Beautiful Soup 4.2.0 文檔

Beautiful Soup 是一個可以從 HTML 或 XML 檔中提取資料的 Python 庫.它能夠通過你喜歡的轉換器實現慣用的文檔導航,查找,修改文檔的方式.Beautiful Soup 會幫你節省數小時甚至數天的工作時間.

這篇文檔介紹了 BeautifulSoup4 中所有主要特性,並切有小例子.讓我來向你展示它適合做什麼,如何工作,怎樣使用,如何達到你想要的效果,和處理異常情況.

文檔中出現的例子在 Python2.7 和 Python3.2 中的執行結果相同

你可能在尋找 Beautiful Soup3 的文檔,Beautiful Soup 3 目前已經停止開發,我們推薦在現在的項目中使用 Beautiful Soup 4, 移植到 BS4

尋求幫助

如果你有關於 BeautifulSoup 的問題,可以發送郵件到 討論群組 .如果你的問題包含了一段需要轉換的 HTML 代碼,那麼確保你提的問題描述中附帶這段 HTML 文檔的 代碼診斷 [1]

快速開始

下面的一段 HTML 代碼將作為例子被多次用到.這是 *愛麗絲夢遊仙境的* 的一段內容(以後內容中簡稱為 *愛麗絲* 的文檔):

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and
their names were
<a href="http://example.com/elsie" class="sister"
id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
and
<a href="http://example.com/tillie" class="sister"
id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
```

使用 BeautifulSoup 解析這段代碼,能夠得到一個 BeautifulSoup 的物件,並能按照標準的縮進格式的結構輸出:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc)

print(soup.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#   <p class="title">
#    <b>
#     The Dormouse's story
#    </b>
#   </p>
#   <p class="story">
#    Once upon a time there were three little sisters; and their names
were
#    <a class="sister" href="http://example.com/elsie" id="link1">
#     Elsie
#    </a>
#    ,
#    <a class="sister" href="http://example.com/lacie" id="link2">
#     Lacie
#    </a>
#    and
#    <a class="sister" href="http://example.com/tillie" id="link2">
#     Tillie
#    </a>
#    ; and they lived at the bottom of a well.
#   </p>
#   <p class="story">
#    ...
#   </p>
#  </body>
# </html>
```

幾個簡單的流覽結構化資料的方法:

```
soup.title
# <title>The Dormouse's story</title>

soup.title.name
# u'title'

soup.title.string
# u'The Dormouse's story'

soup.title.parent.name
# u'head'

soup.p
# <p class="title"><b>The Dormouse's story</b></p>

soup.p['class']
# u'title'

soup.a
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>

soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.find(id="link3")
# <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>
```

從文檔中找到所有<a>標籤的連結:

```
for link in soup.find_all('a'):
    print(link.get('href'))
    # http://example.com/elsie
```

```
    # http://example.com/lacie
    # http://example.com/tillie
```

從文檔中獲取所有文字內容:

```
print(soup.get_text())
# The Dormouse's story
#
# The Dormouse's story
#
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
#
# ...
```

這是你想要的嗎?別著急,還有更好用的

安裝 Beautiful Soup

如果你用的是新版的 Debain 或 ubuntu,那麼可以通過系統的套裝軟體管理來安裝:

```
$ apt-get install Python-bs4
```

Beautiful Soup 4 通過PyPi 發佈,所以如果你無法使用系統包管理安裝,那麼也可以通過 easy_install 或pip 來安裝. 包的名字是beautifulsoup4 ,這個包相容 Python2 和 Python3.

```
$ easy_install beautifulsoup4
$ pip install beautifulsoup4
```

(在 PyPi 中還有一個名字是BeautifulSoup 的包,但那可能不是你想要的,那是 Beautiful Soup3 的發佈版本,因為很多專案還在使用 BS3, 所以 BeautifulSoup 包依然有效.但是如果你在編寫新項目,那麼你應該安裝的beautifulsoup4)

如果你沒有安裝 easy_install 或pip ,那你也可以 下載 BS4 的源碼 ,然後通過 setup.py 來安裝.

```
$ Python setup.py install
```

如果上述安裝方法都行不通,Beautiful Soup 的發佈協議允許你將 BS4 的代碼打包在你的項目中,這樣無須安裝即可使用.

作者在 Python2.7 和 Python3.2 的版本下開發 Beautiful Soup, 理論上 Beautiful Soup 應該在所有當前的 Python 版本中正常工作

安裝完成後的問題

Beautiful Soup 發佈時打包成 Python2 版本的代碼,在 Python3 環境下安裝時,會自動轉換成 Python3 的代碼,如果沒有一個安裝的過程,那麼代碼就不會被轉換.

如果代碼拋出了 ImportError 的異常: “No module named HTMLParser”, 這是因為你在 Python3 版本中執行 Python2 版本的代碼.

如果代碼拋出了 ImportError 的異常: “No module named html.parser”, 這是因為你在 Python2 版本中執行 Python3 版本的代碼.

如果遇到上述 2 種情況,最好的解決方法是重新安裝 BeautifulSoup4.

如果在 ROOT_TAG_NAME = u’[document]’代碼處遇到 SyntaxError “Invalid syntax”錯誤,需要將把 BS4 的 Python 代碼版本從 Python2 轉換到 Python3. 可以重新安裝 BS4:

```
$ Python3 setup.py install
```

或在 bs4 的目錄中執行 Python 代碼版本轉換腳本

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

安裝解析器

Beautiful Soup 支持 Python 標準庫中的 HTML 解析器,還支持一些協力廠商的解析器,其中一個是 lxml .根據作業系統不同,可以選擇下列方法來安裝 lxml:

```
$ apt-get install Python-lxml
$ easy_install lxml
$ pip install lxml
```

另一個可供選擇的解析器是純 Python 實現的 html5lib , html5lib 的解析方式與流覽器相同,可以選擇下列方法來安裝 html5lib:

```
$ apt-get install Python-html5lib
$ easy_install html5lib
$ pip install html5lib
```

下表列出了主要的解析器,以及它們的優缺點:

解析器	使用方法	優勢	劣勢
Python 標準庫	BeautifulSoup(markup, "html.parser")	Python 的內置標準庫 執行速度適中 文檔容錯能力強	Python 2.7.3 or 3.2.2)前 的版本中文檔容錯能力差
lxml HTML 解析器	BeautifulSoup(markup, "lxml")	速度快 文檔容錯能力強	需要安裝C語言庫
lxml XML 解析器	BeautifulSoup(markup, ["lxml", "xml"]) BeautifulSoup(markup, "xml")	速度快 唯一支持 XML 的解析器	需要安裝C語言庫
html5lib	BeautifulSoup(markup, "html5lib")	最好的容錯性	速度慢 不依賴外部

解析器	使用方法	優勢	劣勢
		以流覽器的方式解析文檔 生成 HTML5 格式的文檔	擴展

推薦使用 lxml 作為解析器,因為效率更高. 在 Python2.7.3 之前的版本和 Python3 中 3.2.2 之前的版本,必須安裝 lxml 或 html5lib, 因為那些 Python 版本的標準庫中內置的 HTML 解析方法不夠穩定.

提示: 如果一段 HTML 或 XML 文檔格式不正確的話,那麼在不同的解析器中返回的結果可能是不一樣的,查看 解析器之間的區別 瞭解更多細節

如何使用

將一段文檔傳入 BeautifulSoup 的構造方法,就能得到一個文檔的物件, 可以傳入一段字串或一個檔案控制代碼.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(open("index.html"))

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

首先,文檔被轉換成 Unicode,並且 HTML 的實例都被轉換成 Unicode 編碼

```
BeautifulSoup("Sacr&eacute; bleu!")
<html><head></head><body>Sacré bleu!</body></html>
```

然後,Beautiful Soup 選擇最合適的解析器來解析這段文檔,如果手動指定解析器那麼 Beautiful Soup 會選擇指定的解析器來解析文檔.(參考 解析成 XML).

對象的種類

Beautiful Soup 將複雜 HTML 文檔轉換成一個複雜的樹形結構,每個節點都是 Python 物件,所有物件可以歸納為 4 種: Tag , NavigableString , BeautifulSoup , Comment .

Tag

Tag 物件與 XML 或 HTML 原生文檔中的 tag 相同:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Tag 有很多方法和屬性,在 遍歷文檔樹 和 搜索文檔樹 中有詳細解釋.現在介紹一下 tag 中最重要的屬性: name 和 attributes

Name

每個 tag 都有自己的名字,通過 .name 來獲取:

```
tag.name
# u'b'
```

如果改變了 tag 的 name,那將影響所有通過當前 Beautiful Soup 物件生成的 HTML 文檔:

```
tag.name = "blockquote"
tag
# <blockquote class="boldest">Extremely bold</blockquote>
```

Attributes

一個 tag 可能有很多個屬性. tag <b class="boldest"> 有一個 “class” 的屬性,值為 “boldest” . tag 的屬性的操作方法與字典相同:

```
tag['class']
# u'boldest'
```

也可以直接”點”取屬性, 比如: .attrs :

```
tag.attrs
# {u'class': u'boldest'}
```

tag 的屬性可以被添加,刪除或修改. 再說一次, tag 的屬性操作方法與字典一樣

```
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>

tag['class']
# KeyError: 'class'
print(tag.get('class'))
# None
```

多值屬性

HTML 4 定義了一系列可以包含多個值的屬性.在 HTML5 中移除了一些,卻增加更多.最常見的多值的屬性是 class (一個 tag 可以有多個 CSS 的 class). 還有一些屬性 rel , rev , accept-charset , headers , accesskey . 在 Beautiful Soup 中多值屬性的返回類型是 list:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ["body", "strikeout"]

css_soup = BeautifulSoup('<p class="body"></p>')
```

```
css_soup.p['class']
# ["body"]
```

如果某個屬性看起來好像有多個值,但在任何版本的 HTML 定義中都沒有被定義為多值屬性,那麼 Beautiful Soup 會將這個屬性作為字串返回

