

뷰티플수프 문서

한글판 johnsonj 2012.11.08 [원문 위치](#)

[뷰티플수프](#)는 HTML과 XML 파일로부터 데이터를 뽑아내기 위한 파이썬 라이브러리이다. 여러분이 선호하는 해석기와 함께 사용하여 일반적인 방식으로 해석 트리를 향해, 검색, 변경할 수 있다. 주로 프로그래머의 수고를 덜어준다.

이 지도서에서는 뷰티플수프 4의 중요한 특징들을 예제와 함께 모두 보여준다. 이 라이브러리가 어느 곳에 유용한지, 어떻게 작동하는지, 또 어떻게 사용하는지, 어떻게 원하는대로 바꿀 수 있는지, 예상을 빗나갔을 때 어떻게 해야 하는지를 보여준다.

이 문서의 예제들은 파이썬 2.7과 Python 3.2에서 똑 같이 작동한다.

혹시 [뷰티플수프 3](#)에 관한 문서를 찾고 계신다면 뷰티플수프 3는 더 이상 개발되지 않는다는 사실을 꼭 아셔야겠다. 새로 프로젝트를 시작한다면 뷰티플수프 4를 적극 추천한다. 뷰티플수프 3와 뷰티플수프 4의 차이점은 [BS4 코드 이식하기](#)를 참조하자.



도움 얻기

뷰티플수프에 의문이 있거나, 문제에 봉착하면 [토론 그룹에 메일을 보내자](#).

바로 시작

다음은 이 문서에서 예제로 사용할 HTML 문서이다. 이상한 나라의 앨리스 이야기의 일부이다:

```
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>
"""
```

“three sisters” 문서를 뷰티플수프에 넣으면 BeautifulSoup 객체가 나오는데, 이 객체는 문서를 내포된 데이터 구조로 나타낸다:

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

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

```
# <a class="sister" href="http://example.com/tillie" id="link2">
# Tillie
# </a>
# ; and they lived at the bottom of a well.
# </p>
# <p class="story">
# ...
# </p>
# </body>
# </html>
```

다음은 간단하게 데이터 구조를 항해하는 몇 가지 방법이다:

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

```
soup.title.name
# u'title'
```

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

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

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

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

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

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

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

일반적인 과업으로 한 페이지에서 <a> 태그에 존재하는 모든 URL을 뽑아 낼 일이 많다:

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

또 다른 과업으로 페이지에서 텍스트를 모두 뽑아낼 일이 많다:

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

이것이 여러분이 필요한 것인가? 그렇다면, 계속 읽어 보자.

뷰티풀 수프 설치하기

데비안이나 우분투 리눅스 최신 버전을 사용중이라면, 시스템 꾸러미 관리자로 뷰티풀수프를 설치하자:

```
$ apt-get install python-bs4
```

뷰티블수프 4는 PyPi를 통하여도 출간되어 있으므로, 시스템 꾸러미 관리자로 설치할 수 없을 경우, `easy_install`로 설치하거나 `pip`로 설치할 수 있다. 꾸러미 이름은 `beautifulsoup4`이며, 같은 꾸러미로 파이썬 2 그리고 파이썬 3에 작동한다.

```
$ easy_install beautifulsoup4
```

```
$ pip install beautifulsoup4
```

(이 BeautifulSoup 꾸러미가 혹시 원하는 것이 아니라면. 이전 버전으로 [뷰티플수프 3](#)가 있다. 많은 소프트웨어에서 BS3를 사용하고 있으므로, 여전히 사용할 수 있다. 그러나 새로 코드를 작성할 생각이라면 `beautifulsoup4`를 설치하시기 바란다.)

`easy_install`도 `pip`도 설치되어 있지 않다면, [뷰티플수프 4 소스](#)를 내려 받아 `setup.py`로 설치하실 수 있다.

```
$ python setup.py install
```

다른 모든 것이 실패하더라도, 뷰티플수프 라이선스는 여러분의 어플리케이션에 통채로 꾸러 넣는 것을 허용하므로 전혀 설치할 필요없이 소스를 내려받아 `bs4` 디렉토리를 통채로 코드베이스에 복사해서 사용하셔도 된다.

본인은 파이썬 2.7과 파이썬 3.2에서 뷰티플수프를 개발하였지만, 다른 최신 버전에도 작동하리라 믿는 바이다.

설치 이후의 문제

뷰티플 수프는 파이썬 2 코드로 꾸러 놓여져 있다. 파이썬 3에 사용하기 위해 설치하면, 파이썬 3 코드로 자동으로 변환된다. 꾸러미가 설치되어 있지 않다면, 당연히 변환되지 않는다. 또한 윈도우즈 머신이라면 잘못된 버전이 설치되어 있다고 보고된다.

“No module named HTMLParser”와 같은 `ImportError` 에러가 일어나면, 파이썬 3 아래에서 파이썬 2 버전의 코드를 실행하고 있기 때문이다.

“No module named html.parser”와 같은 `ImportError` 에러라면, 파이썬 3 버전의 코드를 파이썬 2 아래에서 실행하고 있기 때문이다.

두 경우 모두 최선의 선택은 시스템에서 (압축파일을 풀 때 만들어진 디렉토리를 모두 포함하여) 뷰티플수프를 제거하고 다시 설치하는 것이다.

다음 `ROOT_TAG_NAME = u'[document]'` 줄에서 `SyntaxError "Invalid syntax"`를 맞이한다면, 파이썬 2 코드를 파이썬 3 코드로 변환할 필요가 있다. 이렇게 하려면 다음과 같이 패키지를 설치하거나:

```
$ python3 setup.py install
```

아니면 직접 파이썬의 2to3 변환 스크립트를 `bs4` 디렉토리에 실행하면 된다:

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

해석기 설치하기

뷰티플수프는 파이썬 표준 라이브러리에 포함된 HTML 해석기를 지원하지만, 또 수 많은 제-삼자 파이썬 해석기도 지원한다. 그 중 하나는 [lxml 해석기](#)이다. 설정에 따라, 다음 명령어들 중 하나로 `lxml`을 설치하는 편이 좋을 경우가 있다:

```
$ apt-get install python-lxml
```

```
$ easy_install lxml
```

```
$ pip install lxml
```

파이썬 2를 사용중이라면, 또다른 대안은 순수-파이썬 [html5lib 해석기](#)를 사용하는 것인데, 이 해석기는 HTML을 웹 브라우저가 해석하는 방식으로 해석한다. 설정에 따라 다음 명령어중 하나로 `html5lib`를 설치하는 것이 좋을 때가 있다:

```
$ apt-get install python-html5lib
```

```
$ easy_install html5lib
```

```
$ pip install html5lib
```

다음 표에 각 해석 라이브러리의 장점과 단점을 요약해 놓았다:

해석기	전형적 사용방법	장점	단점
파이썬의 <code>html.parser</code>	<code>BeautifulSoup(markup, "html.parser")</code>	<ul style="list-style-type: none">• 각종 기능 완비• 적절한 속도• 관대함 (파이썬 2.7.3과 3.2에서.)	<ul style="list-style-type: none">• 별로 관대하지 않음 (파이썬 2.7.3이나 3.2.2 이전 버전에서)
<code>lxml</code> 의 HTML 해석기	<code>BeautifulSoup(markup, "lxml")</code>	<ul style="list-style-type: none">• 아주 빠름	<ul style="list-style-type: none">• 외부 C 라이브러리

		• 관대함	리 의존
lxml의 XML 해석기	BeautifulSoup(markup, ["lxml", "xml"]) BeautifulSoup(markup, "xml")	<ul style="list-style-type: none"> • 아주 빠름 • 유일하게 XML 해석기 지원 	<ul style="list-style-type: none"> • 외부 C 라이브러리 의존
html5lib	BeautifulSoup(markup, html5lib)	<ul style="list-style-type: none"> • 아주 관대함 • 웹 브라우저의 방식으로 페이지를 해석함 • 유효한 HTML5를 생성함 	<ul style="list-style-type: none"> • 아주 느림 • 외부 파이썬 라이브러리 의존 • 파이썬 2 전용

가능하다면, 속도를 위해 lxml을 설치해 사용하시기를 권장한다. 2.7.3 이전의 파이썬2, 또는 3.2.2 이전의 파이썬 3 버전을 사용하면, lxml을 사용하는 것이 필수이다. 그렇지 않고 구형 버전의 파이썬 내장 HTML 해석기 html5lib는 별로 좋지 않다.

문서가 유효하지 않을 경우 해석기마다 다른 뷰티풀수프 트리를 생산한다는 사실을 주목하자. 자세한 것은 [해석기들 사이의 차이점들](#)을 살펴보자.

