



- » [Документации](#)
- » [BeautifulSoup](#)
- » [Книги](#)
- » [Документация](#)
- » [Пакеты](#)
- » [ПоследниеИзменения](#)
- » [НайтиСтраницу](#)
- » [ПомощьПоГлавам](#)
- » [BeautifulSoup](#)

Страница


- » [Неизменяемая страница](#)
- » [Информация](#)
- » [Прикреплённые файлы](#)
- » ▼

Пользователь

- » [Войти](#)

Beautiful Soup



 [Beautiful Soup](#) - это парсер для синтаксического разбора файлов HTML/XML, написанный на языке программирования Python, который может преобразовать даже неправильную разметку в дерево синтаксического разбора. Он поддерживает простые и естественные способы навигации, поиска и модификации дерева синтаксического разбора. В большинстве случаев он поможет программисту сэкономить часы и дни работы. Написанный на языке программирования Ruby порт называется Rubyful Soup.

Данный документ иллюстрирует основные возможности BeautifulSoup версии 3.0 на примерах. Вы увидите, для чего лучше использовать данную библиотеку, как она работает, как ее использовать, как добиться необходимых вам результатов и что делать, когда она не оправдывает ваших ожиданий.

Содержание

1. [Быстрый старт](#)
2. [Синтаксический разбор документа](#)

1. [Синтаксический разбор HTML](#)
2. [Синтаксический разбор XML](#)
3. [Если это не работает](#)
3. [Beautiful Soup дает тебе Unicode, черт побери](#)
4. [Печать документа](#)
5. [Дерево синтаксического разбора](#)
 1. [Атрибуты `Tag`-ов](#)
6. [Навигация по дереву синтаксического разбора](#)
 1. [`parent`](#)
 2. [`contents`](#)
 3. [`string`](#)
 4. [`nextSibling` и `previousSibling`](#)
 5. [`next` и `previous`](#)
 6. [Итерации над объектом `Tag`](#)
 7. [Используем имена тегов как элементы](#)
7. [Поиск в дереве синтаксического разбора](#)
 1. [Основной метод поиска: `findAll\(name, attrs, recursive, text, limit, **kwargs\)`](#)
 1. [Поиск класса CSS](#)
 2. [Вызов тега аналогично вызову `findall`](#)
 2. [`find\(name, attrs, recursive, text, **kwargs\)`](#)
 3. [Что произошло с методом `first`?](#)
8. [Поиск внутри дерева синтаксического разбора](#)
 1. [`findNextSiblings\(name, attrs, text, limit, **kwargs\)` и `findNextSibling\(name, attrs, text, **kwargs\)`](#)
 2. [`findPreviousSiblings\(name, attrs, text, limit, **kwargs\)` и `findPreviousSibling\(name, attrs, text, **kwargs\)`](#)
 3. [`findAllNext\(name, attrs, text, limit, **kwargs\)` и `findNext\(name, attrs, text, **kwargs\)`](#)
 4. [`findAllPrevious\(name, attrs, text, limit, **kwargs\)` и `findPrevious\(name, attrs, text, **kwargs\)`](#)
 5. [`findParents\(name, attrs, limit, **kwargs\)` и `findParent\(name, attrs, **kwargs\)`](#)
9. [Модификация дерева синтаксического разбора](#)
 1. [Изменение значений атрибутов](#)
 2. [Удаление элементов](#)
 3. [Замена одного элемента на другой](#)
 4. [Добавление другого нового элемента](#)
10. [Известные проблемы](#)
 1. [Почему BeautifulSoup не может вывести на экран не-ASCII символы, которые я передал ему?](#)
 2. [Beautiful Soup потерял данные, которые я ему передал! Почему? ПОЧЕМУ?????](#)
 3. [Синтаксическое дерево разбора, построенное классом `BeautifulSoup`, меня раздражает!](#)
 4. [Beautiful Soup слишком медленно работает!](#)
11. [Дополнительные темы](#)
 1. [Генераторы](#)
 2. [Другие встроенные парсеры](#)
 3. [Настраиваем парсер](#)
 4. [Преобразование сущностей](#)
 5. [Очистка от плохих данных с помощью регулярных выражений](#)
 6. [Наслаждаемся `SoupStrainer`-ми](#)
 7. [Улучшаем производительность за счет синтаксического разбора только части документа](#)
 8. [Улучшаем использование памяти при помощи `extract`](#)
12. [Смотрите также](#)
 1. [Приложения, использующие BeautifulSoup](#)
 2. [Похожие библиотеки](#)
13. [Заключение](#)

Быстрый старт

Скачать BeautifulSoup можно [здесь](#). [Список изменений](#) содержит отличия версии 3.0 от более ранних.

Подключить BeautifulSoup к вашему приложению можно с помощью одной из ниже приведенных строк:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup          # Для обработки HTML
from BeautifulSoup import BeautifulSoup         # Для обработки XML
import BeautifulSoup                             # Для обработки и того
и другого
```

Следующий код демонстрирует основные возможности BeautifulSoup. Можете скопировать его в сессию Python и запустить.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
import re

doc = ['<html><head><title>Page title</title></head>',
      '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
      '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
      '</html>']
soup = BeautifulSoup(''.join(doc))

print soup.prettify()
# <html>
# <head>
# <title>
#   Page title
# </title>
# </head>
# <body>
# <p id="firstpara" align="center">
#   This is paragraph
#   <b>
#     one
#   </b>
#   .
# </p>
# <p id="secondpara" align="blah">
#   This is paragraph
#   <b>
#     two
```

```
# </b>
# .
# </p>
# </body>
# </html>
```

Продemonстрируем несколько способов навигации по супу:

Переключить отображение номеров строк

```
soup.contents[0].name
# u'html'

soup.contents[0].contents[0].name
# u'head'

head = soup.contents[0].contents[0]
head.parent.name
# u'html'

head.next
# <title>Page title</title>

head.nextSibling.name
# u'body'

head.nextSibling.contents[0]
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>

head.nextSibling.contents[0].nextSibling
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>
```

А вот как искать в супе определенные теги или теги с заданными атрибутами:

Переключить отображение номеров строк

```
titleTag = soup.html.head.title
titleTag
# <title>Page title</title>

titleTag.string
# u'Page title'

len(soup('p'))
# 2

soup.findAll('p', align="center")
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]

soup.find('p', align="center")
```

```
# <p id="firstpara" align="center">This is paragraph <b>one</b>. </p>

soup('p', align="center")[0]['id']
# u'firstpara'

soup.find('p', align=re.compile('^b.*'))['id']
# u'secondpara'

soup.find('p').b.string
# u'one'

soup('p')[1].b.string
# u'two'
```

Изменять суп также весьма просто:

Переключить отображение номеров строк

```
titleTag['id'] = 'theTitle'
titleTag.contents[0].replaceWith("New title")
soup.html.head
# <head><title id="theTitle">New title</title></head>

soup.p.extract()
soup.prettify()
# <html>
# <head>
# <title id="theTitle">
#   New title
# </title>
# </head>
# <body>
# <p id="secondpara" align="blah">
#   This is paragraph
#   <b>
#     two
#   </b>
#   .
# </p>
# </body>
# </html>

soup.p.replaceWith(soup.b)
# <html>
# <head>
# <title id="theTitle">
#   New title
# </title>
# </head>
# <body>
```

```
# <b>
# two
# </b>
# </body>
# </html>

soup.body.insert(0, "This page used to have ")
soup.body.insert(2, " &lt;p&gt; tags!")
soup.body
# <body>This page used to have <b>two</b> &lt;p&gt; tags!</body>
```

Приведем реальный пример. Скачаем [🌐 ICC Commercial Crime Services weekly piracy report](#), произведем его синтаксический разбор с помощью BeautifulSoup и выведем на экран сообщения о случаях пиратства (piracy incidents):

Переключить отображение номеров строк

```
import urllib2
from BeautifulSoup import BeautifulSoup

page = urllib2.urlopen("http://www.icc-ccs.org/prc/piracyreport.php")
soup = BeautifulSoup(page)
for incident in soup('td', width="90%"):
    where, linebreak, what = incident.contents[:3]
    print where.strip()
    print what.strip()
    print
```

Синтаксический разбор документа

Для работы конструктору BeautifulSoup требуется документ XML или HTML в виде строки (или открытого файлоподобного объекта). Он произведет синтаксический разбор и создаст в памяти структуры данных, соответствующие документу.

Если обработать с помощью BeautifulSoup хорошо оформленный документ, то разобранный документ будет выглядеть также как и исходный документ. Но если его разметка будет содержать ошибки, то BeautifulSoup использует эвристические методы для построения наиболее подходящей структуры данных.

Синтаксический разбор HTML

Используйте класс BeautifulSoup для синтаксического разбора документа HTML. Несколько фактов, которые необходимо знать о BeautifulSoup:

Некоторые теги могут быть вложенными (<BLOCKQUOTE>), а некоторые - нет (<P>). Таблицы и списки тегов имеют естественный порядок вложенности. Например, теги <TD> появляются только в обрамлении тегов <TR> и никак иначе. Содержимое тега <SCRIPT> не будет участвовать в разборе HTML. Тег <META> может определять кодировку документа. Вот как это работает:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
html = "<html><p>Para 1<p>Para 2<blockquote>Quote 1<blockquote>Quote 2"
soup = BeautifulSoup(html)
print soup.prettify()
# <html>
#   <p>
#     Para 1
#   </p>
#   <p>
#     Para 2
#     <blockquote>
#       Quote 1
#     <blockquote>
#       Quote 2
#     </blockquote>
#   </blockquote>
# </p>
# </html>
```

Обратите внимание на то, что BeautifulSoup вычисляет наиболее вероятные места для закрывающих тегов, даже если они отсутствуют в исходном документе.

Хотя приведенный документ HTML является невалидным, с ним все же можно работать. А вот действительно отвратительный документ. Помимо всего прочего он содержит тег <FORM>, который начинается вне тега <TABLE>, но закрывается внутри тега <TABLE>. (Пример такого HTML был найден на веб-сайте одной из ведущих веб-компаний.)

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
html = """
<html>
<form>
  <table>
    <td><input name="input1">Row 1 cell 1
    <tr><td>Row 2 cell 1
  </form>
  <td>Row 2 cell 2<br>This</br> sure is a long cell
</body>
</html>"""
```

Beautiful Soup справится с обработкой и такого документа:

Переключить отображение номеров строк

```
print BeautifulSoup(html).prettify()
# <html>
#   <form>
```

```
# <table>
# <td>
#   <input name="input1" />
#   Row 1 cell 1
# </td>
# <tr>
#   <td>
#     Row 2 cell 1
#   </td>
# </tr>
# </table>
# </form>
# <td>
#   Row 2 cell 2
#   <br />
#   This
#   sure is a long cell
# </td>
# </html>
```

Последняя ячейка таблицы находится вне тега <TABLE>; BeautifulSoup решил закрыть тег <TABLE> там, где закрыт тег <FORM>. Автор исходного документа планировал, вероятно, продлить действие тега <FORM> до конца таблицы, но BeautifulSoup не сможет об этом догадаться. Даже в таком необычном случае BeautifulSoup произведет синтаксический разбор и предоставит вам доступ ко всем данным документа.

Синтаксический разбор XML

Класс BeautifulSoup содержит эвристики, полностью аналогичные применяющимся в веб-браузерах, что позволяет делать предположения о замыслах авторов HTML файлов. Но в XML нет фиксированного порядка тегов такие эвристики здесь не пригодятся. Поэтому BeautifulSoup не сможет хорошо работать с XML.