```
id_soup = BeautifulSoup('<p id="my id"></p>')
id_soup.p['id']
# 'my id'
```

將 tag 轉換成字串時,多值屬性會合並為一個值

```
rel_soup = BeautifulSoup('<p>Back to the <a
rel="index">homepage</a></p>')
rel_soup.a['rel']
# ['index']
rel_soup.a['rel'] = ['index', 'contents']
print(rel_soup.p)
# <p>Back to the <a rel="index contents">homepage</a></p>
```

如果轉換的文檔是 XML 格式,那麼 tag 中不包含多值屬性

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

可以遍歷的字串

字串常被包含在 tag 內.Beautiful Soup 用 NavigableString 類來包裝 tag 中的字串:

```
tag.string
# u'Extremely bold'
type(tag.string)
# <class 'bs4.element.NavigableString'>
```

一個 NavigableString 字串與 Python 中的 Unicode 字串相同,並且還支援包含在 遍歷文檔樹 和 搜索文檔樹 中的一些特性. 通過 unicode() 方法可以直接將 NavigableString 物件轉換成 Unicode 字串:

```
unicode_string = unicode(tag.string)
unicode_string
# u'Extremely bold'
type(unicode_string)
# <type 'unicode'>
```

tag 中包含的字串不能編輯,但是可以被替換成其它的字串,用 replace_with() 方法:

```
tag.string.replace_with("No longer bold")
tag
# <blockquote>No longer bold</blockquote>
```

NavigableString 對象支援 遍歷文檔樹 和 搜索文檔樹 中定義的大部分屬性, 並非全部.尤其是,一個字串不能包含

其它內容(tag 能夠包含字串或是其它 tag),字串不支援 .contents 或 .string 屬性或 find() 方法.

如果想在 Beautiful Soup 之外使用 NavigableString 物件,需要調用 unicode() 方法,將該物件轉換成普通的 Unicode 字串,否則就算 Beautiful Soup 已方法已經執行結束,該物件的輸出也會帶有物件的引用位址.這樣會浪費記憶體.

BeautifulSoup

BeautifulSoup 物件表示的是一個文檔的全部內容.大部分時候,可以把它當作 Tag 物件,它支援 遍歷文檔樹 和 搜索文檔樹 中描述的大部分的方法.

因為 BeautifulSoup 物件並不是真正的 HTML 或 XML 的 tag,所以它沒有 name 和 attribute 屬性.但有時查看它的 .name 屬性是很方便的,所以 BeautifulSoup 物件包含了一個值為 “[document]” 的特殊屬性 .name

```
soup.name
# u'[document]'
```

注釋及特殊字串

Tag , NavigableString , BeautifulSoup 幾乎覆蓋了 html 和 xml 中的所有內容,但是還有一些特殊物件.容易讓人擔心的內容是文檔的注釋部分:

```
markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
soup = BeautifulSoup(markup)
comment = soup.b.string
type(comment)
# <class 'bs4.element.Comment'>
```

Comment 物件是一個特殊類型的 NavigableString 物件:

```
comment
# u'Hey, buddy. Want to buy a used parser'
```

但是當它出現在 HTML 文檔中時, Comment 物件會使用特殊的格式輸出:

```
print(soup.b.prettify())
# <b>
#  <!--Hey, buddy. Want to buy a used parser?-->
# </b>
```

Beautiful Soup 中定義的其它類型都可能會出現在 XML 的文檔中:CData , ProcessingInstruction , Declaration , Doctype .與 Comment 物件類似,這些類都是 NavigableString 的子類,只是添加了一些額外的方法的字串獨享.下面是用 CDATA 來替代注釋的例子:

```
from bs4 import CData
cdata = CData("A CDATA block")
comment.replace_with(cdata)

print(soup.b.prettify())
# <b>
#  <![CDATA[A CDATA block]]>
# </b>
```

遍歷文檔樹

還拿”愛麗絲夢遊仙境”的文檔來做例子:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and
their names were
<a href="http://example.com/elsie" class="sister"
id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
and
<a href="http://example.com/tillie" class="sister"
id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc)
```

通過這段例子來演示怎樣從文檔的一段內容找到另一段內容

子節點

一個 Tag 可能包含多個字串或其它的 Tag,這些都是這個 Tag 的子節點.Beautiful Soup 提供了許多操作和遍歷子節點的屬性.

注意: Beautiful Soup 中字串節點不支援這些屬性,因為字串沒有子節點

tag 的名字

操作文檔樹最簡單的方法就是告訴它你想獲取的 tag 的 name.如果想獲取 <head> 標籤,只要用 soup.head :

```
soup.head
# <head><title>The Dormouse's story</title></head>

soup.title
# <title>The Dormouse's story</title>
```

這是個獲取 tag 的小竅門,可以在文檔樹的 tag 中多次調用這個方法.下面的代碼可以獲取<body>標籤中的第一個<b>標籤:

```
soup.body.b
# <b>The Dormouse's story</b>
```

通過點取屬性的方式只能獲得當前名字的第一個 tag:

```
soup.a
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>
```

如果想要得到所有的<a>標籤,或是通過名字得到比一個 tag 更多的內容的時候,就需要用到 *Searching the tree* 中描述的方法,比如: find_all()

```
soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

.contents 和 .children

tag 的 .contents 屬性可以將 tag 的子節點以清單的方式輸出:

```
head_tag = soup.head
head_tag
# <head><title>The Dormouse's story</title></head>

head_tag.contents
[<title>The Dormouse's story</title>]

title_tag = head_tag.contents[0]
title_tag
# <title>The Dormouse's story</title>
title_tag.contents
# [u'The Dormouse's story']
```

BeautifulSoup 物件本身一定會包含子節點,也就是說<html>標籤也是 BeautifulSoup 物件的子節點:

```
len(soup.contents)
# 1
soup.contents[0].name
# u'html'
```

字串沒有 .contents 屬性,因為字串沒有子節點:

```
text = title_tag.contents[0]
text.contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

通過 tag 的 .children 生成器,可以對 tag 的子節點進行迴圈:

```
for child in title_tag.children:
    print(child)
    # The Dormouse's story
```

.descendants

.contents 和 .children 屬性僅包含 tag 的直接子節點.例如,<head>標籤只有一個直接子節點<title>

```
head_tag.contents
# [<title>The Dormouse's story</title>]
```

但是<title>標籤也包含一個子節點:字串 “The Dormouse’s story”,這種情況下字串 “The Dormouse’s story”也屬於<head>標籤的子孫節點. .descendants 屬性可以對所有 tag 的子孫節點進行遞迴迴圈 [5]:

```
for child in head_tag.descendants:
    print(child)
    # <title>The Dormouse's story</title>
    # The Dormouse's story
```

上面的例子中, <head>標籤只有一個子節點,但是有 2 個子孫節點:<head>節點和<head>的子節點, BeautifulSoup 有一個直接子節點(<html>節點),卻有很多子孫節點:

```
len(list(soup.children))
# 1
len(list(soup.descendants))
# 25
```

.string

如果 tag 只有一個 NavigableString 類型子節點,那麼這個 tag 可以使用 .string 得到子節點:

```
title_tag.string
# u'The Dormouse's story'
```

如果一個 tag 僅有一個子節點,那麼這個 tag 也可以使用 .string 方法,輸出結果與當前唯一子節點的 .string 結果相同:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

head_tag.string
# u'The Dormouse's story'
```

如果 tag 包含了多個子節點,tag 就無法確定 .string 方法應該調用哪個子節點的內容, .string 的輸出結果是 None :

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

.strings 和 stripped_strings

如果 tag 中包含多個字串 [2],可以使用 .strings 來迴圈獲取:

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

```
    # u"The Dormouse's story"
    # u'\n\n'
    # u"The Dormouse's story"
    # u'\n\n'
    # u'Once upon a time there were three little sisters; and their names
were\n'
    # u'Elsie'
    # u',\n'
    # u'Lacie'
    # u' and\n'
    # u'Tillie'
    # u';\nand they lived at the bottom of a well.'
    # u'\n\n'
    # u'...'
    # u'\n'
```

輸出的字串中可能包含了很多空格或空行,使用 .stripped_strings 可以去除多餘空白內容:

```
for string in soup.stripped_strings:
    print(repr(string))
    # u"The Dormouse's story"
    # u"The Dormouse's story"
    # u'Once upon a time there were three little sisters; and their names
were'
    # u'Elsie'
    # u','
    # u'Lacie'
    # u'and'
    # u'Tillie'
    # u';\nand they lived at the bottom of a well.'
    # u'...'
```

全部是空格的行會被忽略掉,段首和段末的空白會被刪除

父節點

繼續分析文檔樹,每個 tag 或字串都有父節點:被包含在某個 tag 中

.parent

通過 .parent 屬性來獲取某個元素的父節點.在例子“愛麗絲”的文檔中,<head>標籤是<title>標籤的父節點:

```
title_tag = soup.title
title_tag
# <title>The Dormouse's story</title>
```

```
title_tag.parent
# <head><title>The Dormouse's story</title></head>
```

文檔 title 的字串也有父節點:<title>標籤

```
title_tag.string.parent
# <title>The Dormouse's story</title>
```

文檔的頂層節點比如<html>的父節點是 BeautifulSoup 物件:

```
html_tag = soup.html
type(html_tag.parent)
# <class 'bs4.BeautifulSoup'>
```

BeautifulSoup 對象的 .parent 是 None:

```
print(soup.parent)
# None
```

.parents

通過元素的 .parents 屬性可以遞迴得到元素的所有父輩節點,下面的例子使用了 .parents 方法遍歷了<a>標籤到根節點的所有節點.

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>
for parent in link.parents:
    if parent is None:
        print(parent)
    else:
        print(parent.name)
# p
# body
# html
# [document]
# None
```