수프 만들기

문서를 해석하려면, 문서를 BeautifulSoup 구성자에 건네주자. 문자열 혹은 열린 파일 핸들을 건네면 된다:

```
from bs4 import BeautifulSoup

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

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

먼저, 문서는 유니코드로 변환되고 HTML 개체는 유니코드 문자로 변환된다:

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

다음 뷰티풀수프는 문서를 가장 적당한 해석기를 사용하여 해석한다. 특별히 XML 해석기를 사용하라고 지정해 주지 않으면 HTML 해석기를 사용한다. ([XML 해석하기](#) 참조.)

객체의 종류

뷰티풀수프는 복합적인 HTML 문서를 파이썬 객체로 구성된 복합적인 문서로 변환한다. 그러나 객체의 종류를 다루는 법만 알면 된다.

태그

Tag 객체는 원래 문서의 XML 태그 또는 HTML 태그에 상응한다:

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

태그는 많은 속성과 메소드가 있지만, 그 대부분을 나중에 [트리 항해하기](#) 그리고 [트리 검색하기](#)에서 다룰 생각이다. 지금은 태그의 가장 중요한 특징인 이름과 속성을 설명한다.

이름

태그마다 이름이 있고, 다음 .name 과 같이 접근할 수 있다:

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

태그의 이름을 바꾸면, 그 변화는 뷰티풀수프가 생산한 HTML 조판에 반영된다:

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

속성

태그는 속성을 여러개 가질 수 있다. <b class="boldest"> 태그는 속성으로 "class"가 있는데 그 값은 "boldest"이다. 태그의 속성에는 사전처럼 태그를 반복해 접근하면 된다:

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

사전에 .attrs와 같이 바로 접근할 수 있다:

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

태그의 속성을 추가, 제거, 변경할 수 있다. 역시 태그를 사전처럼 취급해서 처리한다:

```
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
```

값이-여럿인 속성

HTML 4에서 몇몇 속성은 값을 여러 개 가질 수 있도록 정의된다. HTML 5에서 그 중 2개는 제거되었지만, 몇 가지가 더 정의되었다. 가장 흔한 다중값 속성은 class이다 (다시 말해, 태그가 하나 이상의 CSS 클래스를 가질 수 있다). 다른 것으로는 rel, rev, accept-charset, headers, 그리고 accesskey가 포함된다. 뷰티풀수프는 다중-값 속성의 값들을 리스트로 나타낸다:

```
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"]
```

속성에 하나 이상의 값이 있는 것처럼 보이지만, HTML 표준에 정의된 다중-값 속성이 아니라면, 뷰티풀수프는 그 속성을 그대로 둔다:

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

태그를 다시 문자열로 바꾸면, 다중-값 속성은 합병된다:

```
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>
```

문서를 XML로 해석하면, 다중-값 속성은 없다:

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

NavigableString

문자열은 태그 안에 있는 일군의 텍스트에 상응한다. 뷰티풀수프는 NavigableString 클래스 안에다 이런 텍스트를 보관한다:

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

NavigableString은 파이썬의 유니코드 문자열과 똑 같은데, 단 [트리 항해하기](#)와 [트리 탐색하기](#)에 기술된 특징들도 지원한다는 점이 다르다. NavigableString을 유니코드 문자열로 변환하려면 unicode()를 사용한다:

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

문자열을 바로바로 편집할 수는 없지만, [replace_with\(\)](#)을 사용하면 한 문자열을 또다른 문자열로 바꿀 수 있다:

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

NavigableString은 [트리 항해하기](#)와 [트리 탐색하기](#)에 기술된 특징들을 모두는 아니지만, 대부분 지원한다. 특히, (태그에는 다른 문자열이나 또다른 태그가 담길 수 있지만) 문자열에는 다른 어떤 것도 담길 수 없기 때문에, 문자열은 .contents나 .string 속성, 또는 find() 메소드를 지원하지 않는다.

BeautifulSoup

BeautifulSoup 객체 자신은 문서 전체를 대표한다. 대부분의 목적에, 그것을 Tag 객체로 취급해도 좋다. 이것은 곧 [트리 항해하기](#)와 [트리 검색하기](#)에 기술된 메소드들을 지원한다는 뜻이다.

BeautifulSoup 객체는 실제 HTML 태그나 XML 태그에 상응하지 않기 때문에, 이름도 속성도 없다. 그러나 가끔 그의 이름 .name을 살펴보는 것이 유용할 경우가 있다. 그래서 특별히 .name에 "[document]"라는 이름이 주어졌다:

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

주석과 기타 특수 문자열들

Tag, NavigableString, 그리고 BeautifulSoup 정도면 HTML이나 XML 파일에서 보게될 거의 모든 것들을 망라한다. 그러나 몇 가지 남은 것들이 있다. 아마도 신경쓸 필요가 있는 것이 유일하게 있다면 바로 주석이다:

```
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 객체는 그냥 특별한 유형의 NavigableString이다:

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

그러나 HTML 문서의 일부에 나타나면, Comment는 특별한 형태로 화면에 표시된다:

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

뷰티플수프는 XML 문서에 나올만한 것들을 모두 클래스에다 정의한다: CData, ProcessingInstruction, Declaration, 그리고 Doctype이 그것이다. Comment와 똑같이, 이런 클래스들은 NavigableString의 하위클래스로서 자신의 문자열에 다른 어떤것들을 추가한다. 다음은 주석을 CDATA 블록으로 교체하는 예이다:

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

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

트리 항해하기

다시 또 "Three sisters" HTML 문서를 보자:

```
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)
```

이 예제로 한 문서에서 일부를 다른 곳으로 이동하는 법을 보여주겠다.

내려가기

태그에는 또다른 태그가 담길 수 있다. 이런 요소들은 그 태그의 자손(children)이라고 부른다. 뷰티풀수프는 한 태그의 자손을 향해 하고 반복하기 위한 속성을 다양하게 제공한다.

뷰티풀수프의 문자열은 이런 속성들을 제공하지 않음에 유의하자. 왜냐하면 문자열은 자손을 가질 수 없기 때문이다.

태그 이름을 사용하여 항해하기

가장 단순하게 해석 트리를 항해하는 방법은 원하는 태그의 이름을 지정해 주는 것이다. <head> 태그를 원한다면, 그냥 soup.head 라고 지정하면 된다:

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

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

이 트리를 반복해 사용하면 해석 트리의 특정 부분을 확대해 볼 수 있다. 다음 코드는 <body> 태그 아래에서 첫 번째 태그를 얻는다:

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

태그 이름을 속성으로 사용하면 오직 그 이름으로 첫 번째 태그만 얻는다:

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

<a> 태그를 모두 얻거나, 특정이름으로 첫 번째 태그 말고 좀 더 복잡한 어떤 것을 얻고 싶다면, [트리 탐색하기](#)에 기술된 메소드들을 사용해야 한다. 예를 들어, find_all()과 같은 메소드를 사용하면 된다:

```
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 **그리고** .children

태그의 자손은 .contents라고 부르는 리스트로 얻을 수 있다:

```
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 객체 자체에 자손이 있다. 이 경우, <html> 태그가 바로 BeautifulSoup 객체의 자손이다.:

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

문자열은 .contents를 가질 수 없는데, 왜냐하면 문자열 안에는 아무것도 담을 수 없기 때문이다:

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

자손을 리스트로 얻는 대신에, .children 발생자를 사용하면 태그의 자손을 반복할 수 있다:

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

.descendants

내용물(.contents)과 자손(.children) 속성은 오직 한 태그의 직계(direct) 자손만 고려한다. 예를 들면, <head> 태그는 오직 한 개의 직계 자손으로 <title> 태그가 있다:

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

그러나 <title> 태그 자체에 자손이 하나 있다: 문자열 "The Dormouse's story"가 그것이다. 그 문자열도 역시 <head> 태그의 자손이다. .descendants 속성은 한 태그의 자손들을 모두 재귀적으로, 반복할 수 있도록 해준다: 그의 직계 자손, 그 직계 자손의 자손, 등등:

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

<head> 태그는 오직 자손이 하나이지만, 후손은 둘이다: <title> 태그와 <title> 태그의 자손이 그것이다. BeautifulSoup 객체는 오직 하나의 직계 자손(<html> 태그)만 있지만, 수 많은 후손을 가진다:

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

.string

태그에 오직 자손이 하나라면, 그리고 그 자손이 NavigableString이라면, 그 자손은 .string으로 얻을 수 있다:

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

태그의 유일한 자손이 또다른 태그라면, 그리고 그 태그가 .string을 가진다면, 그 부모 태그는 같은 .string을 그의 자손으로 가진다고 간주된다:

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

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

태그에 하나 이상의 태그가 있다면, .string이 무엇을 가리킬지 확실하지 않다. 그래서 그럴 경우 .string은 None으로 정의된다:

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

.strings **그리고** stripped_strings

한 태그 안에 여러개의 태그가 있더라도 여전히 문자열을 볼 수 있다. .strings 발생자를 사용하자:

```
for string in soup.strings:
    print(repr(string))
# u"The Dormouse's story"
# u'WnWn'
```



```
# 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'
```

이런 문자열들은 공백이 쓸데 없이 많은 경향이 있으므로, 대신에 `.stripped_strings` 발생자를 사용해 제거해 버릴 수 있다:

```
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'...'
# u'\n'
```

여기에서, 전적으로 공백만으로 구성된 문자열은 무시되고 문자열 앞과 뒤의 공백은 제거된다.

올라가기

“가족 트리” 비유를 계속 사용해 보자. 태그마다 그리고 문자열마다 부모(`parent`)가 있다: 즉 자신을 담고 있는 태그가 있다.

```
.parent
```

한 요소의 부모는 `.parent` 속성으로 접근한다. 예제 “three sisters”문서에서, `<head>` 태그는 `<title>` 태그의 부모이다:

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

`title` 문자열 자체로 부모가 있다: 그 문자열을 담고 있는 `<title>` 태그가 그것이다:

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

`<html>` 태그와 같은 최상위 태그의 부모는 `BeautifulSoup` 객체 자신이다:

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

`BeautifulSoup` 객체의 `.parent`는 `None`으로 정의된다:

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

```
.parents
```

`.parents`로 한 요소의 부모들을 모두 다 반복할 수 있다. 다음 예제는 `.parents`를 사용하여 문서 깊숙히 묻힌 `<a>` 태그로부터 시작하여, 문서의 최상단까지 순회한다:

```
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
```

옆으로 가기

다음과 같은 간단한 문서를 생각해 보자:

```
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>
```

`` 태그와 `<c>` 태그는 같은 수준에 있다: 둘 다 같은 태그의 직계 자손이다. 이를 형제들(siblings)이라고 부른다. 문서가 pretty-printed로 출력되면, 형제들은 같은 들여쓰기 수준에서 나타난다. 이런 관계를 코드 작성에도 이용할 수 있다.