Используйте класс BeautifulSoup для синтаксического разбора документов XML. Это основной класс, не требующий знания диалекта XML и имеющий очень простые правила о вложенности тегов: Вот он в действии:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
xml = "<doc><tag1>Contents 1<tag2>Contents 2<tag1>Contents 3"
soup = BeautifulSoup(xml)
print soup.prettify()
# <doc>
# <tag1>
#   Contents 1
# <tag2>
#   Contents 2
# </tag2>
# </tag1>
```



```
# <tag1>
#   Contents 3
# </tag1>
# </doc>
```

Одним из общеизвестных недостатков BeautifulSoup является то, что он не знает о самозакрывающихся тегах. В HTML имеется фиксированный набор самозакрывающихся тегов, но в случае с XML все зависит от того, что записано в DTD. Вы можете сообщить BeautifulSoup, что определенные теги являются самозакрывающимися, передав их имена через аргумент конструктора `selfClosingTags`:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
xml = "<tag>Text 1<selfclosing>Text 2"
print BeautifulSoup(xml).prettify()
# <tag>
#   Text 1
#   <selfclosing>
#     Text 2
#   </selfclosing>
# </tag>

print BeautifulSoup(xml, selfClosingTags=
['selfclosing']).prettify()
# <tag>
#   Text 1
#   <selfclosing />
#   Text 2
# </tag>
```

Если это не работает

Имеется несколько [других классов парсеров](#), эвристики которых отличаются от двух описанных выше. Также вы можете [создать подкласс и настроить парсер](#) и задать в нем свои собственные эвристики.

Beautiful Soup дает тебе Unicode, черт побери

Во время синтаксического разбора документа он перекодировается в Unicode. В своих структурах данных BeautifulSoup хранит только строки Unicode.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("Hello")
soup.contents[0]
# u'Hello'
soup.originalEncoding
# 'ascii'
```

Вот пример с японским документом в кодировке UTF-8:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf")
soup.contents[0]
# u'\u3053\u308c\u306f'
soup.originalEncoding
# 'utf-8'

str(soup)
# '\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf'

# Note: this bit uses EUC-JP, so it only works if you have cjkcodecs
# installed, or are running Python 2.4.
soup.__str__('euc-jp')
# '\xa4\xb3\xa4\xec\xa4\xcf'
```

Beautiful Soup использует класс с именем `UnicodeDammit` для определения кодировки передаваемых вами документов и перекодировки его в Unicode, не беспокойтесь об этом. Если это также необходимо для других документов (без их синтаксического разбора с помощью Beautiful Soup), то вы можете использовать `UnicodeDammit` отдельно. Он в значительной мере основан на коде из [🌐 Universal Feed Parser](#).

Если вы работаете с Python более ранних версий, чем 2.4, то заранее скачайте и установите [🌐 `cjkcodecs`](#) и [`iconvcodec`](#), которые добавляют в Python поддержку многих кодеков, особенно кодеков CJK. Для лучшего автоопределения кодировки установите также библиотеку [🌐 `chardet`](#).

Перед перекодировкой в Unicode Beautiful Soup проверяет кодировки в следующем порядке:

- » Кодировка, переданная конструктору супа в параметре `fromEncoding`.
- » Кодировка, обнаруженная в самом документе: например, в декларации XML или (для документов HTML) в атрибуте `http-equiv` тега META. Как только Beautiful Soup обнаружит подобное указание о кодировке документа, он повторно приступит к синтаксическому разбору документа с самого начала, но уже применяя найденную кодировку. Избежать этого можно только явным заданием кодировки, которая будет работать: тогда любая найденная в документе кодировка будет игнорироваться.
- » Кодировка, вычисленная по нескольким первым байтам файла. Если кодировка определяется на этом этапе, то она будет либо UTF-*, либо EBCDIC, либо ASCII.
- » Кодировка, вычисленная библиотекой [🌐 `chardet`](#), в случае если она была установлена.
- » UTF-8
- » Windows-1252

Если Beautiful Soup сможет сделать предположение, то почти всегда оно будет верным. Но для документов без декларации или в неизвестной кодировке он не сможет сделать каких-либо предположений. В таком случае – скорее всего, ошибочно, – будет использоваться кодировка Windows-1252. Вот пример в кодировке EUC-JP, когда предположение Beautiful Soup о

кодировке будет ошибочным. (Кроме того, поскольку используется кодировка EUC-JP пример будет работать только под Python 2.4 или если установлен cjkcodecs):

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
euc_jp = '\xa4\xb3\xa4\xec\xa4\xcf'

soup = BeautifulSoup(euc_jp)
soup.originalEncoding
# 'windows-1252'

str(soup)
# '\xc2\xa4\xc2\xb3\xc2\xa4\xc3\xac\xc2\xa4\xc3\x8f'      # Неправильно!
```

Но если задать кодировку с помощью fromEncoding, то синтаксический разбор документа будет корректным и его можно будет перекодировать в UTF-8 или обратно – в EUC-JP.

Переключить отображение номеров строк

```
soup = BeautifulSoup(euc_jp, fromEncoding="euc-jp")
soup.originalEncoding
# 'windows-1252'

str(soup)
# '\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf'      # Правильно!

soup.__str__(self, 'euc-jp') == euc_jp
# True
```

Если обрабатывать с помощью BeautifulSoup документ в кодировке Windows-1252 (или подобных, например, ISO-8859-1 или ISO-8859-2), то BeautifulSoup обнаружит и уничтожит изящные кавычки и другие символы, специфичные для Windows. Чтобы этого избежать BeautifulSoup преобразует эти символы в сущности HTML (BeautifulSoup) или в сущности XML (BeautifulStoneSoup).

Чтобы этого избежать можно передать параметр smartQuotesTo=None в конструктор супа: тогда кавычки будут конвертироваться в Unicode также, как и другие символы данной кодировки. Для изменения поведения BeautifulSoup и BeautifulSoup можно передать в параметре smartQuotesTo значения "xml" или "html".

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, BeautifulSoup
text = "Deploy the \x91SMART QUOTES\x92!"

str(BeautifulSoup(text))
# 'Deploy the &lsquo;SMART QUOTES&rsquo;!'

str(BeautifulStoneSoup(text))
# 'Deploy the &#x2018;SMART QUOTES&#x2019;!'
```

```
str(BeautifulSoup(text, smartQuotesTo="xml"))
# 'Deploy the &#x2018;SMART QUOTES&#x2019;!'

BeautifulSoup(text, smartQuotesTo=None).contents[0]
# u'Deploy the \u2018SMART QUOTES\u2019!'
```

Печать документа

Документ BeautifulSoup (или любое их подмножество) можно превратить в строку с помощью функции `str` или методов `prettify` или `renderContents`. Также можно использовать функцию `unicode` для получения всего документа в виде строки Unicode.

Метод `prettify` добавляет значимые переводы строки и пробелы для придания структуре документа лучшей читабельности. Метод также удаляет текстовые узлы, состоящие только из пробелов, что может изменить смысл XML документа. Функции `str` и `unicode` не удаляют текстовые узлы, состоящие только из пробелов, и не добавляют пробелы между узлами.

Приведем пример.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = "<html><h1>Heading</h1><p>Text"
soup = BeautifulSoup(doc)

str(soup)
# '<html><h1>Heading</h1><p>Text</p></html>'
soup.renderContents()
# '<html><h1>Heading</h1><p>Text</p></html>'
soup.__str__()
# '<html><h1>Heading</h1><p>Text</p></html>'
unicode(soup)
# u'<html><h1>Heading</h1><p>Text</p></html>'

soup.prettify()
# '<html>\n <h1>\n  Heading\n </h1>\n <p>\n  Text\n </p>\n</html>'\n\nprint soup.prettify()
# <html>
#   <h1>
#     Heading
#   </h1>
#   <p>
#     Text
#   </p>
# </html>
```

Обратите внимание, что функции `str` и `renderContents` дают различный результат, когда используются для тегов внутри документа. Функция `str` печатает и теги и их содержимое, а

функция renderContents - только содержимое.

Переключить отображение номеров строк

```
heading = soup.h1
str(heading)
# '<h1>Heading</h1>'
heading.renderContents()
# 'Heading'
```

При вызове функций `__str__`, `prettify` или `renderContents`, вы можете задать кодировку вывода. Кодировкой по умолчанию (используется функцией `str`) является UTF-8. Вот пример разбора строки ISO-8859-1 с последующим выводом на экран той же строки в других кодировках:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = "Sacr\xe9 bleu!"
soup = BeautifulSoup(doc)
str(soup)
# 'Sacr\xc3\xa9 bleu!' # UTF-8
soup.__str__("ISO-8859-1")
# 'Sacr\xe9 bleu!'
soup.__str__("UTF-16")
# '\xff\xfeS\x00a\x00c\x00r\x00\xe9\x00 \x00b\x00l\x00e\x00u\x00!\x00'
soup.__str__("EUC-JP")
# 'Sacr\x8f\xab\xbl bleu!'
```

Если в исходном документе в декларации указывалась кодировка, то при обратном преобразовании в строку BeautifulSoup запишет в декларацию новую кодировку. Это означает, что если загрузить документ HTML в BeautifulSoup, а потом распечатать его, то HTML не только будет вычищен, но и прозрачно перекодирован в UTF-8.

Пример HTML:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = """<html>
<meta http-equiv="Content-type" content="text/html; charset=ISO-Latin-1" >
Sacr\xe9 bleu!
</html>"""

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

Пример XML:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = """<?xml version="1.0" encoding="ISO-Latin-1">Sacr\xe9 bleu!"""

print BeautifulSoup(doc).prettify()
# <?xml version='1.0' encoding='utf-8'>
# Sacré bleu!
```

Дерево синтаксического разбора

До сих пор мы рассматривали загрузку и запись документов. Однако, большую часть времени будет приковывать к себе дерево синтаксического разбора: структуры данных BeautifulSoup, которые создаются по мере синтаксического разбора документа.

Объект парсера (экземпляр класса BeautifulSoup или BeautifulSoup) обладает большой глубиной вложенности связанных структур данных, соответствующих структуре документа XML или HTML. Объект парсера состоит из объектов двух других типов: объектов Tag, которые соответствуют тегам, к примеру, тегу <TITLE> и тегу ; и объекты NavigableString, соответствующие таким строкам как "Page title" или "This is paragraph".

Класс NavigableString имеет несколько подклассов (CDATA, Comment, Declaration и ProcessingInstruction), которые соответствуют специальным конструкциям в XML. Они работают также как NavigableString-и, за исключением того, что когда приходит время выводить их на экран, они содержат некоторые дополнительные данные. Вот документ с включенным в него комментарием:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
import re
hello = "Hello! <!--I've got to be nice to get what I want.-->"
commentSoup = BeautifulSoup(hello)
comment = commentSoup.find(text=re.compile("nice"))

comment.__class__
# <class 'BeautifulSoup.Comment'>
comment
# u'I've got to be nice to get what I want."
comment.previousSibling
# u'Hello! '

str(comment)
# "<!--I've got to be nice to get what I want.-->"
print commentSoup
# Hello! <!--I've got to be nice to get what I want.-->
```