兄弟節點

看一段簡單的例子:

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></b></a>")
print(sibling_soup.prettify())
# <html>
#  <body>
#   <a>
#    <b>
```

```
#      text1
#    </b>
#    <c>
#      text2
#    </c>
#   </a>
#  </body>
# </html>
```

因為<b>標籤和<c>標籤是同一層:他們是同一個元素的子節點,所以<b>和<c>可以被稱為兄弟節點.一段文檔以標準格式輸出時,兄弟節點有相同的縮進級別.在代碼中也可以使用這種關係.

.next_sibling 和 .previous_sibling

在文檔樹中,使用 .next_sibling 和 .previous_sibling 屬性來查詢兄弟節點:

```
sibling_soup.b.next_sibling
# <c>text2</c>

sibling_soup.c.previous_sibling
# <b>text1</b>
```

<b>標籤有 .next_sibling 屬性,但是沒有 .previous_sibling 屬性,因為<b>標籤在同級節點中是第一個.同理,<c>標籤有 .previous_sibling 屬性,卻沒有 .next_sibling 屬性:

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

例子中的字串“text1”和“text2”不是兄弟節點,因為它們的父節點不同:

```
sibling_soup.b.string
# u'text1'

print(sibling_soup.b.string.next_sibling)
# None
```

實際文檔中的 tag 的 .next_sibling 和 .previous_sibling 屬性通常是字串或空白. 看看“愛麗絲”文檔:

```
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
<a href="http://example.com/tillie" class="sister"
id="link3">Tillie</a>
```

如果以為第一個<a>標籤的 .next_sibling 結果是第二個<a>標籤,那就錯了,真實結果是第一個<a>標籤和第二個<a>標籤之間的頓號和分行符號:

```
link = soup.a
link
```

```
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>

link.next_sibling
# u',\n'
```

第二個<a>標籤是頓號的 .next_sibling 屬性:

```
link.next_sibling.next_sibling
# <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>
```

.next_siblings 和 .previous_siblings

通過 .next_siblings 和 .previous_siblings 屬性可以對當前節點的兄弟節點反覆運算輸出:

```
for sibling in soup.a.next_siblings:
    print(repr(sibling))
    # u',\n'
    # <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>
    # u' and\n'
    # <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>
    # u'; and they lived at the bottom of a well.'
    # None

for sibling in soup.find(id="link3").previous_siblings:
    print(repr(sibling))
    # ' and\n'
    # <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>
    # u',\n'
    # <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>
    # u'Once upon a time there were three little sisters; and their names
were\n'
    # None
```

回退和前進

看一下“愛麗絲”文檔:

```
<html><head><title>The Dormouse's story</title></head>
<p class="title"><b>The Dormouse's story</b></p>
```

HTML 解析器把這段字串轉換成一連串的事件: “打開<html>標籤”,”打開一個<head>標籤”,”打開一個<title>標籤”,”添加一段字串”,”關閉<title>標籤”,”打開<p>標籤”,等等.Beautiful Soup 提供了重現解析器初始化過程的方法.

.next_element 和 .previous_element

.next_element 屬性指向解析過程中下一個被解析的物件(字串或 tag),結果可能與 .next_sibling 相同,但通常是不一樣的.

這是“愛麗絲”文檔中最後一個<a>標籤,它的 .next_sibling 結果是一個字串,因為當前的解析過程 [2] 因為當前的解析過程因為遇到了<a>標籤而中斷了:

```
last_a_tag = soup.find("a", id="link3")
last_a_tag
# <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>

last_a_tag.next_sibling
# '; and they lived at the bottom of a well.'
```

但這個<a>標籤的 .next_element 屬性結果是在<a>標籤被解析之後的解析內容,不是<a>標籤後的句子部分,應該是字串”Tillie”:

```
last_a_tag.next_element
# u'Tillie'
```

這是因為在原始文檔中,字串“Tillie” 在分號前出現,解析器先進入<a>標籤,然後是字串“Tillie”,然後關閉標籤,然後是分號和剩餘部分.分號與<a>標籤在同一層級,但是字串“Tillie”會被先解析.

.previous_element 屬性剛好與 .next_element 相反,它指向當前被解析的物件的前一個解析物件:

```
last_a_tag.previous_element
# u' and\n'
last_a_tag.previous_element.next_element
# <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>
```

.next_elements 和 .previous_elements

通過 .next_elements 和 .previous_elements 的反覆運算器就可以向前或向後訪問文檔的解析內容,就好像文檔正在被解析一樣:

```
for element in last_a_tag.next_elements:
    print(repr(element))
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...</p>
# u'...'
# u'\n'
# None
```

搜索文檔樹

Beautiful Soup 定義了很多搜索方法,這裡著重介紹 2 個: find() 和 find_all() .其它方法的參數和用法類似,請讀者舉一反三.

再以“愛麗絲”文檔作為例子:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and
their names were
<a href="http://example.com/elsie" class="sister"
id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
and
<a href="http://example.com/tillie" class="sister"
id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc)
```

使用 find_all() 類似的方法可以查找到想要查找的文檔內容

篩檢程式

介紹 find_all() 方法前,先介紹一下篩檢程式的類型 [3] ,這些篩檢程式貫穿整個搜索的 API.篩檢程式可以被用在 tag 的 name 中,節點的屬性中,字串中或他們的混合中.

字串

最簡單的篩檢程式是字串.在搜索方法中傳入一個字串參數,Beautiful Soup 會查找與字串完整匹配的內容,下面的例子用於查找文檔中所有的<b>標籤:

```
soup.find_all('b')
# [<b>The Dormouse's story</b>]
```

如果傳入位元組碼參數,Beautiful Soup 會當作 UTF-8 編碼,可以傳入一段 Unicode 編碼來避免 Beautiful Soup 解析編碼出錯

規則運算式

如果傳入規則運算式作為參數,Beautiful Soup 會通過規則運算式的 match() 來匹配內容.下面例子中找出所有以 b 開頭的標籤,這表示<body>和<b>標籤都應該被找到:

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

下面代碼找出所有名字中包含"t"的標籤:

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

列表

如果傳入列表參數,Beautiful Soup 會將與清單中任一元素匹配的內容返回.下面代碼找到文檔中所有<a>標籤和<b>標籤:

```
soup.find_all(["a", "b"])
# [<b>The Dormouse's story</b>,
#  <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

True

True 可以匹配任何值,下面代碼查找到所有的 tag,但是不會返回字串節點

```
for tag in soup.find_all(True):
    print(tag.name)
# html
# head
# title
# body
# p
# b
# p
# a
# a
# a
# p
```

方法

如果沒有合適篩檢程式,那麼還可以定義一個方法,方法只接受一個元素參數 [4],如果這個方法返回 True 表示當前元素匹配並且被找到,如果不是則反回 False

下面方法校驗了當前元素,如果包含 class 屬性卻不包含 id 屬性,那麼將返回 True:

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

將這個方法作為參數傳入 find_all() 方法,將得到所有<p>標籤:

```
soup.find_all(has_class_but_no_id)
# [<p class="title"><b>The Dormouse's story</b></p>,
#  <p class="story">Once upon a time there were...</p>,
#  <p class="story">...</p>]
```

返回結果中只有<p>標籤沒有<a>標籤,因為<a>標籤還定義了"id",沒有返回<html>和<head>,因為<html>和<head>中沒有定義"class"屬性.

下面代碼找到所有被文字包含的節點內容:

```
from bs4 import NavigableString
def surrounded_by_strings(tag):
    return (isinstance(tag.next_element, NavigableString)
            and isinstance(tag.previous_element, NavigableString))

for tag in soup.find_all(surrounded_by_strings):
    print tag.name
# p
# a
# a
# a
# p
```

現在來瞭解一下搜索方法的細節

find_all()

find_all(name , attrs , recursive , text , **kwargs)

find_all() 方法搜索當前 tag 的所有 tag 子節點,並判斷是否符合篩檢程式的條件.這裡有幾個例子:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]

soup.find_all("p", "title")
# [<p class="title"><b>The Dormouse's story</b></p>]

soup.find_all("a")
# [<a class="sister" href="http://example.com/elsie"
```

```
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.find_all(id="link2")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]

import re
soup.find(text=re.compile("sisters"))
# u'Once upon a time there were three little sisters; and their names
were\n'
```

有幾個方法很相似,還有幾個方法是新的,參數中的 text 和 id 是什麼含義? 為什麼 find_all("p", "title") 返回的是 CSS Class 為"title"的<p>標籤? 我們來仔細看一下 find_all() 的參數

name 參數

name 參數可以查找所有名字為 name 的 tag,字串物件會被自動忽略掉.

簡單的用法如下:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]
```

重申: 搜索 name 參數的值可以使任一類型的篩檢程式,字元竄,規則運算式,清單,方法或是 True .

keyword 參數

如果一個指定名字的參數不是搜索內置的參數名,搜索時會把該參數當作指定名字 tag 的屬性來搜索,如果包含一個名字為 id 的參數,Beautiful Soup 會搜索每個 tag 的"id"屬性.

