## Python 简介

## Python环境搭建 pycharm安装

## Python基础语法

## Python制作爬虫

### 4.1 什么是爬虫

爬虫，即网络爬虫，大家可以理解为在网络上爬行的一直蜘蛛，互联网就比作一张大网，而爬虫便是在这张网上爬来爬去的蜘蛛咯，如果它遇到资源，那么它就会抓取下来。当然这个抓取的过程是由你控制的。

比如它在抓取一个网页，在这个网中他发现了一条道路，其实就是指向网页的超链接，那么它就可以爬到另一张网上来获取数据。这样，整个连在一起的大网对这之蜘蛛来说触手可及，分分钟爬下来不是事儿。

### 4.2 浏览网页的过程

在用户浏览网页的过程中，我们可能会看到许多好看的图片，比如 <http://image.baidu.com/> ，我们会看到几张的图片以及百度搜索框，这个过程其实就是用户输入网址之后，经过DNS服务器，找到服务器主机，向服务器发出一个请求，服务器经过解析之后，发送给用户的浏览器 HTML、JS、CSS 等文件，浏览器解析出来，用户便可以看到形形色色的图片了。

因此，用户看到的网页实质是由 HTML 代码构成的，爬虫爬来的便是这些内容，通过分析和过滤这些 HTML 代码，实现对图片、文字等资源的获取。

### 4.3 URL的含义

URL，即统一资源定位符，也就是我们说的网址，统一资源定位符是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

URL的格式由三部分组成：  
①第一部分是协议(或称为服务方式)。  
②第二部分是存有该资源的主机IP地址(有时也包括端口号)。  
③第三部分是主机资源的具体地址，如目录和文件名等。

爬虫爬取数据时必须要有一个目标的URL才可以获取数据，因此，它是爬虫获取数据的基本依据，准确理解它的含义对爬虫学习有很大帮助。

### 4.4 Python requests库的用法

urllib和urllib2库是学习Python爬虫最基本的库，利用这个库我们可以得到网页的内容，并对内容提取分析，得到我们想要的结果。

$ pip install requests

<http://cn.python-requests.org/zh_CN/latest/user/quickstart.html>

<https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/>

### 4.5 BeautifulSoup

Beautiful Soup 是一个可以从HTML或XML文件中提取数据的Python库.它能够通过你喜欢的转换器实现惯用的文档导航,查找,修改文档的方式.Beautiful Soup会帮你节省数小时甚至数天的工作时间.

#### 4.5.1安装 Beautiful Soup

Beautiful Soup 4 可以通过 pip 来安装.包的名字是 beautifulsoup4 ,这个包兼容Python2和Python3.

**$ pip install beautifulsoup4**

(在PyPi中还有一个名字是 BeautifulSoup 的包,但那可能不是你想要的,那是 [Beautiful Soup3](http://www.crummy.com/software/BeautifulSoup/bs3/documentation.html) 的发布版本,因为很多项目还在使用BS3, 所以 BeautifulSoup 包依然有效.但是如果你在编写新项目,那么你应该安装的 beautifulsoup4 )

#### 4.5.2安装解析器

Beautiful Soup支持Python标准库中的HTML解析器,还支持一些第三方的解析器,其中一个是 [lxml](http://lxml.de/) .可以用下列方法来安装lxml:

**$ pip install lxml**

这里我们不用Python标准库的解析器，而使用第三方的lxml HTML 解析器

#### 4.5.3使用示例

将一段文档传入BeautifulSoup 的构造方法,就能得到一个文档的对象, 可以传入一段字符串或一个文件句柄.

from bs4 import BeautifulSoup

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

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

首先,文档被转换成Unicode,并且HTML的实例都被转换成Unicode编码

BeautifulSoup("Sacr&eacute; bleu!")

<html><head></head><body>Sacré bleu!</body></html>

然后,Beautiful Soup选择最合适的解析器来解析这段文档,如果手动指定解析器那么Beautiful Soup会选择指定的解析器来解析文档。

#### 4.5.4对象的种类

Beautiful Soup将复杂HTML文档转换成一个复杂的树形结构,每个节点都是Python对象,所有对象可以归纳为4种: Tag , NavigableString , BeautifulSoup , Comment .

##### Tag

Tag 对象与XML或HTML原生文档中的tag相同:

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

tag = soup.b

type(tag)

# <class 'bs4.element.Tag'>

Tag有很多方法和属性,在 [遍历文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id15) 和 [搜索文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id24) 中有详细解释.现在介绍一下tag中最重要的属性: name和attributes

###### Name

每个tag都有自己的名字,通过 .name 来获取:

tag.name

# u'b'

如果改变了tag的name,那将影响所有通过当前Beautiful Soup对象生成的HTML文档:

tag.name = "blockquote"

tag

# <blockquote class="boldest">Extremely bold</blockquote>

###### Attributes

一个tag可能有很多个属性. tag <b class="boldest"> 有一个 “class” 的属性,值为 “boldest” . tag的属性的操作方法与字典相同:

tag['class']

# u'boldest'

也可以直接”点”取属性, 比如: .attrs :

tag.attrs

# {u'class': u'boldest'}

tag的属性可以被添加,删除或修改. 再说一次, tag的属性操作方法与字典一样

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

#### 多值属性

HTML 4定义了一系列可以包含多个值的属性.在HTML5中移除了一些,却增加更多.最常见的多值的属性是 class (一个tag可以有多个CSS的class). 还有一些属性 rel , rev , accept-charset , headers , accesskey . 在Beautiful Soup中多值属性的返回类型是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"]*

