm
https://www.tutorialspoint.com/tutor_connect/index.php
https://www.tutorialspoint.com/whiteboard.htm
https://www.tutorialspoint.com/netmeeting.php
https://www.tutorialspoint.com/index.htm
https://www.tutorialspoint.com/tutorialslibrary.htm
https://www.tutorialspoint.com/videotutorials/index.php
https://store.tutorialspoint.com
https://www.tutorialspoint.com/gate_exams_tutorials.htm
https://www.tutorialspoint.com/html_online_training/index.asp
https://www.tutorialspoint.com/css_online_training/index.asp
https://www.tutorialspoint.com/3d_animation_online_training/index.asp
https://www.tutorialspoint.com/swift_4_online_training/index.asp
https://www.tutorialspoint.com/blockchain_online_training/index.asp
https://www.tutorialspoint.com/reactjs_online_training/index.asp
https://www.tutorix.com
https://www.tutorialspoint.com/videotutorials/top-courses.php
https://www.tutorialspoint.com/the_full_stack_web_development/index.asp
….
….
https://www.tutorialspoint.com/online_dev_tools.htm
https://www.tutorialspoint.com/free_web_graphics.htm
https://www.tutorialspoint.com/online_file_conversion.htm
https://www.tutorialspoint.com/netmeeting.php
https://www.tutorialspoint.com/free_online_whiteboard.htm
```

```
http://www.tutorialspoint.com
https://www.facebook.com/tutorialspointindia
https://plus.google.com/u/0/+tutorialspoint
http://www.twitter.com/tutorialspoint
http://www.linkedin.com/company/tutorialspoint
https://www.youtube.com/channel/UCVLbzhxVTiTLiVKeGV7WEBg
https://www.tutorialspoint.com/index.htm
/about/about_privacy.htm#cookies
/about/faq.htm
/about/about_helping.htm
/about/contact_us.htm
```

Similarly, we can extract useful information using beautifulsoup4.

Now let us understand more about “soup” in above example.

3. Beautiful Soup — Souping the Page

In the previous code example, we parse the document through beautiful constructor using a string method. Another way is to pass the document through open filehandle.

```
from bs4 import BeautifulSoup


with open("example.html") as fp:

    soup = BeautifulSoup(fp)


soup = BeautifulSoup("<html>data</html>")
```

First the document is converted to Unicode, and HTML entities are converted to Unicode characters:

```
import bs4


html = '''<b>tutorialspoint</b>, <i>&web scraping &data science;</i>'''


soup = bs4.BeautifulSoup(html, 'lxml')


print(soup)
```

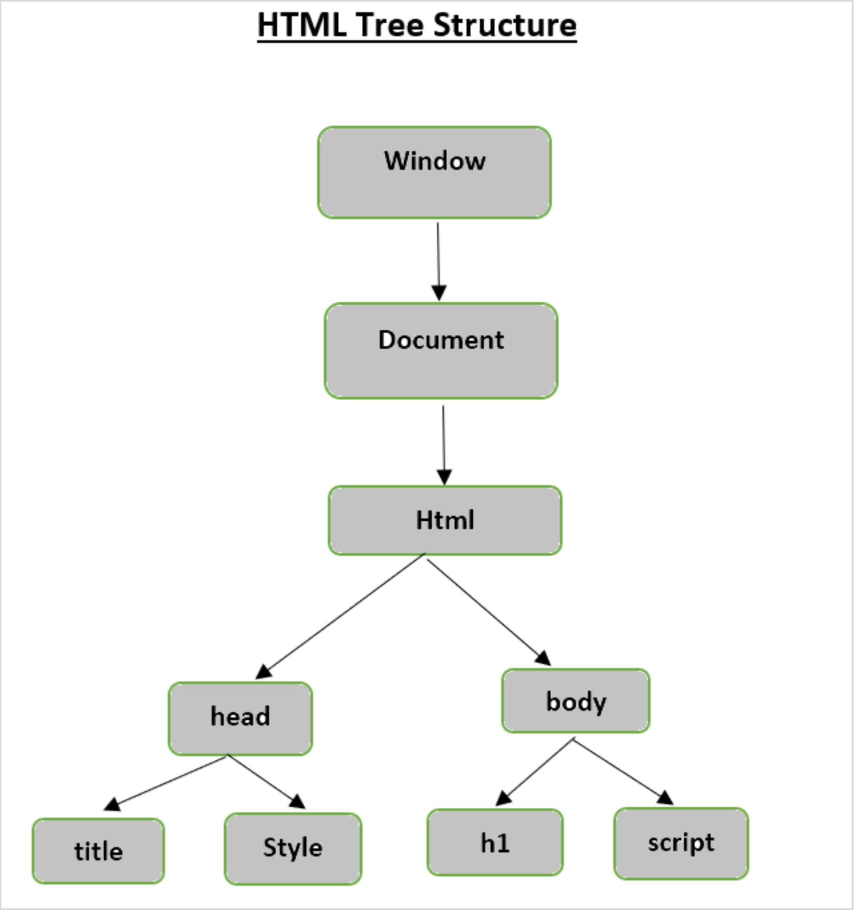
Output

```
<html><body><b>tutorialspoint</b>, <i>&amp;web scraping &amp;data
science;</i></body></html>
```

BeautifulSoup then parses the data using HTML parser or you explicitly tell it to parse using an XML parser.

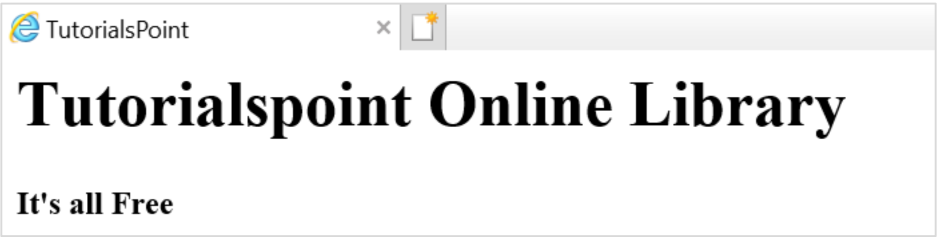
HTML tree Structure

Before we look into different components of a HTML page, let us first understand the HTML tree structure.



The root element in the document tree is the html, which can have parents, children and siblings and this determines by its position in the tree structure. To move among HTML elements, attributes and text, you have to move among nodes in your tree structure.

Let us suppose the webpage is as shown below:



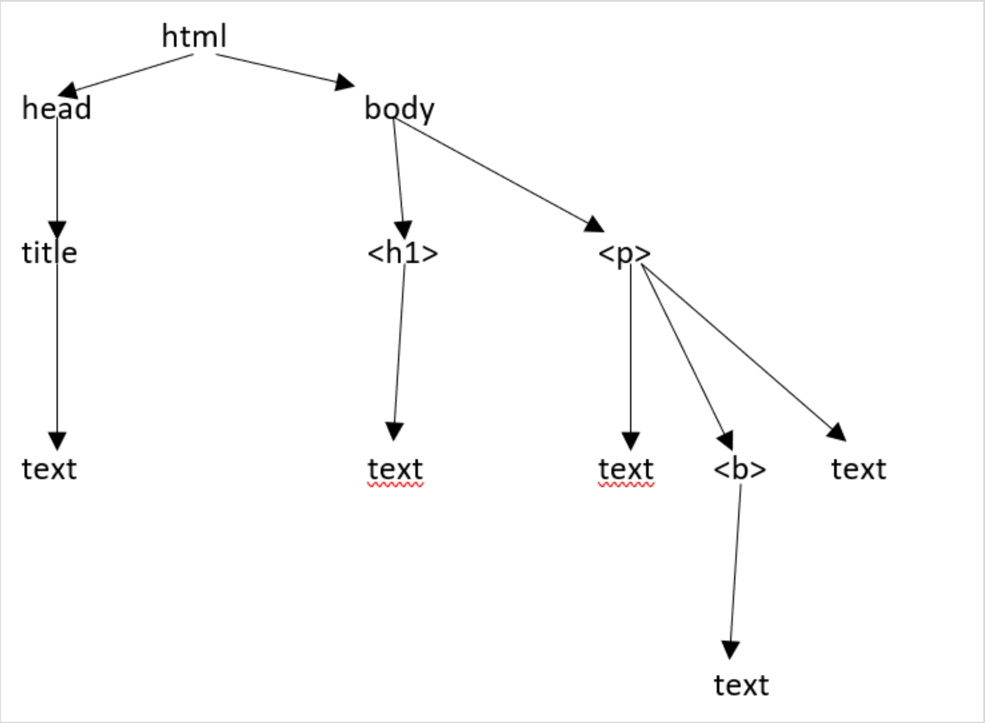
Tutorialspoint Online Library

It's all Free

Which translates to an html document as follows:

<html><head><title>TutorialsPoint</title></head><h1>Tutorialspoint Online Library</h1><p><b>It's all Free</b></p></body></html>

Which simply means, for above html document, we have a html tree structure as follows:


html
head
body
title
<h1>
<p>
text
text
text
<b>
text
text

4. Beautiful Soup — Kinds of objects

When we passed a html document or string to a beautifulsoup constructor, beautifulsoup basically converts a complex html page into different python objects. Below we are going to discuss four major kinds of objects:

- Tag
- NavigableString
- BeautifulSoup
- Comments

Tag Objects

A HTML tag is used to define various types of content. A tag object in BeautifulSoup corresponds to an HTML or XML tag in the actual page or document.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup('<b class="boldest">TutorialsPoint</b>')
>>> tag = soup.html
>>> type(tag)
<class 'bs4.element.Tag'>
```

