BeautifulSoup4

Beautiful Soup est une bibliothèque Python permettant d'extraire des données depuis des fichiers HTML et XML. Elle fonctionne avec votre parseur préféré pour fournir des moyens idiomatiques de naviguer, rechercher et modifier l'arbre d'analyse. Elle permet souvent aux programmeurs d'économiser des heures, voire des jours de travail.

Démarrage rapide

Voici un document HTML que j'utiliserai comme exemple tout au long de ce document. Il fait partie d'une histoire tirée d'Alice au pays des merveilles :

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

<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> an
d
<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>
"""
```

Exécuter le document des "trois sœurs" à travers Beautiful Soup nous donne un objet BeautifulSoup, qui représente le document comme une structure de données imbriquée :

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

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="link3">
    Tillie
   </a>
   ; and they lived at the bottom of a well.
  </p>
  <p class="story">
   ...
  </p>
 </body>
</html>
```

Voici quelques méthodes simples pour naviguer dans cette structure de données :

```
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>
```

Une tâche courante consiste à extraire toutes les URL contenues dans les balises `<a>` d'une page :

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

Une autre tâche fréquente est d'extraire tout le texte d'une page :

```
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.
#
# ...
```

Installation de Beautiful Soup

Beautiful Soup 4 est publié via PyPi, donc si vous ne pouvez pas l'installer avec le gestionnaire de paquets de votre système, vous pouvez l'installer avec easy_install ou pip. Le nom du paquet est **beautifulsoup4**. Assurez-vous d'utiliser la bonne version de pip ou easy_install correspondant à votre version de Python (ces outils peuvent être nommés pip3 et easy_install3 respectivement).

```
$ easy_install beautifulsoup4

$ pip install beautifulsoup4
```

(**Le paquet BeautifulSoup n'est pas celui que vous voulez.** Il s'agit de la version majeure précédente, Beautiful Soup 3. De nombreux logiciels utilisent encore BS3, il reste donc disponible, mais si vous écrivez du nouveau code, vous devez installer **beautifulsoup4**.)

Créer la soupe

Pour analyser un document, passez-le au constructeur de BeautifulSoup. Vous pouvez transmettre une chaîne de caractères ou un gestionnaire de fichier ouvert :

```
from bs4 import BeautifulSoup

with open("index.html") as fp:
    soup = BeautifulSoup(fp, 'html.parser')

soup = BeautifulSoup("<html>a web page</html>", 'html.parser')
```

D'abord, le document est converti en Unicode, et les entités HTML sont converties en caractères Unicode :

```
print(BeautifulSoup("<html><head></head><body>Sacr&eacute; bleu!</body></html>", "html.parser"))
# <html><head></head><body>Sacré bleu!</body></html>
```

Beautiful Soup analyse ensuite le document en utilisant le meilleur parseur disponible. Il utilisera un parseur HTML, sauf si vous lui indiquez spécifiquement d'en utiliser un autre (par exemple, un parseur XML).

Types d'objets

Beautiful Soup transforme un document HTML complexe en un arbre complexe d'objets Python. Mais en pratique, vous n'aurez affaire qu'à environ quatre types d'objets : **Tag**, **NavigableString**, **BeautifulSoup**, et **Comment**. Ces objets représentent les éléments HTML qui composent la page.

Classe Tag

Un objet de type **Tag** correspond à une balise XML ou HTML dans le document original.

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

Les balises ont de nombreux attributs et méthodes, et la plupart seront abordées dans les sections *Naviguer dans l'arbre* et *Rechercher dans l'arbre*.

Pour l'instant, les méthodes les plus importantes d'une balise concernent l'accès à son nom et à ses attributs.

name

Chaque balise a un nom :

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

Si vous changez le nom d'une balise, le changement se reflètera dans tout le balisage généré par Beautiful Soup :

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

attrs

Une balise HTML ou XML peut avoir un nombre quelconque d'attributs. La balise `<b id="boldest">` a un attribut "id" dont la valeur est "boldest". Vous pouvez accéder aux attributs d'une balise en la traitant comme un dictionnaire :

```
tag = BeautifulSoup('<b id="boldest">bold</b>', 'html.parser').b
tag['id']
# 'boldest'
```