如果某个属性看起来好像有多个值,但在任何版本的HTML定义中都没有被定义为多值属性,那么Beautiful Soup会将这个属性作为字符串返回

id\_soup = BeautifulSoup('<p id="my id"></p>')

id\_soup.p['id']

*# 'my id'*

将tag转换成字符串时,多值属性会合并为一个值

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>*

如果转换的文档是XML格式,那么tag中不包含多值属性

xml\_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml')

xml\_soup.p['class']

*# u'body strikeout'*

## 可以遍历的字符串

字符串常被包含在tag内.Beautiful Soup用 NavigableString 类来包装tag中的字符串:

tag.string

*# u'Extremely bold'*

type(tag.string)

*# <class 'bs4.element.NavigableString'>*

一个 NavigableString 字符串与Python中的Unicode字符串相同,并且还支持包含在 [遍历文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id15) 和 [搜索文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id24) 中的一些特性. 通过 unicode() 方法可以直接将 NavigableString 对象转换成Unicode字符串:

unicode\_string = unicode(tag.string)

unicode\_string

*# u'Extremely bold'*

type(unicode\_string)

*# <type 'unicode'>*

tag中包含的字符串不能编辑,但是可以被替换成其它的字符串,用 [replace\_with()](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#replace-with) 方法:

tag.string.replace\_with("No longer bold")

tag

*# <blockquote>No longer bold</blockquote>*

NavigableString 对象支持 [遍历文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id15) 和 [搜索文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id24) 中定义的大部分属性, 并非全部.尤其是,一个字符串不能包含其它内容(tag能够包含字符串或是其它tag),字符串不支持 .contents 或 .string 属性或 find() 方法.

如果想在Beautiful Soup之外使用 NavigableString 对象,需要调用 unicode() 方法,将该对象转换成普通的Unicode字符串,否则就算Beautiful Soup已方法已经执行结束,该对象的输出也会带有对象的引用地址.这样会浪费内存.

## BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容.大部分时候,可以把它当作 Tag 对象,它支持 [遍历文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id15) 和 [搜索文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id24) 中描述的大部分的方法.

因为 BeautifulSoup 对象并不是真正的HTML或XML的tag,所以它没有name和attribute属性.但有时查看它的 .name 属性是很方便的,所以 BeautifulSoup 对象包含了一个值为 “[document]” 的特殊属性 .name

soup.name

*# u'[document]'*

## 注释及特殊字符串

Tag , NavigableString , BeautifulSoup 几乎覆盖了html和xml中的所有内容,但是还有一些特殊对象.容易让人担心的内容是文档的注释部分:

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 对象是一个特殊类型的 NavigableString 对象:

comment

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

但是当它出现在HTML文档中时, Comment 对象会使用特殊的格式输出:

**print**(soup.b.prettify())

*# <b>*

*# <!--Hey, buddy. Want to buy a used parser?-->*

*# </b>*

Beautiful Soup中定义的其它类型都可能会出现在XML的文档中: CData , ProcessingInstruction , Declaration , Doctype .与 Comment 对象类似,这些类都是 NavigableString 的子类,只是添加了一些额外的方法的字符串独享.下面是用CDATA来替代注释的例子:

**from** **bs4** **import** CData

cdata = CData("A CDATA block")

comment.replace\_with(cdata)

**print**(soup.b.prettify())

*# <b>*

*# <![CDATA[A CDATA block]]>*

*# </b>*

# 遍历文档树

还拿”爱丽丝梦游仙境”的文档来做例子:

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)

通过这段例子来演示怎样从文档的一段内容找到另一段内容

## 子节点

一个Tag可能包含多个字符串或其它的Tag,这些都是这个Tag的子节点.Beautiful Soup提供了许多操作和遍历子节点的属性.

注意: Beautiful Soup中字符串节点不支持这些属性,因为字符串没有子节点

### tag的名字

操作文档树最简单的方法就是告诉它你想获取的tag的name.如果想获取 <head> 标签,只要用 soup.head :

soup.head

*# <head><title>The Dormouse's story</title></head>*

soup.title

*# <title>The Dormouse's story</title>*

这是个获取tag的小窍门,可以在文档树的tag中多次调用这个方法.下面的代码可以获取<body>标签中的第一个<b>标签:

soup.body.b

*# <b>The Dormouse's story</b>*

通过点取属性的方式只能获得当前名字的第一个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>标签,或是通过名字得到比一个tag更多的内容的时候,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 属性可以将tag的子节点以列表的方式输出:

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 对象本身一定会包含子节点,也就是说<html>标签也是 BeautifulSoup 对象的子节点:

len(soup.contents)

*# 1*

soup.contents[0].name

*# u'html'*

字符串没有 .contents 属性,因为字符串没有子节点:

text = title\_tag.contents[0]

text.contents

*# AttributeError: 'NavigableString' object has no attribute 'contents'*

通过tag的 .children 生成器,可以对tag的子节点进行循环:

**for** child **in** title\_tag.children:

**print**(child)

*# The Dormouse's story*

### .descendants

.contents 和 .children 属性仅包含tag的直接子节点.例如,<head>标签只有一个直接子节点<title>

head\_tag.contents

*# [<title>The Dormouse's story</title>]*

但是<title>标签也包含一个子节点:字符串 “The Dormouse’s story”,这种情况下字符串 “The Dormouse’s story”也属于<head>标签的子孙节点. .descendants 属性可以对所有tag的子孙节点进行递归循环 [[5]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id86) :

**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>标签只有一个子节点,但是有2个子孙节点:<head>节点和<head>的子节点, BeautifulSoup 有一个直接子节点(<html>节点),却有很多子孙节点:

len(list(soup.children))

*# 1*

len(list(soup.descendants))