Tags contain lot of attributes and methods and two important features of a tag are its name and attributes.

Name (tag.name)

Every tag contains a name and can be accessed through '.name' as suffix. tag.name will return the type of tag it is.

```
>>> tag.name
'html'
```

However, if we change the tag name, same will be reflected in the HTML markup generated by the BeautifulSoup.

```
>>> tag.name = "Strong"
>>> tag
<Strong><body><b class="boldest">TutorialsPoint</b></body></Strong>
>>> tag.name
'Strong'
```

Attributes (tag.attrs)

A tag object can have any number of attributes. The tag <b class=”boldest”> has an attribute ‘class’ whose value is “boldest”. Anything that is NOT tag, is basically an attribute and must contain a value. You can access the attributes either through accessing the keys (like accessing “class” in above example) or directly accessing through “.attrs”

```
>>> tutorialsP = BeautifulSoup("<div class='tutorialsP'></div>",'lxml')
>>> tag2 = tutorialsP.div
>>> tag2['class']
['tutorialsP']
```

We can do all kind of modifications to our tag’s attributes (add/remove/modify).

```
>>> tag2['class'] = 'Online-Learning'
>>> tag2['style'] = '2007'
>>>
>>> tag2
<div class="Online-Learning" style="2007"></div>
>>> del tag2['style']
>>> tag2
<div class="Online-Learning"></div>
>>> del tag['class']
>>> tag
<b SecondAttribute="2">TutorialsPoint</b>
>>>
>>> del tag['SecondAttribute']
>>> tag
<b>TutorialsPoint</b>
>>> tag2['class']
'Online-Learning'
>>> tag2['style']
KeyError: 'style'
```

Multi-valued attributes

Some of the HTML5 attributes can have multiple values. Most commonly used is the class-attribute which can have multiple CSS-values. Others include ‘rel’, ‘rev’, ‘headers’, ‘accesskey’ and ‘accept-charset’. The multi-valued attributes in beautiful soup are shown as list.

```
>>> from bs4 import BeautifulSoup
>>>
>>> css_soup = BeautifulSoup('<p class="body"></p>')
>>> css_soup.p['class']
['body']
>>>
>>> css_soup = BeautifulSoup('<p class="body bold"></p>')
>>> css_soup.p['class']
['body', 'bold']
```

However, if any attribute contains more than one value but it is not multi-valued attributes by any-version of HTML standard, beautiful soup will leave the attribute alone:

```
>>> id_soup = BeautifulSoup('<p id="body bold"></p>')
>>> id_soup.p['id']
'body bold'
>>> type(id_soup.p['id'])
<class 'str'>
```

You can consolidate multiple attribute values if you turn a tag to a string.

```
>>> rel_soup = BeautifulSoup("<p> tutorialspoint Main <a rel='Index'> 
Page</a></p>")
>>> rel_soup.a['rel']
['Index']
>>> rel_soup.a['rel'] = ['Index', ' Online Library, Its all Free']
>>> print(rel_soup.p)
<p> tutorialspoint Main <a rel="Index  Online Library, Its all Free"> 
Page</a></p>
```

By using ‘get_attribute_list’, you get a value that is always a list, string, irrespective of whether it is a multi-valued or not.

```
id_soup.p.get_attribute_list(‘id’)
```

However, if you parse the document as ‘xml’, there are no multi-valued attributes:

```
>>> xml_soup = BeautifulSoup('<p class="body bold"></p>', 'xml')
>>> xml_soup.p['class']
'body bold'
```

NavigableString

The navigablestring object is used to represent the contents of a tag. To access the contents, use ".string" with tag.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup("<h2 id='message'>Hello, Tutorialspoint!</h2>")
>>>
>>> soup.string
'Hello, Tutorialspoint!'
>>> type(soup.string)
<class 'bs4.element.NavigableString'>
```

You can replace the string with another string but you can't edit the existing string.

```
>>> soup = BeautifulSoup("<h2 id='message'>Hello, Tutorialspoint!</h2>")
>>> soup.string.replace_with("Online Learning!")
'Hello, Tutorialspoint!'
>>> soup.string
'Online Learning!'
>>> soup
<html><body><h2 id="message">Online Learning!</h2></body></html>
```

BeautifulSoup

BeautifulSoup is the object created when we try to scrape a web resource. So, it is the complete document which we are trying to scrape. Most of the time, it is treated tag object.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup("<h2 id='message'>Hello, Tutorialspoint!</h2>")
>>> type(soup)
<class 'bs4.BeautifulSoup'>
>>> soup.name
'[document]'
```

Comments

The comment object illustrates the comment part of the web document. It is just a special type of NavigableString.

```
>>> soup = BeautifulSoup('<p><!-- Everything inside it is COMMENTS --></p>')
>>> comment = soup.p.string
>>> type(comment)
<class 'bs4.element.Comment'>
>>> type(comment)
<class 'bs4.element.Comment'>
>>> print(soup.p.prettify())
<p>
 <!-- Everything inside it is COMMENTS -->
</p>
```

NavigableString Objects

The navigablestring objects are used to represent text within tags, rather than the tags themselves.

5. Beautiful Soup — Navigating by Tags

In this chapter, we shall discuss about Navigating by Tags.