Vous pouvez accéder directement au dictionnaire des attributs via `.attrs` :

```
tag.attrs
# {'id': 'boldest'}
tag.attrs.keys()
# dict_keys(['id'])
```

Vous pouvez ajouter, supprimer ou modifier les attributs d'une balise, encore une fois en la traitant comme un dictionnaire :

```
tag['id'] = 'verybold'
tag['another-attribute'] = 1
```

```
tag
# <b another-attribute="1" id="verybold"></b>

del tag['id']
del tag['another-attribute']
tag
# <b>bold</b>

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

Attributs à valeurs multiples

HTML 4 définit quelques attributs pouvant avoir plusieurs valeurs. HTML 5 en retire certains mais en ajoute de nouveaux. L'attribut multi-valeurs le plus courant est **class** (c'est-à-dire qu'une balise peut avoir plusieurs classes CSS). D'autres exemples incluent **rel**, **rev**, **accept-charset**, **headers** et **accesskey**. Par défaut, Beautiful Soup stocke les valeurs d'un attribut multi-valeurs sous forme de **liste** :

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

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

Classe NavigableString

Une balise peut contenir des chaînes de caractères comme éléments de texte. Beautiful Soup utilise la classe **NavigableString** pour contenir ces morceaux de texte :

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>', 'html.pars
er')
```

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

Un **NavigableString** fonctionne comme une chaîne Unicode Python, mais prend aussi en charge certaines fonctionnalités décrites dans *Naviguer dans l'arbre* et *Rechercher dans l'arbre*. Vous pouvez convertir un NavigableString en chaîne Unicode avec `str` :

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

Vous ne pouvez pas modifier une chaîne sur place, mais vous pouvez la remplacer par une autre à l'aide de `replace_with()` :

```
tag.string.replace_with("No longer bold")
tag
# <b class="boldest">No longer bold</b>
```

L'objet BeautifulSoup

L'objet **BeautifulSoup** représente le document analysé dans son ensemble. Dans la plupart des cas, vous pouvez le traiter comme un objet **Tag**. Cela signifie qu'il prend en charge la majorité des méthodes décrites dans les sections *Naviguer dans l'arbre* et *Rechercher dans l'arbre*.

Vous pouvez également passer un objet **BeautifulSoup** à l'une des méthodes définies dans *Modifier l'arbre*, tout comme vous le feriez avec un **Tag**. Cela vous permet par exemple de combiner deux documents analysés :

```
doc = BeautifulSoup("<document><content/>INSERT FOOTER HERE</doc
ument", "xml")
footer = BeautifulSoup("<footer>Here's the footer</footer>", "xml")
```

```
doc.find(text="INSERT FOOTER HERE").replace_with(footer)
# 'INSERT FOOTER HERE'
print(doc)
# <?xml version="1.0" encoding="utf-8"?>
# <document><content/><footer>Here's the footer</footer></document>
```

Puisque l'objet **BeautifulSoup** ne correspond pas à une véritable balise HTML ou XML, il n'a **ni nom ni attributs**. Mais il peut parfois être utile de référencer sa propriété `.name` (par exemple lorsqu'on écrit du code qui doit fonctionner à la fois avec des objets **Tag** et **BeautifulSoup**). C'est pourquoi on lui a donné un nom spécial : `[document]`.

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

Chaînes spéciales

Les classes **Tag**, **NavigableString** et **BeautifulSoup** couvrent presque tout ce que vous rencontrerez dans un fichier HTML ou XML. Mais il reste quelques éléments particuliers. Le plus courant est probablement **Comment**.

Classe Comment

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

L'objet **Comment** est simplement un type spécial de **NavigableString** :

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

Mais lorsqu'un commentaire apparaît dans un document HTML, il est affiché avec un formatage particulier :

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

Parcourir l’arbre

Voici à nouveau le document HTML des « Trois sœurs » :

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

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

<p class="story">...</p>
"""
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

Nous allons utiliser cet exemple pour montrer comment naviguer dans différentes parties d’un document.

Descendre dans l’arbre

Les balises peuvent contenir des chaînes de caractères ou d’autres balises. Ces éléments sont appelés **enfants** de la balise. Beautiful Soup fournit de nombreux attributs pour parcourir ou itérer sur ces enfants.

> Note : Les chaînes de caractères dans Beautiful Soup ne peuvent pas avoir d'enfants, donc elles ne supportent pas ces attributs.

