FREE eBook

LEARNING beautifulsoup

Free unaffiliated eBook created from
Stack Overflow contributors.

#beautifulsoup

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: beautifulsoup

It is an unofficial and free beautifulsoup ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official beautifulsoup.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with beautifulsoup

Remarks

In this section, we discuss what Beautiful Soup is, what it is used for and a brief outline on how to go about using it.

Beautiful Soup is a Python library that uses your pre-installed html/xml parser and converts the web page/html/xml into a tree consisting of tags, elements, attributes and values. To be more exact, the tree consists of four types of objects, Tag, NavigableString, BeautifulSoup and Comment. This tree can then be "queried" using the methods/properties of the BeautifulSoup object that is created from the parser library.

Your need : Often, you may have one of the following needs :

1. You might want to parse a web page to determine, how many of what tags are found, how many elements of each tag are found and their values. You might want to change them.
2. You might want to determine element names and values, so that you can use them in conjunction with other libraries for web page automation, such as Selenium.
3. You might want to transfer/extract data shown in a web page to other formats, such as a CSV file or to a relational database such as SQLite or mysql. In this case, the library helps you with the first step, of understanding the structure of the web page, although you will be using other libraries to do the act of transfer.
4. You might want to find out how many elements are styled with a certain CSS style and which ones.

Sequence for typical basic use in your Python code:

1. Import the Beautiful Soup library
2. Open a web page or html-text with the BeautifulSoup library, by mentioning which parser to be used. The result of this step is a BeautifulSoup object. (Note: This parser name mentioned, must be installed already as part of your Python pacakges. For instance, `html.parser`, is an in-built, 'with-batteries' package shipped with Python. You could install other parsers such as `lxml` or `html5lib`.)
3. "Query" or search the BeautifulSoup object using the syntax `'object.method'` and obtain the result into a collection, such as a Python dictionary. For some methods, the output will be a simple value.
4. Use the result from the previous step to do whatever you want to do with it, in rest of your Python code. You can also modify the element values or attribute values in the tree object. Modifications don't affect the source of the html code, but you can call output formatting methods (such as `prettify`) to create new output from the BeautifulSoup object.

Commonly used methods: Typically, the `.find` and `.find_all` methods are used to search the tree, giving the input arguments.

The input arguments are : the tag name that is being sought, attribute names and other related arguments. These arguments could be presented as : a string, a regular expression, a list or even a function.

Common uses of the BeautifulSoup object include :

1. Search by CSS class
2. Search by Hyperlink address
3. Search by Element Id, tag
4. Search by Attribute name. Attribute value.

If you have a need to filter the tree with a combination of the above criteria, you could also write a function that evaluates to true or false, and search by that function.

Versions

Version	Remarks	Package name	Release date
3.x	Version 3.2.1; Python 2 only	beautifulsoup	2012-02-16
4.x	Version 4.5.0; Python 2 and 3	beautifulsoup4	2016-07-20

Examples

Installation or Setup

pip may be used to install BeautifulSoup. To install Version 4 of BeautifulSoup, run the command:

```
pip install beautifulsoup4
```

Be aware that the package name is `beautifulsoup4` instead of `beautifulsoup`, the latter name stands for old release, see old beautifulsoup

A BeautifulSoup "Hello World" scraping example

```
from bs4 import BeautifulSoup
import requests

main_url = "https://fr.wikipedia.org/wiki/Hello_world"
req = requests.get(main_url)
soup = BeautifulSoup(req.text, "html.parser")

# Finding the main title tag.
title = soup.find("h1", class_ = "firstHeading")
print title.get_text()
```

```
# Finding the mid-titles tags and storing them in a list.
mid_titles = [tag.get_text() for tag in soup.find_all("span", class_ = "mw-headline")]

# Now using css selectors to retrieve the article shortcut links
links_tags = soup.select("li.toclevel-1")
for tag in links_tags:
    print tag.a.get("href")

# Retrieving the side page links by "blocks" and storing them in a dictionary
side_page_blocks = soup.find("div",
                            id = "mw-panel").find_all("div",
                                                      class_ = "portal")
blocks_links = {}
for num, block in enumerate(side_page_blocks):
    blocks_links[num] = [link.get("href") for link in block.find_all("a", href = True)]

print blocks_links[0]
```

Output:

```
"Hello, World!" program
#Purpose
#History
#Variations
#See_also
#References
#External_links
[u'/wiki/Main_Page', u'/wiki/Portal:Contents', u'/wiki/Portal:Featured_content',
u'/wiki/Portal:Current_events', u'/wiki/Special:Random',
u'https://donate.wikimedia.org/wiki/Special:FundraiserRedirector?utm_source=donate&utm_medium=sidebar&u
u'//shop.wikimedia.org']
```

Entering your prefered parser when instanciating Beautiful Soup avoids the usual `Warning` declaring that `no parser was explicitely specified`.

Different methods can be used to find an element within the webpage tree.

Although a handful of other methods exist, `css classes` and `css selectors` are two handy ways to find elements in the tree.

It should be noted that we can look for tags by setting their attribute value to True when searching them.

`get_text()` allows us to retrieve text contained within a tag. It returns it as a single Unicode string. `tag.get("attribute")` allows to get a tag's attribute value.

Read Getting started with beautifulsoup online: https://riptutorial.com/beautifulsoup/topic/1817/getting-started-with-beautifulsoup

Chapter 2: Locating elements

Examples

Locate a text after an element in BeautifulSoup

Imagine you have the following HTML:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

And you need to locate the text "John Smith" after the `label` element.