*# 25*

### .string

如果tag只有一个 NavigableString 类型子节点,那么这个tag可以使用 .string 得到子节点:

title\_tag.string

*# u'The Dormouse's story'*

如果一个tag仅有一个子节点,那么这个tag也可以使用 .string 方法,输出结果与当前唯一子节点的 .string 结果相同:

head\_tag.contents

*# [<title>The Dormouse's story</title>]*

head\_tag.string

*# u'The Dormouse's story'*

如果tag包含了多个子节点,tag就无法确定 .string 方法应该调用哪个子节点的内容, .string 的输出结果是 None :

**print**(soup.html.string)

*# None*

### .strings 和 stripped\_strings

如果tag中包含多个字符串 [[2]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id83) ,可以使用 .strings 来循环获取:

**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'*

输出的字符串中可能包含了很多空格或空行,使用 .stripped\_strings 可以去除多余空白内容:

**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'...'*

全部是空格的行会被忽略掉,段首和段末的空白会被删除

## 父节点

继续分析文档树,每个tag或字符串都有父节点:被包含在某个tag中

### .parent

通过 .parent 属性来获取某个元素的父节点.在例子“爱丽丝”的文档中,<head>标签是<title>标签的父节点:

title\_tag = soup.title

title\_tag

*# <title>The Dormouse's story</title>*

title\_tag.parent

*# <head><title>The Dormouse's story</title></head>*

文档title的字符串也有父节点:<title>标签

title\_tag.string.parent

*# <title>The Dormouse's story</title>*

文档的顶层节点比如<html>的父节点是 BeautifulSoup 对象:

html\_tag = soup.html

type(html\_tag.parent)

*# <class 'bs4.BeautifulSoup'>*

BeautifulSoup 对象的 .parent 是None:

**print**(soup.parent)

*# None*

### .parents

通过元素的 .parents 属性可以递归得到元素的所有父辈节点,下面的例子使用了 .parents 方法遍历了<a>标签到根节点的所有节点.

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*

## 兄弟节点

看一段简单的例子:

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>*

因为<b>标签和<c>标签是同一层:他们是同一个元素的子节点,所以<b>和<c>可以被称为兄弟节点.一段文档以标准格式输出时,兄弟节点有相同的缩进级别.在代码中也可以使用这种关系.

### .next\_sibling 和 .previous\_sibling

在文档树中,使用 .next\_sibling 和 .previous\_sibling 属性来查询兄弟节点:

sibling\_soup.b.next\_sibling

*# <c>text2</c>*

sibling\_soup.c.previous\_sibling

*# <b>text1</b>*

<b>标签有 .next\_sibling 属性,但是没有 .previous\_sibling 属性,因为<b>标签在同级节点中是第一个.同理,<c>标签有 .previous\_sibling 属性,却没有 .next\_sibling 属性:

**print**(sibling\_soup.b.previous\_sibling)

*# None*

**print**(sibling\_soup.c.next\_sibling)

*# None*

例子中的字符串“text1”和“text2”不是兄弟节点,因为它们的父节点不同:

sibling\_soup.b.string

*# u'text1'*

**print**(sibling\_soup.b.string.next\_sibling)

*# None*

实际文档中的tag的 .next\_sibling 和 .previous\_sibling 属性通常是字符串或空白. 看看“爱丽丝”文档:

<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>

如果以为第一个<a>标签的 .next\_sibling 结果是第二个<a>标签,那就错了,真实结果是第一个<a>标签和第二个<a>标签之间的顿号和换行符:

link = soup.a

link

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

link.next\_sibling

*# u',\n'*

第二个<a>标签是顿号的 .next\_sibling 属性:

link.next\_sibling.next\_sibling

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

### .next\_siblings 和 .previous\_siblings

通过 .next\_siblings 和 .previous\_siblings 属性可以对当前节点的兄弟节点迭代输出:

**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*

## 回退和前进

看一下“爱丽丝” 文档:

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

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

HTML解析器把这段字符串转换成一连串的事件: “打开<html>标签”,”打开一个<head>标签”,”打开一个<title>标签”,”添加一段字符串”,”关闭<title>标签”,”打开<p>标签”,等等.Beautiful Soup提供了重现解析器初始化过程的方法.

### .next\_element 和 .previous\_element

.next\_element 属性指向解析过程中下一个被解析的对象(字符串或tag),结果可能与 .next\_sibling 相同,但通常是不一样的.

这是“爱丽丝”文档中最后一个<a>标签,它的 .next\_sibling 结果是一个字符串,因为当前的解析过程 [[2]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id83) 因为当前的解析过程因为遇到了<a>标签而中断了:

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.'*

但这个<a>标签的 .next\_element 属性结果是在<a>标签被解析之后的解析内容,不是<a>标签后的句子部分,应该是字符串”Tillie”:

last\_a\_tag.next\_element

*# u'Tillie'*

这是因为在原始文档中,字符串“Tillie” 在分号前出现,解析器先进入<a>标签,然后是字符串“Tillie”,然后关闭</a>标签,然后是分号和剩余部分.分号与<a>标签在同一层级,但是字符串“Tillie”会被先解析.

.previous\_element 属性刚好与 .next\_element 相反,它指向当前被解析的对象的前一个解析对象:

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

通过 .next\_elements 和 .previous\_elements 的迭代器就可以向前或向后访问文档的解析内容,就好像文档正在被解析一样:

**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*

# 搜索文档树

Beautiful Soup定义了很多搜索方法,这里着重介绍2个: find() 和 find\_all() .其它方法的参数和用法类似,请读者举一反三.