`.next_sibling` **그리고** `.previous_sibling`

`.next_sibling`과 `.previous_sibling`를 사용하면 해석 트리에서 같은 수준에 있는 페이지 요소들 사이를 항해할 수 있다:

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

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

`` 태그는 `.next_sibling`이 있지만, `.previous_sibling`은 없는데, 그 이유는 `` 태그 앞에 트리에서 같은 수준에 아무것도 없기 때문이다. 같은 이유로, `<c>` 태그는 `.previous_sibling`은 있지만 `.next_sibling`은 없다:

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

문자열“text1”과 “text2”는 형제 사이가 아니다. 왜냐하면 부모가 같지 않기 때문이다:

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

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

실제 문서에서, 한 태그의 `.next_sibling`이나 `.previous_sibling`은 보통 공백이 포함된 문자열이다. “three sisters” 문서로 되돌아가보자:

```
<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>
```

첫번째 `<a>` 태그의 `.next_sibling`이 두 번째 `<a>` 태그가 될 것이라고 생각할지 모르겠다. 그러나 실제로는 문자열이 다음 형제이다: 즉, 첫 번째 `<a>` 태그와 두 번째 태그를 가르는 쉼표와 새줄 문자가 그것이다:

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

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

두 번째 `<a>` 태그는 실제로는 그 쉼표의 `.next_sibling`이다:

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

.next_siblings **그리고** .previous_siblings

태그의 형제들은 .next_siblings이나 .previous_siblings로 반복할 수 있다:

```
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
```

앞뒤로 가기

“three sisters” 문서의 앞부분을 살펴보자:

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

HTML 해석기는 이 문자열들을 취해서 일련의 이벤트로 변환한다: “<html> 태그 열기”, “<head> 태그 열기”, “<title> 태그 열기”, “문자열 추가”, “<title> 태그 닫기”, “<p> 태그 열기”, 등등. 뷰티풀수프는 문서의 최초 해석 상태를 재구성하는 도구들을 제공한다.

.next_element **그리고** .previous_element

문자열이나 태그의 .next_element 속성은 바로 다음에 해석된 것을 가리킨다. .next_sibling과 같을 것 같지만, 보통 완전히 다르다.

다음은 “three sisters” 문서에서 마지막 <a> 태그이다. 그의 .next_sibling은 문자열이다: <a> 태그가 시작되어 중단되었던 문장의 끝부분이다:

```
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.'
```

그러나 <a> 태그의 .next_element는, 다시 말해 <a> 태그 바로 다음에 해석된 것은, 나머지 문장이 아니다: 그것은 단어 “Tillie”이다:

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

그 이유는 원래의 조판에서 단어 “Tillie”가 쌍반점보다 먼저 나타나기 때문이다. 해석기는 <a> 태그를 맞이하고, 다음으로 단어 “Tillie”, 그 다음 닫는 태그, 그 다음에 쌍반점과 나머지 문장을 맞이한다. 쌍반점은 <a> 태그와 같은 수준에 있지만, 단어 “Tillie”를 먼저 만난다.

.previous_element 속성은 .next_element와 정반대이다. 바로 앞에 해석된 요소를 가리킨다:

```
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 **그리고** .previous_elements

이제 이해가 가셨으리라 믿는다. 이런 반복자들을 사용하면 문서에서 해석하는 동안 앞 뒤로 이동할 수 있다:

```

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

```

트리 탐색하기

뷰티풀수프에는 해석 트리를 탐색하기 위한 메소드들이 많이 정의되어 있지만, 모두 다 거의 비슷하다. 가장 많이 사용되는 두 가지 메소드를 설명하는데 시간을 많이 할애할 생각이다: `find()`와 `find_all()`이 그것이다. 다른 메소드는 거의 똑 같은 인자를 취한다. 그래서 그것들은 그냥 간략하게 다루겠다.

다시 또, “three sisters” 문서를 예제로 사용하자:

```

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)

```

`find_all()`과 같이 인자에 여과기를 건네면, 얼마든지 문서에서 관심있는 부분을 뜯어낼 수 있다.

여과기의 종류

`find_all()`과 유사 메소드들에 관하여 자세히 설명하기 전에 먼저, 이런 메소드들에 건넬 수 있는 다양한 여과기의 예제들을 보여 주고 싶다. 이런 여과기들은 탐색 API 전체에 걸쳐서 나타나고 또 나타난다. 태그의 이름, 그의 속성, 문자열 텍스트, 또는 이런 것들을 조합하여 여과할 수 있다.

문자열

가장 단순한 여과기는 문자열이다. 문자열을 탐색 메소드에 건네면 뷰티풀수프는 그 정확한 문자열에 맞게 부합을 수행한다. 다음 코드는 문서에서 `` 태그를 모두 찾는다:

```

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

```

바이트 문자열을 건네면, 뷰티풀수프는 그 문자열이 UTF-8로 인코딩되어 있다고 간주한다. 이를 피하려면 대신에 유니코드 문자열을 건네면 된다.

정규 표현식

정규 표현식 객체를 건네면, 뷰티풀수프는 `match()` 메소드를 사용하여 그 정규 표현식에 맞게 여과한다. 다음 코드는 이름이 “b”로 시작하는 태그를 모두 찾는다; 이 경우, `<body>` 태그와 `` 태그를 찾을 것이다:

```

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

```

다음 코드는 이름에 ‘t’가 포함된 태그를 모두 찾는다:

```

for tag in soup.find_all(re.compile("t")):
    print(tag.name)

```

```
# html
# title
```

리스트

리스트를 건네면, 뷰티풀수프는 그 리스트에 담긴 항목마다 문자열 부합을 수행한다. 다음 코드는 모든 <a> 태그 그리고 모든 태그를 찾는다:

```
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 값은 참이면 모두 부합시킨다. 다음 코드는 문서에서 태그를 모두 찾지만, 텍스트 문자열은 전혀 찾지 않는다:

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

함수

다른 어떤 부합 기준도 마음에 안든다면, 요소를 그의 유일한 인자로 취하는 함수를 정의하면 된다. 함수는 인자가 부합하면 True를 돌려주고, 그렇지 않으면 False를 돌려주어야 한다.

다음은 태그에 "class"속성이 정의되어 있지만 "id" 속성은 없으면 True 를 돌려주는 함수이다:

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

이 함수를 find_all()에 건네면 <p> 태그를 모두 얻게 된다:

```
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>]
```

이 함수는 <p> 태그만 얻는다. <a> 태그는 획득하지 않는데, 왜냐하면 "class"와 "id"가 모두 정의되어 있기 때문이다. <html>과 <title>도 얻지 않는데, 왜냐하면 "class"가 정의되어 있지 않기 때문이다.

다음은 태그가 문자열 객체로 둘러 싸여 있으면 True를 돌려주는 함수이다:

```
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
```

이제 탐색 메소드들을 자세하게 살펴볼 준비가 되었다.

find_all()

서명: find_all([name](#), [attrs](#), [recursive](#), [text](#), [limit](#), [**kwargs](#))

find_all() 메소드는 태그의 후손들을 찾아서 지정한 여과기에 부합하면 모두 추출한다. [몇 가지 여과기](#)에서 예제들을 제시했지만, 여기에 몇 가지 더 보여주겠다:

```
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'
```

어떤 것은 익숙하지만, 다른 것들은 새로운 것이다. text 혹은 id에 값을 건넨다는 것이 무슨 뜻인가? 왜 다음 find_all("p", "title")은 CSS 클래스가 "title"인 <p> 태그를 찾는가? find_all()에 건넨 인자들을 살펴보자.

name 인자

인자를 name에 건네면 뷰티풀수프는 특정 이름을 가진 태그에만 관심을 가진다. 이름이 부합되지 않는 태그와 마찬가지로, 텍스트 문자열은 무시된다.

다음은 가장 단순한 사용법이다:

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

[여과기의 종류](#)에서 보았듯이 name에 건넨 값이 [문자열](#), [정규 표현식](#), [리스트](#), [함수](#), 또는 [True](#) 값일 수 있다는 사실을 기억하자.

키워드 인자

인지되지 않는 인자는 한 태그의 속성중 하나에 대한 여과기로 변환된다. id라는 인자에 대하여 값을 하나 건네면, 뷰티풀수프는 각 태그의 'id'속성에 대하여 걸러낸다:

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

href에 대하여 값을 건네면, 뷰티풀수프는 각 태그의 'href'속성에 대하여 걸러낸다:

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

[문자열](#), [정규 표현식](#), [리스트](#), [함수](#), 또는 [True](#) 값에 기반하여 속성을 걸러낼 수 있다.

다음 코드는 그 값이 무엇이든 상관없이, id 속성을 가진 태그를 모두 찾는다:

```
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>]
```

하나 이상의 키워드 인자를 건네면 한 번에 여러 값들을 걸러낼 수 있다:

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

CSS 클래스로 탐색하기

특정 CSS 클래스를 가진 태그를 탐색하면 아주 유용하지만, CSS 속성의 이름인 "class"는 파이썬에서 예약어이다. 키워드 인자로 class를 사용하면 구문 에러를 만나게 된다. 뷰티풀 4.1.2 부터, CSS 클래스로 검색할 수 있는데 class_ 키워드 인자를 사용하면 된다:

```
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>]
```

다른 키워드 인자와 마찬가지로, class_에 문자열, 정규 표현식, 함수, 또는 True를 건넬 수 있다:

```
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>]
```

[기억하자](#). 하나의 태그에 그의 "class" 속성에 대하여 값이 여러개 있을 수 있다. 특정 CSS 클래스에 부합하는 태그를 탐색할 때, 그의 CSS 클래스들 모두에 대하여 부합을 수행하는 것이다:

```
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>]
```

class 속성의 정확한 문자열 값을 탐색할 수도 있다:

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

그러나 문자열 값을 변형해서 탐색하면 작동하지 않는다:

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

class_를 위한 간편한 방법이 뷰티풀수프 모든 버전에 존재한다. find()-유형의 메소드에 건네는 두 번째 인자는 attrs인데, 문자열을 attrs에 건네면 그 문자열을 CSS 클래스처럼 탐색한다:

```
soup.find_all("a", "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>]
```

정규 표현식, 함수 또는 사전을 제외하고 True-유형으로도 건넬 수 있다. 무엇을 건네든지 그 CSS 클래스를 탐색하는데 사용된다. class_ 키워드 인자에 건넬 때와 똑같다:

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

사전을 attrs에 건네면, 단지 그 CSS 클래스만 아니라 한번에 많은 HTML 속성을 탐색할 수 있다. 다음 코드 두 줄은 동등하다:

```
soup.find_all(href=re.compile("elsie"), id='link1')
soup.find_all(attrs={'href': re.compile("elsie"), 'id': 'link1'})
```

이것은 별로 유용한 특징은 아니다. 왜냐하면 보통 키워드 인자를 사용하는 편이 더 쉽기 때문이다.

text 인자

text 인자로 태그 대신 문자열을 탐색할 수 있다. name과 키워드 인자에서처럼, [문자열](#), [정규 표현식](#), [리스트](#), [함수](#), 또는 [True](#) 값을 건넬 수 있다. 다음은 몇 가지 예이다:

```
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'...']
```

text가 문자열 찾기에 사용되지만, 태그를 찾는 인자와 결합해 사용할 수 있다: 뷰티플수프는 text에 대한 값에 자신의 .string이 부합하는 태그를 모두 찾는다. 다음 코드는 자신의 .string이 "Elsie"인 <a> 태그를 찾는다:

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

limit 인자

find_all() 메소드는 여과기에 부합하는 문자열과 태그를 모두 돌려준다. 이런 방법은 문서가 방대하면 시간이 좀 걸릴 수 있다. 결과로 가조리 필요한 것은 아니라면, limit에 숫자를 건넬 수 있다. 이 방법은 SQL에서의 LIMIT 키워드와 정확히 똑같이 작동한다. 뷰티플수프에게 특정 횟수를 넘어서면 결과 수집을 중지하라고 명령한다.

