Python

How to Parse XML Files Using Python’s BeautifulSoup

2 years ago • by Habeeb Kenny Shopeju

Data is literally everywhere, in all kinds of documents. But not all of it is useful, hence the need to parse it to get the parts that are needed. XML documents are one of such documents that hold data. They are very similar to HTML files, as they have almost the same kind of structure. Hence, you’ll need to parse them to get vital information, just as you would when working with HTML.

There are two major aspects to parsing XML files. They are:

- Finding Tags
- Extracting from Tags

You'll need to find the tag that holds the information you want, then extract that information. You'll learn how to do both when working with XML files before the end of this article.

Installation

BeautifulSoup is one of the most used libraries when it comes to web scraping with Python. Since XML files are similar to HTML files, it is also capable of parsing them. To parse XML files using BeautifulSoup though, it's best that you make use of Python's **lxml** parser.

You can install both libraries using the **pip** installation tool, through the command below:

```
pip install bs4 lxml
```

To confirm that both libraries are successfully installed, you can activate the interactive shell and try importing both. If no error pops up, then you are ready to go with the rest of the article.

Here's an example:

```
$python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20)
[MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import bs4
>>> import lxml
>>>
```

Before moving on, you should create an XML file from the code snippet below. It's quite simple, and should suit the use cases you'll learn about in the rest of the article. Simply copy, paste in your editor and save; a name like **sample.xml** should suffice.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root testAttr="testValue">
The Tree
<children>
<child name="Jack">First</child>
<child name="Rose">Second</child>
<child name="Blue Ivy">
Third
<grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>
</child>
<child name="Jane">Fourth</child>
</children>
</root>
```

Now, in your Python script; you'll need to read the XML file like a normal file, then pass it into BeautifulSoup. The remainder of this article will make use of the **bs_content** variable, so it's important that you take this step.

```
# Import BeautifulSoup
from bs4 import BeautifulSoup as bs
content = []
# Read the XML file
with open("sample.xml", "r") as file:
    # Read each line in the file, readlines() returns a list of lines
    content = file.readlines()
    # Combine the lines in the list into a string
    content = "".join(content)
    bs content = bs(content, "lxml")
```

The code sample above imports **BeautifulSoup**, then it reads the XML file like a regular file. After that, it passes the content into the imported **BeautifulSoup** library as well as the parser of choice.

You'll notice that the code doesn't import **lxml**. It doesn't have to as **BeautifulSoup** will choose the **lxml** parser as a result of passing **"lxml"** into the object.

Now, you can proceed with the rest of the article.

Finding Tags

One of the most important stages of parsing XML files is searching for tags. There are various ways to go about this when using BeautifulSoup; so you need to know about a handful of them to have the best tools for the appropriate situation.

You can find tags in XML documents by:

- Names
- Relationships

There are two BeautifulSoup methods you can use when finding tags by names. However, the use cases differ; let's take a look at them.

find

From personal experience, you'll use the **find** method more often than the other methods for finding tags in this article. The find tag receives the name of the tag you want to get, and returns a BeautifulSoup object of the tag if it finds one; else, it returns **None**.

Here's an example:

```
>>> result = bs_content.find("data")
>>> print(result)
<data>One</data>
>>> result = bs_content.find("unique")
>>> print(result)
<unique>Twins</unique>
>>> result = bs_content.find("father")
>>> print(result)
None
>>> result = bs_content.find("mother")
>>> print(result)
None
```

If you take a look at the example, you'll see that the **find** method returns a tag if it matches the name, else it returns None. However, if you take a closer look at it, you'll see it only returns a single tag.

For example, when **find("data")** was called, it only returned the first data tag, but didn't return the other ones.

GOTCHA: The **find** method will only return the first tag that matches its query.

So how do you get to find other tags too? That leads us to the next method.

find_all

The **find_all** method is quite similar to the **find** method. The only difference is that it returns a list of tags that match its query. When it doesn't find any tag, it simply returns an empty list. Hence, **find_all** will always return a list.

Here's an example:

```
>>> result = bs_content.find_all("data")
>>> print(result)
[<data>One</data>, <data>Two</data>]
>>> result = bs_content.find_all("child")
>>> print(result)
[<child>First</child>, <child>Second</child>, <child>
Third
<grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>
</child>, <child>Fourth</child>]
>>> result = bs_content.find_all("father")
>>> print(result
[]
>>> result = bs_content.find_all("mother")
>>> print(result)
[]
```

Now that you know how to use the **find** and **find_all** methods, you can search for tags anywhere in the XML document. However, you can make your searches more powerful.

Here's how:

Some tags may have the same name, but different attributes. For example, the **child** tags have a **name** attribute and different values. You can make specific searches based on those.

Have a look at this:

```
>>> result = bs_content.find("child", {"name": "Jack"})
>>> print(result)
<child name="Jack">First</child>
>>> result = bs_content.find_all("child", {"name": "Jack"})
>>> print(result)
[<child name="Jack">First</child>]
```

You'll see that there is something different about the use of the **find** and **find_all** methods here: they both have a second parameter.

When you pass in a dictionary as a second parameter, the **find** and **find_all** methods further their search to get tags that have attributes and values that fit the provided key:value pair.

For example, despite using the **find** method in the first example, it returned the second **child** tag (instead of the first **child** tag), because that's the first tag that matches the query. The **find_all** tag follows the same principle, except that it returns all the tags that match the query, not just the first.

Finding Tags By Relationships

While less popular than searching by tag names, you can also search for tags by relationships. In the real sense though, it's more of navigating than searching.

There are three key relationships in XML documents:

- **Parent**: The tag in which the reference tag exists.
- **Children**: The tags that exist in the reference tag.
- **Siblings**: The tags that exist on the same level as the reference tag.

From the explanation above, you may infer that the reference tag is the most important factor in searching for tags by relationships. Hence, let's look for the reference tag, and continue the article.

Take a look at this:

```
>>> third_child = bs_content.find("child", {"name": "Blue Ivy"})
>>> print(third_child)
<child name="Blue Ivy">
Third
<grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>
</child>
```

From the code sample above, the reference tag for the rest of this section will be the third **child** tag, stored in a **third_child** variable. In the subsections below, you'll see how to search for tags based on their parent, sibling, and children relationship with the reference tag.

Finding Parents

To find the parent tag of a reference tag, you'll make use of the **parent** attribute. Doing this returns the parent tag, as well as the tags under it. This behaviour is quite understandable, since the children tags are part of the parent tag.

Here's an example:

```
>>> result = third_child.parent
>>> print(result)

<children>
<child name="Jack">First</child>
<child name="Rose">Second</child>
<child name="Blue Ivy">
Third
<grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>
</child>
```

To find the children tags of a reference tag, you'll make use of the **children** attribute. Doing this returns the children tags, as well as the sub-tags under each one of them. This behaviour is also understandable, as the children tags often have their own children tags too.

One thing you should note is that the **children** attribute returns the children tags as a generator. So if you need a list of the children tags, you'll have to convert the generator to a list.

Here's an example:

```
>>> result = list(third_child.children)
>>> print(result)

['\n        Third\n     ', <grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>, '\n']
```

If you take a closer look at the example above, you'll notice that some values in the list are not tags. That's something you need to watch out for.

GOTCHA: The **children** attribute doesn't only return the children tags, it also returns the text in the reference tag.

Finding Siblings

The last in this section is finding tags that are siblings to the reference tag. For every reference tag, there may be sibling tags before and after it. The **previous_siblings** attribute will return the sibling tags before the reference tag, and the **next_siblings** attribute will return the sibling tags after it.

Just like the **children** attribute, the **previous_siblings** and **next_siblings** attributes will return generators. So you need to convert to a list if you need a list of siblings.