```
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]
```

如果傳入 href 參數,Beautiful Soup 會搜索每個 tag 的"href"屬性:

```
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]
```

搜索指定名字的屬性時可以使用的參數值包括 字串 , 規則運算式 , 列表, True .

下面的例子在文檔樹中查找所有包含 id 屬性的 tag,無論 id 的值是什麼:

```
soup.find_all(id=True)
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
```

```
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

使用多個指定名字的參數可以同時過濾 tag 的多個屬性:

```
soup.find_all(href=re.compile("elsie"), id='link1')
# [<a class="sister" href="http://example.com/elsie"
id="link1">three</a>]
```

有些 tag 屬性在搜索不能使用,比如 HTML5 中的 data-* 屬性:

```
data_soup = BeautifulSoup('<div data-foo="value">foo!</div>')
data_soup.find_all(data-foo="value")
# SyntaxError: keyword can't be an expression
```

但是可以通過 find_all() 方法的 attrs 參數定義一個字典參數來搜索包含特殊屬性的 tag:

```
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

按 CSS 搜索

按照 CSS 類名搜索 tag 的功能非常實用,但標識 CSS 類名的關鍵字 class 在 Python 中是保留字,使用 class 做參數會導致語法錯誤.從 Beautiful Soup 的 4.1.1 版本開始,可以通過 class_ 參數搜索有指定 CSS 類名的 tag:

```
soup.find_all("a", class_="sister")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

class_ 參數同樣接受不同類型的 篩檢程式 ,字串,規則運算式,方法或 True :

```
soup.find_all(class_=re.compile("itl"))
# [<p class="title"><b>The Dormouse's story</b></p>]

def has_six_characters(css_class):
    return css_class is not None and len(css_class) == 6

soup.find_all(class_=has_six_characters)
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

tag 的 class 屬性是 多值屬性 .按照 CSS 類名搜索 tag 時,可以分別搜索 tag 中的每個 CSS 類名:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.find_all("p", class_="strikeout")
# [<p class="body strikeout"></p>]

css_soup.find_all("p", class_="body")
# [<p class="body strikeout"></p>]
```

搜索 class 屬性時也可以通過 CSS 值完全匹配:

```
css_soup.find_all("p", class_="body strikeout")
# [<p class="body strikeout"></p>]
```

完全匹配 class 的值時,如果 CSS 類名的順序與實際不符,將搜索不到結果:

```
soup.find_all("a", attrs={"class": "sister"})
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

text 參數

通過 text 參數可以搜搜文檔中的字串內容.與 name 參數的可選值一樣, text 參數接受 字串 , 規則運算式 , 列表 , True . 看例子:

```
soup.find_all(text="Elsie")
# [u'Elsie']

soup.find_all(text=["Tillie", "Elsie", "Lacie"])
# [u'Elsie', u'Lacie', u'Tillie']

soup.find_all(text=re.compile("Dormouse"))
[u"The Dormouse's story", u"The Dormouse's story"]

def is_the_only_string_within_a_tag(s):
    """Return True if this string is the only child of its parent tag."""
    return (s == s.parent.string)

soup.find_all(text=is_the_only_string_within_a_tag)
# [u"The Dormouse's story", u"The Dormouse's story", u'Elsie', u'Lacie',
u'Tillie', u'...']
```

雖然 text 參數用於搜索字串,還可以與其它參數混合使用來過濾 tag.Beautiful Soup 會找到 .string 方法與 text 參

數值相符的 tag.下面代碼用來搜索內容裡面包含“Elsie”的<a>標籤:

```
soup.find_all("a", text="Elsie")
# [<a href="http://example.com/elsie" class="sister"
id="link1">Elsie</a>]
```

limit 參數

find_all() 方法返回全部的搜索結構,如果文檔樹很大那麼搜索會很慢.如果我們不需要全部結果,可以使用 limit 參數限制返回結果的數量.效果與 SQL 中的 limit 關鍵字類似,當搜索到的結果數量達到 limit 的限制時,就停止搜索返回結果.

文檔樹中有 3 個 tag 符合搜索條件,但結果只返回了 2 個,因為我們限制了返回數量:

```
soup.find_all("a", limit=2)
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]
```

recursive 參數

調用 tag 的 find_all() 方法時,Beautiful Soup 會檢索當前 tag 的所有子孫節點,如果只想搜索 tag 的直接子節點,可以使用參數 recursive=False .

一段簡單的文檔:

```
<html>
 <head>
  <title>
   The Dormouse's story
  </title>
 </head>
...
```

是否使用 recursive 參數的搜索結果:

```
soup.html.find_all("title")
# [<title>The Dormouse's story</title>]

soup.html.find_all("title", recursive=False)
# []
```

像調用 find_all() 一樣調用 tag

find_all() 幾乎是 Beautiful Soup 中最常用的搜索方法,所以我們定義了它的簡寫方法. BeautifulSoup 物件和 tag 物件可以被當作一個方法來使用,這個方法的執行結果與調用這個物件的 find_all() 方法相同,下面兩行代碼是等價的:

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

這兩行代碼也是等價的:

```
soup.title.find_all(text=True)
soup.title(text=True)
```

find()

find(name , attrs , recursive , text , **kwargs)

find_all() 方法將返回文檔中符合條件的所有 tag,儘管有時候我們只想得到一個結果.比如文檔中只有一個 <body>標籤,那麼使用 find_all() 方法來查找<body>標籤就不太合適, 使用 find_all 方法並設置 limit=1 參數不如直接使用 find() 方法.下面兩行代碼是等價的:

```
soup.find_all('title', limit=1)
# [<title>The Dormouse's story</title>]

soup.find('title')
# <title>The Dormouse's story</title>
```

唯一的區別是 find_all() 方法的返回結果是值包含一個元素的清單,而 find() 方法直接返回結果.

find_all() 方法沒有找到目標是返回空清單, find() 方法找不到目標時,返回 None .

```
print(soup.find("nosuchtag"))
# None
```

soup.head.title 是 tag 的名字 方法的簡寫.這個簡寫的原理就是多次調用當前 tag 的 find() 方法:

```
soup.head.title
# <title>The Dormouse's story</title>

soup.find("head").find("title")
# <title>The Dormouse's story</title>
```

find_parents() 和 find_parent()

find_parents(name , attrs , recursive , text , **kwargs)

find_parent(name , attrs , recursive , text , **kwargs)

我們已經用了很大篇幅來介紹 find_all() 和 find() 方法,Beautiful Soup 中還有 10 個用於搜索的 API.它們中的五個用的是與 find_all() 相同的搜索參數,另外 5 個與 find() 方法的搜索參數類似.區別僅是它們搜索文檔的不同部分.

記住: find_all() 和 find() 只搜索當前節點的所有子節點,孫子節點等. find_parents() 和 find_parent() 用來搜索當前節點的父輩節點,搜索方法與普通 tag 的搜索方法相同,搜索文檔搜索文檔包含的內容. 我們從一個文檔中的一個葉子節點開始:

```
a_string = soup.find(text="Lacie")
a_string
# u'Lacie'

a_string.find_parents("a")
```

```
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]

a_string.find_parent("p")
# <p class="story">Once upon a time there were three little sisters; and
their names were
#  <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a> and
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>;
#  and they lived at the bottom of a well.</p>

a_string.find_parents("p", class="title")
# []
```

文檔中的一個<a>標籤是是當前葉子節點的直接父節點,所以可以被找到.還有一個<p>標籤,是目標葉子節點的間接父輩節點,所以也可以被找到.包含 class 值為"title"的<p>標籤不是不是目標葉子節點的父輩節點,所以通過 find_parents() 方法搜索不到.

find_parent() 和 find_parents() 方法會讓人聯想到 .parent 和 .parents 屬性.它們之間的聯繫非常緊密.搜索父輩節點的方法實際上就是對 .parents 屬性的反覆運算搜索.

find_next_siblings() 合 find_next_sibling()

find_next_siblings(name , attrs , recursive , text , **kwargs)

find_next_sibling(name , attrs , recursive , text , **kwargs)

這 2 個方法通過 .next_siblings 屬性對當 tag 的所有後面解析 [5] 的兄弟 tag 節點進行反覆運算, find_next_siblings() 方法返回所有符合條件的後面的兄弟節點, find_next_sibling() 只返回符合條件的後面的第一個 tag 節點.

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>

first_link.find_next_siblings("a")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

```
first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_next_sibling("p")
# <p class="story">...</p>
```

find_previous_siblings() 和 find_previous_sibling()

find_previous_siblings(name , attrs , recursive , text , **kwargs)

find_previous_sibling(name , attrs , recursive , text , **kwargs)

這 2 個方法通過 .previous_siblings 屬性對當前 tag 的前面解析 [5]的兄弟 tag 節點進行反覆運算, find_previous_siblings() 方法返回所有符合條件的前面的兄弟節點, find_previous_sibling() 方法返回第一個符合條件的前面的兄弟節點:

```
last_link = soup.find("a", id="link3")
last_link
# <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>

last_link.find_previous_siblings("a")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_previous_sibling("p")
# <p class="title"><b>The Dormouse's story</b></p>
```

find_all_next() 和 find_next()

find_all_next(name , attrs , recursive , text , **kwargs)

find_next(name , attrs , recursive , text , **kwargs)

這 2 個方法通過 .next_elements 屬性對當前 tag 的之後的 [5]tag 和字串進行反覆運算, find_all_next() 方法返回所有符合條件的節點, find_next() 方法返回第一個符合條件的節點:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>

first_link.find_all_next(text=True)
# [u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
```

```
#  u';\nand they lived at the bottom of a well.', u'\n\n', u'...', u'\n']

first_link.find_next("p")
# <p class="story">...</p>
```

第一個例子中,字串 “Elsie”也被顯示出來,儘管它被包含在我們開始查找的<a>標籤的裡面.第二個例子中,最後一個<p>標籤也被顯示出來,儘管它與我們開始查找位置的<a>標籤不屬於同一部分.例子中,搜索的重點是要匹配篩檢程式的條件,並且在文檔中出現的順序而不是開始查找的元素的位置.

find_all_previous() 和 find_previous()

find_all_previous(name , attrs , recursive , text , **kwargs)

find_previous(name , attrs , recursive , text , **kwargs)

這 2 個方法通過 .previous_elements 屬性對當前節點前面 [5] 的 tag 和字串進行反覆運算, find_all_previous() 方法返回所有符合條件的節點, find_previous() 方法返回第一個符合條件的節點.