"three sisters" 문서에 링크가 세 개 있지만, 다음 코드는 앞의 두 링크만 찾는다:

```
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 인자

mytag.find_all()를 호출하면, 뷰티플수프는 mytag의 후손을 모두 조사한다: 그의 자손, 그 자손의 자손, 그리고 등등. 뷰티플수프에게 직계 자손만 신경쓰라고 시키고 싶다면, recursive=False를 건네면 된다. 다음에 차이점을 살펴보자:

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

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

다음은 예제 문서의 일부이다:

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

<title> 태그는 <html> 태그 아래에 있지만, <html> 태그 바로 아래에 있는 것은 아니다: <head> 태그가 사이에 있다. 뷰티플수프는 <html> 태그의 모든 후손을 찾아 보도록 허용해야만 <title> 태그를 발견한다. 그러나 recursive=False가 검색을 <html> 태그의 직접 자손으로 제한하기 때문에, 아무것도 찾지 못한다.

뷰티플수프는 트리-탐색 메소드들을 다양하게 제공한다 (아래에 다름). 대부분 find_all()과 같은 인자를 취한다: name, attrs, text, limit, 그리고 키워드 인자를 취한다. 그러나 recursive 인자는 다르다: find_all()과 find()만 유일하게 지원한다. recursive=False를 find_parents() 같은 인자에 건네면 별로 유용하지 않을 것이다.

태그를 호출하는 것은 find_all()을 호출하는 것과 똑같다

find_all()는 뷰티플수프 탐색 API에서 가장 많이 사용되므로, 그에 대한 간편 방법을 사용할 수 있다. BeautifulSoup 객체나 Tag 객체를 마치 함수처럼 다루면, 그 객체에 대하여 find_all()를 호출하는 것과 똑같다. 다음 코드 두 줄은 동등하다:

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

다음 두 줄도 역시 동등하다:

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

find()

서명: find([name](#), [attrs](#), [recursive](#), [text](#), [**kwargs](#))

find_all() 메소드는 전체 문서를 훑어서 결과를 찾지만, 어떤 경우는 결과 하나만 원할 수도 있다. 문서에 오직 <body> 태그가 하나 뿐임을 안다면, 전체 문서를 훑어 가면서 더 찾는 것은 시간 낭비이다. find_all 메소드를 호출할 때마다, limit=1을 건네기 보다는 find() 메소드를 사용하는 편이 좋다. 다음 코드 두 줄은 거의 동등하다:

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



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

유일한 차이점은 `find_all()` 메소드가 단 한개의 결과만 담고 있는 리스트를 돌려주고, `find()`는 그냥 그 결과를 돌려준다는 점이다.

`find_all()`이 아무것도 찾을 수 없다면, 빈 리스트를 돌려준다. `find()`가 아무것도 찾을 수 없다면, `None`을 돌려준다:

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

[태그 이름을 사용하여 항해하기](#)에서 `soup.head.title` 트릭을 기억하십니까? 그 트릭은 반복적으로 `find()`를 호출해서 작동한다:

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

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

`find_parents()` **그리고** `find_parent()`

서명: `find_parents(name, attrs, text, limit, **kwargs)`

서명: `find_parent(name, attrs, text, **kwargs)`

많은 시간을 할애해 `find_all()`과 `find()`를 다루었다. 뷰티풀수프 API에는 트리 탐색을 위해 다른 메소드가 열가지 정의되어 있지만, 걱정하지 말자. 이런 메소드중 다섯가지는 기본적으로 `find_all()`과 똑같고, 다른 다섯가지는 기본적으로 `find()`와 똑같다. 유일한 차이점은 트리의 어떤 부분을 검색할 것인가에 있다.

먼저 `find_parents()`와 `find_parent()`를 살펴보자. `find_all()`과 `find()`는 트리를 내려 오면서, 태그의 후손들을 찾음을 기억하자. 다음 메소드들은 정 반대로 일을 한다: 트리를 위로 올라가며, 한 태그의 (또는 문자열의) 부모를 찾는다. 시험해 보자. "three daughters" 문서 깊숙히 묻힌 문자열부터 시작해 보자:

```
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")
# []
```

세가지 `<a>` 태그 중 하나는 해당 문자열의 직계 부모이다. 그래서 탐색해서 그것을 찾는다. 세가지 `<p>` 태그 중 하나는 그 문자열의 방계 부모이고, 그것도 역시 잘 탐색한다. CSS 클래스가 "title"인 `<p>` 태그가 문서 어딘가에 존재하지만, 그것은 이 문자열의 부모가 아니므로, `find_parents()`로 부모를 찾을 수 없다.

아마도 `find_parent()`와 `find_parents()`, 그리고 앞서 언급한 `.parent`와 `.parents` 속성 사이에 관련이 있으리라 짐작했을 것이다. 이 관련은 매우 강력하다. 이 탐색 메소드들은 실제로 `.parents`로 부모들을 모두 찾아서, 제공된 여과기준에 부합하는지 하나씩 점검한다.

`find_next_siblings()` **그리고** `find_next_sibling()`

서명: `find_next_siblings(name, attrs, text, limit, **kwargs)`

서명: `find_next_sibling(name, attrs, text, **kwargs)`

이 메소드들은 `.next_siblings`을 사용하여 트리에서 한 요소의 나머지 형제들을 반복한다. `find_next_siblings()` 메소드는 부합하는 형제들을 모두 돌려주고, `find_next_sibling()` 메소드는 그 중 첫 째만 돌려준다:

```
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()` **그리고** `find_previous_sibling()`

서명: `find_previous_siblings(name, attrs, text, limit, **kwargs)`

서명: `find_previous_sibling(name, attrs, text, **kwargs)`

이 메소드들은 `.previous_siblings`를 사용하여 트리에서 한 원소의 앞에 나오는 형제들을 반복한다. `find_previous_siblings()` 메소드는 부합하는 형제들을 모두 돌려주고, `find_previous_sibling()`는 첫 째만 돌려준다:

```
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()` **그리고** `find_next()`

서명: `find_all_next(name, attrs, text, limit, **kwargs)`

서명: `find_next(name, attrs, text, **kwargs)`

이 메소드들은 `.next_elements`를 사용하여 문서에서 한 태그의 뒤에 오는 태그이든 문자열이든 무엇이든지 반복한다. `find_all_next()` 메소드는 부합하는 것들을 모두 돌려주고, `find_next()`는 첫 번째 부합하는 것만 돌려준다:

```
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',Wn', u'Lacie', u' andWn', u'Tillie',
#  u';Wnand they lived at the bottom of a well.', u'WnWn', u'...', u'Wn']
```

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

첫 예제에서, 문자열 "Elsie"가 나타났다. 물론 그 안에 우리가 시작했던 `<a>` 태그 안에 포함되어 있음에도 불구하고 말이다. 두 번째 예제를 보면, 문서의 마지막 `<p>` 태그가 나타났다. 물론 트리에서 우리가 시작했던 `<a>` 태그와 같은 부분에 있지 않음에도 불구하고 말이다. 이런 메소드들에게, 유일한 관심 사항은 원소가 여과 기준에 부합하는가 그리고 시작 원소 말고 나중에 문서에 나타나는가이다.

`find_all_previous()` **그리고** `find_previous()`

서명: `find_all_previous(name, attrs, text, limit, **kwargs)`

서명: `find_previous(name, attrs, text, **kwargs)`

이 메소드들은 `.previous_elements`를 사용하여 문서에서 앞에 오는 태그나 문자열들을 반복한다. `find_all_previous()` 메소드는 부합하는 모든 것을 돌려주고, `find_previous()`는 첫 번째 부합만 돌려준다:

```
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")`를 호출하면 문서에서 첫 번째 문단(`class="title"`)을 찾지만, 두 번째 문단 `<p>` 태그도 찾는다. 이 안에 우리가 시작한 `<a>` 태그가 들어 있다. 이것은 그렇게 놀랄 일이 아니다: 시작한 위치보다 더 앞에 나타나는 태그들을 모두 찾고 있는 중이다. `<a>` 태그가 포함된 `<p>` 태그는 자신 안에 든 `<a>` 태그보다 먼저 나타나는 것이 당연하다.

CSS 선택자

뷰티풀수프는 [CSS 선택자 표준](#)의 부분집합을 지원한다. 그냥 문자열로 선택자를 구성하고 그것을 Tag의 `.select()` 메소드 또는 BeautifulSoup 객체 자체에 건네면 된다.