```
>>> previous_siblings = list(third_child.previous_siblings)
>>> print(previous_siblings)

['\n', <child name="Rose">Second</child>, '\n',
<child name="Jack">First</child>, '\n']

>>> next_siblings = list(third_child.next_siblings)
>>> print(next_siblings)

['\n', <child name="Jane">Fourth</child>]

>>> print(previous_siblings + next_siblings)

['\n', <child name="Rose">Second</child>, '\n', <child name="Jack">First</child>,
 '\n', '\n', <child name="Jane">Fourth</child>, '\n']
```

The first example shows the previous siblings, the second shows the next siblings; then both results are combined to generate a list of all the siblings for the reference tag.

Extracting From Tags

When parsing XML documents, a lot of the work lies in finding the right tags. However, when you find them, you may also want to extract certain information from those tags, and that's what this section will teach you.

You'll see how to extract the following:

- Tag Attribute Values
- Tag Text
- Tag Content

Extracting Tag Attribute Values

Sometimes, you may have a reason to extract the values for attributes in a tag. In the following attribute-value pairing for example: **name="Rose"**, you may want to extract "Rose."

To do this, you can make use of the **get** method, or accessing the attribute's name using **[]** like an index, just as you would when working with a dictionary.

Here's an example:

```
>>> result = third_child.get("name")
>>> print(result)

Blue Ivy

>>> result = third_child["name"]
>>> print(result)

Blue Ivy
```

Extracting Tag Text

When you want to access the text values of a tag, you can use the **text** or **strings** attribute. Both will return the text in a tag, and even the children tags. However, the **text** attribute will return them as a single string, concatenated; while the **strings** attribute will return them as a generator which you can convert to a list.

Here's an example:

```
>>> result = third_child.text
>>> print(result)

'\n    Third\n      \nOne\nTwo\nTwins\n\n'

>>> result = list(third_child.strings)
>>> print(result)

['\n  Third\n      ', '\n', 'One', '\n', 'Two', '\n', 'Twins', '\n', '\n']
```

Extracting Tag Content

Asides extracting the attribute values, and tag text, you can also extract all of a tags content. To do this, you can use the **contents** attribute; it is a bit similar to the **children** attribute and will yield the same results. However, while the **children** attribute returns a generator, the **contents** attribute returns a list.

Here's an example:

```
['\n        Third\n     ', <grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>, '\n']
```

Printing Beautiful

So far, you’ve seen some important methods and attributes that are useful when parsing XML documents using BeautifulSoup. But if you notice, when you print the tags to the screen, they have some kind of clustered look. While appearance may not have a direct impact on your productivity, it can help you parse more effectively and make the work less tedious.

Here’s an example of printing the normal way:

```
>>> print(third_child)

<child name="Blue Ivy">
Third
<grandchildren>
<data>One</data>
<data>Two</data>
<unique>Twins</unique>
</grandchildren>
</child>
```

However, you can improve its appearance by using the **prettify** method. Simply call the **prettify** method on the tag while printing, and you’ll get something visually pleasing.

Take a look at this:

```
>>> print(third_child.prettify())

<child name="Blue Ivy">
 Third
 <grandchildren>
  <data>
   One
  </data>
  <data>
   Two
  </data>
  <unique>
   Twins
  </unique>
 </grandchildren>
</child>
```

Conclusion

Parsing documents is an important aspect of sourcing for data. XML documents are pretty popular, and hopefully you are better equipped to take them on, and extract the data you want.

From this article, you are now able to:

- search for tags either by names, or relationships
- extract data from tags

If you feel quite lost, and are pretty new to the BeautifulSoup library, you can check out the BeautifulSoup tutorial for beginners.

#Beautiful Soup #Web Scraping

Habeeb Kenny Shopeju

I love building software, very proficient with Python and JavaScript. I'm very comfortable with the linux terminal and interested in machine learning. In my spare time, I write prose, poetry and tech articles.

View all posts

RELATED LINUX HINT POSTS

- Python csv skip header row
- Remove Special Characters from String Python
- How to Use Python Csv Dictreader
- How to Append a New Row to CSV Python
- Python Object to String
- Convert a String to JSON Python
- How Do You Repeat a String n Times in Python?