一、页面解析

1.页面解析和数据提取

一般来讲对我们而言，需要抓取的是某个网站或者某个应用的内容，提取有用的价值。内容一般分为两部分，非结构化的数据 和 结构化的数据。

- 非结构化数据：先有数据，再有结构，
- 结构化数据：先有结构、再有数据
- 不同类型的数据，我们需要采用不同的方式来处理。

2.非结构化的数据处理

1) 文本、电话号码、邮箱地址

- 正则表达式

2) HTML 文件

- 正则表达式
- XPath
- CSS 选择器

3.结构化的数据处理

1) JSON 文件

- JSON Path
- 转化成 Python 类型进行操作（json 类）

2) XML 文件

- 转化成 Python 类型（xmltodict）
- XPath
- CSS 选择器
- 正则表达式

二、正则表达式

正则表达式，又称规则表达式，通常被用来检索、替换那些符合某个模式(规则)的文本。

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特 定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过 滤逻辑。 给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

- 给定的字符串是否符合正则表达式的过滤逻辑（“匹配”）；
- 通过正则表达式，从文本字符串中获取我们想要的特定部分（“过滤”）。

	匹配单个字符的元字符	
.	点号	匹配单个任意字符
[…]	字符组	匹配单个列出的字符
[^…]	排除型字符	匹配单个未列出的字符
\w	\w	匹配任意字母、数字、下划线。等价于[A-Za-z0-9_]
\W	\W	\w的反义
\s	\s	匹配任意的空白字符
\S	\S	\s的反义
\d	\d	匹配任意的数字，例如：\d{2}表示前面的数字连续出现2次，即2个数字。
\b	\b	匹配单词的开始或结束，也就是单词的分界处，它只匹配一个字符位置。
\B	\B	/b的反义
\char	转义字符	若char是元字符，或转义序列无特殊含义时，匹配char对应的普通字符。
	提供计数功能的元字符	
?	问号	允许匹配一次，但非必须
*	星号	可以匹配任意多次，也可以不匹配
+	加号	至少需要匹配一次，至多可能匹配任意多次
{min,max}	区间量词	至少要min次，至多允许max次
	匹配位置的元字符	
^	脱字符	匹配一行的开头位置
$	美元符	匹配一行的结束位置
\<	单词分界符	匹配单词开始的位置
\>	单词分界符	匹配单词结束的位置
	其他元字符	
|	竖划线	匹配任意分割的表达式
(…)	括号	限定多选结构的范围，标注量词作用的元素，为反向引用捕获文本。
\1\2	反向引用	匹配之前的第一，第二组括号内的表达式匹配的文本

1. Python 的 re 模块

在 Python 中，我们可以使用内置的 re 模块来使用正则表达式。

有一点需要特别注意的是，正则表达式使用 对特殊字符进行转义，所以如果我们要使用原始字符串，只需加一个 r 前缀，示例：

```
r'hengqijiaoyu\t\.\tpython'
```

1) re 模块的一般使用步骤：

1. 使用 compile() 函数将正则表达式的字符串形式编译为一个 Pattern 对象
2. 通过 Pattern 对象提供的一系列方法对文本进行匹配查找，获得匹配结果:一个 Match 对象。
3. 最后使用 Match 对象提供的属性和方法获得信息，根据需要进行其他的操作

2) compile 函数

compile 函数用于编译正则表达式，生成一个 Pattern 对象，它的一般使用形式如下：

```
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
```

在上面，我们已将一个正则表达式编译成 Pattern 对象，接下来，我们就可以利用 pattern 的一系列方法对文本进行匹配查找了。

Pattern 对象的一些常用方法主要有：

match 方法：从起始位置开始查找，一次匹配 search 方法：从任何位置开始查找，一次匹配 findall 方法：全部匹配，返回列表 finditer 方法：全部匹配，返回迭代器 split 方法：分割字符串，返回列表 sub 方法：替换

3) match 方法

match 方法用于查找字符串的头部（也可以指定起始位置），它是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果。它的一般使用形式如下：

match(string[, pos[, endpos]])