다음과 같이 태그를 검색할 수 있다:

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

다른 태그 아래의 태그를 찾을 수 있다:

```
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>]
```

다른 태그 바로 아래에 있는 태그를 찾을 수 있다:

```
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("body > a")
# []
```

CSS 클래스로 태그를 찾는다:

```
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>]
```

ID로 태그를 찾는다:

```
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>]
```

속성이 존재하는지 테스트 한다:

```
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>]
```

속성 값으로 태그를 찾는다:

```
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>]
```

```
# <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>]
```

언어 코덱을 일치 시킨다:

```
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>]
```

이것은 CSS 선택자 구문을 알고 있는 사용자에게 유용하다. 이 모든 일들을 뷰티풀수프 API로 할 수 있다. CSS 선택자만 필요하다면, lxml을 직접 사용하는 편이 좋을 것이다. 왜냐하면, 더 빠르기 때문이다. 그러나 이렇게 하면 간단한 CSS 선택자들을 뷰티풀수프 API와 조합해 사용할 수 있다.

트리 변형하기

뷰티풀수프의 강점은 해석 트리를 검색 하는데에 있다. 그러나 또한 해석 트리를 변형해서 새로운 HTML 또는 XML 문서로 저장할 수도 있다.

태그 이름과 속성 바꾸기

이에 관해서는 [속성](#) 부분에서 다룬 바 있지만, 다시 반복할 가치가 있다. 태그 이름을 바꾸고 그의 속성 값들을 바꾸며, 속성을 새로 추가하고, 속성을 삭제할 수 있다:

```
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>
```

.string 변경하기

태그의 .string 속성을 설정하면, 태그의 내용이 주어진 문자열로 교체된다:

```
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>
```

주의하자: 태그에 또 다른 태그가 들어 있다면, 그 태그는 물론 모든 내용이 사라진다.

append()

Tag.append()로 태그에 내용을 추가할 수 있다. 파이썬 리스트에 .append()를 호출한 것과 똑같이 작동한다:

```
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()` **그리고** `.new_tag()`

문자열을 문서에 추가하고 싶다면, 파이썬 문자열을 `append()`에 건네기만 하면 된다. 아니면 `BeautifulSoup.new_string()` 공장 메소드를 호출하면 된다:

```
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']
```

완전히 새로 태그를 만들어야 한다면 어떻게 할까? 최선의 해결책은 `BeautifulSoup.new_tag()` 공장 메소드를 호출하는 것이다:

```
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>
```

오직 첫 번째 인자, 즉 태그 이름만 있으면 된다.

`insert()`

`Tag.insert()`는 `Tag.append()`와 거의 같은데, 단, 새 요소가 반드시 그의 부모의 `.contents` 끝에 갈 필요는 없다. 원하는 위치 어디든지 삽입된다. 파이썬 리스트의 `.insert()`와 똑같이 작동한다:

```
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()` **그리고** `insert_after()`

`insert_before()` 메소드는 태그나 문자열을 해석 트리에서 어떤 것 바로 앞에 삽입한다:

```
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()` 메소드는 해석 트리에서 다른 어떤 것 바로 뒤에 나오도록 태그나 문자열을 이동시킨다:

```
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()은 태그의 내용을 제거한다:

```
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()는 해석 트리에서 태그나 문자열을 제거한다. 추출하고 남은 태그나 문자열을 돌려준다:

```
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
```

이 시점에서 두 가지 해석 트리를 가지는 효과가 있다: 하나는 문서를 해석하는데 사용된 BeautifulSoup 객체에 뿌리를 두고, 또 하나는 추출된 그 태그에 뿌리를 둔다. 더 나아가 추출한 요소의 자손들에다 extract를 호출할 수 있다:

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

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

```
decompose()
```

Tag.decompose()는 태그를 트리에서 제거한 다음, 그와 그의 내용물을 완전히 파괴한다:

```
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()는 트리에서 태그나 문자열을 제거하고 그것을 지정한 태그나 문자열로 교체한다:

```
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()`는 교체된 후의 태그나 문자열을 돌려준다. 그래서 검사해 보거나 다시 트리의 다른 부분에 추가할 수 있다.

`wrap()`

`PageElement.wrap()`는 지정한 태그에 요소를 둘러싸서 새로운 포장자를 돌려준다:

```
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>
```

다음 메소드는 뷰티풀수프 4.0.5에 새로 추가되었다.

`unwrap()`

`Tag.unwrap()`은 `wrap()`의 반대이다. 태그를 그 태그 안에 있는 것들로 교체한다. 조판을 걷어내 버릴 때 좋다:

```
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>
```

`replace_with()`처럼, `unwrap()`은 교체된 후의 태그를 돌려준다.

(이전 뷰티풀수프 버전에서, `unwrap()`는 `replace_with_children()`이라고 불리웠으며, 그 이름은 여전히 작동한다.)

출력

예쁘게-인쇄하기

`prettify()` 메소드는 뷰티풀수프 해석 트리를 멋지게 모양을 낸 유니코드 문자열로 변환한다. HTML/XML 태그마다 따로따로 한 줄에 표시된다:

```
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>
```

최상위 `BeautifulSoup` 객체에 `prettify()`를 호출할 수 있으며, 또는 `Tag` 객체에 얼마든지 호출할 수 있다:

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

있는-그대로 인쇄하기

멋진 모양 말고 그냥 문자열을 원한다면, BeautifulSoup 객체, 또는 그 안의 Tag에 `unicode()` 또는 `str()`을 호출하면 된다:

```
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()` 함수는 UTF-8로 인코딩된 문자열을 돌려준다. 다른 옵션은 [인코딩](#)을 살펴보자.

또 `encode()`를 호출하면 `bytestring`을 얻을 수 있고, `decode()`로는 유니코드를 얻는다.

출력 포맷터

뷰티플수프 문서에 ““”와 같은 HTML 개체가 들어 있다면, 그 개체들은 유니코드 문자로 변환된다:

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
unicode(soup)
# u'<html><head></head><body>Wu201cDammit!Wu201d he said.</body></html>'
```

문서를 문자열로 변환하면, 유니코드 문자들은 UTF-8로 인코딩된다. HTML 개체는 다시 복구할 수 없다:

```
str(soup)
# '<html><head></head><body>Wxe2Wx80Wx9cDammit!Wxe2Wx80Wx9d he said.</body></html>'
```

기본 값으로, 출력에서 피싱 처리가 되는 유일한 문자들은 앰퍼센드와 옆격쇠 문자들이다. 이런 문자들은 “&”와 “<”, 그리고 “>”로 변환된다. 그래서 뷰티플수프는 무효한 HTML이나 XML을 생성하는 실수를 하지 않게 된다:

```
soup = BeautifulSoup("<p>The law firm of Dewey, Cheatem, & Howe</p>")
soup.p
# <p>The law firm of Dewey, Cheatem, & Howe</p>

soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a>')
soup.a
# <a href="http://example.com/?foo=val1&amp;bar=val2">A link</a>
```

이 행위를 바꾸려면 `formatter` 인자용 값을 `prettify()`, `encode()`, 또는 `decode()`에 제공하면 된다. 뷰티플수프는 `formatter`에 대하여 가능한 네 가지 값을 인지한다.

기본값은 `formatter="minimal"`이다. 문자열은 뷰티플수프가 유효한 HTML/XML을 생산한다고 확신할 만큼 처리된다:

```
french = "<p>Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;</p>"
soup = BeautifulSoup(french)
print(soup.prettify(formatter="minimal"))
# <html>
# <body>
# <p>
#   Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;
# </p>
# </body>
# </html>
```

`formatter="html"`을 건네면, 뷰티플수프는 유니코드 문자를 가능한한 HTML 개체로 변환한다:

```
print(soup.prettify(formatter="html"))
# <html>
# <body>
# <p>
#   Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;
# </p>
# </body>
# </html>
```

`formatter=None`을 건네면, 뷰티플수프는 출력시 전혀 문자열을 건드리지 않는다. 이것이 가장 빠른 선택이지만, 다음 예제에서와 같이 잘못해서 뷰티플수프가 무효한 HTML/XML을 생산할 가능성이 있다:

```
print(soup.prettify(formatter=None))
# <html>
# <body>
# <p>
#   Il a dit <<Sacr&eacute; bleu!>>
# </p>
# </body>
# </html>
```



```
link_soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a>')
print(link_soup.a.encode(formatter=None))
# <a href="http://example.com/?foo=val1&bar=val2">A link</a>
```

마지막으로, `formatter`에 함수를 건네면, 뷰티풀수프는 문서에서 문자열과 속성 값에 대하여 하나하나 그 함수를 한 번 호출한다. 이 함수에서 무엇이든 할 수 있다. 다음은 문자열을 대문자로 바꾸고 다른 일은 절대로 하지 않는 포맷터이다:

```
def uppercase(str):
    return str.upper()

print(soup.prettify(formatter=uppercase))
# <html>
# <body>
# <p>
#   IL A DIT <<SACRÉ BLEU!>>
# </p>
# </body>
# </html>

print(link_soup.a.prettify(formatter=uppercase))
# <a href="HTTP://EXAMPLE.COM/?FOO=VAL1&BAR=VAL2">
#   A LINK
# </a>
```

따로 함수를 작성하고 있다면, `bs4.dammit` 모듈에 있는 `EntitySubstitution` 클래스에 관하여 알아야 한다. 이 클래스는 뷰티풀수프의 표준 포맷터를 클래스 메소드로 구현한다: “html” 포맷터는 `EntitySubstitution.substitute_html`이고, “minimal” 포맷터는 `EntitySubstitution.substitute_xml`이다. 이 함수들을 사용하면 `formatter=html`나 `formatter=minimal`를 흉내낼 수 있지만, 더 처리해야 할 일이 있다.

다음은 가능하면 유니코드 문자를 HTML 개체로 교체하는 예제이다. 그러나 또한 모든 문자열을 대문자로 바꾼다:

```
from bs4.dammit import EntitySubstitution
def uppercase_and_substitute_html_entities(str):
    return EntitySubstitution.substitute_html(str.upper())

print(soup.prettify(formatter=uppercase_and_substitute_html_entities))
# <html>
# <body>
# <p>
#   IL A DIT &lt;&lt;SACR&Eacute; BLEU!&gt;&gt;
# </p>
# </body>
# </html>
```

마지막 단점: `CData` 객체를 만들면, 그 객체 안의 텍스트는 언제나 포매팅 없이도, 정확하게 똑같이 나타난다. 문서에서 문자열 같은 것들을 세는 메소드를 손수 만들 경우, 뷰티풀수프는 포맷터 메소드를 호출한다. 그러나 반환 값은 무시된다.