Naviguer avec les noms de balises

La méthode la plus simple pour naviguer est d'utiliser `.find()` :

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

Par commodité, vous pouvez simplement utiliser `soup.head` au lieu de `soup.find("head")` :

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

De même :

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

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

`.find()` ne retourne que **la première** balise trouvée :

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

Pour obtenir **toutes** les balises `<a>` :

```
soup.find_all("a")
# [<a href=...>Elsie</a>, <a href=...>Lacie</a>, <a href=...>Tillie</a>]
```

`.contents` et `.children`

Les enfants directs d'une balise sont accessibles via `.contents` :

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

Et l'enfant de `<title>` :

```
title_tag = head_tag.contents[0]
title_tag.contents
# ["The Dormouse's story"]
```

Même l'objet `soup` a des enfants :

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

Une chaîne **n'a pas** d'enfants :

```
text = title_tag.contents[0]
text.contents
# AttributeError
```

Pour itérer sur les enfants :

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

> ⚠️ Ne modifiez pas .contents directement.

`.descendants`

`.descendants` permet d'itérer **récursivement** sur tous les descendants d'une balise :

```
for child in head_tag.descendants:
```

```
    print(child)
```

Cela inclut :

- `<title>`
- son contenu `"The Dormouse's story"`

```
len(list(soup.children))     # 1
len(list(soup.descendants))  # 26
```

`.string`

Si une balise contient **un seul enfant**, et que c'est une chaîne, alors :

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

Cela fonctionne aussi si l'unique enfant est une balise qui contient une chaîne :

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

Mais s'il y a plusieurs éléments, `.string` est `None` :

```
soup.html.string
# None
```

`.strings` et `.stripped_strings`

Pour récupérer **toutes les chaînes**, y compris celles séparées par des balises :

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

Utilisez `.stripped_strings` pour ignorer les espaces et retours à la ligne :

```
for string in soup.stripped_strings:
```

```
    print(repr(string))
```

Monter dans l'arbre

Chaque balise ou chaîne a un **parent**.

.parent

```
title_tag = soup.title
title_tag.parent
# <head>...</head>

title_tag.string.parent
# <title>...</title>

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

print(soup.parent)
# None
```

.parents

Pour parcourir tous les ancêtres :

```
link = soup.a
for parent in link.parents:
    print(parent.name)
# p
# body
# html
# [document]
```

.self_and_parents

Comme `.parents`, mais inclut **l'élément lui-même** :

```
for parent in link.self_and_parents:
    print(parent.name)
# link
# p
# body
# html
# [document]
```

Aller de côté

Considérons un document simple :

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></a>", 'html.
parser')
print(sibling_soup.prettify())
```

Affichage :

```
<a>
 <b>
  text1
 </b>
 <c>
  text2
 </c>
</a>
```

Les balises `<b>` et `<c>` sont **au même niveau** : elles sont toutes deux des enfants directs de la balise `<a>`. On les appelle **frères et sœurs (siblings)**. Lorsqu'un document est affiché avec indentation, les balises sœurs apparaissent au même niveau d'indentation. Vous pouvez exploiter cette relation dans votre code.

`.next_sibling` et `.previous_sibling`

Ces attributs permettent de naviguer **entre éléments situés au même niveau** dans l'arborescence :

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

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

La balise `<b>` n'a pas de `.previous_sibling` car aucun élément ne la précède au même niveau :

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

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

Les chaînes `"text1"` et `"text2"` **ne sont pas des frères et sœurs**, car elles n'ont pas le même parent :

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

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

Dans des documents HTML réels, `.next_sibling` ou `.previous_sibling` retourne souvent une chaîne de caractères contenant **des espaces ou retours à la ligne**.

Revenons au document des « Trois sœurs » :

```
<a href="...">Elsie</a>,
<a href="...">Lacie</a> and
<a href="...">Tillie</a>;
```

On pourrait croire que le `.next_sibling` de la première balise `<a>` est la deuxième balise `<a>`, mais en réalité c'est une **chaîne** contenant la **virgule et un saut de ligne** :

```
link = soup.a
link
# <a href="..." id="link1">Elsie</a>

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

La deuxième balise `<a>` est alors le `.next_sibling` **de cette chaîne** :

```
link.next_sibling.next_sibling
# <a href="..." id="link2">Lacie</a>
```