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。因此，当你不指定 pos 和 endpos 时，match 方法默认匹配字符串的头部。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

```
import re

pattern = re.compile(r'\d+')  # 用于匹配至少一个数字

m = pattern.match('one12twothree34four')  # 查找头部，没有匹配
print(m)
```

```
n = pattern.match('one12twothree34four', 2, 10)  # 从'e'的位置开始匹配，没有匹配
print(n)

o = pattern.match('one12twothree34four', 3, 10)  # 从'1'的位置开始匹配，正好匹配
print(o)

print(o.group())
print(o.start())
print(o.end())
print(o.span())
```

```
None
None
<_sre.SRE_Match object; span=(3, 5), match='12'>
12
3
5
(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- group([group1, ...]) 方法用于获得一个或多个分组匹配的字符串，当要获得整个匹配的子串时，可直接使用 group() 或 group(0)；
- start([group]) 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
- end([group]) 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
- span([group]) 方法返回 (start(group), end(group))。

```
import re

pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)  # re.I 表示忽略大小写
m = pattern.match('Hello World Wide Web')

print(m)

print(m.group(0))  # 返回匹配成功的整个子串

print(m.span(0))  # 返回匹配成功的整个子串的索引

print(m.group(1))  # 返回第一个分组匹配成功的子串

print(m.span(1))  # 返回第一个分组匹配成功的子串的索引
```

```
print(m.group(2))  # 返回第二个分组匹配成功的子串

print(m.span(2))  # 返回第二个分组匹配成功的子串

print(m.groups())  # 等价于 (m.group(1), m.group(2), ...)

print(m.group(3))  # 不存在第三个分组
```

```
<_sre.SRE_Match object; span=(0, 11), match='Hello World'>
Hello World
(0, 11)
Hello
(0, 5)
World
(6, 11)
('Hello', 'World')

Traceback (most recent call last):
  File "/home/python/PycharmProjects/untitled2/test.py", line 22, in <m
odule>
    print(m.group(3))  # 不存在第三个分组
IndexError: no such group
```

4) search 方法

search 方法用于查找字符串的任何位置，它也是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果，它的一般使用形式如下：

search(string[, pos[, endpos]])

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

让我们看看例子：

```
import re

pattern = re.compile('\d+')
m = pattern.search('one12twothree34four')  # 这里如果使用 match 方法则不匹配
print(m)

print(m.group())

o = pattern.search('one12twothree34four', 10, 30)  # 指定字符串区间
print(o)
```

```
print(o.group())

print(o.span())
```

```
<_sre.SRE_Match object; span=(3, 5), match='12'>
12
<_sre.SRE_Match object; span=(13, 15), match='34'>
34
(13, 15)
```

```
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
# 使用 search() 查找匹配的子串，不存在匹配的子串时将返回 None
# 这里使用 match() 无法成功匹配
m = pattern.search('hello 123456 789')
if m:
    # 使用 Match 获得分组信息
    print ('matching string:',m.group())
    # 起始位置和结束位置
    print ('position:',m.span())
```

```
matching string: 123456
position: (6, 12)
```

5) findall 方法

上面的 match 和 search 方法都是一次匹配，只要找到了一个匹配的结果就返回。然而，在大多数时候，我们需要搜索整个字符串，获得所有匹配的结果。

findall 方法的使用形式如下：

findall(string[, pos[, endpos]])

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

findall 以列表形式返回全部能匹配的子串，如果没有匹配，则返回一个空列表。

```
import re
pattern = re.compile(r'\d+')   # 查找数字

result1 = pattern.findall('hello 123456 789')
result2 = pattern.findall('one1two2three3four4', 0, 10)

print result1
print result2
```

```
['123456', '789']
['1', '2']
```

```
import re

#re 模块提供一个方法叫 compile 模块，提供我们输入一个匹配的规则
#然后返回一个 pattern 实例，我们根据这个规则去匹配字符串
pattern = re.compile(r'\d+\.\d*')

#通过 partten.findall()方法就能够全部匹配到我们得到的字符串
result = pattern.findall("123.141593, 'bigcat', 232312, 3.15")