再以“爱丽丝”文档作为例子:

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() 类似的方法可以查找到想要查找的文档内容

## 过滤器

介绍 find\_all() 方法前,先介绍一下过滤器的类型 [[3]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id84) ,这些过滤器贯穿整个搜索的API.过滤器可以被用在tag的name中,节点的属性中,字符串中或他们的混合中.

### 字符串

最简单的过滤器是字符串.在搜索方法中传入一个字符串参数,Beautiful Soup会查找与字符串完整匹配的内容,下面的例子用于查找文档中所有的<b>标签:

soup.find\_all('b')

*# [<b>The Dormouse's story</b>]*

如果传入字节码参数,Beautiful Soup会当作UTF-8编码,可以传入一段Unicode 编码来避免Beautiful Soup解析编码出错

### 正则表达式

如果传入正则表达式作为参数,Beautiful Soup会通过正则表达式的 match() 来匹配内容.下面例子中找出所有以b开头的标签,这表示<body>和<b>标签都应该被找到:

**import** **re**

**for** tag **in** soup.find\_all(re.compile("^b")):

**print**(tag.name)

*# body*

*# b*

下面代码找出所有名字中包含”t”的标签:

**for** tag **in** soup.find\_all(re.compile("t")):

**print**(tag.name)

*# html*

*# title*

### 列表

如果传入列表参数,Beautiful Soup会将与列表中任一元素匹配的内容返回.下面代码找到文档中所有<a>标签和<b>标签:

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,下面代码查找到所有的tag,但是不会返回字符串节点

**for** tag **in** soup.find\_all(True):

**print**(tag.name)

*# html*

*# head*

*# title*

*# body*

*# p*

*# b*

*# p*

*# a*

*# a*

*# a*

*# p*

### 方法

如果没有合适过滤器,那么还可以定义一个方法,方法只接受一个元素参数 [[4]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id85) ,如果这个方法返回 True 表示当前元素匹配并且被找到,如果不是则反回 False

下面方法校验了当前元素,如果包含 class 属性却不包含 id 属性,那么将返回 True:

**def** has\_class\_but\_no\_id(tag):

**return** tag.has\_attr('class') **and** **not** tag.has\_attr('id')

将这个方法作为参数传入 find\_all() 方法,将得到所有<p>标签:

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>]*

返回结果中只有<p>标签没有<a>标签,因为<a>标签还定义了”id”,没有返回<html>和<head>,因为<html>和<head>中没有定义”class”属性.

下面代码找到所有被文字包含的节点内容:

**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*

现在来了解一下搜索方法的细节

## find\_all()

find\_all( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_all() 方法搜索当前tag的所有tag子节点,并判断是否符合过滤器的条件.这里有几个例子:

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'*

有几个方法很相似,还有几个方法是新的,参数中的 text 和 id 是什么含义? 为什么 find\_all("p", "title") 返回的是CSS Class为”title”的<p>标签? 我们来仔细看一下 find\_all() 的参数

### name 参数

name 参数可以查找所有名字为 name 的tag,字符串对象会被自动忽略掉.

简单的用法如下:

soup.find\_all("title")

*# [<title>The Dormouse's story</title>]*

重申: 搜索 name 参数的值可以使任一类型的 [过滤器](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id25) ,字符窜,正则表达式,列表,方法或是 True .

### keyword 参数

如果一个指定名字的参数不是搜索内置的参数名,搜索时会把该参数当作指定名字tag的属性来搜索,如果包含一个名字为 id 的参数,Beautiful Soup会搜索每个tag的”id”属性.

soup.find\_all(id='link2')

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

如果传入 href 参数,Beautiful Soup会搜索每个tag的”href”属性:

soup.find\_all(href=re.compile("elsie"))

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

搜索指定名字的属性时可以使用的参数值包括 [字符串](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id27) , [正则表达式](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id28) , [列表](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id29), [True](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#true) .

下面的例子在文档树中查找所有包含 id 属性的tag,无论 id 的值是什么:

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>]*

使用多个指定名字的参数可以同时过滤tag的多个属性:

soup.find\_all(href=re.compile("elsie"), id='link1')

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

有些tag属性在搜索不能使用,比如HTML5中的 data-\* 属性:

data\_soup = BeautifulSoup('<div data-foo="value">foo!</div>')

data\_soup.find\_all(data-foo="value")

*# SyntaxError: keyword can't be an expression*

但是可以通过 find\_all() 方法的 attrs 参数定义一个字典参数来搜索包含特殊属性的tag:

data\_soup.find\_all(attrs={"data-foo": "value"})

*# [<div data-foo="value">foo!</div>]*

### 按CSS搜索

按照CSS类名搜索tag的功能非常实用,但标识CSS类名的关键字 class 在Python中是保留字,使用 class 做参数会导致语法错误.从Beautiful Soup的4.1.1版本开始,可以通过 class\_ 参数搜索有指定CSS类名的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\_ 参数同样接受不同类型的 过滤器 ,字符串,正则表达式,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 属性是 [多值属性](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id12) .按照CSS类名搜索tag时,可以分别搜索tag中的每个CSS类名:

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 属性时也可以通过CSS值完全匹配:

css\_soup.find\_all("p", class\_="body strikeout")

*# [<p class="body strikeout"></p>]*

完全匹配 class 的值时,如果CSS类名的顺序与实际不符,将搜索不到结果:

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 参数

通过 text 参数可以搜搜文档中的字符串内容.与 name 参数的可选值一样, text 参数接受 [字符串](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id27) , [正则表达式](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id28) , [列表](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id29), [True](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#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'...']