`.next_siblings` et `.previous_siblings`

Vous pouvez itérer sur tous les frères et sœurs suivants ou précédents d'une balise :

```
for sibling in soup.a.next_siblings:
    print(repr(sibling))
```

Affiche :

```
',\n'
<a href="..." id="link2">Lacie</a>
' and\n'
<a href="..." id="link3">Tillie</a>
'; and they lived at the bottom of a well.'
```

Pour les frères et sœurs **précédents** (ordre inverse) :

```
for sibling in soup.find(id="link3").previous_siblings:
    print(repr(sibling))
```

Affiche :

```
' and\n'
<a href="..." id="link2">Lacie</a>
```

```
',\n'
<a href="..." id="link1">Elsie</a>
'Once upon a time there were three little sisters; and their names were\n'
```

Aller en avant et en arrière

Jetons un coup d'œil au début du document « trois sœurs » :

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

Un parseur HTML transforme cette chaîne de caractères en une série d'événements :

- « ouvrir une balise `<html>` »,
- « ouvrir une balise `<head>` »,
- « ouvrir une balise `<title>` »,
- « ajouter une chaîne de caractères »,
- « fermer la balise `<title>` »,
- « ouvrir une balise `<p>` », etc.

L'ordre dans lequel ces balises et chaînes sont rencontrées s'appelle **l'ordre du document (document order)**.

Beautiful Soup permet de parcourir les éléments **dans cet ordre exact**.

`.next_element` et `.previous_element`

L'attribut `.next_element` d'une chaîne ou balise pointe vers **ce qui a été analysé immédiatement après** l'ouverture de la balise courante ou après la chaîne en question.

Il peut parfois être le même que `.next_sibling`, mais en général **c'est très différent**.

Prenons la dernière balise `<a>` du document :

```
last_a_tag = soup.find("a", id="link3")
last_a_tag
```

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

Son `.next_sibling` est une **chaîne** contenant la fin de la phrase :

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

Mais son `.next_element` est **le mot "Tillie"**, c'est-à-dire **le texte contenu dans la balise** `<a>` :

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

Pourquoi ? Parce que dans le HTML d'origine, le mot "Tillie" apparaît **avant le point-virgule**.

Le parseur a vu : `<a>`, puis `"Tillie"`, ensuite `</a>`, **puis** le `;` et le reste de la phrase.

Donc :

- `.next_element` suit **l'ordre de parsing réel**
- `.next_sibling` suit **la hiérarchie du DOM** (arbre).

`.previous_element`

L'attribut `.previous_element` pointe vers l'élément (balise ou chaîne) **immédiatement avant** :

```
last_a_tag.previous_element
# ' and\n'
```

Et naturellement :

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

`.next_elements` et `.previous_elements`

Ces attributs sont **des itérateurs** qui permettent de **parcourir tous les éléments suivants ou précédents** dans l'ordre du document :

```
for element in last_a_tag.next_elements:
    print(repr(element))
```

Résultat :

```
'Tillie'
';\nand they lived at the bottom of a well.'
'\n'
<p class="story">...</p>
'...'
'\n'
```

Ces méthodes sont utiles pour tout analyser **dans l'ordre réel d'apparition dans le fichier HTML**.

Rechercher dans l'arbre

Beautiful Soup propose de nombreuses méthodes pour rechercher dans l'arbre d'analyse, mais elles sont toutes très similaires. Nous allons passer du temps à expliquer les deux plus populaires : `find()` et `find_all()`.

Les autres méthodes utilisent presque exactement les mêmes arguments, donc nous ne les couvrirons que brièvement.