#findall 以 列表形式 返回全部能匹配的子串给 result
for item in result:
    print item
```

```
123.141593
3.15
```

6) finditer 方法

finditer 方法的行为跟 findall 的行为类似，也是搜索整个字符串，获得所有匹配的结果。但它返回一个顺序访问每一个匹配结果（Match 对象）的迭代器。

```
import re
pattern = re.compile(r'\d+')

result_iter1 = pattern.finditer('hello 123456 789')
result_iter2 = pattern.finditer('one1two2three3four4', 0, 10)

print type(result_iter1)
print type(result_iter2)

print 'result1...'
for m1 in result_iter1:   # m1 是 Match 对象
    print 'matching string: {}, position: {}'.format(m1.group(), m1.spa
n())

print 'result2...'
for m2 in result_iter2:
    print 'matching string: {}, position: {}'.format(m2.group(), m2.spa
n())
```

```
<type 'callable-iterator'>
<type 'callable-iterator'>
result1...
matching string: 123456, position: (6, 12)
matching string: 789, position: (13, 16)
result2...
```

```
matching string: 1, position: (3, 4)
matching string: 2, position: (7, 8)
```

7) split 方法

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

split(string[, maxsplit])

其中，maxsplit 用于指定最大分割次数，不指定将全部分割。

```
import re
p = re.compile(r'[\s\,\;]+')
print p.split('a,b;; c   d')
```

```
['a', 'b', 'c', 'd']
```

8) sub 方法

sub 方法用于替换。它的使用形式如下：

```
sub(repl, string[, count])
```

其中，repl 可以是字符串也可以是一个函数：

如果 repl 是字符串，则会使用 repl 去替换字符串每一个匹配的子串，并返回替换后的字符串，另外，repl 还可以使用 id 的形式来引用分组，但不能使用编号 0；

如果 repl 是函数，这个方法应当只接受一个参数（Match 对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。

count 用于指定最多替换次数，不指定时全部替换。

```
import re
p = re.compile(r'(\w+) (\w+)') # \w = [A-Za-z0-9_]
s = 'hello 123, hello 456'

print p.sub(r'hello world', s)  # 使用 'hello world' 替换 'hello 123' 和
 'hello 456'
print p.sub(r'\2 \1', s)        # 引用分组

def func(m):
    return 'hi' + ' ' + m.group(2)

print p.sub(func, s)
print p.sub(func, s, 1)         # 最多替换一次
```

```
hello world, hello world
123 hello, 456 hello
hi 123, hi 456
hi 123, hello 456
```

9) 匹配中文

在某些情况下，我们想匹配文本中的汉字，有一点需要注意的是，中文的 unicode 编码范围 主要在 [u4e00-u9fa5]，这里说主要是因为这个范围并不完整，比如没有包括全角（中文）标点，不过，在大部分情况下，应该是够用的。

假设现在想把字符串 title = '你好，hello，世界' 中的中文提取出来，可以这么做：

```
import re

title = '你好，hello，世界'
pattern = re.compile(r'[\u4e00-\u9fa5]+')
result = pattern.findall(title)