Итак, давайте более внимательно посмотрим на тот документ, что приводился в начале документации:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
      '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
      '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
      '</html>']
soup = BeautifulSoup(''.join(doc))

print soup.prettify()
# <html>
#   <head>
#     <title>
#       Page title
#     </title>
#   </head>
#   <body>
#     <p id="firstpara" align="center">
#       This is paragraph
#       <b>
#         one
#       </b>
#     .
#   </p>
#   <p id="secondpara" align="blah">
#     This is paragraph
#     <b>
#       two
#     </b>
#     .
#   </p>
# </body>
# </html>
```

Атрибуты `Tag`-ов

Объекты Tag и NavigableString имеют множество полезных элементов, большая часть которых описывается в разделах [Навигация по дереву синтаксического разбора](#) и [Поиск внутри дерева синтаксического разбора](#). Тем не менее, рассмотрим здесь один аспект объектов Tag: атрибуты.

Теги SGML имеют атрибуты: например, каждый тег <P> в [приведенном выше примере HTML](#) имеет атрибуты "id" и "align". К атрибутам тегов можно обращаться таким же образом, как если бы объект Tag был словарем:

Переключить отображение номеров строк

```
firstPTag, secondPTag = soup.findAll('p')

firstPTag['id']
# u'firstPara'

secondPTag['id']
# u'secondPara'
```

Объекты `NavigableString` не имеют атрибутов; только объекты `Tag` имеют их.

Навигация по дереву синтаксического разбора

Все объекты `Tag` содержат элементы, перечисленные ниже (тем не менее, фактическое значение элемента может равняться `None`). Объекты `NavigableString` имеют все из них за исключением `contents` и `string`.

`parent`

В [примере выше](#), родителем объекта `<HEAD>` `Tag` является объект `<HTML>` `Tag`. Родителем объекта `<HTML>` `Tag` является сам объект парсера `BeautifulSoup`. Родитель объекта парсера равен `None`. Передвигаясь по объектам `parent`, можно перемещаться по дереву синтаксического разбора:

Переключить отображение номеров строк

```
soup.head.parent.name
# u'html'
soup.head.parent.parent.__class__.__name__
# 'BeautifulSoup'
soup.parent == None
# True
```

`contents`

С помощью `parent` вы перемещаетесь вверх по дереву синтаксического разбора. С помощью `contents` вы перемещаетесь вниз по дереву синтаксического разбора. `contents` является упорядоченным списком объектов `Tag` и `NavigableString`, содержащихся в элементе страницы (page element). Только объект парсера самого высокого уровня и объекты `Tag` содержат `contents`. Объекты `NavigableString` являются простыми строками и не могут содержать подэлементов, поэтому они не содержат `contents`.

В [примере выше](#), элемент `contents` первого объекта `<P>` `Tag` является списком, содержащим объект `NavigableString` ("This is paragraph "), объекта `` `Tag` и еще одного объекта `NavigableString` ("."). Элемент `contents` объекта `` `Tag`: список, состоящий из одного объекта `NavigableString` ("one").

Переключить отображение номеров строк

```
pTag = soup.p
pTag.contents
```



```
# [u'This is paragraph ', <b>one</b>, u'.']
pTag.contents[1].contents
# [u'one']
pTag.contents[0].contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

`string`

Для вашего удобства сделано так, что в случае, когда тег имеет только один дочерний узел, который является строкой, дочерний узел будет доступен через `tag.string` точно также как и через `tag.contents[0]`. В [примере выше](#) `soup.b.string` является объектом `NavigableString` отображающим Unicode-строку "one". Эта строка содержится в первом объекте `` Tag дерева синтаксического разбора.

Переключить отображение номеров строк

```
soup.b.string
# u'one'
soup.b.contents[0]
# u'one'
```

Но `soup.p.string` равен `None`, поскольку первый объект `<P>` Tag в дереве синтаксического разбора имеет более одного дочернего элемента. `soup.head.string` также равен `None`, хотя объект `<HEAD>` Tag имеет только один дочерний элемент, поскольку этот дочерний элемент – Tag (объект `<TITLE>` Tag), а не объект `NavigableString`.

Переключить отображение номеров строк

```
soup.p.string == None
# True
soup.head.string == None
# True
```

`nextSibling` и `previousSibling`

Эти элементы позволяют пропускать следующий или предыдущий элемент на этом же уровне дерева синтаксического разбора. В [представленном выше документе](#), элемент `nextSibling` объекта `<HEAD>` Tag равен объекту `<BODY>` Tag, поскольку объект `<BODY>` Tag является следующим вложенным элементом по отношению к объекту `<html>` Tag. Элемент `nextSibling` объекта `<BODY>` Tag равен `None`, поскольку в нем больше нет вложенных по отношению к объекту `<HTML>` Tag элементов.

Переключить отображение номеров строк

```
soup.head.nextSibling.name
# u'body'
soup.html.nextSibling == None
# True
```

И наоборот, элемент `previousSibling` объекта `<BODY> Tag` равен объекту `<HEAD> tag`, а элемент `previousSibling` объекта `<HEAD> Tag` равен `None`:

[Переключить отображение номеров строк](#)

```
soup.body.previousSibling.name
# u'head'
soup.head.previousSibling == None
# True
```

Несколько примеров: элемент `nextSibling` первого объекта `<P> Tag` равен второму объекту `<P> Tag`. Элемент `previousSibling` объекта ` Tag` внутри второго объекта `<P> Tag` равен объекту `NavigableString "This is paragraph"`. Элемент `previousSibling` данного объекта `NavigableString` равен `None`, внутри первого объекта `<P> Tag`.

[Переключить отображение номеров строк](#)

```
soup.p.nextSibling
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>

secondBTag = soup.findAll('b')[1]
secondBTag.previousSibling
# u'This is paragraph'
secondBTag.previousSibling.previousSibling == None
# True
```

`next` и `previous`

Данные элементы позволяют передвигаться по элементам документа в том порядке, в котором они были обработаны парсером, а не в порядке появления в дереве. Например, элемент `next` для объекта `<HEAD> Tag` равен объекту `<TITLE> Tag`, а не объекту `<BODY> Tag`. Это потому, что в [исходном документе](#), тег `<TITLE>` идет сразу после тега `<HEAD>`.

[Переключить отображение номеров строк](#)

```
soup.head.next
# u'title'
soup.head.nextSibling.name
# u'body'
soup.head.previous.name
# u'html'
```

Поскольку элементы `next` и `previous` связаны, содержимое элемента `contents` объекта `Tag` обновляется до того как элемент `nextSibling`. Как правило, эти элементы не используют, но иногда это наиболее быстрый способ получить что-либо скрытое в глубине дерева синтаксического разбора.

Итерации над объектом `Tag`

Над элементом `contents` объекта `Tag` можно производить итерации, рассматривая его в качестве списка. Это полезное упрощение. Подобным образом можно узнать, сколько дочерних узлов имеет объект `Tag`, вызвав функцию `len(tag)` вместо `len(tag.contents)`. В терминах [документа выше](#):

Переключить отображение номеров строк

```
for i in soup.body:
    print i
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>

len(soup.body)
# 2
len(soup.body.contents)
# 2
```

Используем имена тегов как элементы

Гораздо легче перемещаться по дереву синтаксического разбора, если в качестве имен тегов выступали элементы парсера или объекта `Tag`. Будем так делать на протяжении следующих примеров. В терминах [документа выше](#), `soup.head` возвращает нам первый (как и следовало ожидать, единственный) объекта `<HEAD>` `Tag` документа:

Переключить отображение номеров строк

```
soup.head
# <head><title>Page title</title></head>
```

Вообще, вызов `mytag.foo` возвратит первого потомка `mytag`, чем, как ни странно, будет объект `<FOO>` `Tag`. Если каких-либо объектов `<FOO>` `Tag` внутри `mytag` нет, то `mytag.foo` возвратит `None`. Вы можете использовать это для очень быстрого обхода дерева синтаксического разбора:

Переключить отображение номеров строк

```
soup.head.title
# <title>Page title</title>

soup.body.p.b.string
# u'one'
```

Также это можно использовать для быстрого перехода к определенной части дерева синтаксического разбора. Например, если вас не беспокоит отсутствие тегов `<TITLE>` на предусмотренных для них месте, вы можете просто использовать `soup.title` для получения названия документа HTML. Вам не нужно использовать `soup.head.title`:

Переключить отображение номеров строк

```
soup.title.string
# u'Page title'
```

soup.p перейдет к первому тегу <P> внутри документа, где бы тот ни был. soup.table.tr.td перейдет к первому столбцу первой строки первой же таблицы документа.

Фактически эти элементы – алиасы для метода first, описываемого [ниже](#). Я упоминаю их здесь потому, что алиасы делают очень легким увеличение масштаба интересующей вас части хорошо знакомого дерева синтаксического разбора.

Альтернативная форма этого стиля позволяет обращаться к первому тегу <FOO> как .fooTag вместо .foo. Например, soup.table.tr.td можно также отобразить как soup.tableTag.trTag.tdTag или даже soup.tableTag.tr.tdTag. Это полезно если вы предпочитаете быть более осведомленным о том, что делаете или если вы разбираете XML, в котором имена тегов конфликтуют с именами методов и элементов BeautifulSoup.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulSoup(xml)

xmlSoup.person.parent                # A BeautifulSoup member
# <person name="Bob"><parent rel="mother" name="Alice"></parent>
</person>
xmlSoup.person.parentTag              # A tag name
# <parent rel="mother" name="Alice"></parent>
```

Если вы присмотритесь к именам тегов, то увидите что они не являются корректными идентификаторами Python (как hyphenated-name), вам нужно использовать first.

Поиск в дереве синтаксического разбора

Beautiful Soup предоставляет множество методов для обхода дерева синтаксического разбора, отбирая по заданным критериям объекты Tag и NavigableString. Для определения критериев отбора объектов BeautifulSoup есть несколько способов. Продемонстрируем доскональное исследование наиболее общего из всех методов поиска BeautifulSoup, findAll. Как и раньше, показывать будем на следующем документе:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
      '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
      '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
      '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
# <head>
# <title>
#   Page title
# </title>
```

```
# </head>
# <body>
# <p id="firstpara" align="center">
#   This is paragraph
#   <b>
#     one
#   </b>
#   .
# </p>
# <p id="secondpara" align="blah">
#   This is paragraph
#   <b>
#     two
#   </b>
#   .
# </p>
# </body>
# </html>
```

Между прочим, оба описываемых в этом разделе метода (`findAll` и `find`) доступны только для объектов `Tag` и объектов парсера самого высокого уровня, но не для объектов `NavigableString`. Методы, описываемые в разделе [Поиск в дереве синтаксического разбора](#), доступны и для объектов `NavigableString`.

Основной метод поиска: `findAll(name, attrs, recursive, text, limit, **kwargs)`

Метод обхода дерева `findAll` начинает с заданной точки и ищет все объекты `Tag` и `NavigableString`, соответствующие заданным критериям. Сигнатура метода `findall` следующая:

Переключить отображение номеров строк

```
findAll(name=None, attrs={}, recursive=True, text=None, limit=None,
**kwargs) '''
```

Эти аргументы появляются снова и снова повсюду в BeautifulSoup API. Наиболее важными являются аргументы `name` и именованные аргументы.

» Аргумент `name` ограничивает набор имен тегов. Имеется несколько способов ограничить имена и все они появляются снова и снова повсюду в BeautifulSoup API.