Encore une fois, nous allons utiliser le document « three sisters » :

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

<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> an
```

```
d
<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, 'html.parser')
```

En passant un **filtre** à une méthode comme `find_all()`, vous pouvez cibler **les parties du document qui vous intéressent**.

Types de filtres

Avant d'entrer dans les détails de `find_all()` et des méthodes similaires, voici différents **types de filtres** que vous pouvez utiliser.

Ces filtres s'utilisent **partout** dans l'API de recherche.

Vous pouvez filtrer selon :

- le nom de la balise,
- ses attributs,
- le texte qu'elle contient,
- ou une combinaison de ces critères.

Une chaîne de caractères

Le filtre le plus simple est **une chaîne de caractères**.

Si vous passez une chaîne à une méthode de recherche, Beautiful Soup va chercher **les balises portant exactement ce nom**.

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

> ⚠️ Si vous passez une chaîne d'octets (bytes), Beautiful Soup supposera qu'elle est encodée en UTF-8. Pour éviter ça, utilisez des chaînes Unicode (par défaut en Python 3).

Une expression régulière

Si vous passez une **expression régulière** (objet `re.compile`), Beautiful Soup l'utilisera pour chercher des noms de balise correspondants :

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

Cherche les balises dont le nom **commence par "b"**.

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

Cherche les balises contenant **la lettre "t"**.

La valeur `True`

Passer `True` retourne **toutes les balises**, mais pas les chaînes de texte :

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

Une fonction personnalisée

Si aucun des filtres ci-dessus ne vous suffit, vous pouvez définir **votre propre fonction**.

Elle doit **prendre un seul argument** (le tag), et **retourner** `True` **ou** `False`.

Exemple : balises qui ont un attribut `class` mais pas d'attribut `id` :

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

soup.find_all(has_class_but_no_id)
```

Résultat :

```
[
 <p class="title"><b>The Dormouse's story</b></p>,
 <p class="story">Once upon a time...</p>,
 <p class="story">...</p>
]
```

> ✅ Les balises <a> sont ignorées car elles ont à la fois class et id.

Autre exemple : balises **entourées de chaînes de caractères** :

```
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)
# body
# p
# a
# a
```

```
# a
# p
```

Une liste

Vous pouvez passer **une liste** de filtres (chaînes, regex, fonctions) et Beautiful Soup retournera tout ce qui **correspond à l'un d'eux** :

```
soup.find_all(["a", "b"])
```

Résultat :

```
[
 <b>The Dormouse's story</b>,
 <a class="sister" href=...>Elsie</a>,
 <a class="sister" href=...>Lacie</a>,
 <a class="sister" href=...>Tillie</a>
]
```

Maintenant que vous connaissez les différents types de filtres, vous êtes prêt·e à plonger dans les méthodes de recherche elles-mêmes.

---

find_all()

Signature de la méthode :

```
find_all(name, attrs, recursive, string, limit, **kwargs)
```

La méthode `find_all()` parcourt les **descendants** d'une balise et récupère tous ceux qui correspondent à vos **filtres**.

Vous avez déjà vu plusieurs exemples dans la section *Types de filtres*, mais en voici quelques autres :

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

Certains exemples vous sont peut-être familiers, d'autres non.

Pourquoi `find_all("p", "title")` trouve-t-il une balise `<p>` avec la classe CSS `"title"` ?

Voyons cela en explorant les **arguments** de `find_all()`.

L'argument `name`

En passant une valeur à `name`, vous indiquez à Beautiful Soup de ne considérer **que les balises portant ce nom**.

Les chaînes de texte seront **ignorées**, tout comme les balises dont le nom ne correspond pas.

Exemple simple :

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

Rappel : la valeur de `name` peut être une chaîne, une expression régulière, une liste, une fonction, ou la valeur `True`.

Les arguments par mots-clés (`*kwargs`)

Tous les **arguments nommés** que Beautiful Soup ne reconnaît pas seront **interprétés comme des attributs HTML** à filtrer.

Par exemple, si vous passez `id="link2"` :

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

> ✅ Cela filtre les balises dont l'attribut id vaut "link2".

Comme pour les balises, vous pouvez filtrer selon :

- une **chaîne**,
- une **regex**,
- une **liste**,
- une **fonction**,
- ou la valeur `True`.

Exemple avec une regex :

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

Utiliser `True` retourne toutes les balises qui **possèdent cet attribut** :

```
soup.find_all(id=True)
# [<a class="sister" href=... id="link1">Elsie</a>,
#  <a class="sister" href=... id="link2">Lacie</a>,
#  <a class="sister" href=... id="link3">Tillie</a>]
```

Utiliser une fonction comme filtre d'attribut

Vous pouvez écrire une fonction qui prend **la valeur d'un attribut** comme seul argument, et retourne `True` ou `False`.