虽然 text 参数用于搜索字符串,还可以与其它参数混合使用来过滤tag.Beautiful Soup会找到 .string 方法与 text 参数值相符的tag.下面代码用来搜索内容里面包含“Elsie”的<a>标签:

soup.find\_all("a", text="Elsie")

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

### limit 参数

find\_all() 方法返回全部的搜索结构,如果文档树很大那么搜索会很慢.如果我们不需要全部结果,可以使用 limit 参数限制返回结果的数量.效果与SQL中的limit关键字类似,当搜索到的结果数量达到 limit 的限制时,就停止搜索返回结果.

文档树中有3个tag符合搜索条件,但结果只返回了2个,因为我们限制了返回数量:

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 参数

调用tag的 find\_all() 方法时,Beautiful Soup会检索当前tag的所有子孙节点,如果只想搜索tag的直接子节点,可以使用参数 recursive=False .

一段简单的文档:

<html>

<head>

<title>

The Dormouse's story

</title>

</head>

...

是否使用 recursive 参数的搜索结果:

soup.html.find\_all("title")

*# [<title>The Dormouse's story</title>]*

soup.html.find\_all("title", recursive=False)

*# []*

## 像调用 find\_all() 一样调用tag

find\_all() 几乎是Beautiful Soup中最常用的搜索方法,所以我们定义了它的简写方法. BeautifulSoup 对象和 tag 对象可以被当作一个方法来使用,这个方法的执行结果与调用这个对象的 find\_all() 方法相同,下面两行代码是等价的:

soup.find\_all("a")

soup("a")

这两行代码也是等价的:

soup.title.find\_all(text=True)

soup.title(text=True)

## find()

find( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_all() 方法将返回文档中符合条件的所有tag,尽管有时候我们只想得到一个结果.比如文档中只有一个<body>标签,那么使用 find\_all() 方法来查找<body>标签就不太合适, 使用 find\_all 方法并设置 limit=1 参数不如直接使用 find() 方法.下面两行代码是等价的:

soup.find\_all('title', limit=1)

*# [<title>The Dormouse's story</title>]*

soup.find('title')

*# <title>The Dormouse's story</title>*

唯一的区别是 find\_all() 方法的返回结果是值包含一个元素的列表,而 find() 方法直接返回结果.

find\_all() 方法没有找到目标是返回空列表, find() 方法找不到目标时,返回 None .

**print**(soup.find("nosuchtag"))

*# None*

soup.head.title 是 [tag的名字](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id17) 方法的简写.这个简写的原理就是多次调用当前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](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_parent( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

我们已经用了很大篇幅来介绍 find\_all() 和 find() 方法,Beautiful Soup中还有10个用于搜索的API.它们中的五个用的是与 find\_all() 相同的搜索参数,另外5个与 find() 方法的搜索参数类似.区别仅是它们搜索文档的不同部分.

记住: find\_all() 和 find() 只搜索当前节点的所有子节点,孙子节点等. find\_parents() 和 find\_parent() 用来搜索当前节点的父辈节点,搜索方法与普通tag的搜索方法相同,搜索文档搜索文档包含的内容. 我们从一个文档中的一个叶子节点开始:

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")

# []

文档中的一个<a>标签是是当前叶子节点的直接父节点,所以可以被找到.还有一个<p>标签,是目标叶子节点的间接父辈节点,所以也可以被找到.包含class值为”title”的<p>标签不是不是目标叶子节点的父辈节点,所以通过 find\_parents() 方法搜索不到.

find\_parent() 和 find\_parents() 方法会让人联想到 [.parent](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#parent) 和 [.parents](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#parents) 属性.它们之间的联系非常紧密.搜索父辈节点的方法实际上就是对 .parents 属性的迭代搜索.

## find\_next\_siblings() 合 find\_next\_sibling()

find\_next\_siblings( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_next\_sibling( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

这2个方法通过 [.next\_siblings](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#next-siblings-previous-siblings) 属性对当tag的所有后面解析 [[5]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id86) 的兄弟tag节点进行迭代, find\_next\_siblings() 方法返回所有符合条件的后面的兄弟节点, find\_next\_sibling() 只返回符合条件的后面的第一个tag节点.

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](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_previous\_sibling( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

这2个方法通过 [.previous\_siblings](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#next-siblings-previous-siblings) 属性对当前tag的前面解析 [[5]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id86) 的兄弟tag节点进行迭代, find\_previous\_siblings() 方法返回所有符合条件的前面的兄弟节点, find\_previous\_sibling() 方法返回第一个符合条件的前面的兄弟节点:

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](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_next( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

这2个方法通过 [.next\_elements](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#next-elements-previous-elements) 属性对当前tag的之后的 [[5]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id86) tag和字符串进行迭代, find\_all\_next() 方法返回所有符合条件的节点, find\_next() 方法返回第一个符合条件的节点:

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>*

第一个例子中,字符串 “Elsie”也被显示出来,尽管它被包含在我们开始查找的<a>标签的里面.第二个例子中,最后一个<p>标签也被显示出来,尽管它与我们开始查找位置的<a>标签不属于同一部分.例子中,搜索的重点是要匹配过滤器的条件,并且在文档中出现的顺序而不是开始查找的元素的位置.

## find\_all\_previous() 和 find\_previous()

find\_all\_previous( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

find\_previous( [name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) )

这2个方法通过 [.previous\_elements](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#next-elements-previous-elements) 属性对当前节点前面 [[5]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id86) 的tag和字符串进行迭代, find\_all\_previous() 方法返回所有符合条件的节点, find\_previous() 方法返回第一个符合条件的节点.