```
from bs4.element import CData
soup = BeautifulSoup("<a></a>")
soup.a.string = CData("one < three")
print(soup.a.prettify(formatter="xml"))
# <a> # <![CDATA[one < three]]> # </a>
```

get_text()

문서나 태그에서 텍스트 부분만 추출하고 싶다면, `get_text()` 메소드를 사용할 수 있다. 이 메소드는 문서나 태그 아래의 텍스트를, 유니코드 문자열 하나로 모두 돌려준다:

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

soup.get_text()
u'WnI linked to example.comWn'
soup.i.get_text()
u'example.com'
```

텍스트를 합칠 때 사용될 문자열을 지정해 줄 수 있다:

```
# soup.get_text("|")
u'WnI linked to |example.com|Wn'
```

뷰티풀수프에게 각 테스트의 앞과 뒤에 있는 공백을 건너내라고 알려줄 수 있다:

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

그러나 이 시점에서 대신에 [stripped_strings](#) 발생자를 사용해서, 텍스트를 손수 처리하고 싶을 수 있겠다:

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

사용할 해석기 지정하기

단지 HTML만 해석하고 싶을 경우, 조판을 BeautifulSoup 구성자에 넣기만 하면, 아마도 잘 처리될 것이다. 뷰티풀수프는 해석기를 여러분 대신 선택해 데이터를 해석한다. 그러나 어느 해석기를 사용할지 바꾸기 위해 구성자에 건넬 수 있는 인자가 몇 가지 더 있다.

BeautifulSoup 구성자에 건네는 첫 번째 인자는 문자열이나 열린 파일 핸들-즉 해석하기를 원하는 조판이 첫 번째 인자이다. 두 번째 인자는 그 조판이 어떻게 해석되기를 바라는지 지정한다.

아무것도 지정하지 않으면, 설치된 해석기중 최적의 HTML 해석기가 배당된다. 뷰티풀수프는 lxml 해석기를 최선으로 취급하고, 다음에 html5lib 해석기, 그 다음이 파이썬의 내장 해석기를 선택한다. 이것은 다음 중 하나로 덮어쓸 수 있다:

- 해석하고 싶은 조판의 종류. 현재 "html", "xml", 그리고 "html5"가 지원된다.
- 사용하고 싶은 해석기의 이름. 현재 선택은 "lxml", "html5lib", 그리고 "html.parser" (파이썬의 내장 HTML 해석기)이다.

[해석기 설치하기](#) 섹션에 지원 해석기들을 비교해 놓았다.

적절한 해석기가 설치되어 있지 않다면, 뷰티풀수프는 여러분의 요구를 무시하고 다른 해석기를 선택한다. 지금 유일하게 지원되는 XML 해석기는 lxml이다. lxml 해석기가 설치되어 있지 않으면, XML 해석기를 요구할 경우 아무것도 얻을 수 없고, "lxml"을 요구하더라도 얻을 수 없다.

해석기 사이의 차이점들

뷰티풀수프는 다양한 해석기에 대하여 인터페이스가 같다. 그러나 각 해석기는 다르다. 해석기마다 같은 문서에서 다른 해석 트리를 만들어낸다. 가장 큰 차이점은 HTML 해석기와 XML 해석기 사이에 있다. 다음은 HTML로 해석된 짧은 문서이다:

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

빈 태그는 유효한 HTML이 아니므로, 해석기는 그것을 태그 쌍으로 변환한다.

다음 똑같은 문서를 XML로 해석한 것이다 (이를 실행하려면 lxml이 설치되어 있어야 한다). 빈 태그가 홀로 남았음에 유의하자. 그리고 <html> 태그를 출력하는 대신에 XML 선언이 주어졌음을 주목하자:

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

HTML 해석기 사이에서도 차이가 있다. 뷰티풀수프에 완벽하게 모양을 갖춘 HTML 문서를 주면, 이 차이는 문제가 되지 않는다. 비록 해석기마다 속도에 차이가 있기는 하지만, 모두 원래의 HTML 문서와 정확하게 똑같이 보이는 데이터 구조를 돌려준다.

그러나 문서가 불완전하게 모양을 갖추었다면, 해석기마다 결과가 다르다. 다음은 짧은 무효한 문서를 lxml의 HTML 해석기로 해석한 것이다. 나홀로 </p> 태그는 그냥 무시된다:

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

다음은 같은 문서를 html5lib로 해석하였다:

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

나홀로 </p> 태그를 무시하는 대신에, html5lib는 여는 <p> 태그로 짝을 맞추어 준다. 이 해석기는 또한 빈 <head> 태그를 문서에 추가한다.

다음은 같은 문서를 파이썬 내장 HTML 해석기로 해석한 것이다:

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

html5lib처럼, 이 해석기는 닫는 </p> 태그를 무시한다. html5lib와 다르게, 이 해석기는 <body> 태그를 추가해서 모양을 갖춘 HTML 문서를 생성하려고 아무 시도도 하지 않는다. lxml과 다르게, 심지어 <html> 태그를 추가하는 것에도 신경쓰지 않는다.

문서 "<a></p>"는 무효하므로, 이 테크닉중 어느 것도 "올바른" 처리 방법이 아니다. html5lib 해석기는 HTML5 표준에 있는 테크닉을 사용하므로, 아무래도 "가장 올바른" 방법이라고 주장할 수 있지만, 세 가지 테크닉 모두 같은 주장을 할 수 있다.

해석기 사이의 차이점 때문에 스크립트가 영향을 받을 수 있다. 스크립트를 다른 사람들에게 나누어 줄 계획이 있다면, 또는 여러 머신에서 실행할 생각이라면, BeautifulSoup 구성자에 해석기를 지정해 주는 편이 좋다. 그렇게 해야 여러분이 해석한 방식과 다르게 사용자가 문서를 해석할 위험성이 감소한다.

인코딩

HTML이든 XML이든 문서는 ASCII나 UTF-8 같은 특정한 인코딩으로 작성된다. 그러나 문서를 뷰티풀수프에 적재하면, 문서가 유니코드로 변환되었음을 알게 될 것이다:

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

마법이 아니다(확실히 좋은 것이다.). 뷰티풀수프는 [Unicode, Dammit](#)라는 하위 라이브러리를 사용하여 문서의 인코딩을 탐지하고 유니코드로 변환한다. 자동 인코딩 탐지는 BeautifulSoup 객체의 `.original_encoding` 속성으로 얻을 수 있다:

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



Unicode, Dammit은 대부분 올바르게 추측하지만, 가끔은 실수가 있다. 가끔 올바르게 추측하지만, 문서를 바이트 하나 하나 오랫동안 탐색한 후에야 그렇다. 혹시 문서의 인코딩을 미리 안다면, 그 인코딩을 BeautifulSoup 구성자에 `from_encoding`로 건네면 실수를 피하고 시간을 절약할 수 있다.

다음은 ISO-8859-8로 작성된 문서이다. 이 문서는 Unicode, Dammit이 충분히 살펴보기에는 너무 짧아서, ISO-8859-7로 잘못 인식한다:

```
markup = b"<h1>                </h1>"
soup = BeautifulSoup(markup)
soup.h1
<h1>                </h1>
soup.original_encoding
'ISO-8859-7'
```

이를 해결하려면 올바른 `from_encoding`을 건네면 된다:

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

아주 드물게 (보통 UTF-8 문서 안에 텍스트가 완전히 다른 인코딩으로 작성되어 있을 경우), 유일하게 유니코드를 얻는 방법은 몇 가지 문자를 특별한 유니코드 문자 “REPLACEMENT CHARACTER” (U+FFFD, )로 교체하는 것이다. Unicode, Dammit이 이를 필요로 하면, UnicodeDammit이나 BeautifulSoup 객체에 대하여 `.contains_replacement_characters` 속성에 True를 설정할 것이다. 이렇게 하면 유니코드 표현이 원래의 정확한 표현이 아니라는 사실을 알 수 있다. 약간 데이터가 손실된다. 문서에 가 있지만, `.contains_replacement_characters`가 False라면, 원래부터 거기에 있었고 데이터 손실을 감내하지 않는다는 사실을 알게 될 것이다.

출력 인코딩

뷰티풀수프로 문서를 작성할 때, UTF-8 문서를 얻는다. 그 문서가 처음에는 UTF-8이 아니었다고 할지라도 말이다. 다음은 Latin-1 인코딩으로 작성된 문서이다:

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

<meta> 태그가 재작성되어 문서가 이제 UTF-8이라는 사실을 반영하고 있음을 주목하자.

UTF-8이 싫으면, 인코딩을 `prettify()`에 건넬 수 있다:

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

또 `encode()`를 BeautifulSoup 객체, 또는 수프의 다른 어떤 요소에라도 호출할 수 있다. 마치 파이썬 문자열처럼 말이다:

```
soup.p.encode("latin-1")
# '<p>SacrWxe9 bleu!</p>'

soup.p.encode("utf-8")
# '<p>SacrWxc3Wxa9 bleu!</p>'
```

선택한 인코딩에서 표현이 불가능한 문자는 숫자의 XML 개체 참조로 변환된다. 다음은 유니코드 문자 SNOWMAN이 포함된 문자 이다:

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

눈사람 문자는 UTF-8 문서에 포함될 수 있지만 (☺처럼 생김), ISO-Latin-1이나 ASCII에 그 문자에 대한 표현이 없다. 그래서 "☃"으로 변환된다:

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

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

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

이런, 유니코드군

뷰티풀수프를 사용하지 않더라도 유니코드를 사용할 수 있다. 인코딩을 알 수 없는 데이터가 있을 때마다 그냥 유니코드가 되어 주었으면 하고 바라기만 하면 된다:

```
from bs4 import UnicodeDammit
dammit = UnicodeDammit("SacrWxc3Wxa9 bleu!")
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'utf-8'
```