Exemple : chercher les liens **sauf celui contenant "lacie"** :

```
def not_lacie(href):
    return href and not re.compile("lacie").search(href)

soup.find_all(href=not_lacie)
```

```
# [<a class="sister" href=...>Elsie</a>,
#  <a class="sister" href=...>Tillie</a>]
```

Utiliser une liste

Vous pouvez passer une liste de filtres (chaînes, regex, fonctions, etc.) :

```
soup.find_all(id=["link1", re.compile("3$")])
# [<a ... id="link1">Elsie</a>,
#  <a ... id="link3">Tillie</a>]
```

Combiner plusieurs attributs

Vous pouvez filtrer sur plusieurs attributs en les passant comme plusieurs arguments :

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

Rechercher avec `attrs={...}`

Certains attributs HTML (comme ceux commençant par `data-`) **ne peuvent pas** être utilisés en tant que mots-clés dans Python.

Exemple : `data-foo` est **invalide** comme nom d'argument :

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

Mais vous pouvez les passer via le dictionnaire `attrs` :

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

Même problème avec l'attribut HTML `"name"` :

Il entre en conflit avec l'argument `name` qui désigne **le nom de la balise**. Pour rechercher sur l'attribut `name`, utilisez `attrs` :

```
name_soup = BeautifulSoup('<input name="email"/>', 'html.parser')

name_soup.find_all(name="email")
# [] (cherche une balise nommée "email")

name_soup.find_all(attrs={"name": "email"})
# [<input name="email"/>]
```

Recherche par classe CSS

Il est très utile de rechercher une balise qui possède une certaine classe CSS, mais le nom de l'attribut CSS, `"class"`, est un mot réservé en Python. Utiliser `class` comme argument de mot-clé entraînerait une erreur de syntaxe. Depuis **Beautiful Soup 4.1.2**, vous pouvez rechercher par classe CSS en utilisant l'argument de mot-clé `class_` (notez le souligné à la fin).

Voici un exemple de recherche de balises `<a>` ayant la classe `"sister"` :

```
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>]
```

Comme pour tout argument de mot-clé, vous pouvez passer à `class_` une **chaîne**, une **expression régulière**, une **fonction**, ou la valeur **True**.

Exemple avec une expression régulière :

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

Exemple avec une fonction :

```
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>]
```

Classes multiples dans un même tag

Rappelez-vous qu'une balise peut avoir **plusieurs valeurs** pour son attribut `"class"`. Lorsque vous recherchez une balise correspondant à une certaine classe CSS, vous recherchez contre **n'importe laquelle** de ses classes CSS.

Exemple :

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>', 'html.parser')

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

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

Recherche de la valeur exacte de l'attribut `class`

Vous pouvez aussi rechercher exactement la valeur de la classe CSS :

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

Cependant, la recherche de variantes de la valeur de chaîne **ne fonctionne pas**. Par exemple, rechercher `"strikeout body"` ne renverra aucun résultat car l'ordre des classes dans l'attribut `class` compte :

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

L'argument string

Avec l'argument **string**, vous pouvez rechercher des chaînes de texte au lieu de balises. Comme pour les arguments **name** et les attributs, vous pouvez passer une **chaîne**, une **expression régulière**, une **fonction**, une **liste**, ou la valeur **True**. Voici quelques exemples :

Exemple de recherche d'une chaîne exacte :

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

Recherche de plusieurs chaînes à la fois :

```
soup.find_all(string=["Tillie", "Elsie", "Lacie"])
# ['Elsie', 'Lacie', 'Tillie']
```

Recherche avec une expression régulière :

```
soup.find_all(string=re.compile("Dormouse"))
# ["The Dormouse's story", "The Dormouse's story"]
```