Below is our html document:

```
>>> html_doc = """
<html><head><title>Tutorials Point</title></head>
<body>
<p class="title"><b>The Biggest Online Tutorials Library, It's all Free</b></p>
<p class="prog">Top 5 most used Programming Languages are:
<a href="https://www.tutorialspoint.com/java/java_overview.htm" class="prog"
id="link1">Java</a>,
<a href="https://www.tutorialspoint.com/cprogramming/index.htm" class="prog"
id="link2">C</a>,
<a href="https://www.tutorialspoint.com/python/index.htm" class="prog"
id="link3">Python</a>,
<a href="https://www.tutorialspoint.com/javascript/javascript_overview.htm"
class="prog" id="link4">JavaScript</a> and
<a href="https://www.tutorialspoint.com/ruby/index.htm" class="prog"
id="link5">C</a>;


as per online survey.</p>
<p class="prog">Programming Languages</p>
"""
>>>


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

Based on the above document, we will try to move from one part of document to another.

Going down

One of the important pieces of element in any piece of HTML document are tags, which may contain other tags/strings (tag's children). Beautiful Soup provides different ways to navigate and iterate over's tag's children.

Navigating using tag names

Easiest way to search a parse tree is to search the tag by its name. If you want the <head> tag, use soup.head:

```
>>> soup.head

<head><title>Tutorials Point</title></head>

>>> soup.title

<title>Tutorials Point</title>
```

To get specific tag (like first <b> tag) in the <body> tag.

```
>>> soup.body.b

<b>The Biggest Online Tutorials Library, It's all Free</b>
```

Using a tag name as an attribute will give you only the first tag by that name:

```
>>> soup.a

<a class="prog" href="https://www.tutorialspoint.com/java/java_overview.htm"
id="link1">Java</a>
```

To get all the tag's attribute, you can use find_all() method:

```
>>> soup.find_all("a")

[<a class="prog" href="https://www.tutorialspoint.com/java/java_overview.htm"
id="link1">Java</a>, <a class="prog"
href="https://www.tutorialspoint.com/cprogramming/index.htm" id="link2">C</a>,
<a class="prog" href="https://www.tutorialspoint.com/python/index.htm"
id="link3">Python</a>, <a class="prog"
href="https://www.tutorialspoint.com/javascript/javascript_overview.htm"
id="link4">JavaScript</a>, <a class="prog"
href="https://www.tutorialspoint.com/ruby/index.htm" id="link5">C</a>]
```

.contents and .children

We can search tag's children in a list by its .contents:

```
>>> head_tag = soup.head

>>> head_tag

<head><title>Tutorials Point</title></head>

>>> Htag = soup.head

>>> Htag

<head><title>Tutorials Point</title></head>

>>>

>>> Htag.contents

[<title>Tutorials Point</title>]
```

```
>>>
>>> Ttag = head_tag.contents[0]
>>> Ttag
<title>Tutorials Point</title>


>>> Ttag.contents
['Tutorials Point']
```

The BeautifulSoup object itself has children. In this case, the <html> tag is the child of the BeautifulSoup object:

```
>>> len(soup.contents)
2
>>> soup.contents[1].name
'html'
```

A string does not have .contents, because it can’t contain anything:

```
>>> text = Ttag.contents[0]
>>> text.contents
self.__class__.__name__, attr))
AttributeError: 'NavigableString' object has no attribute 'contents'
```

Instead of getting them as a list, use .children generator to access tag’s children:

```
>>> for child in Ttag.children:
     print(child)



Tutorials Point
```

.descendants

The .descendants attribute allows you to iterate over all of a tag’s children, recursively: its direct children and the children of its direct children and so on:

```
>>> for child in Htag.descendants:
     print(child)


     <title>Tutorials Point</title>
Tutorials Point
```

The <head> tag has only one child, but it has two descendants: the <title> tag and the <title> tag's child. The beautifulsoup object has only one direct child (the <html> tag), but it has a whole lot of descendants:

```
>>> len(list(soup.children))
2
>>> len(list(soup.descendants))
33
```

.string

If the tag has only one child, and that child is a NavigableString, the child is made available as .string:

```
>>> Ttag.string
'Tutorials Point'
```

If a tag's only child is another tag, and that tag has a .string, then the parent tag is considered to have the same .string as its child:

```
>>> Htag.contents
[<title>Tutorials Point</title>]
>>>
>>> Htag.string
'Tutorials Point'
```

However, if a tag contains more than one thing, then it's not clear what .string should refer to, so .string is defined to None:

```
>>> print(soup.html.string)
None
```

.strings and stripped_strings

If there's more than one thing inside a tag, you can still look at just the strings. Use the .strings generator:

```
>>> for string in soup.strings:
     print(repr(string))



'\n'
'Tutorials Point'
'\n'
```

```
'\n'
"The Biggest Online Tutorials Library, It's all Free"
'\n'
'Top 5 most used Programming Languages are: \n'
'Java'
',\n'
'C'
',\n'
'Python'
',\n'
'JavaScript'
' and\n'
'C'
';\n \nas per online survey.'
'\n'
'Programming Languages'
'\n'
```

To remove extra whitespace, use .stripped_strings generator:

```
>>> for string in soup.stripped_strings:
     print(repr(string))



'Tutorials Point'
"The Biggest Online Tutorials Library, It's all Free"
'Top 5 most used Programming Languages are:'
'Java'
','
'C'
','
'Python'
','
'JavaScript'
'and'
'C'
';\n \nas per online survey.'
'Programming Languages'
```

Going up

In a “family tree” analogy, every tag and every string has a parent: the tag that contain it:

.parent

To access the element’s parent element, use .parent attribute.

```
>>> Ttag = soup.title

>>> Ttag

<title>Tutorials Point</title>

>>> Ttag.parent

<head><title>Tutorials Point</title></head>
```

In our html_doc, the title string itself has a parent: the <title> tag that contain it:

```
>>> Ttag.string.parent

<title>Tutorials Point</title>
```

The parent of a top-level tag like <html> is the Beautifulsoup object itself:

```
>>> htmltag = soup.html

>>> type(htmltag.parent)

<class 'bs4.BeautifulSoup'>
```

The .parent of a Beautifulsoup object is defined as None:

```
>>> print(soup.parent)

None
```

.parents

To iterate over all the parents elements, use .parents attribute.

```
>>> link = soup.a

>>> link

<a class="prog" href="https://www.tutorialspoint.com/java/java_overview.htm"
id="link1">Java</a>

>>>

>>> for parent in link.parents:

     if parent is None:

            print(parent)

     else:

            print(parent.name)
```

```
p
body
html
[document]
```

Going sideways

Below is one simple document:

```
>>> sibling_soup = BeautifulSoup("<a><b>TutorialsPoint</b><c><strong>The
Biggest Online Tutorials Library, It's all Free</strong></b></a>")
>>> print(sibling_soup.prettify())
<html>
 <body>
  <a>
   <b>
    TutorialsPoint
   </b>
   <c>
    <strong>
     The Biggest Online Tutorials Library, It's all Free
    </strong>
   </c>
  </a>
 </body>
</html>
```

In the above doc, <b> and <c> tag is at the same level and they are both children of the same tag. Both <b> and <c> tag are siblings.

.next_sibling and .previous_sibling

Use .next_sibling and .previous_sibling to navigate between page elements that are on the same level of the parse tree:

```
>>> sibling_soup.b.next_sibling
<c><strong>The Biggest Online Tutorials Library, It's all Free</strong></c>
>>>
>>> sibling_soup.c.previous_sibling
<b>TutorialsPoint</b>
```

The <b> tag has a .next_sibling but no .previous_sibling, as there is nothing before the <b> tag on the same level of the tree, same case is with <c> tag.