print result
```

注意到，我们在正则表达式前面加上了两个前缀 r，其中 r 表示使用原始字符串，

```
[u'\u4f60\u597d', u'\u4e16\u754c']
```

注意：贪婪模式与非贪婪模式

1. 贪婪模式：在整个表达式匹配成功的前提下，尽可能多的匹配 (*)；
2. 非贪婪模式：在整个表达式匹配成功的前提下，尽可能少的匹配 (*?)；
3. Python 里数量词默认是贪婪的。

例一： 源字符串：abbbc

- 使用贪婪的数量词的正则表达式 ab* ，匹配结果： abbb。

* 决定了尽可能多匹配 b，所以 a 后面所有的 b 都出现了。

- 使用非贪婪的数量词的正则表达式 ab*?，匹配结果： a。

即使前面有 *，但是 ? 决定了尽可能少匹配 b，所以没有 b。

例二： 源字符串：aa test1 bb test2 cc

- 使用贪婪的数量词的正则表达式： .*
- 匹配结果： test1 bb test2

这里采用的是贪婪模式。在匹配到第一个 时已经可以使整个表达式匹配成功，但是由于采用的是贪婪模式，所以仍然要向右尝试匹配，查看是否还有更长的可以成功匹配的子串。匹配到第二个 后，向右再没有可以成功匹配的子串，匹配结束，匹配结果为
test1 bb test2

- 使用非贪婪的数量词的正则表达式： .*?

- 匹配结果： test1

正则表达式二采用的是非贪婪模式，在匹配到第一个“”时使整个表达式匹配成功，由于采用的是非贪婪模式，所以结束匹配，不再向右尝试，匹配结果为“ test1 ”。

2.正则表达式爬虫案例

1) 需求：

提取猫眼电影 TOP100 的电影名称，时间，评分，图片等信息

2) 完整代码：

```
import json
import requests
from requests.exceptions import RequestException
import re
import time

# 抓取首页
def get_one_page(url):
    try:
        headers = {
            'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13
_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/
537.36'
        }
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            return response.text
        return None
    except RequestException:
        return None

# 正则提取
def parse_one_page(html):
    pattern = re.compile('<dd>.*?board-index.*?>(\d+)</i>.*?data-src="(.
*?)".*?name"><a'
                         + '.*?>(.*?)</a>.*?star">(.*?)</p>.*?releaseti
me">(.*?)</p>'
                         + '.*?integer">(.*?)</i>.*?fraction">(.*?)</i>.
*?</dd>', re.S)
    items = re.findall(pattern, html)
    for item in items:
        yield {
            'index': item[0],
            'image': item[1],
            'title': item[2],
```

```
            'actor': item[3].strip()[3:],
            'time': item[4].strip()[5:],
            'score': item[5] + item[6]
        }

# 写入文件
def write_to_file(content):
    with open('result.txt', 'a', encoding='utf-8') as f:
        f.write(json.dumps(content, ensure_ascii=False) + '\n')


def main(offset):
    url = 'http://maoyan.com/board/4?offset=' + str(offset)
    html = get_one_page(url)

    #分页爬取
    for item in parse_one_page(html):
        print(item)
        write_to_file(item)


if __name__ == '__main__':
    for i in range(10):
        main(offset=i * 10)