Recherche avec une fonction personnalisée :

```
def is_the_only_string_within_a_tag(s):
    """Retourne True si cette chaîne est le seul enfant de sa balise parent
e."""
    return (s == s.parent.string)

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

Si vous utilisez l'argument **string** lors d'une recherche de balises, Beautiful Soup trouvera toutes les balises dont l'attribut `.string` correspond à votre valeur pour **string**. Par exemple, voici comment trouver les balises `<a>` dont le texte est "Elsie" :

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

L'argument **string** a été introduit dans **Beautiful Soup 4.4.0**. Dans les versions précédentes, cet argument s'appelait **text** :

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

L'argument limit

La méthode **find_all()** renvoie toutes les balises et chaînes correspondant à vos filtres. Cela peut prendre un certain temps si le document est volumineux. Si vous ne avez pas besoin de tous les résultats, vous pouvez passer un nombre à l'argument **limit**. Cela fonctionne de la même manière que le mot-clé **LIMIT** en SQL. Il indique à Beautiful Soup de cesser de collecter des résultats après avoir trouvé un certain nombre de correspondances.

Voici un exemple où il y a trois liens dans le document, mais où nous limitons le nombre de résultats à deux :

```
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>]
```

L'argument recursive

Par défaut, **mytag.find_all()** examine tous les descendants de **mytag** : ses enfants, les enfants de ses enfants, etc. Pour n'examiner que les **enfants directs**, vous pouvez passer **recursive=False**. Voici la différence dans cet exemple :

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

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

Dans cet extrait du document, le tag `<title>` se trouve sous le tag `<html>`, mais il n'est pas directement sous ce tag. Le tag `<head>` se trouve entre eux. Par défaut, Beautiful Soup trouve le tag `<title>` lorsqu'il peut examiner tous les descendants du tag `<html>`, mais lorsque vous définissez **recursive=False** pour restreindre la recherche aux enfants directs du tag `<html>`, il ne trouve rien.

L'argument **recursive** est spécifique aux méthodes **find_all()** et **find()**. Passer **recursive=False** dans des méthodes comme **find_parents()** ne serait pas utile.

Appeler une balise comme une méthode find_all()

Pour plus de commodité, appeler un objet **BeautifulSoup** ou un objet **Tag** comme une fonction est équivalent à appeler **find_all()** (si aucune méthode intégrée ne correspond au nom de la balise que vous recherchez). Ces deux lignes de code sont équivalentes :

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

De même, ces deux lignes sont également équivalentes :

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

---

find()

Signature de la méthode : `find(name, attrs, recursive, string, **kwargs)`

La méthode **find_all()** parcourt tout le document à la recherche de résultats, mais parfois vous ne voulez trouver qu'un seul résultat. Si vous savez qu'un document ne contient qu'une seule balise `<body>`, il est inutile de parcourir tout le document à la recherche d'autres balises. Plutôt que de passer `limit=1` à chaque fois que vous appelez **find_all**, vous pouvez utiliser la méthode **find()**. Ces deux lignes de code sont presque équivalentes :

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

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

La seule différence est que **find_all()** renvoie une liste contenant le résultat unique, tandis que **find()** renvoie simplement le résultat.

Si **find_all()** ne trouve rien, il renvoie une liste vide. Si **find()** ne trouve rien, il renvoie **None** :

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

Rappelez-vous du truc `soup.head.title` mentionné dans **Naviguer en utilisant les noms de balises** ? Ce truc fonctionne en appelant à plusieurs reprises **find()** :

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

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

---

find_parents() and find_parent()

Signatures des méthodes :

- `find_parents(name, attrs, string, limit, **kwargs)`
- `find_parent(name, attrs, string, **kwargs)`

J'ai passé beaucoup de temps à expliquer **find_all()** et **find()**. L'API Beautiful Soup définit dix autres méthodes pour explorer l'arbre, mais ne vous inquiétez pas. Cinq de ces méthodes sont essentiellement les mêmes que **find_all()**, et les cinq autres sont essentiellement les mêmes que **find()**. La seule différence réside dans la manière dont elles se déplacent d'une partie de l'arbre à une autre.

Commençons par examiner **find_parents()** et **find_parent()**. Rappelez-vous que **find_all()** et **find()** parcourent l'arbre en descendant, en cherchant les descendants d'une balise. Ces méthodes fonctionnent à l'inverse : elles

remontent l'arbre, en cherchant les parents d'une balise (ou d'une chaîne). Essayons-les dans le document "three daughters" :