1. Самый простой способ – передать имя тега. Приведем код, который ищет все объекты `` Tag в документе:

Переключить отображение номеров строк

```
soup.findAll('b')
# [<b>one</b>, <b>two</b>]
```

2. Также можно передать регулярное выражение. Код, который ищет все теги, имена которых *начинаются* на английскую букву "B":

Переключить отображение номеров строк

```
import re
tagsStartingWithB = soup.findAll(re.compile('^b'))
[tag.name for tag in tagsStartingWithB]
# [u'body', u'b', u'b']
```

3. Можно передать список или словарь. Эти два вызова ищут все теги <TITLE> и <P>. Принцип работы у них одинаков, но второй вызов отработает быстрее:

Переключить отображение номеров строк

```
soup.findAll(['title', 'p'])
# [<title>Page title</title>,
#  <p id="firstpara" align="center">This is paragraph
  <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
  <b>two</b>.</p>]

soup.findAll({'title' : True, 'p' : True})
# [<title>Page title</title>,
#  <p id="firstpara" align="center">This is paragraph
  <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
  <b>two</b>.</p>]
```

4. Можно передать специальное значение True, которому соответствуют все теги с любыми именами.

Переключить отображение номеров строк

```
allTags = soup.findAll(True)
[tag.name for tag in allTags]
[u'html', u'head', u'title', u'body', u'p', u'b', u'p',
u'b']
```

Это не выглядит полезным, но True очень полезно, когда нужно ограничить значения атрибутов.

1. Можно передать вызываемый объект, который принимает объект Tag как единственный аргумент и возвращает логическое значение. Каждый объект Tag, который находит findAll, будет передан в этот объект и если его вызов возвращает True, то необходимый тег найден.

Вот код, который ищет теги с двумя и только двумя атрибутами:

Переключить отображение номеров строк

```
soup.findAll(lambda tag: len(tag.attrs) == 2)
# [<p id="firstpara" align="center">This is paragraph
  <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph
  <b>two</b>.</p>]
```

Данный код ищет теги, имена которых состоят из одной буквы и которые не имеют атрибутов:

Переключить отображение номеров строк

```
soup.findAll(lambda tag: len(tag.name) == 1 and not tag.attrs)
# [<b>one</b>, <b>two</b>]
```

» Аргументы ключевых слов (keyword arguments) налагают ограничения на атрибуты тега. Вот простой пример поиска всех тегов, атрибут "align" которых имеет значение "center":

» Переключить отображение номеров строк

```
soup.findAll(align="center")
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>]
```

Как и в случае с аргументом name вы можете передать именованный аргумент различными видами объектов для наложения разных ограничений на соответствующие атрибуты. Можно передать строку, как показано выше, чтобы ограничить значение атрибута единственным значением. Можно также передать регулярное выражение, список, хэш, специальные значения True или None, или вызываемый объект, который получает значение атрибута в качестве аргумента (обратите внимание на то, что значение может быть и None). Несколько примеров:

Переключить отображение номеров строк

```
soup.findAll(id=re.compile("para$"))
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
# <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]

soup.findAll(align=["center", "blah"])
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
# <p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]

soup.findAll(align=lambda(value): value and len(value) < 5)
# [<p id="secondpara" align="blah">This is paragraph
<b>two</b>.</p>]
```

Специальные значения True и None особо интересны. Значению True соответствует тег, заданный атрибут которого имеет *любое* значение, а None соответствует тегу, у которого заданный атрибут *не* содержит значения. Несколько примеров:

Переключить отображение номеров строк

```
soup.findAll(align=True)
# [<p id="firstpara" align="center">This is paragraph
<b>one</b>.</p>,
# <p id="secondpara" align="blah">This is paragraph
```

```
<b>two</b>.</p>]
```

```
[tag.name for tag in soup.findAll(aligned=None)]
# [u'html', u'head', u'title', u'body', u'b', u'b']
```

Если необходимо наложить сложные или взаимосвязанные ограничения на атрибуты тегов, передавайте вызываемый объект для name, [как показано выше](#), и работайте с объектом Tag.

- » Здесь вы должны обратить внимание на одну проблему. Что делать, если в вашем документе есть тег, который определяет атрибут с именем name? Поскольку методы поиска BeautifulSoup всегда определяют аргумент name, вы не можете использовать именованный аргумент с именем name. В качестве именованного аргумента также нельзя использовать зарезервированные слова Python, такие как for.

Beautiful Soup предоставляет специальный аргумент с именем attrs, который можно использовать в таких ситуациях. attrs представляет собой словарь, который работает также как именованные аргументы:

Переключить отображение номеров строк

```
soup.findAll(id=re.compile("para$"))
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]

soup.findAll(attrs={'id' : re.compile("para$")})
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

Можно использовать attrs если необходимо наложить ограничения на атрибуты, имена которых совпадают с зарезервированными словами Python такими, как class, for или import; или атрибуты, имена которых являются неименованными аргументами методов поиска BeautifulSoup: name, recursive, limit, text или сам attrs.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulSoup(xml)

xmlSoup.findAll(name="Alice")
# []

xmlSoup.findAll(attrs={"name" : "Alice"})
# [parent rel="mother" name="Alice"></parent>]
```

Поиск класса CSS

Аргумент `attrs` был бы приятным средством, если бы его не портила одна вещь: CSS. Очень полезно находить теги, которые принадлежат определенному классу CSS, но имя атрибута CSS, `class`, также является зарезервированным словом Python.

Искать класс CSS можно с помощью `soup.find("tagName", { "class" : "cssClass" })`, но это требует слишком большого объема кода для столь простой операции. Вместо этого можно передать строку для `attrs` взамен словаря. Строка будет использована для ограничения класса CSS.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("""Bob's <b>Bold</b> Barbeque Sauce now available
in
                                <b class="hickory">Hickory</b> and <b
class="lime">Lime</a>""")

soup.find("b", { "class" : "lime" })
# <b class="lime">Lime</b>

soup.find("b", "hickory")
# <b class="hickory">Hickory</b>
```

- » **text** – аргумент, позволяющий находить вместо объектов `NavigableString` объекты `Tag`. Его значением может быть строка, регулярное выражение, список или словарь, `True` или `None`, или вызываемый объект, который получает объект `NavigableString` в качестве аргумента:

Переключить отображение номеров строк

```
soup.findAll(text="one")
# [u'one']
soup.findAll(text=u'one')
# [u'one']

soup.findAll(text=["one", "two"])
# [u'one', u'two']

soup.findAll(text=re.compile("paragraph"))
# [u'This is paragraph ', u'This is paragraph ']

soup.findAll(text=True)
# [u'Page title', u'This is paragraph ', u'one', u'.', u'This is
paragraph ',
#  u'two', u'.']

soup.findAll(text=lambda(x): len(x) < 12)
# [u'Page title', u'one', u'.', u'two', u'.']
```

Если вы используете `text`, то любые значения передаются в `name` и именованные аргументы игнорируются.

- » **recursive** – логический аргумент (по умолчанию равен `True`), который сообщает BeautifulSoup о том, нужно ли обходить все поддерево или искать лишь среди непосредственных потомков объекта `Tag` или объекта парсера. Вот в чем различие:

Переключить отображение номеров строк

```
[tag.name for tag in soup.html.findAll()]
# [u'head', u'title', u'body', u'p', u'b', u'p', u'b']

[tag.name for tag in soup.html.findAll(recursive=False)]
# [u'head', u'body']
```

Когда аргумент `recursive` ложен, ищутся только непосредственные потомки тега `<HTML>`. Если вы знаете, что нужно найти, то с помощью этого способа можно сэкономить время.

- » Установка аргумента `limit` позволяет останавливать поиск после того, как будет найдено заданное число совпадений. Если в документе тысячи таблиц, а нужно только четыре, то передайте в аргументе `limit` значение 4 и сэкономьте время. По умолчанию, ограничения нет.

Переключить отображение номеров строк

```
soup.findAll('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]

soup.findAll('p', limit=100)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

Вызов тега аналогично вызову `findall`

Небольшое упрощение. Если вызвать объекта парсера или объект `Tag` как функцию, то можно передать ему все аргументы метода `findall` и вызывать также как и `findall`. В терминах документа выше:

Переключить отображение номеров строк

```
soup(text=lambda(x): len(x) < 12)
# [u'Page title', u'one', u'.', u'two', u'.']

soup.body('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]
```

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

Теперь давайте взглянем на другие методы поиска. Все они получают примерно те же аргументы, что и `findAll`.

Метод `find` почти в точности совпадает с `findAll`, за исключением того, что он ищет первое вхождение искомого объекта, а не все. Это похоже на установку для результирующего множества `limit` равным 1 и затем извлечения единственного результата из массива. В терминах [документа выше](#):

Переключить отображение номеров строк

```
soup.findAll('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]

soup.find('p', limit=1)
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>

soup.find('nosuchtag', limit=1) == None
# True
```

В общем, когда взглянете на метод поиска с множественными именами (такой, как `findAll` или `findNextSiblings`), этот метод получает аргумент `limit` и возвращает список результатов. Когда взгляните на метод поиска без множественных имен (такой, как `find` или `findNextSibling`), вы знаете что метод не получает `limit` и возвращает единственный результат.

Что произошло с методом `'first'`?

В предыдущих версиях BeautifulSoup имелись такие методы как `first`, `fetch` и `fetchPrevious`. Эти методы по-прежнему есть, но использовать их нежелательно и в будущем они могут быть убраны. Общий эффект этих имен очень сбивал с толку. Новые имена присвоены единообразно: как упоминалось выше, если имя метода является множественным или ссылается на `All`, то он возвращает множественные объекты. В противном случае он возвращает единственный объект.

Поиск внутри дерева синтаксического разбора

Методы описанные выше — `findAll` и `find`, — стартуют с определенной точки в дереве синтаксического разбора и движутся по нему вниз. Они рекурсивно производят итерацию по элементу `contents` объектов до тех пор, пока не достигнут конца документа.

Это означает, что вызывать эти методы для объектов `NavigableString` не получится, поскольку в них нет элемента `contents`: они всегда являются листьями дерева синтаксического разбора.

Но спуск по дереву - не единственный способ итерации по документу. Не так давно в [Навигации по дереву синтаксического разбора](#) я продемонстрировал много других способов: `parent`, `nextSibling` и т.д. Каждый из этих методов итерации имеют два соответствующих метода: один из них работает аналогично `findAll`, а другой — аналогично `find`. И поскольку объекты `NavigableString` *поддерживают* эти операции и эти методы, то эти методы можно вызывать для них также как и для объектов `Tag` и основного объекта парсера.