유니코드에 더 많은 데이터를 줄 수록, Dammit은 더 정확하게 추측할 것이다. 나름대로 어떤 인코딩일지 짐작이 간다면, 그것들을 리스트로 건넬 수 있다:

```
dammit = UnicodeDammit("SacrWxe9 bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

Unicode, Dammit는 뷰티풀수프가 사용하지 않는 특별한 특징이 두 가지 있다.

지능형 따옴표

Unicode, Dammit을 사용하여 마이크로소프트 지능형 따옴표를 HTML이나 XML 개체로 변환할 수 있다:

```
markup = b"<p>I just Wx93loveWx94 Microsoft WordWx92s 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>'
```

또 마이크로소프트 지능형 따옴표를 ASCII 따옴표로 변환할 수 있다:

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

모쪼록 이 특징이 쓸모가 있기를 바라지만, 뷰티풀수프는 사용하지 않는다. 뷰티풀수프는 기본 행위를 선호하는데, 기본적으로 마이크로소프트 지능형 따옴표를 다른 모든 것과 함께 유니코드 문자로 변환한다:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just Wu201cloveWu201d Microsoft WordWu2019s smart quotes</p>'
```

비 일관적인 인코딩

어떤 경우 문서 대부분이 UTF-8이지만, 안에 (역시) 마이크로소프트 지능형 따옴표와 같이 Windows-1252 문자가 들어 있는 경우가 있다. 한 웹 사이트에 여러 소스로 부터 데이터가 포함될 경우에 이런 일이 일어날 수 있다. UnicodeDammit.detwingle()을 사용하여 그런 문서를 순수한 UTF-8 문서로 변환할 수 있다. 다음은 간단한 예이다:

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

이 문서는 뒤죽박죽이다. 눈사람은 UTF-8인데 따옴표는 Windows-1252이다. 눈사람 아니면 따옴표를 화면에 나타낼 수 있지만, 둘 다 나타낼 수는 없다:

```
print(doc)
#       I like snowmen!      
```

```
print(doc.decode("windows-1252"))
#  ~f ~f f "I like snowmen!"
```

문서를 UTF-8로 디코딩하면 UnicodeDecodeError가 일어나고, Windows-1252로 디코딩하면 알 수 없는 글자들이 출력된다. 다행스럽게도, UnicodeDammit.detwingle()는 그 문자열을 순수 UTF-8로 변환해 주므로, 유니코드로 디코드하면 눈사람과 따옴표를 동시에 화면에 보여줄 수 있다:

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

UnicodeDammit.detwingle()는 오직 UTF-8에 임베드된 (또는 그 반대일 수도 있지만) Windows-1252을 다루는 법만 아는데, 이것이 가장 일반적인 사례이다.

BeautifulSoup이나 UnicodeDammit 구성자에 건네기 전에 먼저 데이터에 UnicodeDammit.detwingle()을 호출하는 법을 반드시 알아야 한다. 뷰티풀수프는 문서에 하나의 인코딩만 있다고 간주한다. 그것이 무엇이든 상관없이 말이다. UTF-8과 Windows-1252를 모두 포함한 문서를 건네면, 전체 문서가 Windows-1252라고 생각할 가능성이 높고, 그 문서는 다음 ` ~f ~f f"I like snowmen!"`처럼 보일 것이다.

UnicodeDammit.detwingle()은 뷰티풀수프 4.1.0에서 새로 추가되었다.

문서의 일부만을 해석하기

뷰티풀수프를 사용하여 문서에서 <a> 태그를 살펴보고 싶다고 해보자. 전체 문서를 해석해서 훑어가며 <a> 태그를 찾는 일은 시간 낭비이자 메모리 낭비이다. 처음부터 <a> 태그가 아닌 것들을 무시하는 편이 더 빠를 것이 분명하다. SoupStrainer 클래스는 문서에 어느 부분을 해석할지 고르도록 해준다. 그냥 SoupStrainer를 만들고 그것을 BeautifulSoup 구성자에 parse_only 인자로 건네면 된다.

(이 특징은 html5lib 해석기를 사용중이라면 작동하지 않음을 주목하자. html5lib을 사용한다면, 어쨌거나 문서 전체가 해석된다. 이것은 html5lib가 작업하면서 항상 해석 트리를 재정렬하기 때문이다. 문서의 일부가 실제로 해석 트리에 맞지 않을 경우, 충돌을 일으킨다. 혼란을 피하기 위해, 아래의 예제에서 뷰티풀수프에게 파이썬의 내장 해석기를 사용하라고 강제하겠다.)

SoupStrainer

SoupStrainer 클래스는 [트리 탐색하기](#)의 전형적인 메소드와 같은 인자들을 취한다: [name](#), [attrs](#), [text](#), 그리고 [**kwargs](#)이 그 인자들이다. 다음은 세 가지 SoupStrainer 객체이다:

```
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)
```

다시 한 번 더 "three sisters" 문서로 돌아가 보겠다. 문서를 세 가지 SoupStrainer 객체로 해석하면 어떻게 보이는지 살펴보자:

```
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
# ...
#
```

또한 SoupStrainer를 [트리 탐색하기](#)에서 다룬 메소드에 건넬 수 있다. 이는 별로 유용하지는 않지만, 그럼에도 언급해 둔다:

```
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']
```

문제 해결

버전 불일치 문제

- SyntaxError: Invalid syntax (다음 ROOT_TAG_NAME = u'[document]' 줄에서): 코드를 변경하지 않고서, 파이썬 2 버전의 뷰티풀수프를 파이썬 3 아래에서 사용하기 때문에 야기된다.
- ImportError: No module named HTMLParser - 파이썬 2 버전의 뷰티풀수프를 파이썬 3 아래에서 사용하기 때문에 야기된다.
- ImportError: No module named html.parser - 파이썬 3 버전의 뷰티풀수프를 파이썬 2에서 실행하기 때문에 야기된다.

- ImportError: No module named BeautifulSoup - 뷰티플수프 3 코드를 BS3가 설치되어 있지 않은 시스템에서 실행할 때 야기된다. 또는 꾸러미 이름이 bs4로 바뀌었음을 알지 못하고 뷰티플수프 4 코드를 실행하면 야기된다.
- ImportError: No module named bs4 - 뷰티플수프 4 코드를 BS4가 설치되어 있지 않은 시스템에서 실행하면 야기된다.

XML 해석하기

기본값으로, 뷰티플수프는 문서를 HTML로 해석한다. 문서를 XML로 해석하려면, "xml"를 두 번째 인자로 BeautifulSoup 구성자에 건네야 한다:

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

[lxml이 설치되어 있어야 한다.](#)

기타 해석기 문제

- 스크립트가 한 컴퓨터에서는 잘 되는데 다른 컴퓨터에서는 작동하지 않는다면, 아마도 두 컴퓨터가 다른 해석기를 가지고 있기 때문일 것이다. 예를 들어, lxml이 설치된 컴퓨터에서 스크립트를 개발해 놓고, 그것을 html5lib만 설치된 컴퓨터에서 실행하려고 했을 수 있다. 왜 이것이 문제가 되는지는 [해석기들 사이의 차이점](#)을 참고하고, BeautifulSoup 구성자에 특정 라이브러리를 지정해서 문제를 해결하자.
- HTMLParser.HTMLParseError: malformed start tag or HTMLParser.HTMLParseError: bad end tag - 파이썬의 내장 HTML 해석기에 처리가 불가능한 문서를 건네면 야기된다. 다른 HTMLParseError도 아마 같은 문제일 것이다. 해결책: [lxml이나 html5lib를 설치하자.](#)
- 알고 있는데 문서에서 그 태그를 발견할 수 없다면 (다시 말해, find_all()이 []를 돌려주거나 find()가 None을 돌려줄 경우), 아마도 파이썬의 내장 HTML 해석기를 사용하고 있을 가능성이 높다. 이 해석기는 가끔 이해하지 못하면 그 태그를 무시하고 지나간다. 해결책: [lxml이나 html5lib를 설치하자.](#)
- [HTML 태그와 속성](#)은 대소문자를 구별하므로, 세가지 HTML 해석기 모두 태그와 속성 이름을 소문자로 변환한다. 다시 말해, 다음 조판 <TAG> </TAG>는 <tag> </tag>로 변환된다. 태그와 속성에 대소문자 혼합 또는 대문자를 그대로 유지하고 싶다면, [문서를 XML로 해석할 필요가 있다.](#)

기타

- KeyError: [attr] - tag['attr']에 접근했는데 해당 태그에 attr 속성이 정의되어 있지 않을 때 야기된다. 가장 흔한 예라는 KeyError: 'href' 그리고 KeyError: 'class'이다. attr이 정의되어 있는지 잘 모르겠다면, 파이썬 사전에 그렇게 하듯이, tag.get('attr')을 사용하자.
- UnicodeEncodeError: 'charmap' codec can't encode character u'Wxfoo' in position bar (또는 그냥 기타 다른 UnicodeEncodeError에 관한 모든 것) - 이 예라는 뷰티플수프에 관련된 문제가 아니다. 이 문제는 두 가지 상황에서 출현한다. 첫 째, 유니코드 문자열을 인쇄했는데 콘솔이 표시할 줄 모를 경우가 있다. ([파이썬 위키에서](#) 도움을 받자.) 둘째, 파일에 쓰는데 기본 인코딩으로 지원되지 않는 유니코드 문자열을 건넨 경우가 있다. 이런 경우, 가장 쉬운 해결책은 u.encode("utf8")을 지정해서 그 유니코드 문자열을 UTF-8로 명시적으로 인코딩하는 것이다.

수행성능 개선

뷰티플수프는 그 밑에 깔린 해석기보다 더 빠를 수는 없다. 응답 시간이 중요하다면, 다시 말해, 시간제로 컴퓨터를 쓰고 있거나 아니면 컴퓨터 시간이 프로그래머 시간보다 더 가치가 있는 다른 이유가 조금이라도 있다면, 그렇다면 뷰티플수프는 잊어 버리고 직접 [lxml](#) 위에 작업하는 편이 좋을 것이다.

그렇지만, 뷰티플수프의 속도를 높일 수 있는 방법이 있다. 아래에 해석기로 lxml을 사용하고 있지 않다면, [당장 시작해 보기를](#) 조언한다. 뷰티플수프는 html.parser나 html5lib를 사용하는 것보다 lxml을 사용하는 것이 문서를 상당히 더 빠르게 해석한다.