```
>>> print(sibling_soup.b.previous_sibling)
None
>>> print(sibling_soup.c.next_sibling)
None
```

The two strings are not siblings, as they don’t have the same parent.

```
>>> sibling_soup.b.string
'TutorialsPoint'
>>>
>>> print(sibling_soup.b.string.next_sibling)
None
```

.next_siblings and .previous_siblings

To iterate over a tag’s siblings use .next_siblings and .previous_siblings.

```
>>> for sibling in soup.a.next_siblings:
     print(repr(sibling))



',\n'
<a class="prog" href="https://www.tutorialspoint.com/cprogramming/index.htm"
id="link2">C</a>
',\n'
<a class="prog" href="https://www.tutorialspoint.com/python/index.htm"
id="link3">Python</a>
',\n'
<a class="prog"
href="https://www.tutorialspoint.com/javascript/javascript_overview.htm"
id="link4">JavaScript</a>
' and\n'
<a class="prog" href="https://www.tutorialspoint.com/ruby/index.htm"
id="link5">C</a>
';\n \nas per online survey.'


>>> for sibling in soup.find(id="link3").previous_siblings:
     print(repr(sibling))
```

```
',\n'

<a class="prog" href="https://www.tutorialspoint.com/cprogramming/index.htm"
id="link2">C</a>

',\n'

<a class="prog" href="https://www.tutorialspoint.com/java/java_overview.htm"
id="link1">Java</a>

'Top 5 most used Programming Languages are: \n'
```

Going back and forth

Now let us get back to first two lines in our previous "html_doc" example:

```
<html><head><title>Tutorials Point</title></head>

<body>

<h4 class="tagLine"><b>The Biggest Online Tutorials Library, It's all
Free</b></h4>
```

An HTML parser takes above string of characters and turns it into a series of events like "open an <html> tag", "open an <head> tag", "open the <title> tag", "add a string", "close the </title> tag", "close the </head> tag", "open a <h4> tag" and so on. BeautifulSoup offers different methods to reconstructs the initial parse of the document.

.next_element and .previous_element

The .next_element attribute of a tag or string points to whatever was parsed immediately afterwards. Sometimes it looks similar to .next_sibling, however it is not same entirely.

Below is the final <a> tag in our "html_doc" example document.

```
>>> last_a_tag = soup.find("a", id="link5")

>>> last_a_tag

<a class="prog" href="https://www.tutorialspoint.com/ruby/index.htm"
id="link5">C</a>

>>> last_a_tag.next_sibling

';\n \nas per online survey.'
```

However the .next_element of that <a> tag, the thing that was parsed immediately after the <a> tag, is not the rest of that sentence: it is the word "C":

```
>>> last_a_tag.next_element

'C'
```

Above behavior is because in the original markup, the letter "C" appeared before that semicolon. The parser encountered an <a> tag, then the letter "C", then the closing </a> tag, then the semicolon and rest of the sentence. The semicolon is on the same level as the <a> tag, but the letter "C" was encountered first.

The .previous_element attribute is the exact opposite of .next_element. It points to whatever element was parsed immediately before this one.

```
>>> last_a_tag.previous_element

' and\n'

>>>

>>> last_a_tag.previous_element.next_element

<a class="prog" href="https://www.tutorialspoint.com/ruby/index.htm"
id="link5">C</a>
```

.next_elements and .previous_elements

We use these iterators to move forward and backward to an element.

```
>>> for element in last_a_tag.next_e  lements:

     print(repr(element))



'C'

';\n \nas per online survey.'

'\n'

<p class="prog">Programming Languages</p>

'Programming Languages'

'\n'
```

6. Beautiful Soup — Searching the tree

There are many Beautifulsoup methods, which allows us to search a parse tree. The two most common and used methods are find() and find_all().

Before talking about find() and find_all(), let us see some examples of different filters you can pass into these methods.

Kinds of Filters

We have different filters which we can pass into these methods and understanding of these filters is crucial as these filters used again and again, throughout the search API. We can use these filters based on tag's name, on its attributes, on the text of a string, or mixed of these.

A string

One of the simplest types of filter is a string. Passing a string to the search method and Beautifulsoup will perform a match against that exact string.

Below code will find all the <p> tags in the document:

```
>>> markup = BeautifulSoup('<p>Top Three</p><p><pre>Programming Languages
are:</pre></p><p><b>Java, Python, Cplusplus</b></p>')

>>> markup.find_all('p')

[<p>Top Three</p>, <p></p>, <p><b>Java, Python, Cplusplus</b></p>]
```

Regular Expression

You can find all tags starting with a given string/tag. Before that we need to import the re module to use regular expression.

```
>>> import re

>>> markup = BeautifulSoup('<p>Top Three</p><p><pre>Programming Languages
are:</pre></p><p><b>Java, Python, Cplusplus</b></p>')

>>>

>>> markup.find_all(re.compile('^p'))

[<p>Top Three</p>, <p></p>, <pre>Programming Languages are:</pre>, <p><b>Java,
Python, Cplusplus</b></p>]
```

List

You can pass multiple tags to find by providing a list. Below code finds all the <b> and <pre> tags:

```
>>> markup.find_all(['pre', 'b'])
[<pre>Programming Languages are:</pre>, <b>Java, Python, Cplusplus</b>]
```

True

True will return all tags that it can find, but no strings on their own:

```
>>> markup.find_all(True)
[<html><body><p>Top Three</p><p></p><pre>Programming Languages
are:</pre><p><b>Java, Python, Cplusplus</b></p></body></html>, <body><p>Top
Three</p><p></p><pre>Programming Languages are:</pre><p><b>Java, Python,
Cplusplus</b></p></body>, <p>Top Three</p>, <p></p>, <pre>Programming Languages
are:</pre>, <p><b>Java, Python, Cplusplus</b></p>, <b>Java, Python,
Cplusplus</b>]
```

To return only the tags from the above soup:

```
>>> for tag in markup.find_all(True):
     (tag.name)
```

```
'html'
'body'
'p'
'p'
'pre'
'p'
'b'
```

find_all()

You can use find_all to extract all the occurrences of a particular tag from the page response as:

Syntax

```
find_all(name, attrs, recursive, string, limit, **kwargs)
```

Let us extract some interesting data from IMDB-"Top rated movies" of all time.

```
>>> url="https://www.imdb.com/chart/top/?ref_=nv_mv_250"
>>> content = requests.get(url)
>>> soup = BeautifulSoup(content.text, 'html.parser')


#Extract title Page
```

```
>>> print(soup.find('title'))
<title>IMDb Top 250 - IMDb</title>

#Extracting main heading
>>> for heading in soup.find_all('h1'):
     print(heading.text)



Top Rated Movies

#Extracting sub-heading
>>> for heading in soup.find_all('h3'):
     print(heading.text)



IMDb Charts
You Have Seen
 IMDb Charts
 Top India Charts
Top Rated Movies by Genre
Recently Viewed
```

From above, we can see find_all will give us all the items matching the search criteria we define. All the filters we can use with find_all() can be used with find() and other searching methods too like find_parents() or find_siblings().

find()

We have seen above, find_all() is used to scan the entire document to find all the contents but something, the requirement is to find only one result. If you know that the document contains only one <body> tag, it is waste of time to search the entire document. One way is to call find_all() with limit=1 every time or else we can use find() method to do the same:

Syntax

```
find(name, attrs, recursive, string, **kwargs)
```

So below two different methods gives the same output:

```
>>> soup.find_all('title',limit=1)
[<title>IMDb Top 250 - IMDb</title>]
>>>
>>> soup.find('title')
<title>IMDb Top 250 - IMDb</title>
```

In the above outputs, we can see the find_all() method returns a list containing single item whereas find() method returns single result.

Another difference between find() and find_all() method is:

```
>>> soup.find_all('h2')
[]
>>>
>>> soup.find('h2')
```

If soup.find_all() method can’t find anything, it returns empty list whereas find() returns None.

find_parents() and find_parent()

Unlike the find_all() and find() methods which traverse the tree, looking at tag’s descendents, find_parents() and find_parents methods() do the opposite, they traverse the tree upwards and look at a tag’s (or a string’s) parents.

Syntax

```
find_parents(name, attrs, string, limit, **kwargs)


find_parent(name, attrs, string, **kwargs)
```

```
>>> a_string = soup.find(string="The Godfather")
>>> a_string
'The Godfather'
>>> a_string.find_parents('a')
[<a href="/title/tt0068646/" title="Francis Ford Coppola (dir.), Marlon Brando,
Al Pacino">The Godfather</a>]
>>> a_string.find_parent('a')
<a href="/title/tt0068646/" title="Francis Ford Coppola (dir.), Marlon Brando,
Al Pacino">The Godfather</a>


>>> a_string.find_parent('tr')
<tr>
```

```
<td class="posterColumn">
<span data-value="2" name="rk"></span>
<span data-value="9.149038526210072" name="ir"></span>
<span data-value="6.93792E10" name="us"></span>
<span data-value="1485540" name="nv"></span>
<span data-value="-1.850961473789928" name="ur"></span>
<a href="/title/tt0068646/"> <img alt="The Godfather" height="67"
src="https://m.media-
amazon.com/images/M/MV5BM2MyNjYxNmUtYTAwNi00MTYxLWJmNWYtYzZlODY3ZTk3OTFlXkEyXkF
qcGdeQXVyNzkwMjQ5NzM@._V1_UY67_CR1,0,45,67_AL_.jpg" width="45"/>
</a> </td>
<td class="titleColumn">
      2.
      <a href="/title/tt0068646/" title="Francis Ford Coppola (dir.), Marlon
Brando, Al Pacino">The Godfather</a>
<span class="secondaryInfo">(1972)</span>
</td>
<td class="ratingColumn imdbRating">
<strong title="9.1 based on 1,485,540 user ratings">9.1</strong>
</td>
<td class="ratingColumn">
<div class="seen-widget seen-widget-tt0068646 pending" data-
titleid="tt0068646">
<div class="boundary">
<div class="popover">
<span
class="delete"> </span><ol><li>1<li>2<li>3<li>4<li>5<li>6<li>7<li>8<li>9<li>10<
/li></li></li></li></li></li></li></li></li></li></ol> </div>
</div>
<div class="inline">
<div class="pending"></div>
<div class="unseeable">NOT YET RELEASED</div>
<div class="unseen"> </div>
<div class="rating"></div>
<div class="seen">Seen</div>
</div>
</div>
</td>
<td class="watchlistColumn">
```

```
<div class="wlb_ribbon" data-recordmetrics="true" data-
tconst="tt0068646"></div>
</td>
</tr>
>>>


>>> a_string.find_parents('td')
[<td class="titleColumn">
      2.
      <a href="/title/tt0068646/" title="Francis Ford Coppola (dir.), Marlon
Brando, Al Pacino">The Godfather</a>
<span class="secondaryInfo">(1972)</span>
</td>]
```

There are eight other similar methods:

```
find_next_siblings(name, attrs, string, limit, **kwargs)
find_next_sibling(name, attrs, string, **kwargs)


find_previous_siblings(name, attrs, string, limit, **kwargs)
find_previous_sibling(name, attrs, string, **kwargs)


find_all_next(name, attrs, string, limit, **kwargs)
find_next(name, attrs, string, **kwargs)


find_all_previous(name, attrs, string, limit, **kwargs)
find_previous(name, attrs, string, **kwargs)
```

Where,

find_next_siblings() and **find_next_sibling()** methods will iterate over all the siblings of the element that come after the current one.

find_previous_siblings() and **find_previous_sibling()** methods will iterate over all the siblings that come before the current element.

find_all_next() and **find_next()** methods will iterate over all the tags and strings that come after the current element.

find_all_previous and **find_previous()** methods will iterate over all the tags and strings that come before the current element.

CSS selectors

The BeautifulSoup library to support the most commonly-used CSS selectors. You can search for elements using CSS selectors with the help of the select() method.

Here are some examples:

```
>>> soup.select('title')

[<title>IMDb Top 250 - IMDb</title>, <title>IMDb Top Rated Movies</title>]

>>>

>>> soup.select("p:nth-of-type(1)")

[<p>The Top Rated Movie list only includes theatrical features.</p>, <p
class="imdb-footer__copyright _2-iNNCFskmr4l2OFN2DRsf">© 1990-<!-- -->2019<!--
--> by IMDb.com, Inc.</p>]

>>> len(soup.select("p:nth-of-type(1)"))

2

>>> len(soup.select("a"))

609

>>> len(soup.select("p"))

2


>>> soup.select("html head title")

[<title>IMDb Top 250 - IMDb</title>, <title>IMDb Top Rated Movies</title>]

>>> soup.select("head > title")

[<title>IMDb Top 250 - IMDb</title>]


#print HTML code of the tenth li elemnet

>>> soup.select("li:nth-of-type(10)")

[<li class="subnav_item_main">

<a
href="/search/title?genres=film_noir&amp;sort=user_rating,desc&amp;title_type=f
eature&amp;num_votes=25000,"> Film-Noir

</a> </li>]
```

7. Beautiful Soup — Modifying the tree

One of the important aspects of BeautifulSoup is search the parse tree and it allows you to make changes to the web document according to your requirement. We can make changes to tag's properties using its attributes, such as the .name, .string or .append() method. It allows you to add new tags and strings to an existing tag with the help of the .new_string() and .new_tag() methods. There are other methods too, such as .insert(), .insert_before() or .insert_after() to make various modification to your HTML or XML document.

Changing tag names and attributes

Once you have created the soup, it is easy to make modification like renaming the tag, make modification to its attributes, add new attributes and delete attributes.

```
>>> soup = BeautifulSoup('<b class="bolder">Very Bold</b>')
>>> tag = soup.b
```

Modification and adding new attributes are as follows:

```
>>> tag.name = 'Blockquote'
>>> tag['class'] = 'Bolder'
>>> tag['id'] = 1.1
>>> tag
<Blockquote class="Bolder" id="1.1">Very Bold</Blockquote>
```

Deleting attributes are as follows:

```
>>> del tag['class']
>>> tag
<Blockquote id="1.1">Very Bold</Blockquote>

>>> del tag['id']
>>> tag
<Blockquote>Very Bold</Blockquote>
```

Modifying .string

You can easily modify the tag's .string attribute:

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">Must for every <i>Learner</i></a>'
>>> Bsoup = BeautifulSoup(markup)
>>> tag = Bsoup.a

>>> tag.string = "My Favourite spot."

>>> tag

<a href="https://www.tutorialspoint.com/index.htm">My Favourite spot.</a>
```

From above, we can see if the tag contains any other tag, they and all their contents will be replaced by new data.

append()

Adding new data/contents to an existing tag is by using tag.append() method. It is very much similar to append() method in Python list.