Почему это полезно? Иногда вы не сможете применить методы `findAll` или `find`, чтобы получить желаемый объект `Tag` или `NavigableString`. Например, рассмотрим такой HTML:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup(''


  <li>An unrelated list
</ul>

<h1>Heading</h1>
<p>This is <b>the list you want</b>:</p>
<ul><li>The data you want</ul>'')
```

Для навигации по тегу , содержащим необходимые данные, существуют несколько способов. Наиболее очевидный:

Переключить отображение номеров строк

```
soup('li', limit=2)[1]
# <li>The data you want</li>
```

В равной степени очевидно, что это будет не самым стабильным способом получения этого тега . Если эта страница скачивается (scraping) единожды это не имеет значения, но если ее предполагается скачивать неоднократно в течение значительного периода времени, такие соображения становятся существенными. Если в несущественный список (irrelevant list) будет вставлен тег , то вместо искомого тега будет получен этот ненужный тег и работа скрипта прервется или же он выдаст ошибочные данные.

Переключить отображение номеров строк

```
soup('ul', limit=2)[1].li
# <li>The data you want</li>
```

Несколько лучше, поскольку теперь учитываются несущественные изменения в списке. Но если выше в документ будет вставлен другой ненужный список, то будет получен первый тег из этого списка вместо того, который вам нужен. Более надежным способом указания необходимого тега ul будет служить указание на его место в структуре документа.

Когда вы смотрите на этот фрагмент HTML, вы думаете о нужном списке примерно так: 'тег , который идет сразу после тега <H1>'. Проблема состоит в том, что этот тег не содержится внутри тега <H1>; так уж получилось, что он идет после него. Довольно просто перейти к тегу <H1>, но нет способа перейти с этого места к тегу с помощью first и fetch, поскольку эти методы проверяют только элемент contents тега <H1>. Необходимо переместиться к тегу с помощью элементов next или nextSibling:

Переключить отображение номеров строк

```
s = soup.h1
while getattr(s, 'name', None) != 'ul':
    s = s.nextSibling
s.li
# <li>The data you want</li>
```

Или, если вы считаете это более стабильным:

Переключить отображение номеров строк

```
s = soup.find(text='Heading')
while getattr(s, 'name', None) != 'ul':
    s = s.next
s.li
# <li>The data you want</li>
```

Но для достижения нужной вам цели это слишком сложно. Методы из этого раздела предоставляют полезные упрощения. Ими можно воспользоваться, когда возникает желание написать цикл `while` над одним из элементов перемещения. Получив начальную точку где-нибудь в дереве, они позволяют перемещаться по дереву в том же направлении и отслеживают объекты `Tag` или `NavigableString`, соответствующие заданным критериям. Вместо первого цикла в примере кода выше можно написать:

Переключить отображение номеров строк

```
soup.h1.findNextSibling('ul').li
# <li>The data you want</li>
```

Вместо второго цикла можно написать так:

Переключить отображение номеров строк

```
soup.find(text='Heading').findNext('ul').li
# <li>The data you want</li>
```

Циклы заменены вызовами `findNextSibling` и `findNext`. В оставшейся части раздела приведена справочная информация для всех методов такого рода. С другой стороны, каждый метод имеет два варианта: один возвращает список аналогично `findAll` и другой - возвращающий одно значение также как `find`. И напоследок, примера ради, давайте загрузим документ в уже знакомый нам суп:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
      '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
      '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
      '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
# <head>
# <title>
#   Page title
# </title>
```

```
# </head>
# <body>
# <p id="firstpara" align="center">
#   This is paragraph
#   <b>
#     one
#   </b>
#   .
# </p>
# <p id="secondpara" align="blah">
#   This is paragraph
#   <b>
#     two
#   </b>
#   .
# </p>
# </body>
# </html>
```

`findNextSiblings(name, attrs, text, limit, **kwargs)` и `findNextSibling(name, attrs, text, **kwargs)`

Эти методы часто следуют за элементом `nextSibling` объектов, собирая объекты `Tag` или `NavigableText`, соответствующие заданным критериям. В терминах [документа выше](#):

Переключить отображение номеров строк

```
paraText = soup.find(text='This is paragraph ')
paraText.findNextSiblings('b')
# [<b>one</b>]

paraText.findNextSibling(text = lambda(text): len(text) == 1)
# u'.'
```

`findPreviousSiblings(name, attrs, text, limit, **kwargs)` и `findPreviousSibling(name, attrs, text, **kwargs)`

Эти методы часто следуют за элементом `previousSibling` объектов, собирая объекты `Tag` или `NavigableText`, соответствующие заданным критериям. В терминах [документа выше](#):

Переключить отображение номеров строк

```
paraText = soup.find(text='.')
paraText.findPreviousSiblings('b')
# [<b>one</b>]

paraText.findPreviousSibling(text = True)
# u'This is paragraph '
```

`findAllNext(name, attrs, text, limit, **kwargs)` и `findNext(name, attrs, text, **kwargs)`

Эти методы часто следуют за элементом `Next` объектов, собирая объекты `Tag` или `NavigableText`, соответствующие заданным критериям. В терминах [документа выше](#):

Переключить отображение номеров строк

```
pTag = soup.find('p')
pTag.findAllNext(text=True)
# [u'This is paragraph ', u'one', u'.', u'This is paragraph ', u'two',
# u'.']

pTag.findNext('p')
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>

pTag.findNext('b')
# <b>one</b>
```

`findAllPrevious(name, attrs, text, limit, **kwargs)` и `findPrevious(name, attrs, text, **kwargs)`

Эти методы часто следуют за элементом `previous` объектов, собирая объекты `Tag` или `NavigableText`, соответствующие заданным критериям. В терминах [документа выше](#):

Переключить отображение номеров строк

```
lastPTag = soup('p')[-1]
lastPTag.findAllPrevious(text=True)
# [u'.', u'one', u'This is paragraph ', u'Page title']
# Note the reverse order!

lastPTag.findPrevious('p')
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>

lastPTag.findPrevious('b')
# <b>one</b>
```

`findParents(name, attrs, limit, **kwargs)` и `findParent(name, attrs, **kwargs)`

Эти методы часто следуют за элементом `parent` объектов, собирая объекты `Tag` или `NavigableText`, соответствующие заданным критериям. Они не принимают аргумент `text`, поскольку не может быть объектов, предком которых был бы объект `NavigableString`. В терминах [документа выше](#):

Переключить отображение номеров строк

```
bTag = soup.find('b')

[tag.name for tag in bTag.findParents()]
```

```
# [u'p', u'body', u'html', '[document]']
# NOTE: "u'[document]'" means that the parser object itself
# matched.

bTag.findParent('body').name
# u'body'
```

Модификация дерева синтаксического разбора

Теперь вам известно как разыскать что-либо в дереве синтаксического разбора. Но, возможно, вы захотите изменить это что-либо и записать обратно. Можно просто вырезать элемент из contents родительского элемента, но в документе останутся ссылки на извлеченный фрагмент. BeautifulSoup предлагает несколько методов, позволяющих изменить дерево синтаксического разбора во время разбора и сохраняющих при этом внутреннюю согласованность данных.

Изменение значений атрибутов

Модифицировать значения атрибутов объекта Tag можно так же, как если бы он был словарем.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<b id=2>Argh!</b>")
print soup
# <b id="2">Argh!</b>
b = soup.b

b['id'] = 10
print soup
# <b id="10">Argh!</b>

b['id'] = "ten"
print soup
# <b id="ten">Argh!</b>

b['id'] = 'one "million"'
print soup
# <b id='one "million"'>Argh!</b>
```

Так же можно удалять значения атрибутов и добавлять новые:

Переключить отображение номеров строк

```
del(b['id'])
print soup
# <b>Argh!</b>

b['class'] = "extra bold and brassy!"
print soup
# <b class="extra bold and brassy!">Argh!</b>
```


Удаление элементов

Если имеется ссылка на элемент, то удалить его из дерева можно с помощью метода `extract`. Приведем код, который удалит все комментарии из документа:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, Comment
soup = BeautifulSoup("""1<!--The loneliest number-->
                        <a>2<!--Can be as bad as one--><b>3""")
comments = soup.findAll(text=lambda text: isinstance(text, Comment))
[comment.extract() for comment in comments]
print soup
# 1
# <a>2<b>3</b></a>
```

Код, который удалит из документа все поддерево целиком:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<a1></a1><a><b>Amazing content<c><d></a><a2>
</a2>")
soup.a1.nextSibling
# <a><b>Amazing content<c><d></d></c></b></a>
soup.a2.previousSibling
# <a><b>Amazing content<c><d></d></c></b></a>

subtree = soup.a
subtree.extract()

print soup
# <a1></a1><a2></a2>
soup.a1.nextSibling
# <a2></a2>
soup.a2.previousSibling
# <a1></a1>
```

Метод `extract` разъединяет синтаксическое дерево разбора на два несвязанных дерева. Элементы навигации изменяются таким образом, что все выглядит так, будто деревья никогда не были одним целым:

Переключить отображение номеров строк

```
soup.a1.nextSibling
# <a2></a2>
soup.a2.previousSibling
# <a1></a1>
subtree.previousSibling == None
```

```
# True
subtree.parent == None
# True
```

Замена одного элемента на другой

Метод `replaceWith` извлекает один страничный элемент (page element) и заменяет его другим. Новый элемент может быть объектом `Tag` (возможно, с целым деревом синтаксического разбора внутри) или `NavigableString`. Если передавать в метод `replaceWith` простую старую строку, он поместит ее в объект `NavigableString`. Элементы навигации изменятся таким образом, как будто документ прошел синтаксический разбор с самого начала.

Простой пример:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<b>Argh!</b>")
soup.find(text="Argh!").replaceWith("Hooray!")
print soup
# <b>Hooray!</b>

newText = soup.find(text="Hooray!")
newText.previous
# <b>Hooray!</b>
newText.previous.next
# u'Hooray!'
newText.parent
# <b>Hooray!</b>
soup.b.contents
# [u'Hooray!']
```

Вот более сложный пример, в котором один тег замещается другим:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, Tag
soup = BeautifulSoup("<b>Argh!<a>Foo</a></b><i>Blah!</i>")
tag = Tag(soup, "newTag", [("id", 1)])
tag.insert(0, "Hooray!")
soup.a.replaceWith(tag)
print soup
# <b>Argh!<newTag id="1">Hooray!</newTag></b><i>Blah!</i>
```

Можно даже вырезать элемент из одного места документа и вставить в другое:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
text = "<html>There's <b>no</b> business like <b>show</b>
```

```
business</html>"
soup = BeautifulSoup(text)

no, show = soup.findAll('b')
show.replaceWith(no)
print soup
# <html>There's business like <b>no</b> business</html>
```

Добавление другого нового элемента

Класс Tag и классы парсеров поддерживают метод insert. Он работает также как метод insert обычного списка Python: получает индекс элемента contents тега и вставляет в эту позицию новый элемент.