[cchardet](#) 라이브러리를 설치하면 인코딩 탐지 속도를 상당히 높일 수 있다.

가끔 [Unicode, Dammit](#)는 바이트별로 파일을 조사해서 인코딩을 탐지할 수 있을 뿐이다. 이 때문에 뷰티플수프가 기어가는 원인이 된다. 본인의 테스트에 의하면 이런 일은 파이썬 2.x 버전대에서만 일어나고, 러시아나 중국어 인코딩을 사용한 문서에 아주 많이 발생했다. 이런 일이 일어나면, cchardet을 설치하거나, 스크립트에 Python 3를 사용하여 문제를 해결할 수 있다. 혹시 문서의 인코딩을 안다면, 그 인코딩을 BeautifulSoup 구성자에 from_encoding로 건네면, 인코딩 탐지를 완전히 건너뛴다.

[문서의 일부만 해석하기](#)는 문서를 해석하는 시간을 많이 절약해 주지는 못하겠지만, 메모리가 절약되고, 문서를 훨씬 더 빨리 탐색할 수 있을 것이다.

뷰티플수프 3

뷰티플수프 3는 이전의 구형으로서, 더 이상 활발하게 개발되지 않는다. 현재는 주요 리눅스 배포본에 모두 함께 꾸러넣어진다:

```
$ apt-get install python-beautifulsoup
```

또 PyPi를 통하여 BeautifulSoup로 출간되어 있다:

```
$ easy_install BeautifulSoup
```

```
$ pip install BeautifulSoup
```

또한 [뷰티플수프 3.2.0](#) 압축파일을 내려받을 수 있다.

easy_install beautifulsoup이나 easy_install BeautifulSoup을 실행했는데, 코드가 작동하지 않으면, 실수로 뷰티플수프 3을 설치한 것이다. easy_install beautifulsoup4을 실행할 필요가 있다.

[뷰티플수프 3 문서는 온라인에 보관되어 있다](#). 모국어가 중국어라면, [뷰티플수프 3 문서 중국어 번역본](#)을 보는 것이 더 쉬울 것이다. 그 다음에 이 문서를 읽고 뷰티플수프 4에서 변한 것들을 알아보자.

BS4로 코드 이식하기

뷰티플수프 3용 코드는 하나만 살짝 바꾸면 뷰티플수프 4에도 작동한다. 꾸러미 이름을 BeautifulSoup에서 bs4로 바꾸기만 하면 된다. 그래서 다음은:

```
from BeautifulSoup import BeautifulSoup
```

다음과 같이 된다:

```
from bs4 import BeautifulSoup
```

- “No module named BeautifulSoup”와 같이 ImportError를 만난다면, 문제는 뷰티플수프 3 코드를 시도하는데 뷰티플수프 4만 설치되어 있기 때문이다.
- “No module named bs4”와 같은 ImportError를 만난다면, 문제는 뷰티플수프 4 코드를 시도하는데 뷰티플수프 3만 설치되어 있기 때문이다.

BS4는 BS3와 대부분 하위 호환성이 있으므로, 대부분의 메쏘드는 폐기되고 [PEP 8을 준수하기 위해](#) 새로운 이름이 주어졌다. 이름 바꾸기와 변화가 많이 있지만, 그 중에 몇 가지는 하위 호환성이 깨진다.

다음은 BS3 코드를 변환해 BS4에 이식하고자 할 때 알아야 할 것들이다:

해석기가 필요해

뷰티플수프 3는 파이썬의 SGMLParser해석기를 사용했다. 이 모듈은 파이썬 3.0에서 제거되었다. 뷰티플수프 4는 기본적으로 html.parser를 사용하지만, 대신에 lxml이나 html5lib을 설치해 사용할 수 있다. 비교는 [해석기 설치하기](#)를 참조하자.

html.parser는 SGMLParser와 같은 해석기가 아니기 때문에, 무효한 조판을 다르게 취급한다. 보통 “차이점은” 무효한 조판을 다룰 경우 html.parser가 해석기가 충돌을 일으키는 것이다. 이런 경우, 또다른 해석기를 설치할 필요가 있다. 그러나 html.parser는 SGMLParser와는 다른 해석 트리를 생성한다. 이런 일이 일어나면, BS3 코드를 업데이트하여 새로운 트리를 다루도록 해야 할 필요가 있다.

메쏘드 이름

- renderContents -> encode_contents
- replaceWith -> replace_with
- replaceWithChildren -> unwrap
- findAll -> find_all
- findAllNext -> find_all_next
- findAllPrevious -> find_all_previous
- findNext -> find_next
- findNextSibling -> find_next_sibling
- findNextSiblings -> find_next_siblings
- findParent -> find_parent
- findParents -> find_parents
- findPrevious -> find_previous
- findPreviousSibling -> find_previous_sibling
- findPreviousSiblings -> find_previous_siblings
- nextSibling -> next_sibling
- previousSibling -> previous_sibling

뷰티플수프 구성자에 건네는 인자들 중에서 같은 이유로 이름이 바뀌었다:

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

파이썬 3와의 호환을 위해 한 가지 메소드 이름을 바꾸었다:

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

더 정확한 용어를 위해 한 속성의 이름을 바꾸었다:

- `Tag.isSelfClosing` -> `Tag.is_empty_element`

파이썬에서 특별한 의미가 있는 단어들을 피해서 세 가지 속성의 이름을 바꾸었다. 다른 것들과 다르게 이 변경사항은 하위 호환이 되지 않는다. 이런 속성을 BS3에 사용하면, BS4로 이식할 때 코드가 깨질 것이다.

- `UnicodeDammit.unicode` -> `UnicodeDammit.unicode_markup`
- `Tag.next` -> `Tag.next_element`
- `Tag.previous` -> `Tag.previous_element`

발생자

발생자에 PEP 8을-준수하는 이름을 부여하고, 특성으로 변환하였다:

- `childGenerator()` -> `children`
- `nextGenerator()` -> `next_elements`
- `nextSiblingGenerator()` -> `next_siblings`
- `previousGenerator()` -> `previous_elements`
- `previousSiblingGenerator()` -> `previous_siblings`
- `recursiveChildGenerator()` -> `descendants`
- `parentGenerator()` -> `parents`

그래서 다음과 같이 하는 대신에:

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

다음과 같이 작성할 수 있다:

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

(그러나 구형 코드도 여전히 작동한다.)

어떤 발생자들은 일이 끝난후 `None`을 돌려주곤 했다. 그것은 버그였다. 이제 발생자는 그냥 멈춘다.

두 가지 발생자가 새로 추가되었는데, [.strings](#)와 [.stripped_strings](#)가 그것이다. `.strings`는 `NavigableString` 객체를 산출하고, `.stripped_strings`는 공백이 제거된 파이썬 문자열을 산출한다.

XML

이제 XML 해석을 위한 `BeautifulStoneSoup` 클래스는 더 이상 없다. XML을 해석하려면 “xml”을 두번째 인자로 `BeautifulSoup` 구성자에 건네야 한다. 같은 이유로, `BeautifulSoup` 구성자는 더 이상 `isHTML` 인자를 인지하지 못한다.

뷰티풀수프의 빈-원소 XML 태그 처리 방식이 개선되었다. 전에는 XML을 해석할 때 명시적으로 어느 태그가 빈-원소 태그로 간주되는지 지정해야 했었다. 구성자에 `selfClosingTags` 인자를 보내 봐야 더 이상 인지하지 못한다. 대신에, 뷰티풀수프는 빈 태그를 빈-원소 태그로 간주한다. 빈-원소 태그에 자손을 하나 추가하면, 더 이상 빈-원소 태그가 아니다.

개체

HTML이나 XML 개체가 들어 오면 언제나 그에 상응하는 유니코드 문자로 변환된다. 뷰티풀수프 3는 개체들을 다루기 위한 방법이 중첩적으로 많았다. 이제 중복이 제거되었다. `BeautifulSoup` 구성자는 더 이상 `smartQuotesTo`이나 `convertEntities` 인자를 인지하지 않는다. ([Unicode, Dammit](#)은 여전히 `smart_quotes_to`가 있지만, 그의 기본값은 이제 지능형 따옴표를 유니코드로 변환하는 것이다.) `HTML_ENTITIES`, `XML_ENTITIES`, 그리고 `XHTML_ENTITIES` 상수는 제거되었다. 왜냐하면 이제 더 이상 존재하지 않는 특징을 구성하기 때문이다 (유니코드 문자열을 제대로 모두 변환하지 못했다).

유니코드 문자들을 다시 출력시에 HTML 개체로 변환하고 싶다면, 그것들을 UTF-8 문자로 변환하기 보다, [출력 포맷터](#)를 사용할 필요가 있다.

기타

[Tag.string](#)은 이제 재귀적으로 작동한다. 태그 A에 태그 B만 달랑 있고 다른 것이 없다면, `A.string`은 `B.string`과 똑같다. (이전에서는 `None`이었다.)

[다중-값 속성](#)은 class와 같이 문자열이 아니라 문자열 리스트를 그 값으로 가진다. 이 사실은 CSS 클래스로 검색하는 방식에 영향을 미친다.

find* 메소드에 [text](#) 그리고 [name](#) 같은 태그-종속적 인자를 모두 건네면, 뷰티풀수프는 태그-종속적 기준에 부합하고 그 태그의 [Tag.string](#)이 [text](#) 값에 부합하는 태그들을 탐색한다. 문자열 자체는 찾지 않는다. 이전에, 뷰티풀수프는 태그-종속적 인자는 무시하고 문자열을 찾았다.

BeautifulSoup 구성자는 더 이상 markupMassage 인자를 인지하지 않는다. 이제 조판을 제대로 처리하는 일은 해석기의 책임이다..

ICantBelieveItsBeautifulSoup 그리고 BeautifulSoup와 같이 거의-사용되지 않는 해석기 클래스는 제거되었다. 이제 애매모호한 조판을 처리하는 방법은 해석기가 결정한다.

prettify() 메소드는 이제, bytestring이 아니라 유니코드 문자열을 돌려준다.