iterfind() перебирает все элементы, которые соответствуют выражению пути

findall() возвращает список подходящих элементов

find() эффективно возвращает только первое совпадение

findtext() возвращает текстовое содержимое первого совпадения

Наглядные примеры:

>>> root = etree.XML("<root><a x='123'>aText<b/><c/><b/></a></root>")

#Find a child of an Element:

>>> print(root.find("b"))

None

>>> print(root.find("a").tag)

a

#Find an Element anywhere in the tree:

>>> print(root.find(".//b").tag)

b

>>> [ b.tag for b in root.iterfind(".//b") ]

['b', 'b']

#Find Elements with a certain attribute:

>>> print(root.findall(".//a[@x]")[0].tag)

a

>>> print(root.findall(".//a[@y]"))

[]

Элементы списки

Чтобы сделать доступ к этим подэлементам простым и понятным, элементы максимально имитируют поведение обычных списков Python:

>>> child = root[0]

>>> print(child.tag)

child1

>>> print(len(root))

3

>>> root.index(root[1]) # lxml.etree only!

1

>>> children = list(root)

>>> for child in root:

... print(child.tag)

child1

child2

child3

>>> root.insert(0, etree.Element("child0"))

>>> start = root[:1]

>>> end = root[-1:]

>>> print(start[0].tag)

child0

>>> print(end[0].tag)

child3

Проверка на детей:

if root: # this no longer works!

print("The root element has children")

Но так вернее.

>>> print(etree.iselement(root)) # test if it's some kind of Element

True

>>> if len(root): # test if it has children

... print("The root element has children")

The root element has children

обратить внимание

>>> for child in root:

... print(child.tag)

child0

child1

child2

child3

>>> root[0] = root[-1] # this moves the element in lxml.etree!

>>> for child in root:

... print(child.tag)

child3

child1

child2

В этом примере последний элемент перемещается в другую позицию вместо копирования, т.е. он автоматически удаляется из своей предыдущей позиции, когда он помещается в другое место. В списках объекты могут появляться в нескольких позициях одновременно, и указанное выше назначение просто скопирует ссылку на элемент в первую позицию, так что оба содержат один и тот же элемент:

>>> l = [0, 1, 2, 3]

>>> l[0] = l[-1]

>>> l

[3, 1, 2, 3]

Обратите внимание, что в исходном ElementTree один объект Element может находиться в любом количестве мест в любом количестве деревьев, что позволяет выполнять ту же операцию копирования, что и в списках. Очевидный недостаток заключается в том, что модификации такого элемента будут применяться ко всем местам, где он появляется в дереве, что может или не может быть предназначено.

Преимущество этого различия состоит в том, что элемент в lxml.etree всегда имеет ровно одного родителя, к которому можно обратиться с помощью метода getparent () . Это не поддерживается в оригинальном ElementTree

>>> root is root[0].getparent() # lxml.etree only!

True

Если вы хотите скопировать элемент в другую позицию в lxml.etree , рассмотрите возможность создания независимой глубокой копии с использованием модуля копирования из стандартной библиотеки Python:

>>> from copy import deepcopy

>>> element = etree.Element("neu")

>>> element.append( deepcopy(root[1]) )

>>> print(element[0].tag)

child1

>>> print([ c.tag for c in root ])

['child3', 'child1', 'child2']

Братья и сестры (или соседи) элемента доступны как следующий и предыдущий элементы:

>>> root[0] is root[1].getprevious() # lxml.etree only!

True

>>> root[1] is root[0].getnext() # lxml.etree only!

True

Элементы несут атрибуты как словари

Элементы XML поддерживают атрибуты. Вы можете создать так:

>>> root = etree.Element("root", interesting="totally")

>>> etree.tostring(root)

b'<root interesting="totally"/>'

Атрибуты - это просто неупорядоченные пары имя-значение, поэтому очень удобный способ работы с ними - через словарь-интерфейс интерфейса Elements:

>>> print(root.get("interesting"))

totally

>>> print(root.get("hello"))

None

>>> root.set("hello", "Huhu")

>>> print(root.get("hello"))

Huhu

>>> etree.tostring(root)

b'<root interesting="totally" hello="Huhu"/>'

>>> sorted(root.keys())

['hello', 'interesting']

>>> for name, value in sorted(root.items()):

... print('%s = %r' % (name, value))

hello = 'Huhu'

interesting = 'totally'

В случаях, когда вы хотите выполнить поиск элементов или у вас есть другие причины для получения «реального» словарного объекта, например, для его передачи, вы можете использовать свойство attrib :

>>> attributes = root.attrib

>>> print(attributes["interesting"])

totally

>>> print(attributes.get("no-such-attribute"))

None

>>> attributes["hello"] = "Guten Tag"

>>> print(attributes["hello"])

Guten Tag

>>> print(root.get("hello"))

Guten Tag

Обратите внимание, что attrib - это объект, похожий на диктовку, поддерживаемый самим элементом. Это означает, что любые изменения в Элементе отражаются в атрибуте и наоборот. Это также означает, что дерево XML остается в памяти до тех пор, пока используется атрибут одного из его элементов. Чтобы получить независимый снимок атрибутов, который не зависит от дерева XML, скопируйте его в dict:

>>> d = dict(root.attrib)

>>> sorted(d.items())

[('hello', 'Guten Tag'), ('interesting', 'totally')]

Элементы содержат текст

Элементы могут содержать текст:

>>> root = etree.Element("root")

>>> root.text = "TEXT"

>>> print(root.text)

TEXT

>>> etree.tostring(root)

b'<root>TEXT</root>'

Во многих XML-документах (документах, ориентированных на данные ) это единственное место, где можно найти текст. Он инкапсулирован листовым тегом в самом низу древовидной иерархии.