Это было продемонстрировано в предыдущем разделе, когда мы заменяли тег в документе другим новым тегом. Можно использовать insert для построения полного дерева синтаксического разбора с нуля:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, Tag, NavigableString
soup = BeautifulSoup()
tag1 = Tag(soup, "mytag")
tag2 = Tag(soup, "myOtherTag")
tag3 = Tag(soup, "myThirdTag")
soup.insert(0, tag1)
tag1.insert(0, tag2)
tag1.insert(1, tag3)
print soup
# <mytag><myOtherTag></myOtherTag><myThirdTag></myThirdTag></mytag>

text = NavigableString("Hello!")
tag3.insert(0, text)
print soup
# <mytag><myOtherTag></myOtherTag><myThirdTag>Hello!</myThirdTag>
</mytag>
```

В дереве синтаксического разбора элемент может встретиться только в одном месте. Если применить insert к элементу, который уже соединен с объектом супа, он будет отсоединен (с помощью extract) перед тем, как будет подсоединен где-либо. В данном примере я попытался вставить NavigableString во вторую часть супа, но во второй раз он не вставился. Он переместился:

Переключить отображение номеров строк

```
tag2.insert(0, text)
print soup
# <mytag><myOtherTag>Hello!</myOtherTag><myThirdTag></myThirdTag>
</mytag>
```

Это произойдет даже если элемент до этого принадлежал принципиально другому объекту супа. У элемента может быть только один parent, только один nextSibling и т.д., так что в данный момент времени он может находиться только в одном месте.

Известные проблемы

В этом разделе описываются общие проблемы, с которыми сталкиваются при работе с BeautifulSoup.

Почему BeautifulSoup не может вывести на экран не-ASCII символы, которые я передал ему?

Если вы получаете ошибки примерно такого содержания:

"'ascii' codec can't encode character 'x' in position y: ordinal not in range(128)", это значит, что проблема вероятно в установленном Python, а не в BeautifulSoup. Попробуйте вывести на экран не-ASCII символы без использования BeautifulSoup и вы столкнетесь с той же проблемой. Например, попробуйте запустить такой код:

Переключить отображение номеров строк

```
latin1word = 'Sacré bleu!'
unicodeword = unicode(latin1word, 'latin-1')
print unicodeword
```

Если он работает нормально, а при использовании BeautifulSoup все равно выдается сообщение об ошибке – значит ошибка в BeautifulSoup. Однако же если код не работает – проблема с установленным Python. Python выполняет код безопасно и не посылает не-ASCII символы на терминал. Для изменения такого поведения есть два способа.

1. Легкий способ – переназначить стандартный вывод в конвертер, который не боится отправлять символы ISO-Latin-1 или UTF-8 на терминал.

Переключить отображение номеров строк

```
import codecs
import sys
streamWriter = codecs.lookup('utf-8')[-1]
sys.stdout = StreamWriter(sys.stdout)
```

codecs.lookup возвращает количество ограничивающих методов и других объектов, связанных с кодеком. В последней строке объект StreamWriter реализует обертку вокруг потока вывода.

1. Более сложный способ – в каталоге, где установлен Python, создать файл sitecustomize.py, который установит кодировкой по умолчанию ISO-Latin-1 или UTF-8. В этом случае все ваши программы на Python будут использовать эту кодировку для стандартного вывода, не требуя ничего менять в каждой программе. В моем случае имеется файл /usr/lib/python/sitecustomize.py, который выглядит так:

Переключить отображение номеров строк

```
import sys
sys.setdefaultencoding("utf-8")
```

За дополнительной информацией о поддержке Unicode в Python смотрите [Unicode for Programmers](#) или [End to End Unicode Web Applications in Python](#). Также будут очень полезны Рецепт 1.20 и 1.21 Поваренной книги Python.

Помните, что даже если ваш дисплей ограничен ASCII, вы можете использовать BeautifulSoup для синтаксического разбора, обработки и записи документов в UTF-8 и других кодировках. Просто некоторые строки не получится распечатать с помощью `print`.

BeautifulSoup потерял данные, которые я ему передал! Почему? ПОЧЕМУ?????

BeautifulSoup может обрабатывать плохо структурированный SGML, но иногда он теряет данные когда получает нечто, совершенно не похожее на SGML. Это не особенно близко к плохо структурированной разметке, но если у вас в планах построить поисковый агент или что-нибудь вроде этого, то вы наверняка столкнетесь с этим.

Единственным решением является [рассматриваемая далее очистка данных](#) с помощью регулярного выражения. Вот несколько примеров, которые выявили я и пользователи BeautifulSoup:

- » BeautifulSoup интерпретирует плохо оформленные определения XML как данные. Однако он теряет хорошо оформленные определения XML, которые не существуют:

» Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup("< ! FOO @=>")
# < ! FOO @=>
BeautifulSoup("<b><!FOO>!</b>")
# <b>!</b>
```

- » Если документ начинается с декларации и не завершает ее, BeautifulSoup предполагает, что оставшаяся часть документа является частью декларации. Если документ заканчивается посередине декларации, BeautifulSoup игнорирует декларацию полностью. Пара примеров:

» Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup

BeautifulSoup("foo<!bar")
# foo

soup = BeautifulSoup("<html>foo<!bar</html>")
print soup.prettify()
# <html>
#   foo<!bar</html>
# </html>
```

Имеется пара способов исправить это; один из них детально рассмотрен [здесь](#). BeautifulSoup также игнорирует ссылки на сущности, которые не завершены до конца документа:

[Переключить отображение номеров строк](#)

```
BeautifulSoup("<foo>")
# <foo
```

Я никогда не видел такого в реальных веб-страницах, но, наверное, где-нибудь такое встречается.

- » Плохо сформированный комментарий заставит BeautifulSoup проигнорировать оставшийся фрагмент документа. Это раскрывается в примере [Очистка от плохих данных с помощью регулярных выражений](#).

Синтаксическое дерево разбора, построенное классом `BeautifulSoup`, меня раздражает!

Чтобы получить синтаксический разбор вашей разметки другим способом, загляните в раздел [Другие встроенные парсеры](#) или же [Создание собственного парсера](#).

Beautiful Soup слишком медленно работает!

Beautiful Soup никогда не будет столь же быстр как ElementTree или специально разработанный подкласс SGMLParser. ElementTree написан на Си, а с помощью SGMLParser можно написать собственный мини-Beautiful Soup, который будет делать то, что вам необходимо. Суть BeautifulSoup состоит в экономии времени работы программиста, а не процессора.

Это говорит о том, что достаточно сильно ускорить BeautifulSoup можно подвергнув [синтаксическому разбору только нужную часть документа](#), а также вы можете убрать ненужные объекты с помощью `extract` или `decompose`.

Дополнительные темы

Все это необходимо для стандартного использования BeautifulSoup. Но HTML и XML сложны и в реальной ситуации могут слишком сложны. Поэтому BeautifulSoup содержит несколько дополнительных трюков для исправления собственных проблем.

Генераторы

Методы поиска, описываемые выше, управляются с помощью методов генераторов. Вы сами можете использовать эти методы: они называются nextGenerator, previousGenerator, nextSiblingGenerator, previousSiblingGenerator и parentGenerator. Объекты Tag и парсера также имеют в наличии методы childGenerator и recursiveChildGenerator.

Простой пример, в котором при итерациях по документу HTML удаляются все теги и оставшиеся строки объединяются в одну.

[Переключить отображение номеров строк](#)

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("""<div>You <i>bet</i>
<a
href="http://www.crummy.com/software/BeautifulSoup/">BeautifulSoup</a>
rocks!</div>""")

''.join([e for e in soup.recursiveChildGenerator()
         if isinstance(e, unicode)])
# u'You bet\nBeautifulSoup\nrocks!'
```

Конечно, вам действительно не нужен генератор только для поиска текста внутри тега. Этот код делает то же самое что и `findAll(text=True)`.

Переключить отображение номеров строк

```
''.join(soup.findAll(text=True))
# u'You bet\nBeautifulSoup\nrocks!'
```

Вот более сложный пример, в котором используется `recursiveChildGenerator` для итераций по элементам документа, выводя их на экран по мере нахождения.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("1<a>2<b>3")
g = soup.recursiveChildGenerator()
while True:
    try:
        print g.next()
    except StopIteration:
        break
# 1
# <a>2<b>3</b></a>
# 2
# <b>3</b>
# 3
```

Другие встроенные парсеры

Помимо `BeautifulSoup` и `BeautifulStoneSoup` в составе Beautiful Soup имеются еще три класса парсеров:

- » `MinimalSoup` – подкласс `BeautifulSoup`. Он обладает знанием большинства фактов о HTML таких, как какие из тегов самозакрывающиеся, особое поведение тега `<SCRIPT>`, возможности указания кодировки в теге `<META>` и т.д. Но он совсем не имеет вложенных эвристик. Поэтому он не знает, что теги `` всегда вложены в теги `` и никак иначе. Он полезен для синтаксического разбора патологически плохой разметки и для наследования.

- » ICantBelieveItsBeautifulSoup – также подкласс BeautifulSoup. Он содержит эвристики HTML, которые почти соответствуют стандарту HTML, но игнорирует реальное использование HTML. Например, он проверяет вложенность тегов , но в реальном мире такая вложенность тегов почти всегда означает, что автор документа забыл закрыть первый тег . Когда вы столкнетесь с чем-нибудь, где используется вложенность тегов , вы можете использовать ICantBelieveItsBeautifulSoup.
- » BeautifulSoup – подкласс BeautifulSoup. Он полезен при синтаксическом разборе таких документов как сообщения SOAP, которые используют подэлементы, вместо использования атрибутов родительского элемента. Вот пример:

» Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, BeautifulSoup
xml = "<doc><tag>subelement</tag></doc>"
print BeautifulSoup(xml)
# <doc><tag>subelement</tag></doc>
print BeautifulSoup(xml)
<doc tag="subelement"><tag>subelement</tag></doc>
```

С помощью BeautifulSoup можно получить доступ сразу к содержимому тега <TAG> без спуска по дереву к тегу.

Настраиваем парсер

Когда встроенные классы парсеров не делают то, что нужно, тогда необходимо заняться настройкой. Обычно это означает настройку списков вложенности и самозакрываемости тегов. Список самозакрывающихся тегов можно настроить передав в конструктор супа аргумент `'selfClosingTags'`. Для настройки же списков вложенности тегов придется использовать производный класс.

Наиболее полезные классы для создания производных – BeautifulSoup (для HTML) и BeautifulSoup (для XML). Я покажу, как перегрузить RESET_NESTING_TAGS и NESTABLE_TAGS в подклассах. Это наиболее сложная часть BeautifulSoup и я не смогу разъяснить здесь ее всю, но кое-что я все же покажу и смогу улучшить с помощью обратной связи.

Когда BeautifulSoup разбирает документ, он сохраняет открытые теги в стеке. Всякий раз, когда встречается новый открывающий тег, он помещает его на вершину стека. Но до этого он может закрыть некоторые из открытых тегов и удалить их из стека. Какие теги он закроет – зависит от вида найденного тега и от вида тегов в стеке.

Лучшим способом объяснить все это будет пример. Предположим, что стек выглядит таким образом ['html', 'p', 'b'] и BeautifulSoup встретился тег <P>. Если просто поместить еще один 'p' в стек, то это будет означать, что второй тег <P> находится внутри первого тега <P>, не говоря уже о теге . Но теги <P> функционируют по-другому. Вы не сможете вставить один тег <P> в другой тег <P>. Тег <P> вообще «не вкладываемый».

Поэтому когда BeautifulSoup встречает тег <P>, он закрывает и извлекает из стека все теги и их содержимое ранее встреченного тега такого же типа. Это поведение по умолчанию, и BeautifulSoup обращается так с *каждым* тегом. Вот что получится, если тег не упомянут ни в NESTABLE_TAGS, ни в RESET_NESTING_TAGS. Если тег указан в RESET_NESTING_TAGS, но не указан в NESTABLE_TAGS, поведение будет таким же, как и в случае тега.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup.RESET_NESTING_TAGS['p'] == None
# True
BeautifulSoup.NESTABLE_TAGS.has_key('p')
# False

print BeautifulSoup("<html><p>Para<b>one<p>Para two")
# <html><p>Para<b>one</b></p><p>Para two</p></html>
#
#           ^---^--Второй тег <p> предполагает закрытие этих
#           двух тегов
```

Давайте предположим, что стек выглядит следующим образом ['html', 'span', 'b'] и BeautifulSoup встретил тег . Теперь теги могут содержать другие теги без ограничений, так что при встрече тега теперь не нужно убирать со стека предыдущий тег такого же типа. Все это представлено отображением имени тега на пустой список в NESTABLE_TAGS. Этот тип тегов не должен указываться в RESET_NESTING_TAGS: не существует ситуаций, когда при встрече тега придется удалять со стека какие-либо теги.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup.NESTABLE_TAGS['span']
# []
BeautifulSoup.RESET_NESTING_TAGS.has_key('span')
# False

print BeautifulSoup("<html><span>Span<b>one<span>Span two")
# <html><span>Span<b>one<span>Span two</span></b></span></html>
```

Третий пример: предположим, что стек выглядит так ['ol', 'li', 'ul']: т.е. мы получим упорядоченный список, первый элемент которого содержит неупорядоченный список. Теперь предположим, что BeautifulSoup встретил тег . Это не приведет к удалению со стека первого тега , т.к. новый тег является часть неупорядоченного подсписка. С тегом внутри другого тега все в порядке до тех пор, пока встречается теги или .