        # 添加一个延时等待，防止速度过快被反爬
        time.sleep(1)
```

三、XPath

1.XML

- XML 指可扩展标记语言（EXtensible Markup Language）
- XML 是一种标记语言，很类似 HTML
- XML 的设计宗旨是传输数据，而非显示数据
- XML 的标签需要我们自行定义。
- XML 被设计为具有自我描述性。
- XML 是 W3C 的推荐标准

W3School 官方文档：http://www.w3school.com.cn/xml/index.asp

数据格式	描述	设计目标

XML	Extensible Markup Language （可扩展标记语言）	被设计为传输和存储数据，其焦点是数据的内容。
HTML	HyperText Markup Language （超文本标记语言）	显示数据以及如何更好显示数据。
HTML DOM	Document Object Model for HTML (文档对象模型)	通过 HTML DOM，可以访问所有的 HTML 元素，连同它们所包含的文本和属性。可以对其中的内容进行修改和删除，同时也可以创建新的元素

2.什么是 XPath？

Path 是一门在 XML 文档中查找信息的语言。可用来在 XML 文档中对元素和属性进行遍历。是 W3C XSLT 标准的主要元素，并且 XQuery 和 XPointer 都构建于 XPath 表达之上。因此，对 XPath 的理解是很多高级 XML 应用的基础。 W3School 官方文档：http://www.w3school.com.cn/xpath/index.asp

3.XPath 开发工具

1. 开源的 XPath 表达式编辑工具:XMLQuire(XML 格式文件可用)
2. Chrome 插件 XPath Helper
3. Firefox 插件 XPath Checker

4.Xpath 使用

1) 选取节点

XPath 使用路径表达式来选取 XML 文档中的节点或者节点集。这些路径表达式和我们在常规的电脑文件系统中看到的表达式非常相似。

最常用的路径表达式：

表达式	描述
nodename	选取此节点的所有子节点。
/	从根节点选取。
//	从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置。
.	选取当前节点。
..	选取当前节点的父节点。
@	选取属性。

一些路径表达式以及表达式的结果:

路径表达式	结果
bookstore	选取 bookstore 元素的所有子节点。
/bookstore	选取根元素 bookstore。注释：假如路径起始于正斜杠(/)，则此路径始终代表到某元素的绝对路径！
bookstore/book	选取属于 bookstore 的子元素的所有 book 元素。
//book	选取所有 book 子元素，而不管它们在文档中的位置。
bookstore//book	选择属于 bookstore 元素的后代的所有 book 元素，而不管它们位于 bookstore 之下的什么位置。
//@lang	选取名为 lang 的所有属性。

2) 谓语

谓语用来查找某个特定的节点或者包含某个指定的值的节点，被嵌在方括号中。

在下面的表格中，我们列出了带有谓语的一些路径表达式，以及表达式的结果：

路径表达式	结果
/bookstore/book[1]	选取属于 bookstore 子元素的第一个 book 元素。
/bookstore/book[last()]	选取属于 bookstore 子元素的最后一个 book 元素。
/bookstore/book[last()-1]	选取属于 bookstore 子元素的倒数第二个 book 元素。
/bookstore/book[position()< 3]	选取最前面的两个属于 bookstore 元素的子元素的 book 元素。
//title[@lang]	选取所有拥有名为 lang 的属性的 title 元素。
//title[@lang='eng']	选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。
/bookstore/book[price>35.00]	选取 bookstore 元素的所有 book 元素，且其中的 price 元素的值须大于 35.00。
/bookstore/book[price>35.00]/title	选取 bookstore 元素中的 book 元素的所有 title 元素，且其中的 price 元素的值须大于 35.00。

3) 选取未知节点

XPath 通配符可用来选取未知的 XML 元素。

通配符	描述

*	匹配任何元素节点。
@*	匹配任何属性节点。
node()	匹配任何类型的节点。

在下面的表格中，我们列出了一些路径表达式，以及这些表达式的结果：

路径表达式	结果
/bookstore/*	选取 bookstore 元素的所有子元素。
//*	选取文档中的所有元素。
//title[@*]	选取所有带有属性的 title 元素。

5.lxml 库

lxml 是 一个 HTML/XML 的解析器，主要的功能是如何解析和提取 HTML/XML 数据。 lxml 和正则一样，也是用 C 实现的，是一款高性能的 Python HTML/XML 解析器，我们可以利用之前学习的 XPath 语法，来快速的定位特定元素以及节点信息。 lxml python 官方文档：http://lxml.de/index.html 需要安装 C 语言库，可使用 pip 安装：pip install lxml （或通过 wheel 方式安装）

1) 初步使用

```
# 使用 lxml 的 etree 库
from lxml import etree

text = '''
<div>
    <ul>
         <li class="item-0"><a href="link1.html">first item</a></li>
         <li class="item-1"><a href="link2.html">second item</a></li>
         <li class="item-inactive"><a href="link3.html">third item</a><
/li>
         <li class="item-1"><a href="link4.html">fourth item</a></li>
         <li class="item-0"><a href="link5.html">fifth item</a> #此处缺
少一个 </li> 闭合标签
     </ul>
 </div>
'''

#利用etree.HTML，将字符串解析为HTML 文档
html = etree.HTML(text)

# 按字符串序列化 HTML 文档
result = etree.tostring(html)

print(result)
```

输出结果：

```
<html><body>
<div>
    <ul>
         <li class="item-0"><a href="link1.html">first item</a></li>
         <li class="item-1"><a href="link2.html">second item</a></li>
         <li class="item-inactive"><a href="link3.html">third item</a><
/li>
         <li class="item-1"><a href="link4.html">fourth item</a></li>
         <li class="item-0"><a href="link5.html">fifth item</a></li>
</ul>
 </div>
</body></html>
```

lxml 可以自动修正 html 代码，例子里不仅补全了 li 标签，还添加了 body，html 标签。

文件读取： 除了直接读取字符串，lxml 还支持从文件里读取内容。我们新建一个 hello.html 文件：