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>

first_link.find_all_previous("p")
# [<p class="story">Once upon a time there were three little
sisters; ...</p>,
#  <p class="title"><b>The Dormouse's story</b></p>]

first_link.find_previous("title")
# <title>The Dormouse's story</title>
```

find_all_previous("p") 返回了文檔中的第一段(class=”title”的那段),但還返回了第二段,<p>標籤包含了我們開始查找的<a>標籤.不要驚訝,這段代碼的功能是查找所有出現在指定<a>標籤之前的<p>標籤,因為這個<p>標籤包含了開始的<a>標籤,所以<p>標籤一定是在<a>之前出現的.

CSS 選擇器

Beautiful Soup 支持大部分的 CSS 選擇器 [6],在 Tag 或 BeautifulSoup 對象的 .select() 方法中傳入字串參數,即可使用 CSS 選擇器的語法找到 tag:

```
soup.select("title")
# [<title>The Dormouse's story</title>]

soup.select("p nth-of-type(3)")
# [<p class="story">...</p>]
```

通過 tag 標籤逐層查找:

```
soup.select("body a")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.select("html head title")
# [<title>The Dormouse's story</title>]
```

找到某個 tag 標籤下的直接子標籤 [6]：

```
soup.select("head > title")
# [<title>The Dormouse's story</title>]

soup.select("p > a")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.select("p > a:nth-of-type(2)")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]

soup.select("p > #link1")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]

soup.select("body > a")
# []
```

找到兄弟節點標籤:

```
soup.select("#link1 ~ .sister")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

```
soup.select("#link1 + .sister")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]
```

通過 CSS 的類名查找:

```
soup.select(".sister")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.select("[class~=sister]")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

通過 tag 的 id 查找:

```
soup.select("#link1")
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]

soup.select("a#link2")
# [<a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>]
```

通過是否存在某個屬性來查找:

```
soup.select('a[href]')
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]
```

通過屬性的值來查找:

```
soup.select('a[href="http://example.com/elsie"]')
```

```
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]

soup.select('a[href^="http://example.com/"]')
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"
id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.select('a[href$="tillie"]')
# [<a class="sister" href="http://example.com/tillie"
id="link3">Tillie</a>]

soup.select('a[href*=".com/el"]')
# [<a class="sister" href="http://example.com/elsie"
id="link1">Elsie</a>]
```

通過語言設置來查找:

```
multilingual_markup = """
 <p lang="en">Hello</p>
 <p lang="en-us">Howdy, y'all</p>
 <p lang="en-gb">Pip-pip, old fruit</p>
 <p lang="fr">Bonjour mes amis</p>
"""
multilingual_soup = BeautifulSoup(multilingual_markup)
multilingual_soup.select('p[lang|=en]')
# [<p lang="en">Hello</p>,
#  <p lang="en-us">Howdy, y'all</p>,
#  <p lang="en-gb">Pip-pip, old fruit</p>]
```

對於熟悉 CSS 選擇器語法的人來說這是個非常方便的方法.Beautiful Soup 也支援 CSS 選擇器 API,如果你僅僅需要 CSS 選擇器的功能,那麼直接使用 lxml 也可以,而且速度更快,支援更多的 CSS 選擇器語法,但 Beautiful Soup 整合了 CSS 選擇器的語法和自身方便使用 API.

修改文檔樹

Beautiful Soup 的強項是文檔樹的搜索,但同時也可以方便的修改文檔樹

修改 tag 的名稱和屬性

在 Attributes 的章節中已經介紹過這個功能,但是再看一遍也無妨. 重命名一個 tag,改變屬性的值,添加或刪除屬性:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b

tag.name = "blockquote"
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>
```

修改 .string

給 tag 的 .string 屬性賦值,就相當於用當前的內容替代了原來的內容:

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

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

注意: 如果當前的 tag 包含了其它 tag,那麼給它的 .string 屬性賦值會覆蓋掉原有的所有內容包括子 tag

append()

Tag.append() 方法想 tag 中添加內容,就好像 Python 的列表的 .append() 方法:

```
soup = BeautifulSoup("<a>Foo</a>")
soup.a.append("Bar")

soup
# <html><head></head><body><a>FooBar</a></body></html>
soup.a.contents
# [u'Foo', u'Bar']
```

BeautifulSoup.new_string() 和 .new_tag()

如果想添加一段文本內容到文檔中也沒問題,可以調用 Python 的 append() 方法或調用工廠方

法BeautifulSoup.new_string()：

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

如果想要創建一段注釋,或NavigableString 的任何子類,將子類作為 new_string() 方法的第二個參數傳入：

```
from bs4 import Comment
new_comment = soup.new_string("Nice to see you.", Comment)
tag.append(new_comment)
tag
# <b>Hello there<!--Nice to see you.--></b>
tag.contents
# [u'Hello', u' there', u'Nice to see you.']
```

這是 Beautiful Soup 4.2.1 中新增的方法

創建一個 tag 最好的方法是調用工廠方法 BeautifulSoup.new_tag()：

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

new_tag = soup.new_tag("a", href="http://www.example.com")
original_tag.append(new_tag)
original_tag
# <b><a href="http://www.example.com"></a></b>

new_tag.string = "Link text."
original_tag
# <b><a href="http://www.example.com">Link text.</a></b>
```

第一個參數作為 tag 的 name,是必填,其它參數選填

insert()

Tag.insert() 方法與 Tag.append() 方法類似,區別是不會把新元素添加到父節點 .contents 屬性的最後,而是把元素插入到指定的位置.與 Python 列表總的 .insert() 方法的用法下同：

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

```
tag = soup.a

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

insert_before() 和 insert_after()

insert_before() 方法在當前 tag 或文本節點前插入內容:

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

insert_after() 方法在當前 tag 或文本節點後插入內容:

```
soup.b.i.insert_after(soup.new_string(" ever "))
soup.b
# <b><i>Don't</i> ever stop</b>
soup.b.contents
# [<i>Don't</i>, u' ever ', u'stop']
```

clear()

Tag.clear() 方法移除當前 tag 的內容:

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

tag.clear()
tag
# <a href="http://example.com/"></a>
```

extract()

PageElement.extract() 方法將當前 tag 移除文檔樹,並作為方法結果返回:

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

```
a_tag = soup.a

i_tag = soup.i.extract()

a_tag
# <a href="http://example.com/">I linked to</a>

i_tag
# <i>example.com</i>

print(i_tag.parent)
None
```

這個方法實際上產生了 2 個文檔樹: 一個是用來解析原始文檔的 BeautifulSoup 物件,另一個是被移除並且返回的 tag.被移除並返回的 tag 可以繼續調用 extract 方法:

```
my_string = i_tag.string.extract()
my_string
# u'example.com'

print(my_string.parent)
# None
i_tag
# <i></i>
```

decompose()

Tag.decompose() 方法將當前節點移除文檔樹並完全銷毀:

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

soup.i.decompose()

a_tag
# <a href="http://example.com/">I linked to</a>
```

replace_with()

PageElement.replace_with() 方法移除文檔樹中的某段內容,並用新 tag 或文本節點替代它:

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

```
a_tag = soup.a

new_tag = soup.new_tag("b")
new_tag.string = "example.net"
a_tag.i.replace_with(new_tag)

a_tag
# <a href="http://example.com/">I linked to <b>example.net</b></a>
```

replace_with() 方法返回被替代的 tag 或文本節點,可以用來流覽或添加到文檔樹其它地方

wrap()

PageElement.wrap() 方法可以對指定的 tag 元素進行包裝 [8],並返回包裝後的結果:

```
soup = BeautifulSoup("<p>I wish I was bold.</p>")
soup.p.string.wrap(soup.new_tag("b"))
# <b>I wish I was bold.</b>

soup.p.wrap(soup.new_tag("div"))
# <div><p><b>I wish I was bold.</b></p></div>
```

該方法在 Beautiful Soup 4.0.5 中添加

unwrap()

Tag.unwrap() 方法與 wrap() 方法相反.將移除 tag 內的所有 tag 標籤,該方法常被用來進行標記的解包:

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

a_tag.i.unwrap()
a_tag
# <a href="http://example.com/">I linked to example.com</a>
```

與 replace_with() 方法相同,unwrap() 方法返回被移除的 tag

輸出

格式化輸出

prettify() 方法將 Beautiful Soup 的文檔樹格式化後以 Unicode 編碼輸出,每個 XML/HTML 標籤都獨佔一行

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

```
# '<html>\n <head>\n </head>\n <body>\n  <a
href="http://example.com/">\n...'

print(soup.prettify())
# <html>
#  <head>
#  </head>
#  <body>
#   <a href="http://example.com/">
#    I linked to
#    <i>
#     example.com
#    </i>
#   </a>
#  </body>
# </html>
```

BeautifulSoup 物件和它的 tag 節點都可以調用 prettify() 方法:

```
print(soup.a.prettify())
# <a href="http://example.com/">
#  I linked to
#  <i>
#   example.com
#  </i>
# </a>
```

壓縮輸出

如果只想得到結果字串,不重視格式,那麼可以對一個 BeautifulSoup 物件或 Tag 物件使用 Python 的 unicode() 或 str() 方法:

```
str(soup)
# '<html><head></head><body><a href="http://example.com/">I linked to
<i>example.com</i></a></body></html>'

unicode(soup.a)
# u'<a href="http://example.com/">I linked to <i>example.com</i></a>'
```

str() 方法返回 UTF-8 編碼的字串,可以指定 編碼 的設置.

還可以調用 encode() 方法獲得位元組碼或調用 decode() 方法獲得 Unicode.

輸出格式

Beautiful Soup 輸出是會將 HTML 中的特殊字元轉換成 Unicode,比如“&lquot;”:

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
unicode(soup)
# u'<html><head></head><body>\u201cDammit!\u201d he
said.</body></html>'
```