```
a_string = soup.find(string="Lacie")
a_string
# '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 the
ir 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")
# []
```

L'une des trois balises `<a>` est le parent direct de la chaîne recherchée, donc notre recherche la trouve. L'une des trois balises `<p>` est un parent indirect (ancêtre) de la chaîne, et notre recherche la trouve également. Il existe une balise `<p>` avec la classe CSS "title" quelque part dans le document, mais ce n'est pas l'un des parents de cette chaîne, donc nous ne pouvons pas la trouver avec **find_parents()**.

Vous avez peut-être remarqué une similitude entre **find_parent()** et **find_parents()**, ainsi que les attributs **.parent** et **.parents** mentionnés précédemment. Ces méthodes de recherche utilisent en fait l'attribut **.parents** pour parcourir tous les parents (non filtrés), vérifiant chaque parent par rapport au filtre fourni pour voir s'il correspond.

---

find_next_siblings() and find_next_sibling()

Signatures des méthodes :

- `find_next_siblings(name, attrs, string, limit, **kwargs)`

- `find_next_sibling(name, attrs, string, **kwargs)`

Ces méthodes utilisent **.next_siblings** pour parcourir les autres frères et sœurs d'un élément dans l'arbre. La méthode **find_next_siblings()** renvoie tous les frères et sœurs qui correspondent, tandis que **find_next_sibling()** renvoie uniquement le premier :

```
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() and find_previous_sibling()

Signatures des méthodes :

- `find_previous_siblings(name, attrs, string, limit, **kwargs)`
- `find_previous_sibling(name, attrs, string, **kwargs)`

Ces méthodes utilisent **.previous_siblings** pour parcourir les frères et sœurs d'un élément qui le précèdent dans l'arbre. La méthode **find_previous_siblings()** renvoie tous les frères et sœurs qui correspondent, tandis que **find_previous_sibling()** renvoie uniquement le premier :

```
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() and find_next()

Signatures des méthodes :

- `find_all_next(name, attrs, string, limit, **kwargs)`
- `find_next(name, attrs, string, **kwargs)`

Ces méthodes utilisent **.next_elements** pour parcourir tous les éléments de l'arbre qui suivent l'élément initial dans le document. La méthode **find_all_next()** renvoie toutes les correspondances, tandis que **find_next()** renvoie uniquement la première correspondance :

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

first_link.find_all_next(string=True)
# ['Elsie', ',\n', 'Lacie', ' and\n', 'Tillie',
#  ';\nand they lived at the bottom of a well.', '\n', '...', '\n']

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

---

find_all_previous() and find_previous()

Signatures des méthodes :

- `find_all_previous(name, attrs, string, limit, **kwargs)`
- `find_previous(name, attrs, string, **kwargs)`

Ces méthodes utilisent **.previous_elements** pour parcourir les éléments du document qui précédaient l'élément initial dans le document. La méthode **find_all_previous()** renvoie toutes les correspondances, tandis que **find_previous()** renvoie uniquement la première correspondance :

```
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>
```

CSS selectors through the .css property

Les objets **BeautifulSoup** et **Tag** supportent les sélecteurs CSS à travers leur propriété **.css**. L'implémentation des sélecteurs réels est gérée par le package **Soup Sieve**, disponible sur PyPI sous le nom de **soupsieve**. Si vous avez installé Beautiful Soup via **pip**, **Soup Sieve** a été installé en même temps, donc vous n'avez rien à faire de plus.

La documentation de **Soup Sieve** liste tous les sélecteurs CSS actuellement supportés, mais voici quelques exemples de base. Vous pouvez trouver des balises par leur nom :

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

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

Sélecteurs CSS pour différents cas

Vous pouvez également trouver des balises par leur ID, leur classe CSS, ou en utilisant des sélecteurs combinés :

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

```
# Trouver des balises par classe CSS
soup.css.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>]
```

Il existe aussi une méthode appelée **select_one()**, qui trouve uniquement la première balise qui correspond à un sélecteur :

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

Pour simplifier, vous pouvez appeler **select()** et **select_one()** directement sur l'objet **BeautifulSoup** ou **Tag**, sans utiliser la propriété **.css** :

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

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

Les sélecteurs CSS sont pratiques pour ceux qui connaissent déjà la syntaxe des sélecteurs CSS. Vous pouvez accomplir tout cela avec l'API Beautiful Soup, mais si les sélecteurs CSS sont tout ce dont vous avez besoin, vous pouvez passer directement à l'utilisation de **lxml**, qui est beaucoup plus rapide. Cependant, **Soup Sieve** permet de combiner les sélecteurs CSS avec l'API Beautiful Soup.