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">Must for every
<i>Learner</i></a>'

>>> Bsoup = BeautifulSoup(markup)


>>> Bsoup.a.append(" Really Liked it")

>>> Bsoup

<html><body><a href="https://www.tutorialspoint.com/index.htm">Must for every
<i>Learner</i> Really Liked it</a></body></html>


>>> Bsoup.a.contents

['Must for every ', <i>Learner</i>, ' Really Liked it']
```

NavigableString() and .new_tag()

In case you want to add a string to a document, this can be done easily by using the append() or by NavigableString() constructor:

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

>>> tag = soup.b

>>> tag.append("Start")

>>>

>>> new_string = NavigableString(" Your")

>>> tag.append(new_string)

>>> tag

<b>Start Your</b>

>>> tag.contents

['Start', ' Your']
```

Note: If you find any name Error while accessing the NavigableString() function, as follows:

NameError: name 'NavigableString' is not defined

Just import the NavigableString directory from bs4 package:

```
>>> from bs4 import NavigableString
```

We can resolve the above error.

You can add comments to your existing tag's or can add some other subclass of NavigableString, just call the constructor.

```
>>> from bs4 import Comment
>>> adding_comment = Comment("Always Learn something Good!")
>>> tag.append(adding_comment)
>>> tag
<b>Start Your<!--Always Learn something Good!--></b>
>>> tag.contents
['Start', ' Your', 'Always Learn something Good!']
```

Adding a whole new tag (not appending to an existing tag) can be done using the Beautifulsoup inbuilt method, BeautifulSoup.new_tag():

```
>>> soup = BeautifulSoup("<b></b>")
>>> Otag = soup.b
>>>
>>> Newtag = soup.new_tag("a", href="https://www.tutorialspoint.com")
>>> Otag.append(Newtag)
>>> Otag
<b><a href="https://www.tutorialspoint.com"></a></b>
```

Only the first argument, the tag name, is required.

insert()

Similar to .insert() method on python list, tag.insert() will insert new element however, unlike tag.append(), new element doesn't necessarily go at the end of its parent's contents. New element can be added at any position.

```
>>> markup = '<a href="https://www.djangoproject.com/community/">Django
Official website <i>Huge Community base</i></a>'
>>> soup = BeautifulSoup(markup)
>>> tag = soup.a
>>>
>>> tag.insert(1, "Love this framework ")
>>> tag
<a href="https://www.djangoproject.com/community/">Django Official website Love
this framework <i>Huge Community base</i></a>
>>> tag.contents
['Django Official website ', 'Love this framework ', <i>Huge Community
base</i>]
>>>
```

insert_before() and insert_after()

To insert some tag or string just before something in the parse tree, we use insert_before():

```
>>> soup = BeautifulSoup("<b>Brave</b>")
>>> tag = soup.new_tag("i")
>>> tag.string = "Be"
>>>
>>> soup.b.string.insert_before(tag)
>>> soup.b
<b><i>Be</i>Brave</b>
```

Similarly to insert some tag or string just after something in the parse tree, use insert_after().

```
>>> soup.b.i.insert_after(soup.new_string(" Always "))
>>> soup.b
<b><i>Be</i> Always Brave</b>
>>> soup.b.contents
[<i>Be</i>, ' Always ', 'Brave']
```

clear()

To remove the contents of a tag, use tag.clear():

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">For
<i>technical & Non-technical</i> Contents</a>'
>>> soup = BeautifulSoup(markup)
>>> tag = soup.a
>>> tag
<a href="https://www.tutorialspoint.com/index.htm">For <i>technical &amp; Non-
technical</i> Contents</a>
>>>
```

```
>>> tag.clear()

>>> tag

<a href="https://www.tutorialspoint.com/index.htm"></a>
```

extract()

To remove a tag or strings from the tree, use PageElement.extract().

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">For
<i>technical & Non-technical</i> Contents</a>'

>>> soup = BeautifulSoup(markup)

>>> a_tag = soup.a

>>>

>>> i_tag = soup.i.extract()

>>>

>>> a_tag

<a href="https://www.tutorialspoint.com/index.htm">For  Contents</a>

>>>

>>> i_tag

<i>technical &amp; Non-technical</i>

>>>

>>> print(i_tag.parent)

None
```

decompose()

The tag.decompose() removes a tag from the tree and deletes all its contents.

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">For
<i>technical & Non-technical</i> Contents</a>'

>>> soup = BeautifulSoup(markup)

>>> a_tag = soup.a

>>> a_tag

<a href="https://www.tutorialspoint.com/index.htm">For <i>technical &amp; Non-
technical</i> Contents</a>

>>>

>>> soup.i.decompose()

>>> a_tag

<a href="https://www.tutorialspoint.com/index.htm">For  Contents</a>

>>>
```

Replace_with()

As the name suggests, pageElement.replace_with() function will replace the old tag or string with the new tag or string in the tree:

```
>>> markup = '<a href="https://www.tutorialspoint.com/index.htm">Complete
Python <i>Material</i></a>'
>>> soup = BeautifulSoup(markup)
>>> a_tag = soup.a
>>>
>>> new_tag = soup.new_tag("Official_site")
>>> new_tag.string = "https://www.python.org/"
>>> a_tag.i.replace_with(new_tag)
<i>Material</i>
>>>
>>> a_tag
<a href="https://www.tutorialspoint.com/index.htm">Complete Python
<Official_site>https://www.python.org/</Official_site></a>
```

In the above output, you have noticed that replace_with() returns the tag or string that was replaced (like "Material" in our case), so you can examine it or add it back to another part of the tree.

wrap()

The pageElement.wrap() enclosed an element in the tag you specify and returns a new wrapper:

```
>>> soup = BeautifulSoup("<p>tutorialspoint.com</p>")
>>> soup.p.string.wrap(soup.new_tag("b"))
<b>tutorialspoint.com</b>
>>>
>>> soup.p.wrap(soup.new_tag("Div"))
<Div><p><b>tutorialspoint.com</b></p></Div>
```

unwrap()

The tag.unwrap() is just opposite to wrap() and replaces a tag with whatever inside that tag.

```
>>> soup = BeautifulSoup('<a href="https://www.tutorialspoint.com/">I liked
<i>tutorialspoint</i></a>')
>>> a_tag = soup.a
>>>
```

```
>>> a_tag.i.unwrap()
<i></i>
>>> a_tag
<a href="https://www.tutorialspoint.com/">I liked tutorialspoint</a>
```

From above, you have noticed that like replace_with(), unwrap() returns the tag that was replaced.

Below is one more example of unwrap() to understand it better:

```
>>> soup = BeautifulSoup("<p>I <strong>AM</strong> a <i>text</i>.</p>")
>>> soup.i.unwrap()
<i></i>
>>> soup
<html><body><p>I <strong>AM</strong> a text.</p></body></html>
```

unwrap() is good for striping out markup.

8. Beautiful Soup — Encoding

All HTML or XML documents are written in some specific encoding like ASCII or UTF-8. However, when you load that HTML/XML document into BeautifulSoup, it has been converted to Unicode.

```
>>> markup = "<p>I will display &#x00A3;</p>"
>>> Bsoup = BeautifulSoup(markup)
>>> Bsoup.p
<p>I will display £</p>
>>> Bsoup.p.string
'I will display £'
```

Above behavior is because BeautifulSoup internally uses the sub-library called Unicode, Dammit to detect a document's encoding and then convert it into Unicode.

However, not all the time, the Unicode, Dammit guesses correctly. As the document is searched byte-by-byte to guess the encoding, it takes lot of time. You can save some time and avoid mistakes, if you already know the encoding by passing it to the BeautifulSoup constructor as from_encoding.

Below is one example where the BeautifulSoup misidentifies, an ISO-8859-8 document as ISO-8859-7:

```
>>> markup = b"<h1>\xed\xe5\xec\xf9</h1>"
>>> soup = BeautifulSoup(markup)
>>> soup.h1
<h1>νεμω</h1>
>>> soup.original_encoding
'ISO-8859-7'
>>>
```

To resolve above issue, pass it to BeautifulSoup using from_encoding:

```
>>> soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
>>> soup.h1
<h1>שלום</h1>
>>> soup.original_encoding
'iso-8859-8'
>>>
```

Another new feature added from BeautifulSoup 4.4.0 is, exclude_encoding. It can be used, when you don't know the correct encoding but sure that Unicode, Dammit is showing wrong result.