如果將文檔轉換成字串,Unicode 編碼會被編碼成 UTF-8.這樣就無法正確顯示 HTML 特殊字元了:

```
str(soup)
# '<html><head></head><body>\xe2\x80\x9cDammit!\xe2\x80\x9d he
said.</body></html>'
```

get_text()

如果只想得到 tag 中包含的文本內容,那麼可以嗲用 get_text() 方法,這個方法獲取到 tag 中包含的所有文版內容包括子孫 tag 中的內容,並將結果作為 Unicode 字串返回:

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

soup.get_text()
u'\nI linked to example.com\n'
soup.i.get_text()
u'example.com'
```

可以通過參數指定 tag 的文本內容的分隔符號:

```
# soup.get_text("|")
u'\nI linked to |example.com|\n'
```

還可以去除獲得文本內容的前後空白:

```
# soup.get_text("|", strip=True)
u'I linked to|example.com'
```

或者使用 .stripped_strings 生成器,獲得文本清單後手動處理清單:

```
[text for text in soup.stripped_strings]
# [u'I linked to', u'example.com']
```

指定文檔解析器

如果僅是想要解析 HTML 文檔,只要用文檔創建 BeautifulSoup 物件就可以了.Beautiful Soup 會自動選擇一個解析器來解析文檔.但是還可以通過參數指定使用那種解析器來解析當前文檔.

BeautifulSoup 第一個參數應該是要被解析的文檔字串或是檔案控制代碼,第二個參數用來標識怎樣解析文檔.如果第二個參數為空,那麼 Beautiful Soup 根據當前系統安裝的庫自動選擇解析器,解析器的優先數序: lxml, html5lib, Python 標準庫.在下面兩種條件下解析器優先順序會變化:

要解析的文檔是什麼類型: 目前支持, “html”, “xml”, 和 “html5”

指定使用哪種解析器: 目前支持, “lxml”, “html5lib”, 和 “html.parser”

安裝解析器 章節介紹了可以使用哪種解析器,以及如何安裝.

如果指定的解析器沒有安裝,Beautiful Soup 會自動選擇其它方案.目前只有 lxml 解析器支持 XML 文檔的解析,在沒有安裝 lxml 庫的情況下,創建 beautifulsoup 物件時無論是否指定使用 lxml,都無法得到解析後的物件

解析器之間的區別

Beautiful Soup 為不同的解析器提供了相同的介面,但解析器本身時有區別的.同一篇文檔被不同的解析器解析後可能會生成不同結構的樹型文檔.區別最大的是 HTML 解析器和 XML 解析器,看下面片段被解析成 HTML 結構:

```
BeautifulSoup("<a><b /></a>")
# <html><head></head><body><a><b></b></a></body></html>
```

因為空標籤不符合 HTML 標準,所以解析器把它解析成

同樣的文檔使用 XML 解析如下(解析 XML 需要安裝 lxml 庫).注意,空標籤依然被保留,並且文檔前添加了 XML 頭,而不是被包含在<html>標籤內:

```
BeautifulSoup("<a><b /></a>", "xml")
# <?xml version="1.0" encoding="utf-8"?>
# <a><b/></a>
```

HTML 解析器之間也有區別,如果被解析的 HTML 文檔是標準格式,那麼解析器之間沒有任何差別,只是解析速度不同,結果都會返回正確的文檔樹.

但是如果被解析文檔不是標準格式,那麼不同的解析器返回結果可能不同.下面例子中,使用 lxml 解析錯誤格式的文檔,結果</p>標籤被直接忽略掉了:

```
BeautifulSoup("<a></p>", "lxml")
# <html><body><a></a></body></html>
```

使用 html5lib 庫解析相同文檔會得到不同的結果:

```
BeautifulSoup("<a></p>", "html5lib")
# <html><head></head><body><a><p></p></a></body></html>
```

html5lib 庫沒有忽略掉</p>標籤,而是自動補全了標籤,還給文檔樹添加了<head>標籤.

使用 pyhton 內置庫解析結果如下:

```
BeautifulSoup("<a></p>", "html.parser")
# <a></a>
```

與 lxml [7] 庫類似的,Python 內置庫忽略掉了</p>標籤,與 html5lib 庫不同的是標準庫沒有嘗試創建符合標準的文檔格式或將文檔片段包含在<body>標籤內,與 lxml 不同的是標準庫甚至連<html>標籤都沒有嘗試去添加.

因為文檔片段“<a></p>”是錯誤格式,所以以上解析方式都能算作”正確”,html5lib 庫使用的是 HTML5 的部分標準,所以最接近”正確”.不過所有解析器的結構都能夠被認為是”正常”的.

不同的解析器可能影響代碼執行結果,如果在分發給別人的代碼中使用了 BeautifulSoup,那麼最好注明使用了哪種解析器,以減少不必要的麻煩.

編碼

任何 HTML 或 XML 文檔都有自己的編碼方式,比如 ASCII 或 UTF-8,但是使用 Beautiful Soup 解析後,文檔都被轉換成了 Unicode:

```
markup = "<h1>Sacr\xc3\xa9 bleu!</h1>"
soup = BeautifulSoup(markup)
soup.h1
# <h1>Sacré bleu!</h1>
soup.h1.string
# u'Sacr\xe9 bleu!'
```

這不是魔術(但很神奇),Beautiful Soup 用了 編碼自動檢測 子庫來識別當前文檔編碼並轉換成 Unicode 編碼.BeautifulSoup 對象的 .original_encoding 屬性記錄了自動識別編碼的結果:

```
soup.original_encoding
'utf-8'
```

編碼自動檢測 功能大部分時候都能猜對編碼格式,但有時候也會出錯.有時候即使猜測正確,也是在逐個位元組的遍歷整個文檔後才猜對的,這樣很慢.如果預先知道文檔編碼,可以設置編碼參數來減少自動檢查編碼出錯的概率並且提高文檔解析速度.在創建 BeautifulSoup 物件的時候設置 from_encoding 參數.

下面一段文檔用了 ISO-8859-8 編碼方式,這段文檔太短,結果 Beautiful Soup 以為文檔是用 ISO-8859-7 編碼:

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

通過傳入 from_encoding 參數來指定編碼方式:

```
soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
soup.h1
<h1>שלום</h1>
soup.original_encoding
'iso8859-8'
```

少數情況下(通常是 UTF-8 編碼的文檔中包含了其它編碼格式的檔),想獲得正確的 Unicode 編碼就不得不將文檔中少數特殊編碼字元替換成特殊 Unicode 編碼,"REPLACEMENT CHARACTER" (U+FFFD,) [9]. 如果 Beautifu Soup 猜測文檔編碼時作了特殊字元的替換,那麼 Beautiful Soup 會把 UnicodeDammit 或 BeautifulSoup 物件的 .contains_replacement_characters 屬性標記為 True .這樣就可以知道當前文檔進行 Unicode 編碼後丟失了一部分特殊內容字元.如果文檔中包含而 .contains_replacement_characters 屬性是 False ,則表示就是文檔中原來的字元,不是轉碼失敗.

輸出編碼

通過 Beautiful Soup 輸出文檔時,不管輸入文檔是什麼編碼方式,輸出編碼均為 UTF-8 編碼,下面例子輸入文檔是 Latin-1 編碼:

```
markup = b'''
<html>
```

```
  <head>
    <meta content="text/html; charset=ISO-Latin-1"
http-equiv="Content-type" />
  </head>
  <body>
    <p>Sacr\xe9 bleu!</p>
  </body>
</html>
'''

soup = BeautifulSoup(markup)
print(soup.prettify())
# <html>
#  <head>
#   <meta content="text/html; charset=utf-8" http-equiv="Content-type"
/>
#  </head>
#  <body>
#   <p>
#    Sacré bleu!
#   </p>
#  </body>
# </html>
```

注意,輸出文檔中的<meta>標籤的編碼設置已經修改成了與輸出編碼一致的 UTF-8.

如果不想用 UTF-8 編碼輸出,可以將編碼方式傳入 prettify() 方法:

```
print(soup.prettify("latin-1"))
# <html>
#  <head>
#   <meta content="text/html; charset=latin-1"
http-equiv="Content-type" />
# ...
```

還可以調用 BeautifulSoup 物件或任意節點的 encode() 方法,就像 Python 的字串調用 encode() 方法一樣:

```
soup.p.encode("latin-1")
# '<p>Sacr\xe9 bleu!</p>'

soup.p.encode("utf-8")
# '<p>Sacr\xc3\xa9 bleu!</p>'
```

如果文檔中包含當前編碼不支援的字元,那麼這些字元將唄轉換成一系列 XML 特殊字元引用,下面例子中包含了

Unicode 編碼字元 SNOWMAN:

```
markup = u"<b>\N{SNOWMAN}</b>"
snowman_soup = BeautifulSoup(markup)
tag = snowman_soup.b
```

SNOWMAN 字元在 UTF-8 編碼中可以正常顯示(看上去像是☃),但有些編碼不支援 SNOWMAN 字元,比如 ISO-Latin-1 或 ASCII,那麼在這些編碼中 SNOWMAN 字元會被轉換成"&#9731":

```
print(tag.encode("utf-8"))
# <b>☃</b>

print tag.encode("latin-1")
# <b>&#9731;</b>

print tag.encode("ascii")
# <b>&#9731;</b>
```

Unicode, dammit! (靠!)

編碼自動檢測 功能可以在 Beautiful Soup 以外使用,檢測某段未知編碼時,可以使用這個方法:

```
from bs4 import UnicodeDammit
dammit = UnicodeDammit("Sacr\xc3\xa9 bleu!")
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'utf-8'
```

如果 Python 中安裝了 chardet 或 cchardet 那麼編碼檢測功能的準確率將大大提高.輸入的字元越多,檢測結果越精確,如果事先猜測到一些可能編碼,那麼可以將猜測的編碼作為參數,這樣將優先檢測這些編碼:

```
dammit = UnicodeDammit("Sacr\xe9 bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