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") 返回了文档中的第一段(class=”title”的那段),但还返回了第二段,<p>标签包含了我们开始查找的<a>标签.不要惊讶,这段代码的功能是查找所有出现在指定<a>标签之前的<p>标签,因为这个<p>标签包含了开始的<a>标签,所以<p>标签一定是在<a>之前出现的.

## CSS选择器

Beautiful Soup支持大部分的CSS选择器 [[6]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id87) ,在 Tag 或 BeautifulSoup 对象的 .select() 方法中传入字符串参数,即可使用CSS选择器的语法找到tag:

soup.select("title")

*# [<title>The Dormouse's story</title>]*

soup.select("p nth-of-type(3)")

*# [<p class="story">...</p>]*

通过tag标签逐层查找:

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>]*

找到某个tag标签下的直接子标签 [[6]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id87) :

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")

*# []*

找到兄弟节点标签:

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>]*

通过CSS的类名查找:

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>]*

通过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>]*

通过是否存在某个属性来查找:

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>]*

通过属性的值来查找:

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>]*

通过语言设置来查找:

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>]*

对于熟悉CSS选择器语法的人来说这是个非常方便的方法.Beautiful Soup也支持CSS选择器API,如果你仅仅需要CSS选择器的功能,那么直接使用 lxml 也可以,而且速度更快,支持更多的CSS选择器语法,但Beautiful Soup整合了CSS选择器的语法和自身方便使用API.

# 修改文档树

Beautiful Soup的强项是文档树的搜索,但同时也可以方便的修改文档树

## 修改tag的名称和属性

在 [Attributes](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#attributes) 的章节中已经介绍过这个功能,但是再看一遍也无妨. 重命名一个tag,改变属性的值,添加或删除属性:

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

给tag的 .string 属性赋值,就相当于用当前的内容替代了原来的内容:

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: 如果当前的tag包含了其它tag,那么给它的 .string 属性赋值会覆盖掉原有的所有内容包括子tag

## append()

Tag.append() 方法想tag中添加内容,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内容到文档中也没问题,可以调用Python的 append() 方法或调用工厂方法 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']*

如果想要创建一段注释,或 NavigableString 的任何子类,将子类作为 new\_string() 方法的第二个参数传入:

**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.']*

# 这是Beautiful Soup 4.2.1 中新增的方法

创建一个tag最好的方法是调用工厂方法 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>*

第一个参数作为tag的name,是必填,其它参数选填

## insert()

Tag.insert() 方法与 Tag.append() 方法类似,区别是不会把新元素添加到父节点 .contents 属性的最后,而是把元素插入到指定的位置.与Python列表总的 .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() 方法在当前tag或文本节点前插入内容:

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() 方法在当前tag或文本节点后插入内容:

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() 方法移除当前tag的内容:

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() 方法将当前tag移除文档树,并作为方法结果返回:

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

这个方法实际上产生了2个文档树: 一个是用来解析原始文档的 BeautifulSoup 对象,另一个是被移除并且返回的tag.被移除并返回的tag可以继续调用 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() 方法将当前节点移除文档树并完全销毁:

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() 方法移除文档树中的某段内容,并用新tag或文本节点替代它:

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或文本节点,可以用来浏览或添加到文档树其它地方

## wrap()

PageElement.wrap() 方法可以对指定的tag元素进行包装 [[8]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id89) ,并返回包装后的结果:

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>*

该方法在 Beautiful Soup 4.0.5 中添加

## unwrap()

Tag.unwrap() 方法与 wrap() 方法相反.将移除tag内的所有tag标签,该方法常被用来进行标记的解包:

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>*

与 replace\_with() 方法相同, unwrap() 方法返回被移除的tag

# 输出

## 格式化输出

prettify() 方法将Beautiful Soup的文档树格式化后以Unicode编码输出,每个XML/HTML标签都独占一行

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 对象和它的tag节点都可以调用 prettify() 方法:

**print**(soup.a.prettify())

*# <a href="http://example.com/">*

*# I linked to*

*# <i>*

*# example.com*

*# </i>*

*# </a>*

## 压缩输出

如果只想得到结果字符串,不重视格式,那么可以对一个 BeautifulSoup 对象或 Tag 对象使用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编码的字符串,可以指定 [编码](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id51) 的设置.

还可以调用 encode() 方法获得字节码或调用 decode() 方法获得Unicode.

## 输出格式

Beautiful Soup输出是会将HTML中的特殊字符转换成Unicode,比如“&lquot;”:

soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")

unicode(soup)

*# u'<html><head></head><body>\u201cDammit!\u201d he said.</body></html>'*

如果将文档转换成字符串,Unicode编码会被编码成UTF-8.这样就无法正确显示HTML特殊字符了:

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中包含的文本内容,那么可以嗲用 get\_text() 方法,这个方法获取到tag中包含的所有文版内容包括子孙tag中的内容,并将结果作为Unicode字符串返回:

markup = '<a href="http://example.com/">**\n**I linked to <i>example.com</i>**\n**</a>'

soup = BeautifulSoup(markup)

soup.get\_text()

u'**\n**I linked to example.com**\n**'

soup.i.get\_text()

u'example.com'

可以通过参数指定tag的文本内容的分隔符:

*# soup.get\_text("|")*

u'**\n**I linked to |example.com|**\n**'

还可以去除获得文本内容的前后空白:

*# soup.get\_text("|", strip=True)*

u'I linked to|example.com'

或者使用 [.stripped\_strings](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#strings-stripped-strings) 生成器,获得文本列表后手动处理列表:

[text **for** text **in** soup.stripped\_strings]

*# [u'I linked to', u'example.com']*

# 指定文档解析器

如果仅是想要解析HTML文档,只要用文档创建 BeautifulSoup 对象就可以了.Beautiful Soup会自动选择一个解析器来解析文档.但是还可以通过参数指定使用那种解析器来解析当前文档.