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
print BeautifulSoup("<ol><li>1<ul><li>A").prettify()
# <ol>
#   <li>
#     1
#     <ul>
#       <li>
#         A
#       </li>
#     </ul>
```

```
# </li>
# </ol>
```

Но если не встретится или , то один тег не может быть внутри другого:

Переключить отображение номеров строк

```
print BeautifulSoup("<ol><li>1<li>A").prettify()
# <ol>
# <li>
# 1
# </li>
# <li>
# A
# </li>
# </ol>
```

Сообщим BeautifulSoup о том, чтобы он трактовал теги поместив "li" в RESET_NESTING_TAGS, и передавая "li" в элемент NESTABLE_TAGS, чтобы показать список тегов, в которые он может быть вложен.

Переключить отображение номеров строк

```
BeautifulSoup.RESET_NESTING_TAGS.has_key('li')
# True
BeautifulSoup.NESTABLE_TAGS['li']
# ['ul', 'ol']
```

Таким же образом мы обрабатываем вложенность тегов таблиц:

Переключить отображение номеров строк

```
BeautifulSoup.NESTABLE_TAGS['td']
# ['tr']
BeautifulSoup.NESTABLE_TAGS['tr']
# ['table', 'tbody', 'tfoot', 'thead']
BeautifulSoup.NESTABLE_TAGS['tbody']
# ['table']
BeautifulSoup.NESTABLE_TAGS['thead']
# ['table']
BeautifulSoup.NESTABLE_TAGS['tfoot']
# ['table']
BeautifulSoup.NESTABLE_TAGS['table']
# []
```

Это означает, что теги <TD> могут быть вложены внутри тегов <TR>. Теги <TR> могут быть вложены внутри тегов <TABLE>, <TBODY>, <TFOOT> и <THEAD>. Теги <TBODY>, <TFOOT> и <THEAD> могут быть вложены в теги <TABLE> и теги <TABLE>, в свою очередь, могут быть

вложены в другие теги <TABLE>. Если вы освоили таблицы HTML, то эти правила будут вам понятны.

Еще один пример. Предположим, что стек выглядит так ['html', 'p', 'table'] и BeautifulSoup встретил тег <P>.

На первый взгляд это выглядит почти так же как в примере со стеком вида ['html', 'p', 'b'], когда BeautifulSoup встретил тег <P>. В данном примере, мы закрыли теги и <P>, поскольку один параграф не может быть внутри другого.

За исключением того... что вы *можете* иметь параграф, содержащий таблицу, и в этой таблице содержится параграф. Т.е. правильным будет не закрывать какие-либо из этих тегов. BeautifulSoup поступит правильно:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
print BeautifulSoup("<p>Para 1<b><p>Para 2")
# <p>
#   Para 1
#   <b>
#   </b>
# </p>
# <p>
#   Para 2
# </p>

print BeautifulSoup("<p>Para 1<table><p>Para 2").prettify()
# <p>
#   Para 1
#   <table>
#     <p>
#       Para 2
#     </p>
#   </table>
# </p>
```

В чем же разница? Разница заключается в том, что тег <TABLE> содержится в RESET_NESTING_TAGS, а тег - нет. Тег, содержащийся в RESET_NESTING_TAGS, не сможет покинуть стек так же легко как не содержащийся там тег.

Итак, будем надеяться, вы поняли идею. Вот NESTABLE_TAGS для класса BeautifulSoup. Соотнесите его с тем, что вы знаете о HTML и вы сможете создать свой собственный NESTABLE_TAGS для необычных документов HTML, которые не подчиняются нормальным правилам, и для других диалектов XML, имеющих различные правила вложенности.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
nestKeys = BeautifulSoup.NESTABLE_TAGS.keys()
nestKeys.sort()
```

```
for key in nestKeys:
    print "%s: %s" % (key, BeautifulSoup.NESTABLE_TAGS[key])
# bdo: []
# blockquote: []
# center: []
# dd: ['dl']
# del: []
# div: []
# dl: []
# dt: ['dl']
# fieldset: []
# font: []
# ins: []
# li: ['ul', 'ol']
# object: []
# ol: []
# q: []
# span: []
# sub: []
# sup: []
# table: []
# tbody: ['table']
# td: ['tr']
# tfoot: ['table']
# th: ['tr']
# thead: ['table']
# tr: ['table', 'tbody', 'tfoot', 'thead']
# ul: []
```

А вот RESET_NESTING_TAGS класса BeautifulSoup. Важны только ключи: фактически RESET_NESTING_TAGS является списком, помещенным в виде словаря в целях организации быстрого случайного доступа.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
resetKeys = BeautifulSoup.RESET_NESTING_TAGS.keys()
resetKeys.sort()
resetKeys
# ['address', 'blockquote', 'dd', 'del', 'div', 'dl', 'dt', 'fieldset',
#  'form', 'ins', 'li', 'noscript', 'ol', 'p', 'pre', 'table', 'tbody',
#  'td', 'tfoot', 'th', 'thead', 'tr', 'ul']
```

В любом случае получив подкласс, вы можете точно также заменить SELF_CLOSING_TAGS пока вы в нем. Это словарь, который отображает имена самозакрывающихся тегов в какие-либо значения (аналогично RESET_NESTING_TAGS, он фактически является списком в виде словаря). К тому же, вам не захочется передавать этот список в конструктор (аналогично selfClosingTags) каждый раз, когда создаете экземпляр своего подкласса.

Преобразование сущностей

Когда идет синтаксический разбор документа вы можете преобразовать HTML или XML ссылки на сущности в соответствующие символы Unicode. Данный код преобразует HTML сущность "é" в символ Unicode ЛАТИНСКАЯ МАЛЕНЬКАЯ БУКВА Е С АКУТОМ и числовую сущность "e" в символ Unicode ЛАТИНСКУЮ МАЛЕНЬКУЮ БУКВУ E.

[Переключить отображение номеров строк](#)

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup("Sacré; bl#101;u!",

convertEntities=BeautifulSoup.HTML_ENTITIES).contents[0]
# u'Sacr\xe9 bleu!'
```

Это если вы используете HTML_ENTITIES (который является только строкой "html"). Если используется XML_ENTITIES (или строка "xml"), то только числовые сущности и пять XML сущностей ("'", '"', ">", "<", and "&") будут конвертированы. Если используется ALL_ENTITIES (или список ["xml", "html"]), то оба вида сущностей будут конвертированы. Последний случай необходим, поскольку ' - сущность XML, а не HTML.

[Переключить отображение номеров строк](#)

```
BeautifulSoup("Sacré; bl#101;u!",
              convertEntities=BeautifulSoup.XML_ENTITIES)
# Sacré; bleu!

from BeautifulSoup import BeautifulSoup
BeautifulSoup("Il a dit, <<Sacré; bl#101;u!>>",
              convertEntities=BeautifulSoup.XML_ENTITIES)
# Il a dit, <<Sacré; bleu!>>
```

Если вы сообщите BeautifulSoup о необходимости конвертировать XML- или HTML-сущности в соответствующие символы Unicode, то символы Windows-1252 (такие как изящные кавычки Microsoft) также будут трансформированы в символы Unicode. Это произойдет даже если сообщить BeautifulSoup о конвертировании этих символов в сущности.

[Переключить отображение номеров строк](#)

```
from BeautifulSoup import BeautifulSoup
smartQuotesAndEntities = "Il a dit, \x8BSacré; bl#101;u!\x9b"

BeautifulSoup(smartQuotesAndEntities,
smartQuotesTo="html").contents[0]
# u'Il a dit, &lsquo;Sacré; bl#101;u!&rsquo;

BeautifulSoup(smartQuotesAndEntities, convertEntities="html",
              smartQuotesTo="html").contents[0]
# u'Il a dit, \u2039Sacr\xe9 bleu!\u203a'
```

```
BeautifulStoneSoup(smartQuotesAndEntities, convertEntities="xml",
                    smartQuotesTo="xml").contents[0]
# u'Il a dit, \u2039Sacr&eacute; bleu!\u203a'
```

Пока вы заняты преобразованием всех имеющихся сущностей в символы Unicode, создавать новые HTML/XML сущности не имеет смысла.

Очистка от плохих данных с помощью регулярных выражений

Beautiful Soup прекрасно подходит для обработки плохой разметки, когда под "плохой разметкой" понимается неверное расположение тегов. Но иногда разметка настолько деформирована, что базовый парсер не может ее обработать. Поэтому BeautifulSoup применяет к исходному документу регулярные выражения прежде, чем попытается произвести его синтаксический разбор.

По умолчанию BeautifulSoup использует регулярные выражения и функции замены для выполнения поиска и замены в исходных документах. Он находит самозакрывающиеся теги, такие как `
`, и изменяет их на такие `
`. Он находит декларации, содержащие внешние пробелы, такие как `<!--Comment-->`, и удаляет из них пробелы: `<!--Comment-->`.

Если имеется плохая разметка, для исправления которой требуются другие методы, вы можете передать свой список кортежей вида (regular expression, replacement function) в конструктор супа как аргумент `markupMassage`.

Давайте рассмотрим пример: страница имеет деформированный комментарий. Базовый парсер SGML не справился и пропустил комментарий со всем, что было после него:

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
badString = "Foo<!--This comment is malformed.-->Bar<br/>Baz"
BeautifulSoup(badString)
# Foo
```

Давайте исправим это с помощью регулярных выражений и функции:

Переключить отображение номеров строк

```
import re
myMessage = [(re.compile('<!--([^-])'), lambda match: '<!--' +
match.group(1))]
BeautifulSoup(badString, markupMassage=myMessage)
# Foo<!--This comment is malformed.-->Bar
```

Ой, мы к тому же пропустили тег `
`. Наш аргумент `markupMassage` поменял манипуляцию парсера по умолчанию, так что функция поиска и замены по умолчанию не выполнялась. Парсер сделал это, пропустив комментарий, но деформированный самозакрывающийся тег вывел его из строя. Давайте добавим свою функцию манипулирования данными в список умолчаний, а затем выполним все функции.

Переключить отображение номеров строк

```
import copy
myNewMessage = copy.copy(BeautifulSoup.MARKUP_MESSAGE)
myNewMessage.extend(myMessage)
BeautifulSoup(badString, markupMessage=myNewMessage)
# Foo<!--This comment is malformed.-->Bar<br />Baz
```

Теперь мы получили все.

Если вы уверены в том, что разметка не нуждается в применении регулярных выражений, вы можете ускорить время запуска передав значение False в аргументе markupMessage.