編碼自動檢測 功能中有 2 項功能是 Beautiful Soup 庫中用不到的

智能引號

使用 Unicode 時,Beautiful Soup 還會智慧的把引號 [10] 轉換成 HTML 或 XML 中的特殊字元:

```
markup = b"<p>I just \x93love\x94 Microsoft Word\x92s smart quotes</p>"

UnicodeDammit(markup, ["windows-1252"],
smart_quotes_to="html").unicode_markup
# u'<p>I just &ldquo;love&rdquo; Microsoft Word&rsquo;s smart
```

```
quotes</p>'

UnicodeDammit(markup, ["windows-1252"],
smart_quotes_to="xml").unicode_markup
# u'<p>I just &#x201C;love&#x201D; Microsoft Word&#x2019;s smart
quotes</p>'
```

也可以把引號轉換為 ASCII 碼:

```
UnicodeDammit(markup, ["windows-1252"],
smart_quotes_to="ascii").unicode_markup
# u'<p>I just "love" Microsoft Word\'s smart quotes</p>'
```

很有用的功能,但是 Beautiful Soup 沒有使用這種方式.預設情況下,Beautiful Soup 把引號轉換成 Unicode:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'
```

矛盾的編碼

有時文檔的大部分都是用 UTF-8,但同時還包含了 Windows-1252 編碼的字元,就像微軟的智能引號 [10] 一樣.一些包含多個資訊的來源網站容易出現這種情況. UnicodeDammit.detwingle() 方法可以把這類文檔轉換成純 UTF-8 編碼格式,看個簡單的例子:

```
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE
QUOTATION MARK}")
doc = snowmen.encode("utf8") + quote.encode("windows_1252")
```

這段文檔很雜亂,snowmen 是 UTF-8 編碼,引號是 Windows-1252 編碼,直接輸出時不能同時顯示 snowmen 和引號,因為它們編碼不同:

```
print(doc)
# ☃☃☃<?>I like snowmen!<?>

print(doc.decode("windows-1252"))
# â˜ƒâ˜ƒâ˜ƒ“I like snowmen!”
```

如果對這段文檔用 UTF-8 解碼就會得到 UnicodeDecodeError 異常,如果用 Windows-1252 解碼就回得到一堆亂碼.幸好,UnicodeDammit.detwingle() 方法會吧這段字串轉換成 UTF-8 編碼,允許我們同時顯示出文檔中的 snowmen 和引號:

```
new_doc = UnicodeDammit.detwingle(doc)
print(new_doc.decode("utf8"))
# ☃☃☃“I like snowmen!”
```

UnicodeDammit.detwingle() 方法只能解碼包含在 UTF-8 編碼中的 Windows-1252 編碼內容,但這解決了最常見的一類問題.

在創建 BeautifulSoup 或UnicodeDammit 物件前一定要先對文檔調用 UnicodeDammit.detwingle() 確保文檔的編碼方式正確.如果嘗試去解析一段包含 Windows-1252 編碼的 UTF-8 文檔,就會得到一堆亂碼,比如: â˜ƒâ˜ƒâ˜ƒ“I like

snowmen!”.

UnicodeDammit.detwingle() 方法在 Beautiful Soup 4.1.0 版本中新增

解析部分文檔

如果僅僅因為想要查找文檔中的<a>標籤而將整片文檔進行解析,實在是浪費記憶體和時間.最快的方法是從一開始就把<a>標籤以外的東西都忽略掉. SoupStrainer 類可以定義文檔的某段內容,這樣搜索文檔時就不必先解析整篇文檔,只會解析在 SoupStrainer 中定義過的文檔. 創建一個 SoupStrainer 物件並作為 parse_only 參數給 BeautifulSoup 的構造方法即可.

SoupStrainer

SoupStrainer 類接受與典型搜索方法相同的參數：name , attrs , recursive , text , **kwargs 。下面舉例說明三種 SoupStrainer 物件：

```
from bs4 import SoupStrainer

only_a_tags = SoupStrainer("a")

only_tags_with_id_link2 = SoupStrainer(id="link2")

def is_short_string(string):
    return len(string) < 10

only_short_strings = SoupStrainer(text=is_short_string)
```

再拿“愛麗絲”文檔來舉例，來看看使用三種 SoupStrainer 物件做參數會有什麼不同:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and
their names were
<a href="http://example.com/elsie" class="sister"
id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
and
<a href="http://example.com/tillie" class="sister"
id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
```

```
"""

print(BeautifulSoup(html_doc, "html.parser",
parse_only=only_a_tags).prettify())
# <a class="sister" href="http://example.com/elsie" id="link1">
#  Elsie
# </a>
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>
# <a class="sister" href="http://example.com/tillie" id="link3">
#  Tillie
# </a>

print(BeautifulSoup(html_doc, "html.parser",
parse_only=only_tags_with_id_link2).prettify())
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>

print(BeautifulSoup(html_doc, "html.parser",
parse_only=only_short_strings).prettify())
# Elsie
# ,
# Lacie
# and
# Tillie
# ...
#
```

還可以將 SoupStrainer 作為參數傳入搜索文檔樹 中提到的方法.這可能不是個常用用法,所以還是提一下:

```
soup = BeautifulSoup(html_doc)
soup.find_all(only_short_strings)
# [u'\n\n', u'\n\n', u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
#  u'\n\n', u'...', u'\n']
```

常見問題

代碼診斷

如果想知道 Beautiful Soup 到底怎樣處理一份文檔,可以將文檔傳入 diagnose() 方法(Beautiful Soup 4.2.0 中新增),Beautiful Soup 會輸出一份報告,說明不同的解析器會怎樣處理這段文檔,並標出當前的解析過程會使用哪種解析器:

```
from bs4.diagnose import diagnose
data = open("bad.html").read()
diagnose(data)

# Diagnostic running on Beautiful Soup 4.2.0
# Python version 2.7.3 (default, Aug  1 2012, 05:16:07)
# I noticed that html5lib is not installed. Installing it may help.
# Found lxml version 2.3.2.0
#
# Trying to parse your data with html.parser
# Here's what html.parser did with the document:
# ...
```

diagnose() 方法的輸出結果可能幫助你找到問題的原因,如果不行,還可以把結果複製出來以便尋求他人的幫助

文檔解析錯誤

文檔解析錯誤有兩種.一種是崩潰,Beautiful Soup 嘗試解析一段文檔結果卻拋除了異常,通常是 HTMLParser.HTMLParseError .還有一種異常情況,是 Beautiful Soup 解析後的文檔樹看起來與原來的內容相差很多.

這些錯誤幾乎都不是 Beautiful Soup 的原因,這不會是因為 Beautiful Soup 得代碼寫的太優秀,而是因為 Beautiful Soup 沒有包含任何文檔解析代碼.異常產生自被依賴的解析器,如果解析器不能很好的解析出當前的文檔,那麼最好的辦法是換一個解析器.更多細節查看 安裝解析器 章節.

最常見的解析錯誤

是 HTMLParser.HTMLParseError: malformed start tag 和 HTMLParser.HTMLParseError: bad end tag .這都是由 Python 內置的解析器引起的,解決方法是 安裝 lxml 或 html5lib

最常見的異常現象是當前文檔找不到指定的 Tag,而這個 Tag 光是用眼睛就足夠發現的了. find_all() 方法返回 [], 而 find() 方法返回 None .這是 Python 內置解析器的又一個問題: 解析器會跳過那些它不知道的 tag.解決方法還是 安裝 lxml 或 html5lib

版本錯誤

SyntaxError: Invalid syntax (異常位置在代碼行:ROOT_TAG_NAME = u'[document]'),因為 Python2 版本的代碼沒有經過遷移就在 Python3 中窒息感

ImportError: No module named HTMLParser 因為在 Python3 中執行 Python2 版本的 Beautiful Soup

ImportError: No module named html.parser 因為在 Python2 中執行 Python3 版本的 Beautiful Soup

ImportError: No module named BeautifulSoup 因為在沒有安裝 BeautifulSoup3 庫的 Python 環境下執行代碼,或忘記了 BeautifulSoup4 的代碼需要從 bs4 包中引入

ImportError: No module named bs4 因為當前 Python 環境下還沒有安裝 BeautifulSoup4

解析成 XML

預設情況下,Beautiful Soup 會將當前文檔作為 HTML 格式解析,如果要解析 XML 文檔,要在 BeautifulSoup 構造方法中加入第二個參數 “xml”:

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

當然,還需要 安裝 lxml

解析器的錯誤

如果同樣的代碼在不同環境下結果不同,可能是因為兩個環境下使用不同的解析器造成的.例如這個環境中安裝了 lxml,而另一個環境中只有 html5lib, 解析器之間的區別 中說明了原因.修復方法是在 BeautifulSoup 的構造方法中中指定解析器

因為 HTML 標籤是 大小寫敏感 的,所以 3 種解析器再出來文檔時都將 tag 和屬性轉換成小寫.例如文檔中的 <TAG></TAG> 會被轉換為 <tag></tag>.如果想要保留 tag 的大寫的話,那麼應該將文檔 解析成 XML.

雜項錯誤

UnicodeEncodeError: 'charmap' codec can't encode character u'\xfoo' in position bar (或其它類型的 UnicodeEncodeError)的錯誤,主要是兩方面的錯誤(都不是 Beautiful Soup 的原因),第一種是正在使用的終端(console)無法顯示部分 Unicode,參考 Python wiki ,第二種是向檔寫入時,被寫入檔不支援部分 Unicode,這時只要用 u.encode("utf8") 方法將編碼轉換為 UTF-8.