BeautifulSoup 第一个参数应该是要被解析的文档字符串或是文件句柄,第二个参数用来标识怎样解析文档.如果第二个参数为空,那么Beautiful Soup根据当前系统安装的库自动选择解析器,解析器的优先数序: lxml, html5lib, Python标准库.在下面两种条件下解析器优先顺序会变化:

* 要解析的文档是什么类型: 目前支持, “html”, “xml”, 和 “html5”
* 指定使用哪种解析器: 目前支持, “lxml”, “html5lib”, 和 “html.parser”

[安装解析器](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id9) 章节介绍了可以使用哪种解析器,以及如何安装.

如果指定的解析器没有安装,Beautiful Soup会自动选择其它方案.目前只有 lxml 解析器支持XML文档的解析,在没有安装lxml库的情况下,创建 beautifulsoup对象时无论是否指定使用lxml,都无法得到解析后的对象

## 解析器之间的区别

Beautiful Soup为不同的解析器提供了相同的接口,但解析器本身时有区别的.同一篇文档被不同的解析器解析后可能会生成不同结构的树型文档.区别最大的是HTML解析器和XML解析器,看下面片段被解析成HTML结构:

BeautifulSoup("<a><b /></a>")

*# <html><head></head><body><a><b></b></a></body></html>*

因为空标签<b />不符合HTML标准,所以解析器把它解析成<b></b>

同样的文档使用XML解析如下(解析XML需要安装lxml库).注意,空标签<b />依然被保留,并且文档前添加了XML头,而不是被包含在<html>标签内:

BeautifulSoup("<a><b /></a>", "xml")

*# <?xml version="1.0" encoding="utf-8"?>*

*# <a><b/></a>*

HTML解析器之间也有区别,如果被解析的HTML文档是标准格式,那么解析器之间没有任何差别,只是解析速度不同,结果都会返回正确的文档树.

但是如果被解析文档不是标准格式,那么不同的解析器返回结果可能不同.下面例子中,使用lxml解析错误格式的文档,结果</p>标签被直接忽略掉了:

BeautifulSoup("<a></p>", "lxml")

*# <html><body><a></a></body></html>*

使用html5lib库解析相同文档会得到不同的结果:

BeautifulSoup("<a></p>", "html5lib")

*# <html><head></head><body><a><p></p></a></body></html>*

html5lib库没有忽略掉</p>标签,而是自动补全了标签,还给文档树添加了<head>标签.

使用pyhton内置库解析结果如下:

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

*# <a></a>*

与lxml [[7]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id88) 库类似的,Python内置库忽略掉了</p>标签,与html5lib库不同的是标准库没有尝试创建符合标准的文档格式或将文档片段包含在<body>标签内,与lxml不同的是标准库甚至连<html>标签都没有尝试去添加.

因为文档片段“<a></p>”是错误格式,所以以上解析方式都能算作”正确”,html5lib库使用的是HTML5的部分标准,所以最接近”正确”.不过所有解析器的结构都能够被认为是”正常”的.

不同的解析器可能影响代码执行结果,如果在分发给别人的代码中使用了 BeautifulSoup ,那么最好注明使用了哪种解析器,以减少不必要的麻烦.

# 编码

任何HTML或XML文档都有自己的编码方式,比如ASCII 或 UTF-8,但是使用Beautiful Soup解析后,文档都被转换成了Unicode:

markup = "<h1>Sacr**\xc3\xa9** bleu!</h1>"

soup = BeautifulSoup(markup)

soup.h1

*# <h1>Sacré bleu!</h1>*

soup.h1.string

*# u'Sacr\xe9 bleu!'*

这不是魔术(但很神奇),Beautiful Soup用了 [编码自动检测](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#unicode-dammit) 子库来识别当前文档编码并转换成Unicode编码. BeautifulSoup 对象的 .original\_encoding 属性记录了自动识别编码的结果:

soup.original\_encoding

'utf-8'

[编码自动检测](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#unicode-dammit) 功能大部分时候都能猜对编码格式,但有时候也会出错.有时候即使猜测正确,也是在逐个字节的遍历整个文档后才猜对的,这样很慢.如果预先知道文档编码,可以设置编码参数来减少自动检查编码出错的概率并且提高文档解析速度.在创建 BeautifulSoup 对象的时候设置 from\_encoding 参数.

下面一段文档用了ISO-8859-8编码方式,这段文档太短,结果Beautiful Soup以为文档是用ISO-8859-7编码:

markup = b"<h1>\xed\xe5\xec\xf9</h1>"

soup = BeautifulSoup(markup)

soup.h1

<h1>νεμω</h1>

soup.original\_encoding

'ISO-8859-7'

通过传入 from\_encoding 参数来指定编码方式:

soup = BeautifulSoup(markup, from\_encoding="iso-8859-8")

soup.h1

<h1>םולש</h1>

soup.original\_encoding

'iso8859-8'

少数情况下(通常是UTF-8编码的文档中包含了其它编码格式的文件),想获得正确的Unicode编码就不得不将文档中少数特殊编码字符替换成特殊Unicode编码,“REPLACEMENT CHARACTER” (U+FFFD, �) [[9]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id90) . 如果Beautifu Soup猜测文档编码时作了特殊字符的替换,那么Beautiful Soup会把 UnicodeDammit 或 BeautifulSoup 对象的 .contains\_replacement\_characters 属性标记为 True .这样就可以知道当前文档进行Unicode编码后丢失了一部分特殊内容字符.如果文档中包含�而 .contains\_replacement\_characters 属性是 False ,则表示�就是文档中原来的字符,不是转码失败.