<html><body>Hello<br/>World</body></html>

Однако, если XML используется для текстовых документов с тегами, таких как (X) HTML, текст также может появляться между различными элементами прямо в середине дерева:

>>> html = etree.Element("html")

>>> body = etree.SubElement(html, "body")

>>> body.text = "TEXT"

>>> etree.tostring(html)

b'<html><body>TEXT</body></html>'

>>> br = etree.SubElement(body, "br")

>>> etree.tostring(html)

b'<html><body>TEXT<br/></body></html>'

>>> br.tail = "TAIL"

>>> etree.tostring(html)

b'<html><body>TEXT<br/>TAIL</body></html>'

Здесь тег <br/> окружен текстом. Это часто называют XML- документом в стиле документа или со смешанным содержимым . Элементы поддерживают это через свое свойство tail . Он содержит текст, который следует непосредственно за элементом, вплоть до следующего элемента в дереве XML:

Двух свойств .text и .tail достаточно для представления любого текстового содержимого в документе XML. Таким образом, API ElementTree не требует каких-либо специальных текстовых узлов в дополнение к классу Element, которые имеют тенденцию довольно часто мешать (как вы, возможно, знаете из классических API DOM ).

Однако, есть случаи, когда хвостовой текст также мешает. Например, когда вы сериализуете элемент из дерева, вам не всегда нужен его хвостовой текст в результате (хотя вы все равно хотели бы хвостовой текст его дочерних элементов). Для этого функция tostring () принимает аргумент ключевого слова with\_tail :

>>> etree.tostring(br)

b'<br/>TAIL'

>>> etree.tostring(br, with\_tail=False) # lxml.etree only!

b'<br/>'

Если вы хотите прочитать только текст, то есть без каких-либо промежуточных тегов, вы должны рекурсивно объединить все атрибуты текста и хвоста в правильном порядке. Опять же, функция tostring () приходит на помощь, на этот раз используя ключевое слово method :

>>> etree.tostring(html, method="text")

b'TEXTTAIL'

Использование XPath для поиска текста

Другой способ извлечения текстового содержимого дерева - это XPath , который также позволяет извлекать отдельные текстовые блоки в список:

>>> print(html.xpath("string()")) # lxml.etree only!

TEXTTAIL

>>> print(html.xpath("//text()")) # lxml.etree only!

['TEXT', 'TAIL']

Если вы хотите использовать это чаще, вы можете заключить его в функцию:

>>> build\_text\_list = etree.XPath("//text()") # lxml.etree only!

>>> print(build\_text\_list(html))

['TEXT', 'TAIL']

Обратите внимание, что строковый результат, возвращаемый XPath, является специальным «умным» объектом, который знает о его происхождении. Вы можете спросить его, откуда он пришел, с помощью метода getparent () , так же, как и с Elements:

>>> texts = build\_text\_list(html)

>>> print(texts[0])

TEXT

>>> parent = texts[0].getparent()

>>> print(parent.tag)

body

>>> print(texts[1])

TAIL

>>> print(texts[1].getparent().tag)

br

Вы также можете узнать, является ли это обычным текстовым содержимым или хвостовым текстом:

>>> print(texts[0].is\_text)

True

>>> print(texts[1].is\_text)

False

>>> print(texts[1].is\_tail)

True

Хотя это работает для результатов функции text () , lxml не сообщит вам происхождение строкового значения, созданного функциями XPath string () или concat () :

>>> stringify = etree.XPath("string()")

>>> print(stringify(html))

TEXTTAIL

>>> print(stringify(html).getparent())

None

<https://lxml.de/api/index.html>

xpath(self, \_path, namespaces=None, extensions=None, smart\_strings=True, \*\*\_variables)

\_path — берём из хрома элемент Xpath. Пояснения в https://habr.com/ru/post/114772/.

Итерация дерева

Разбор строк и файлов

lxml.etree поддерживает синтаксический анализ XML несколькими способами и из всех важных источников, а именно: строк, файлов, URL-адресов (http / ftp) и файловых объектов. Основными функциями разбора являются fromstring () и parse () , которые вызываются с источником в качестве первого аргумента. По умолчанию они используют стандартный анализатор, но вы всегда можете передать другой анализатор в качестве второго аргумента.