Наслаждаемся `SoupStrainer`-ми

Припомните, что все поисковые методы получают больше или меньше [вот таких аргументов](#). За кулисами, ваши аргументы поисковых методов преобразуются в объект SoupStrainer. Если вызвать один из методов, возвращающих список (например, findAll), объект SoupStrainer станет доступным как свойство source результирующего списка.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulSoup(xml)
results = xmlSoup.findAll(rel='mother')

results.source
# <BeautifulSoup.SoupStrainer instance at 0xb7e0158c>
str(results.source)
# "None|{'rel': 'mother'}"
```

Конструктор SoupStrainer получает большую часть тех же аргументов, что и find: `name`, `attrs`, `text` и `**kwargs`. Можно передать SoupStrainer как аргумент name в любой поисковый метод:

Переключить отображение номеров строк

```
xmlSoup.findAll(results.source) == results
# True

customStrainer = BeautifulSoup.SoupStrainer(rel='mother')
xmlSoup.findAll(customStrainer) == results
# True
```

Не велика важность, не правда ли? Можно передать аргументы в метод и другими способами. Но с другой стороны SoupStrainer можно передать в конструктор супа для ограничения фрагмента документа, который фактически будет подвергнут синтаксическому разбору. Это рассматривается в следующем разделе:

Улучшаем производительность за счет синтаксического разбора только части документа

Beautiful Soup преобразует каждый элемент документа в объект Python и присоединяет его к группе других объектов Python. Если вам необходимо лишь подмножество документа, то это крайне медленно. Но `SoupStrainer` можно передать в качестве аргумента `parseOnlyThese` в конструктор супа. Beautiful Soup проверяет каждый элемент – не является ли он `SoupStrainer`, и только если является преобразует в объект `Tag` или `NavigableText`, и добавляет в дерево.

Если элемент добавлен в дерево, то он имеет потомков – даже если они не являются `SoupStrainer`. Это позволяет разбирать только фрагменты документа, которые содержит нужную вам информацию.

Вот значительно реорганизованный документ:

Переключить отображение номеров строк

```
doc = '''Bob reports <a href="http://www.bob.com/">success</a>
with his plasma breeding <a
href="http://www.bob.com/plasma">experiments</a>. <i>Don't get any on
us, Bob!</i>

<br><br>Ever hear of annular fusion? The folks at <a
href="http://www.boogabooga.net/">BoogaBooga</a> sure seem obsessed
with it. Secret project, or <b>WEB MADNESS?</b> You decide!'''
```

Имеется несколько разных способов синтаксического разбора документа в супе в зависимости от того, какие части необходимы. Все они работают быстрее и используют меньше памяти, нежели синтаксический разбор всего документа и используют `SoupStrainer` для выделения необходимых вам частей.

Переключить отображение номеров строк

```
from BeautifulSoup import BeautifulSoup, SoupStrainer
import re

links = SoupStrainer('a')
[tag for tag in BeautifulSoup(doc, parseOnlyThese=links)]
# [<a href="http://www.bob.com/">success</a>,
#  <a href="http://www.bob.com/plasma">experiments</a>,
#  <a href="http://www.boogabooga.net/">BoogaBooga</a>]

linksToBob = SoupStrainer('a', href=re.compile('bob.com/'))
[tag for tag in BeautifulSoup(doc, parseOnlyThese=linksToBob)]
# [<a href="http://www.bob.com/">success</a>,
#  <a href="http://www.bob.com/plasma">experiments</a>]

mentionsOfBob = SoupStrainer(text=re.compile("Bob"))
[text for text in BeautifulSoup(doc, parseOnlyThese=mentionsOfBob)]
# [u'Bob reports ', u"Don't get any on\nus, Bob!"]
```



```
allCaps = SoupStrainer(text=lambda(t):t.upper()==t)
[text for text in BeautifulSoup(doc, parseOnlyThese=allCaps)]
# [u'. ', u'\n', u'WEB MADNESS?']
```

Между передачей SoupStrainer в метод поиска и его передачей в конструктор супа имеется одно существенное отличие. Припомним, что аргумент name может принимать значение [функции с объектом Tag в качестве аргумента](#). Для аргумента name SoupStrainer-а это сделать нельзя, поскольку SoupStrainer используется для решения: будет или нет вообще создан объект Tag. В аргумент name SoupStrainer-а можно передать функцию, но не объект Tag: можно передать только имя тега и таблицу аргументов.

Переключить отображение номеров строк

```
shortWithNoAttrs = SoupStrainer(lambda name, attrs: \
                                len(name) == 1 and not attrs)
[tag for tag in BeautifulSoup(doc, parseOnlyThese=shortWithNoAttrs)]
# [<i>Don't get any on us, Bob!</i>,
#  <b>WEB MADNESS?</b>]
```

Улучшаем использование памяти при помощи `extract`

Когда BeautifulSoup разбирает документ, то он загружает в память большие, сильно связанные структуры данных. Можно решить, что если необходимо найти строку в этой структуре данных, то достаточно просто вытащить строку и оставить остальное для уборки мусора. Все несколько сложнее. Эта строка является объектом NavigableString. Который имеет элемент parent, указывающий на объект Tag, который в свою очередь указывает на другие объекты Tag и так далее. Поэтому для того, чтобы иметь в распоряжении любую часть дерева, необходимо всю ее хранить в памяти.

Метод extract разрывает эти связи. Если вызвать extract для искомой строки, то она будет оторвана от остального дерева синтаксического разбора. После этого остальное дерево может выйти за пределы видимости и будет удалено из памяти при сборке мусора до тех пор, пока вы используете строку для чего-нибудь еще. Если вам необходим небольшой фрагмент дерева, вы можете вызывать extract для самого верхнего объекта Tag и позволить убрать из памяти остальное дерево при сборке мусора.

И наоборот. Если имеется большой фрагмент документа, *не нужный* вам, то можно вызвать extract для его извлечения из дерева, а затем удалить из памяти с помощью уборки мусора, удерживая в то же время контроль над (меньшим) деревом.

Если extract не устраивает вас, можете попробовать Tag.decompose. Она более медленная чем extract, но более совершенная. Она рекурсивно разбирает Tag и его содержимое, отсоединяя каждую часть дерева ото всех других частей.

Если вы обнаружите, что заняты удалением больших фрагментов дерева, то вы можете сэкономить время, если [сразу исключите из разбора часть дерева](#).

Смотрите также

Приложения, использующие BeautifulSoup

Многие реальные приложения используют BeautifulSoup. Здесь приведены общедоступные приложения, которые я знаю:

- » [!\[\]\(31b03e46ee8a80a1f1467b8c03bd76e8_img.jpg\) Scrape 'N' Feed](#) разработан для работы с BeautifulSoup, чтобы создавать RSS потоки для сайтов, которые их не имеют.
- » [!\[\]\(7d9665ff04f9d2270c38081c6215a724_img.jpg\) htlatex](#) использует BeautifulSoup для поиска выражений LaTeX и формирования их графического представления.
- » [!\[\]\(7cea648fec4dfc1e99934873e9173b69_img.jpg\) chmtopdf](#) конвертирует CHM-файлы в PDF формат. Кто я чтобы спорить с этим?
- » Duncan Gough в [!\[\]\(48ceb66414885cacc3f139b4fa359213_img.jpg\) Fotopic backup](#) использует BeautifulSoup для скачивания (scrape) веб-сайта Fotopic.
- » Iñigo Serna в [!\[\]\(01a1fc700f38e6e09ee62e6a9c54d804_img.jpg\) googlenews.py](#) использует BeautifulSoup для скачивания (scrape) Google News (см. функции `parse_entry` и `parse_category`).
- » Сайт [!\[\]\(833c1865792a2399365d8193854ceab7_img.jpg\) Weather Office Screen Scraper](#) использует BeautifulSoup для скачивания (scrape) Государственного метеорологического сайта Канады (Canadian government's weather office site).
- » [!\[\]\(5b4802b5ab32e2afe0a3214e088c55e2_img.jpg\) News Clues](#) использует BeautifulSoup для синтаксического разбора RSS потоков.
- » [!\[\]\(c1a72aaa635814897c20812b2e4c560c_img.jpg\) BlinkFlash](#) использует BeautifulSoup для автоматизации форм подписки на онлайн-сервисы.
- » Программа [!\[\]\(b89ef0c055b78377f582d5966452ea89_img.jpg\) linky](#) использует BeautifulSoup для поиска ссылок и изображений на странице, которые нужно проверить.
- » [!\[\]\(843cf0c3ada5c46c853d1230936e9604_img.jpg\) Matt Croydon](#) заставил работать BeautifulSoup версии 1.x на своем смартфоне Nokia на платформе Series 60. [!\[\]\(90136a0f77adba2cf51723c9a7ae8606_img.jpg\) C.R. Sandeep](#), используя BeautifulSoup, написал работающий на платформе Series 60 в реальном времени конвертер валют, но он не хочет показать нам как он это сделал.
- » [!\[\]\(cc272731498ccb66601daa96e4c289fa_img.jpg\) Небольшой скрипт](#) с jacobian.org для исправления метаданных в музыкальных файлах, скачанных с сайта allofmp3.com.
- » Сайт [!\[\]\(66766d3efd042fd755814511162914b7_img.jpg\) Python Community Server](#) использует BeautifulSoup в своем детекторе спама.

Похожие библиотеки

Я нашел несколько других парсеров на разных языках программирования, которые могут работать с поврежденной разметкой, обходить дерево или делать еще что-нибудь сверх того, что может ваш стандартный парсер.

- » Я портировал BeautifulSoup на Ruby. Результат называется [!\[\]\(4e333a6106fc298d0ae6dff272a736ef_img.jpg\) Rubyful Soup](#).
- » [!\[\]\(97089f8e07e24e31baa67366e358a709_img.jpg\) Hpricot](#) платит за использование Rubyful Soup.
- » [!\[\]\(9496824b8cff3a19f59b81b37b57d8b6_img.jpg\) ElementTree](#) - быстрый парсер поврежденных файлов XML на Python. Мне он понравился.
- » [!\[\]\(ec8d0f7e486e2280c113cd85015a8548_img.jpg\) Tag Soup](#) - XML/HTML парсер, написанный на Java, который переписывает поврежденный HTML файл в виде, доступном для синтаксического разбора HTML.
- » [!\[\]\(fad66fecb73aae330937d501057cafc9_img.jpg\) HtmlPrag](#) - библиотека языка программирования Scheme для синтаксического разбора поврежденных HTML.
- » [!\[\]\(a94e0943f5ecd6c1adc5223fd7677110_img.jpg\) xmltramp](#) - отличная реализация парсера файлов XML/XHTML, сравнимая со 'стандартными'. Как и большинство парсеров он позволяет обходить дерево разбора, но использовать его проще.
- » [!\[\]\(f14ef06774200ee2342297364295aa0f_img.jpg\) pullparser](#) включает в себя метод обхода дерева разбора (tree-traversal method).

- » Mike Foord не понравилось, как BeautifulSoup изменяет HTML когда записывает его обратно, и он написал [HTML Scraper](#). Основываясь на HTMLParser, он способен обрабатывать поврежденный HTML. Может быть, после выхода BeautifulSoup 3.0 он устареет, но в этом я не уверен.
- » Ka-Ping Yee в своем скрипте [scrape.py](#) сочетает скачивание с открытием URL.

Заключение

Вот и все! Наслаждайтесь! Я написал BeautifulSoup чтобы сэкономить вам время. Однажды воспользовавшись им вы будете способны в течение всего нескольких минут отстоять в споре выходные данные плохо спроектированных веб-сайтов. Присылайте мне по электронной почте свои комментарии, описания возникающих при работе проблем или просто сообщите мне о вашем проекте с применением BeautifulSoup.

Перевод: Сёмка Александр

Документации/BeautifulSoup (последним исправлял пользователь [AleksandrSemka](#) 2010-06-13 14:46:21)

- » [MoinMoin Powered](#)
- » [Python Powered](#)
- » [GPL licensed](#)
- » [Valid HTML 4.01](#)