## 输出编码

通过Beautiful Soup输出文档时,不管输入文档是什么编码方式,输出编码均为UTF-8编码,下面例子输入文档是Latin-1编码:

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,输出文档中的<meta>标签的编码设置已经修改成了与输出编码一致的UTF-8.

如果不想用UTF-8编码输出,可以将编码方式传入 prettify() 方法:

**print**(soup.prettify("latin-1"))

*# <html>*

*# <head>*

*# <meta content="text/html; charset=latin-1" http-equiv="Content-type" />*

*# ...*

还可以调用 BeautifulSoup 对象或任意节点的 encode() 方法,就像Python的字符串调用 encode() 方法一样:

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

*# '<p>Sacr\xe9 bleu!</p>'*

soup.p.encode("utf-8")

*# '<p>Sacr\xc3\xa9 bleu!</p>'*

如果文档中包含当前编码不支持的字符,那么这些字符将呗转换成一系列XML特殊字符引用,下面例子中包含了Unicode编码字符SNOWMAN:

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

snowman\_soup = BeautifulSoup(markup)

tag = snowman\_soup.b

SNOWMAN字符在UTF-8编码中可以正常显示(看上去像是☃),但有些编码不支持SNOWMAN字符,比如ISO-Latin-1或ASCII,那么在这些编码中SNOWMAN字符会被转换成“&#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!)

[编码自动检测](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#unicode-dammit) 功能可以在Beautiful Soup以外使用,检测某段未知编码时,可以使用这个方法:

**from** **bs4** **import** UnicodeDammit

dammit = UnicodeDammit("Sacr**\xc3\xa9** bleu!")

**print**(dammit.unicode\_markup)

*# Sacré bleu!*

dammit.original\_encoding

*# 'utf-8'*

如果Python中安装了 chardet 或 cchardet 那么编码检测功能的准确率将大大提高.输入的字符越多,检测结果越精确,如果事先猜测到一些可能编码,那么可以将猜测的编码作为参数,这样将优先检测这些编码:

dammit = UnicodeDammit("Sacr**\xe9** bleu!", ["latin-1", "iso-8859-1"])

**print**(dammit.unicode\_markup)

*# Sacré bleu!*

dammit.original\_encoding

*# 'latin-1'*

[编码自动检测](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#unicode-dammit) 功能中有2项功能是Beautiful Soup库中用不到的

### 智能引号

使用Unicode时,Beautiful Soup还会智能的把引号 [[10]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id91) 转换成HTML或XML中的特殊字符:

markup = b"<p>I just **\x93**love**\x94** Microsoft Word**\x92**s 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>'*

也可以把引号转换为ASCII码:

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没有使用这种方式.默认情况下,Beautiful Soup把引号转换成Unicode:

UnicodeDammit(markup, ["windows-1252"]).unicode\_markup

*# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'*

### 矛盾的编码

有时文档的大部分都是用UTF-8,但同时还包含了Windows-1252编码的字符,就像微软的智能引号 [[10]](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id91) 一样.一些包含多个信息的来源网站容易出现这种情况. UnicodeDammit.detwingle() 方法可以把这类文档转换成纯UTF-8编码格式,看个简单的例子:

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")

这段文档很杂乱,snowmen是UTF-8编码,引号是Windows-1252编码,直接输出时不能同时显示snowmen和引号,因为它们编码不同:

**print**(doc)

*# ☃☃☃�I like snowmen!�*

**print**(doc.decode("windows-1252"))

*# â˜ƒâ˜ƒâ˜ƒ“I like snowmen!”*

如果对这段文档用UTF-8解码就会得到 UnicodeDecodeError 异常,如果用Windows-1252解码就回得到一堆乱码.幸好, UnicodeDammit.detwingle() 方法会吧这段字符串转换成UTF-8编码,允许我们同时显示出文档中的snowmen和引号:

new\_doc = UnicodeDammit.detwingle(doc)

**print**(new\_doc.decode("utf8"))

*# ☃☃☃“I like snowmen!”*

UnicodeDammit.detwingle() 方法只能解码包含在UTF-8编码中的Windows-1252编码内容,但这解决了最常见的一类问题.

在创建 BeautifulSoup 或 UnicodeDammit 对象前一定要先对文档调用 UnicodeDammit.detwingle() 确保文档的编码方式正确.如果尝试去解析一段包含Windows-1252编码的UTF-8文档,就会得到一堆乱码,比如: â˜ƒâ˜ƒâ˜ƒ“I like snowmen!”.

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

# 解析部分文档

如果仅仅因为想要查找文档中的<a>标签而将整片文档进行解析,实在是浪费内存和时间.最快的方法是从一开始就把<a>标签以外的东西都忽略掉. SoupStrainer 类可以定义文档的某段内容,这样搜索文档时就不必先解析整篇文档,只会解析在 SoupStrainer 中定义过的文档. 创建一个 SoupStrainer 对象并作为 parse\_only 参数给 BeautifulSoup 的构造方法即可.

## SoupStrainer

SoupStrainer 类接受与典型搜索方法相同的参数：[name](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id32) , [attrs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#css) , [recursive](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#recursive) , [text](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#text) , [\*\*kwargs](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#keyword) 。下面举例说明三种 SoupStrainer 对象：

**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“爱丽丝”文档来举例，来看看使用三种 SoupStrainer 对象做参数会有什么不同:

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*

*# ...*

*#*

还可以将 SoupStrainer 作为参数传入 [搜索文档树](https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/#id24) 中提到的方法.这可能不是个常用用法,所以还是提一下:

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']*