Функция fromstring ()

Функция fromstring () - это самый простой способ разбора строки:

>>> some\_xml\_data = "<root>data</root>"

>>> root = etree.fromstring(some\_xml\_data)

>>> print(root.tag)

root

>>> etree.tostring(root)

b'<root>data</root>''

Функция XML ()

Функция XML () ведет себя как функция fromstring () , но обычно используется для записи литералов XML прямо в источник:

>>> root = etree.XML("<root>data</root>")

>>> print(root.tag)

root

>>> etree.tostring(root)

b'<root>data</root>'

Существует также соответствующая функция HTML () для литералов HTML.

>>> root = etree.HTML("<p>data</p>")

>>> etree.tostring(root)

b'<html><body><p>data</p></body></html>'

Функция parse ()

Функция parse () используется для разбора файлов и файловых объектов.

В качестве примера такого файлового объекта следующий код использует класс BytesIO для чтения из строки вместо внешнего файла. Этот класс происходит из модуля io в Python 2.6 и более поздних версиях. В более старых версиях Python вам придется использовать класс StringIO из модуля StringIO . Однако в реальной жизни вы, очевидно, избегаете делать все это вместе и используете функции разбора строк, описанные выше.

>>> from io import BytesIO

>>> some\_file\_or\_file\_like\_object = BytesIO(b"<root>data</root>")

>>> tree = etree.parse(some\_file\_or\_file\_like\_object)

>>> etree.tostring(tree)

b'<root>data</root>'

Обратите внимание, что parse () возвращает объект ElementTree, а не объект Element в качестве функции парсера строк:

>>> root = tree.getroot()

>>> print(root.tag)

root

>>> etree.tostring(root)

b'<root>data</root>'

Причина этого различия заключается в том, что parse () возвращает полный документ из файла, в то время как функции синтаксического анализа строк обычно используются для анализа фрагментов XML.

Функция parse () поддерживает любой из следующих источников:

* открытый объект файла (не забудьте открыть его в двоичном режиме)
* файловый объект, имеющий метод .read (byte\_count), возвращающий строку байтов при каждом вызове
* строка имени файла
* строка URL HTTP или FTP

Обратите внимание, что передача имени файла или URL-адреса обычно происходит быстрее, чем передача открытого файла или файлового объекта. Однако HTTP / FTP-клиент в libxml2 довольно прост, поэтому такие вещи, как HTTP-аутентификация, требуют специальной библиотеки URL-запросов, например, urllib2 или запросов . Эти библиотеки обычно предоставляют файл-подобный объект для результата, который вы можете анализировать во время потоковой передачи ответа.

ElementPath

Библиотека ElementTree поставляется с простым XPath-подобным языком путей, который называется ElementPath . Основное отличие состоит в том, что вы можете использовать нотацию тега {namespace} в выражениях ElementPath. Однако дополнительные функции, такие как сравнение значений и функции, недоступны.

В дополнение к полной реализации XPath , lxml.etree поддерживает язык ElementPath так же, как и ElementTree, даже используя (почти) ту же реализацию. API предоставляет здесь четыре метода, которые вы можете найти в Elements и ElementTrees:

iterfind () выполняет итерацию по всем элементам, которые соответствуют выражению пути

findall () возвращает список подходящих элементов

find () эффективно возвращает только первое совпадение

findtext () возвращает текстовое содержимое первого совпадения.

Вот некоторые примеры:

(Вам нужно использовать .// чтобы указать, что вы хотите найти Body которое является потомком (а не прямым потомком) текущего элемента)

>>> root = etree.XML("<root><a x='123'>aText<b/><c/><b/></a></root>")

Найдите ребенка Элемента:

>>> print(root.find("b"))

None

>>> print(root.find("a").tag)

a

Найдите элемент в любом месте дерева:

>>> print(root.find(".//b").tag)

b

>>> [ b.tag for b in root.iterfind(".//b") ]

['b', 'b']

Найти элементы с определенным атрибутом:

>>> print ( root . findall ( ".//a[@x]" ) [ 0 ] . tag )

a

>>> print ( root . findall ( ".//a[@y]" ))

[]

В lxml 3.4 есть новый помощник для генерации структурного выражения ElementPath для элемента:

>>> Дерево = Этри . ElementTree ( root )

>>> a = root [ 0 ]

>>> print ( tree . Getelementpath ( a [ 0 ]))

a / b [1]

>>> print ( tree . Getelementpath ( a [ 1 ]))

a / c

>>> print ( tree . getelementpath ( a [ 2 ]))

a / b [2]

>>> Дерево . find ( tree . getelementpath ( a [ 2 ])) == a [ 2 ]

True

Пока дерево не изменено, это выражение пути представляет идентификатор для данного элемента, который может быть использован, чтобы позже найти его в том же дереве. По сравнению с XPath, выражения ElementPath имеют то преимущество, что они автономны даже для документов, использующих пространства имен.

Метод .iter () является особым случаем, который находит только определенные теги в дереве по их имени, а не по пути. Это означает, что следующие команды эквивалентны в случае успеха:

>>> print ( root . find ( ".//b" ) . tag )

b

>>> print ( next ( root . iterfind ( ".//b" )) . tag )

b

>>> print ( next ( root . iter ( "b" )) . tag )

b

Обратите внимание, что метод .find () просто возвращает None, если совпадение не найдено, тогда как два других примера вызовут исключение StopIteration