In this case, you can locate the `label` element by text and then use `.next_sibling` property:

```
from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

Prints `John Smith`.

Using CSS selectors to locate elements in BeautifulSoup

BeautifulSoup has a limited support for CSS selectors, but covers most commonly used ones. Use `select()` method to find multiple elements and `select_one()` to find a single element.

Basic example:

```
from bs4 import BeautifulSoup

data = """
<ul>
    <li class="item">item1</li>
    <li class="item">item2</li>
    <li class="item">item3</li>
</ul>
"""

soup = BeautifulSoup(data, "html.parser")
```

```
for item in soup.select("li.item"):
    print(item.get_text())
```

Prints:

```
item1
item2
item3
```

Locating comments

To locate comments in `BeautifulSoup`, use the `text` (or `string` in the recent versions) argument checking the type to be `Comment`:

```
from bs4 import BeautifulSoup
from bs4 import Comment

data = """
<html>
    <body>
        <div>
        <!-- desired text -->
        </div>
    </body>
</html>
"""

soup = BeautifulSoup(data, "html.parser")
comment = soup.find(text=lambda text: isinstance(text, Comment))
print(comment)
```

Prints `desired text`.

Filter functions

BeautifulSoup allows you to filter results by providing a function to `find_all` and similar functions. This can be useful for complex filters as well as a tool for code reuse.

Basic usage

Define a function that takes an element as its only argument. The function should return `True` if the argument matches.

```
def has_href(tag):
    '''Returns True for tags with a href attribute'''
    return  bool(tag.get("href"))

soup.find_all(has_href) #find all elements with a href attribute
#equivilent using lambda:
soup.find_all(lambda tag: bool(tag.get("href")))
```

Another example that finds tags with a `href` value that do not start with

Providing additional arguments to filter functions

Since the function passed to `find_all` can only take one argument, it's sometimes useful to make 'function factories' that produce functions fit for use in `find_all`. This is useful for making your tag-finding functions more flexible.

```
def present_in_href(check_string):
    return lambda tag: tag.get("href") and check_string in tag.get("href")

soup.find_all(present_in_href("/partial/path"))
```

Accessing internal tags and their attributes of initially selected tag

Let's assume you got an `html` after selecting with `soup.find('div', class_='base class')`:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(SomePage, 'lxml')
html = soup.find('div', class_='base class')
print(html)

<div class="base class">
  <div>Sample text 1</div>
  <div>Sample text 2</div>
  <div>
    <a class="ordinary link" href="https://example.com">URL text</a>
  </div>
</div>

<div class="Confusing class"></div>
'''
```

And if you want to access `<a>` tag's `href`, you can do it this way:

```
a_tag = html.a
link = a_tag['href']
print(link)

https://example.com
```

This is useful when you can't directly select `<a>` tag because it's `attrs` don't give you unique identification, there are other "twin" `<a>` tags in parsed page. But you can uniquely select a parent tag which contains needed `<a>`.

Collecting optional elements and/or their attributes from series of pages

Let's consider situation when you parse number of pages and you want to collect value from element that's *optional* (can be presented on one page and can be absent on another) for a paticular page.

Moreover the element itself, for example, is the *most ordinary element* on page, in other words no specific attributes can uniquely locate it. But you see that you can properly select its parent element and you know wanted element's *order number* in the respective nesting level.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(SomePage, 'lxml')
html = soup.find('div', class_='base class') # Below it refers to html_1 and html_2
```

Wanted element is optional, so there could be 2 situations for `html` to be:

```
html_1 = '''
<div class="base class">    # №0
  <div>Sample text 1</div>  # №1
  <div>Sample text 2</div>  # №2
  <div>!Needed text!</div>  # №3
</div>

<div>Confusing div text</div>  # №4
'''

html_2 = '''
<div class="base class">    # №0
  <div>Sample text 1</div>  # №1
  <div>Sample text 2</div>  # №2
</div>

<div>Confusing div text</div>  # №4
'''
```

If you got `html_1` you can collect `!Needed text!` from tag №3 this way:

```
wanted tag = html_1.div.find_next_sibling().find_next_sibling() # this gives you whole tag №3
```

It initially gets №1 `div`, then 2 times switches to next `div` on same nesting level to get to №3.

```
wanted_text = wanted_tag.text # extracting !Needed text!
```

Usefulness of this approach comes when you get `html_2` - approach won't give you error, it will give `None`:

```
print(html_2.div.find_next_sibling().find_next_sibling())
None
```

Using `find_next_sibling()` here is crucial because it limits element search by respective nesting level. If you'd use `find_next()` then tag №4 will be collected and you don't want it:

```
print(html_2.div.find_next().find_next())
<div>Confusing div text</div>
```

You also can explore `find_previous_sibling()` and `find_previous()` which work straight opposite way.

All described functions have their *miltiple* variants to catch all tags, not just the first one:

```
find_next_siblings()
find_previous_siblings()
find_all_next()
find_all_previous()
```

Read Locating elements online: https://riptutorial.com/beautifulsoup/topic/1940/locating-elements

Credits

S. No	Chapters	Contributors
1	Getting started with beautifulsoup	Antti Haapala, Community, Dair, DMPierre, Larry Cai, vikingben, Whirl Mind
2	Locating elements	alecxe, Dmitriy Fialkovskiy, sytech