KeyError: [attr] 因為調用 tag['attr'] 方法而引起,因為這個 tag 沒有定義該屬性.出錯最多的是 KeyError: 'href' 和 KeyError: 'class' .如果不確定某個屬性是否存在時,用 tag.get('attr') 方法去獲取它,跟獲取 Python 字典的 key 一樣

AttributeError: 'ResultSet' object has no attribute 'foo' 錯誤通常是因為把 find_all() 的返回結果當作一個 tag 或文本節點使用,實際上返回結果是一個清單或 ResultSet 物件的字串,需要對結果進行迴圈才能得到每個節點的 .foo 屬性.或者使用 find() 方法僅獲取到一個節點

AttributeError: 'NoneType' object has no attribute 'foo' 這個錯誤通常是在調用了 find() 方法後直節點取某個屬性 .foo 但是 find() 方法並沒有找到任何結果,所以它的返回值是 None .需要找出為什麼 find() 的返回值是 None .

如何提高效率

Beautiful Soup 對文檔的解析速度不會比它所依賴的解析器更快,如果對計算時間要求很高或者電腦的時間比程式師的時間更值錢,那麼就應該直接使用 lxml .

換句話說,還有提高 Beautiful Soup 效率的辦法,使用 lxml 作為解析器.Beautiful Soup 用 lxml 做解析器比用 html5lib 或 Python 內置解析器速度快很多.

安裝 cchardet 後文檔的解碼的編碼檢測會速度更快

解析部分文檔 不會節省多少解析時間,但是會節省很多記憶體,並且搜索時也會變得更快.

Beautiful Soup 3

Beautiful Soup 3 是上一個發佈版本,目前已經停止維護.Beautiful Soup 3 庫目前已經被幾個主要的 linux 平臺添加到源裡:

```
$ apt-get install Python-beautifulsoup
```

在 PyPi 中分發的包名字是 BeautifulSoup:

```
$ easy_install BeautifulSoup
$ pip install BeautifulSoup
```

或通過 Beautiful Soup 3.2.0 源碼包 安裝

Beautiful Soup 3 的線上文檔查看 這裡,當然還有 中文版,然後再讀本片 文檔,來對比 Beautiful Soup 4 中有什新變化.

遷移到 BS4

只要一個小變動就能讓大部分的 Beautiful Soup 3 代碼使用 Beautiful Soup 4 的庫和方法—-修改 BeautifulSoup 物件的引入方式:

```
from BeautifulSoup import BeautifulSoup
```

修改為:

```
from bs4 import BeautifulSoup
```

如果代碼拋出 ImportError 異常“No module named BeautifulSoup”,原因可能是嘗試執行 Beautiful Soup 3,但環境中只安裝了 Beautiful Soup 4 庫

如果代碼跑出 ImportError 異常“No module named bs4”,原因可能是嘗試運行 Beautiful Soup 4 的代碼,但環境中只安裝了 Beautiful Soup 3.

雖然 BS4 相容絕大部分 BS3 的功能,但 BS3 中的大部分方法已經不推薦使用了,就方法按照 PEP8 標準 重新定義了方法名.很多方法都重新定義了方法名,但只有少數幾個方法沒有向下相容.

上述內容就是 BS3 遷移到 BS4 的注意事項

需要的解析器

Beautiful Soup 3 曾使用 Python 的 SGMLParser 解析器,這個模組在 Python3 中已經被移除了.Beautiful Soup 4 預設使用系統的 html.parser,也可以使用 lxml 或 html5lib 擴展庫代替.查看 安裝解析器 章節

因為 html.parser 解析器與 SGMLParser 解析器不同,它們在處理格式不正確的文檔時也會產生不同結果.通常 html.parser 解析器會拋出異常.所以推薦安裝擴展庫作為解析器.有時 html.parser 解析出的文檔樹結構與 SGMLParser 的不同.如果發生這種情況,那麼需要升級 BS3 來處理新的文檔樹.

方法名的變化

renderContents -> encode_contents

replaceWith -> replace_with

replaceWithChildren -> unwrap

findAll -> find_all

findAllNext -> find_all_next

findAllPrevious -> find_all_previous

findNext -> find_next

findNextSibling -> find_next_sibling

findNextSiblings -> find_next_siblings

findParent -> find_parent

findParents -> find_parents

findPrevious -> find_previous

findPreviousSibling -> find_previous_sibling

findPreviousSiblings -> find_previous_siblings

nextSibling -> next_sibling

previousSibling -> previous_sibling

Beautiful Soup 構造方法的參數部分也有名字變化:

BeautifulSoup(parseOnlyThese=...) -> BeautifulSoup(parse_only=...)

BeautifulSoup(fromEncoding=...) -> BeautifulSoup(from_encoding=...)

為了適配 Python3,修改了一個方法名:

Tag.has_key() -> Tag.has_attr()

修改了一個屬性名,讓它看起來更專業點:

Tag.isSelfClosing -> Tag.is_empty_element

修改了下面 3 個屬性的名字,以免雨 Python 保留字衝突.這些變動不是向下相容的,如果在 BS3 中使用了這些屬性,那麼在 BS4 中這些代碼無法執行.

UnicodeDammit.Unicode -> UnicodeDammit.Unicode_markup``

Tag.next -> Tag.next_element

Tag.previous -> Tag.previous_element

生成器

將下列生成器按照 PEP8 標準重新命名,並轉換成物件的屬性:

childGenerator() -> children

nextGenerator() -> next_elements

nextSiblingGenerator() -> next_siblings

previousGenerator() -> previous_elements

previousSiblingGenerator() -> previous_siblings

recursiveChildGenerator() -> descendants

parentGenerator() -> parents

所以遷移到 BS4 版本時要替換這些代碼:

```
for parent in tag.parentGenerator():
    ...
```

替換為:

```
for parent in tag.parents:
    ...
```

(兩種調用方法現在都能使用)

BS3 中有的生成器迴圈結束後會返回 None 然後結束.這是個 bug.新版生成器不再返回 None .

BS4 中增加了 2 個新的生成器, .strings 和 stripped_strings . .strings 生成器返回 NavigableString 對象, .stripped_strings 方法返回去除前後空白的 Python 的 string 物件.

XML

BS4 中移除了解析 XML 的 BeautifulStoneSoup 類.如果要解析一段 XML 文檔,使用 BeautifulSoup 構造方法並在第二個參數設置為“xml”.同時 BeautifulSoup 構造方法也不再識別 isHTML 參數.

Beautiful Soup 處理 XML 空標籤的方法升級了.舊版本中解析 XML 時必須指明哪個標籤是空標籤. 構造方法的 selfClosingTags 參數已經不再使用.新版 Beautiful Soup 將所有空標籤解析為空元素,如果向空元素中添加子節點,那麼這個元素就不再是空元素了.

實體

HTML 或 XML 實體都會被解析成 Unicode 字元,Beautiful Soup 3 版本中有很多處理實體的方法,在新版中都被移除了.BeautifulSoup 構造方法也不再接受 smartQuotesTo 或 convertEntities 參數. 編碼自動檢測 方法依然有 smart_quotes_to 參數,但是默認會將引號轉換成 Unicode.內容配置項 HTML_ENTITIES , XML_ENTITIES 和 XHTML_ENTITIES 在新版中被移除.因為它們代表的特性已經不再被支持.

如果在輸出文檔時想把 Unicode 字元轉換成 HTML 實體,而不是輸出成 UTF-8 編碼,那就需要用到 輸出格式 的方法.

遷移雜項

Tag.string 屬性現在是一個遞迴操作.如果 A 標籤只包含了一個 B 標籤,那麼 A 標籤的.string 屬性值與 B 標籤的.string 屬性值相同.

多值屬性 比如 class 屬性包含一個他們的值的清單,而不是一個字串.這可能會影響到如何按照 CSS 類名哦搜索 tag.

如果使用 find* 方法時同時傳入了 text 參數 和 name 參數 .Beautiful Soup 會搜索指定 name 的 tag,並且這個 tag 的 Tag.string 屬性包含 text 參數的內容.結果中不會包含字串本身.舊版本中 Beautiful Soup 會忽略掉 tag 參數,只搜索 text 參數.

BeautifulSoup 構造方法不再支援 markupMassage 參數.現在由解析器負責文檔的解析正確性.

很少被用到的幾個解析器方法在新版中被移除,比如 ICantBelieveItsBeautifulSoup 和 BeautifulSOAP .現在由解析器完全負責如何解釋模糊不清的文檔標記.

prettify() 方法在新版中返回 Unicode 字串,不再返回位元組流.

BeautifulSoup3 文檔

[1] BeautifulSoup 的 google 討論群組不是很活躍,可能是因為庫已經比較完善了吧,但是作者還是會很熱心的儘量幫你解決問題的.

[2] *(1,2)*文檔被解析成樹形結構,所以下一步解析過程應該是當前節點的子節點

[3] 篩檢程式只能作為搜索文檔的參數,或者說應該叫參數類型更為貼切,原文中用了filter因此翻譯為篩檢程式

[4] 元素參數,HTML 文檔中的一個 tag 節點,不能是文本節點

[5] *(1,2,3,4,5)*採用先序遍歷方式

[6] *(1,2)*CSS 選擇器是一種單獨的文檔搜索語法, 參考http://www.w3school.com.cn/css/css_selector_type.asp

[7] 原文寫的是 html5lib, 譯者覺得這是願文檔的一個筆誤

[8] wrap 含有包裝,打包的意思,但是這裡的包裝不是在外部包裝而是將當前tag的內部內容包裝在一個tag裡.包裝原來內容的新tag依然在執行wrap()方法的tag內

[9] 文檔中特殊編碼字元被替換成特殊字元(通常是 )的過程是 Beautful Soup 自動實現的,如果想要多種編碼格式的文檔被完全轉換正確,那麼,只好,預先手動處理,統一編碼格式

[10] *(1,2)*智慧引號,常出現在 microsoft 的 word 軟體中,即在某一段落中按引號出現的順序每個引號都被自動轉換為左引號,或右引號.

Table Of Contents

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

index

Beautiful Soup 4.2.0 documentation »