```
>>> soup = BeautifulSoup(markup, exclude_encodings=["ISO-8859-7"])
```

Output encoding

The output from a BeautifulSoup is UTF-8 document, irrespective of the entered document to BeautifulSoup. Below a document, where the polish characters are there in ISO-8859-2 format.

```
html_markup = """
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<META HTTP-EQUIV="content-type" CONTENT="text/html; charset=iso-8859-2">
</HEAD>
<BODY>
ą ć ę ł ń ó ś ź ż Ą Ć Ę Ł Ń Ó Ś Ź Ż
</BODY>
</HTML>
"""


>>> soup = BeautifulSoup(html_markup)
>>> print(soup.prettify())
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <meta content="text/html; charset=utf-8" http-equiv="content-type"/>
 </head>
 <body>
  ą ć ę ł ń ó ś ź ż Ą Ć Ę Ł Ń Ó Ś Ź Ż
 </body>
</html>
```

In the above example, if you notice, the <meta> tag has been rewritten to reflect the generated document from BeautifulSoup is now in UTF-8 format.

If you don't want the generated output in UTF-8, you can assign the desired encoding in prettify().

```
>>> print(soup.prettify("latin-1"))

b'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">\n<html>\n
<head>\n  <meta content="text/html; charset=latin-1" http-equiv="content-
type"/>\n </head>\n <body>\n  &#261; &#263; &#281; &#322; &#324; \xf3 &#347;
&#378; &#380; &#260; &#262; &#280; &#321; &#323; \xd3 &#346; &#377; &#379;\n
</body>\n</html>\n'
```

In the above example, we have encoded the complete document, however you can encode, any particular element in the soup as if they were a python string:

```
>>> soup.p.encode("latin-1")

b'<p>My first paragraph.</p>'

>>> soup.h1.encode("latin-1")

b'<h1>My First Heading</h1>'
```

Any characters that can’t be represented in your chosen encoding will be converted into numeric XML entity references. Below is one such example:

```
>>> markup = u"<b>\N{SNOWMAN}</b>"

>>> snowman_soup = BeautifulSoup(markup)

>>> tag = snowman_soup.b

>>> print(tag.encode("utf-8"))

b'<b>\xe2\x98\x83</b>'
```

If you try to encode the above in “latin-1” or “ascii”, it will generate “&#9731”, indicating there is no representation for that.

```
>>> print (tag.encode("latin-1"))

b'<b>&#9731;</b>'

>>> print (tag.encode("ascii"))

b'<b>&#9731;</b>'
```

Unicode, Dammit

Unicode, Dammit is used mainly when the incoming document is in unknown format (mainly foreign language) and we want to encode in some known format (Unicode) and also we don’t need Beautifulsoup to do all this.

9. Beautiful Soup — Beautiful Objects

The starting point of any BeautifulSoup project, is the BeautifulSoup object. A BeautifulSoup object represents the input HTML/XML document used for its creation.

We can either pass a string or a file-like object for Beautiful Soup, where files (objects) are either locally stored in our machine or a web page.

The most common BeautifulSoup Objects are:

- Tag
- NavigableString
- BeautifulSoup
- Comment

Comparing objects for equality

As per the beautiful soup, two navigable string or tag objects are equal if they represent the same HTML/XML markup.

Now let us see the below example, where the two <b> tags are treated as equal, even though they live in different parts of the object tree, because they both look like "<b>Java</b>".

```
>>> markup = "<p>Learn Python and  <b>Java</b> and advanced <b>Java</b>! from
Tutorialspoint</p>"
>>> soup = BeautifulSoup(markup, "html.parser")
>>> first_b, second_b = soup.find_all('b')
>>> print(first_b == second_b)
True
>>> print(first_b.previous_element == second_b.previous_element)
False
```

However, to check if the two variables refer to the same objects, you can use the following:

```
>>> print(first_b is second_b)
False
```

Copying Beautiful Soup objects

To create a copy of any tag or NavigableString, use copy.copy() function, just like below:

```
>>> import copy
>>> p_copy = copy.copy(soup.p)
>>> print(p_copy)
<p>Learn Python and  <b>Java</b> and advanced <b>Java</b>! from
Tutorialspoint</p>
>>>
```

Although the two copies (original and copied one) contain the same markup however, the two do not represent the same object:

```
>>> print(soup.p == p_copy)
True
>>>
>>> print(soup.p is p_copy)
False
>>>
```

The only real difference is that the copy is completely detached from the original Beautiful Soup object tree, just as if extract() had been called on it.

```
>>> print(p_copy.parent)
None
```

Above behavior is due to two different tag objects which cannot occupy the same space at the same time.

10. Beautiful Soup — Parsing only section of a document

There are multiple situations where you want to extract specific types of information (only <a> tags) using Beautifulsoup4. The SoupStrainer class in Beautifulsoup allows you to parse only specific part of an incoming document.

One way is to create a SoupStrainer and pass it on to the Beautifulsoup4 constructor as the parse_only argument.

SoupStrainer

A SoupStrainer tells BeautifulSoup what parts extract, and the parse tree consists of only these elements. If you narrow down your required information to a specific portion of the HTML, this will speed up your search result.

```
product = SoupStrainer('div',{'id': 'products_list'})
soup = BeautifulSoup(html,parse_only=product)
```

Above lines of code will parse only the titles from a product site, which might be inside a tag field.

Similarly, like above we can use other soupStrainer objects, to parse specific information from an HTML tag. Below are some of the examples:

```
from bs4 import BeautifulSoup, SoupStrainer

#Only "a" tags
only_a_tags = SoupStrainer("a")

#Will parse only the below mentioned "ids".
parse_only = SoupStrainer(id=["first", "third", "my_unique_id"])
soup = BeautifulSoup(my_document, "html.parser", parse_only=parse_only)

#parse only where string length is less than 10
def is_short_string(string):
     return len(string) < 10

only_short_strings = SoupStrainer(string=is_short_string)
```

11. Beautiful Soup — Trouble Shooting

Error Handling

There are two main kinds of errors that need to be handled in BeautifulSoup. These two errors are not from your script but from the structure of the snippet because the BeautifulSoup API throws an error.

The two main errors are as follows:

AttributeError

It is caused when the dot notation doesn't find a sibling tag to the current HTML tag. For example, you may have encountered this error, because of missing "anchor tag", cost-key will throw an error as it traverses and requires an anchor tag.

KeyError

This error occurs if the required HTML tag attribute is missing. For example, if we don't have data-pid attribute in a snippet, the pid key will throw key-error.

To avoid the above two listed errors when parsing a result, that result will be bypassed to make sure that a malformed snippet isn't inserted into the databases:

```
except(AttributeError, KeyError) as er:
     pass
```

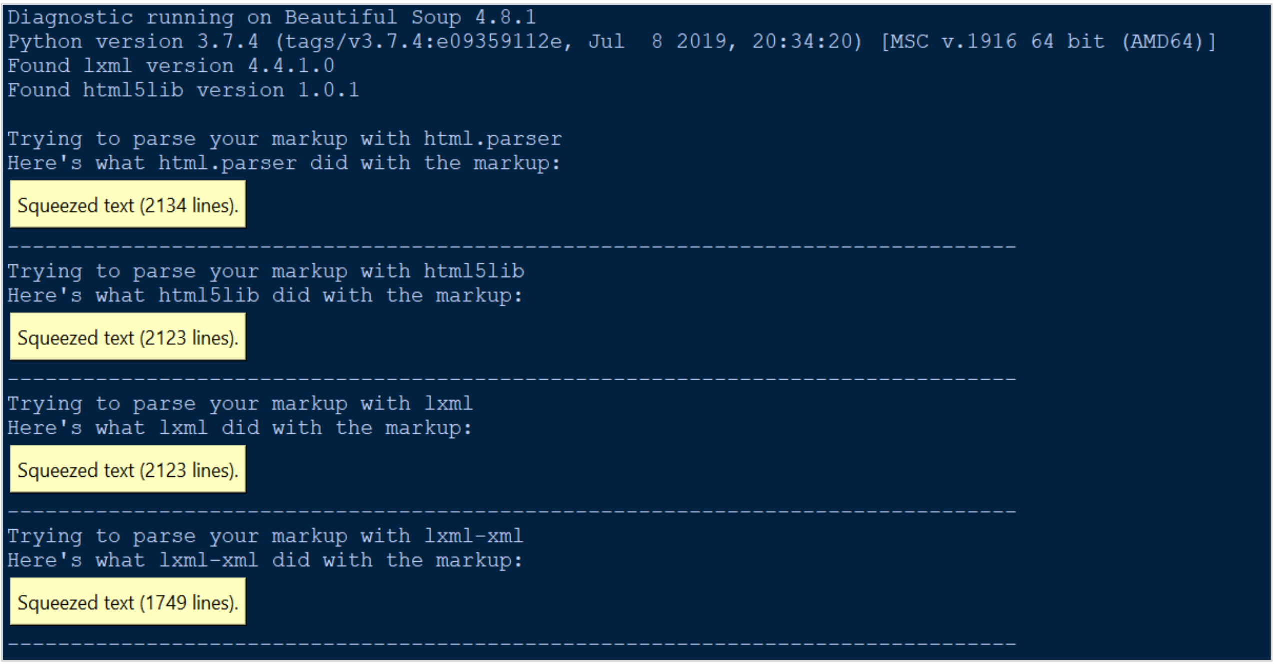
diagnose()

Whenever we find any difficulty in understanding what BeautifulSoup does to our document or HTML, simply pass it to the diagnose() function. On passing document file to the diagnose() function, we can show how list of different parser handles the document.

Below is one example to demonstrate the use of diagnose() function:

```
from bs4.diagnose import diagnose


with open("20 Books.html",encoding="utf8") as fp:
    data = fp.read()


diagnose(data)
```

Output



```
Diagnostic running on Beautiful Soup 4.8.1
Python version 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)]
Found lxml version 4.4.1.0
Found html5lib version 1.0.1

Trying to parse your markup with html.parser
Here's what html.parser did with the markup:
Squeezed text (2134 lines).
--------------------------------------------------------------------------------
Trying to parse your markup with html5lib
Here's what html5lib did with the markup:
Squeezed text (2123 lines).
--------------------------------------------------------------------------------
Trying to parse your markup with lxml
Here's what lxml did with the markup:
Squeezed text (2123 lines).
--------------------------------------------------------------------------------
Trying to parse your markup with lxml-xml
Here's what lxml-xml did with the markup:
Squeezed text (1749 lines).
--------------------------------------------------------------------------------
```

Parsing error

There are two main types of parsing errors. You might get an exception like HTMLParseError, when you feed your document to BeautifulSoup. You may also get an unexpected result, where the BeautifulSoup parse tree looks a lot different from the expected result from the parse document.

None of the parsing error is caused due to BeautifulSoup. It is because of external parser we use (html5lib, lxml) since BeautifulSoup doesn't contain any parser code. One way to resolve above parsing error is to use another parser.

```
from HTMLParser import HTMLParser


try:

    from HTMLParser import HTMLParseError

except ImportError, e:

    # From python 3.5, HTMLParseError is removed. Since it can never be

    # thrown in 3.5, we can just define our own class as a placeholder.

    class HTMLParseError(Exception):

        pass
```

Python built-in HTML parser causes two most common parse errors, HTMLParser.HTMLParserError: malformed start tag and HTMLParser.HTMLParserError: bad end tag and to resolve this, is to use another parser mainly: lxml or html5lib.

Another common type of unexpected behavior is that you can't find a tag that you know is in the document. However, when you run the find_all() returns [] or find() returns None.

This may be due to python built-in HTML parser sometimes skips tags it doesn’t understand.

XML parser Error

By default, BeautifulSoup package parses the documents as HTML, however, it is very easy-to-use and handle ill-formed XML in a very elegant manner using beautifulsoup4.

To parse the document as XML, you need to have lxml parser and you just need to pass the “xml” as the second argument to the Beautifulsoup constructor:

```
soup = BeautifulSoup(markup, "lxml-xml")
```

or

```
soup = BeautifulSoup(markup, "xml")
```

One common XML parsing error is:

```
AttributeError: 'NoneType' object has no attribute 'attrib'
```

This might happen in case, some element is missing or not defined while using find() or findall() function.

Other parsing errors

Given below are some of the other parsing errors we are going to discuss in this section:

Environmental issue

Apart from the above mentioned parsing errors, you may encounter other parsing issues such as environmental issues where your script might work in one operating system but not in another operating system or may work in one virtual environment but not in another virtual environment or may not work outside the virtual environment. All these issues may be because the two environments have different parser libraries available.

It is recommended to know or check your default parser in your current working environment. You can check the current default parser available for the current working environment or else pass explicitly the required parser library as second arguments to the BeautifulSoup constructor.

Case-insensitive

As the HTML tags and attributes are case-insensitive, all three HTML parsers convert tag and attribute names to lowercase. However, if you want to preserve mixed-case or uppercase tags and attributes, then it is better to parse the document as XML.

UnicodeEncodeError

Let us look into below code segment:

```
soup = BeautifulSoup(response, "html.parser")
                 print (soup)
```

Output

```
UnicodeEncodeError: 'charmap' codec can't encode character '\u011f'
```

Above problem may be because of two main situations. You might be trying to print out a unicode character that your console doesn’t know how to display. Second, you are trying to write to a file and you pass in a Unicode character that’s not supported by your default encoding.

One way to resolve above problem is to encode the response text/character before making the soup to get the desired result, as follows:

```
responseTxt = response.text.encode('UTF-8')
```

KeyError: [attr]

It is caused by accessing tag[‘attr’] when the tag in question doesn’t define the attr attribute. Most common errors are: “KeyError: ‘href’” and “KeyError: ‘class’”. Use tag.get(‘attr’) if you are not sure attr is defined.

```
for item in soup.fetch('a'):

        try:

            if (item['href'].startswith('/') or "tutorialspoint" in
item['href']):

            (...)

        except KeyError:

            pass # or some other fallback action
```

AttributeError

You may encounter AttributeError as follows:

```
AttributeError: 'list' object has no attribute 'find_all'
```

The above error mainly occurs because you expected find_all() return a single tag or string. However, soup.find_all returns a python list of elements.

All you need to do is to iterate through the list and catch data from those elements.