```
<div>
    <ul>
         <li class="item-0"><a href="link1.html">first item</a></li>
         <li class="item-1"><a href="link2.html">second item</a></li>
         <li class="item-inactive"><a href="link3.html"><span class="bo
ld">third item</span></a></li>
         <li class="item-1"><a href="link4.html">fourth item</a></li>
         <li class="item-0"><a href="link5.html">fifth item</a></li>
     </ul>
 </div>
```

再利用 etree.parse() 方法来读取文件。

```
from lxml import etree

# 读取外部文件 hello.html
html = etree.parse('./hello.html')
result = etree.tostring(html, pretty_print=True)

print(result)
```

输出结果与之前相同：

```
<html><body>
<div>
    <ul>
         <li class="item-0"><a href="link1.html">first item</a></li>
         <li class="item-1"><a href="link2.html">second item</a></li>
         <li class="item-inactive"><a href="link3.html">third item</a><
/li>
         <li class="item-1"><a href="link4.html">fourth item</a></li>
```

```
        <li class="item-0"><a href="link5.html">fifth item</a></li>
</ul>
 </div>
</body></html>
```

2) XPath 实例测试

1. 获取所有的<li>标签

```
from lxml import etree

html = etree.parse('hello.html')
print(type(html))  # 显示etree.parse() 返回类型

result = html.xpath('//li')

print(result)  # 打印<li>标签的元素集合
print(len(result))
print(type(result))
print(type(result[0]))
```

输出结果：

```
<type 'lxml.etree._ElementTree'>
[<Element li at 0x1014e0e18>, <Element li at 0x1014e0ef0>, <Element li
at 0x1014e0f38>, <Element li at 0x1014e0f80>, <Element li at 0x1014e0fc
8>]
5
<type 'list'>
<type 'lxml.etree._Element'>
```

2. 继续获取 标签的所有 class 属性

```
from lxml import etree

html = etree.parse('hello.html')
result = html.xpath('//li/@class')

print(result)
```

运行结果

```
['item-0', 'item-1', 'item-inactive', 'item-1', 'item-0']
```

3. 继续获取<li>标签下 hre 为 link1.html 的<a>标签

```
from lxml import etree

html = etree.parse('hello.html')
result = html.xpath('//li/a[@href="link1.html"]')
```

```
print(result)
```

运行结果

```
[<Element a at 0x10ffaae18>]
```

4. 获取 标签下的所有 标签

```
from lxml import etree

html = etree.parse('hello.html')

#result = html.xpath('//li/span')
#注意这么写是不对的:
#因为 / 是用来获取子元素的，而 <span> 并不是 <li> 的子元素，所以，要用双斜杠

result = html.xpath('//li//span')

print(result)
```

运行结果

```
[<Element span at 0x10d698e18>]
```

5. 获取 <li> 标签下的<a>标签里的所有 class

```
from lxml import etree

html = etree.parse('hello.html')
result = html.xpath('//li/a//@class')

print(result)
```

运行结果

```
['blod']
```

6. 获取最后一个 <li> 的 <a> 的 href

```
from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li[last()]/a/@href')
# 谓语 [last()] 可以找到最后一个元素

print(result)
```

运行结果

```
['link5.html']
```

7. 获取倒数第二个元素的内容

```
from lxml import etree

html = etree.parse('hello.html')
result = html.xpath('//li[last()-1]/a')

# text 方法可以获取元素内容
print(result[0].text)
```

运行结果

```
fourth item
```

8. 获取 class 值为 bold 的标签名

```
from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//*[@class="bold"]')

# tag 方法可以获取标签名
print(result[0].tag)
```

运行结果

```
span
```

6.XPath 爬虫案例

```
import urllib.parse
import urllib.request
from lxml import etree


class Imagespider:
    def __init__(self):
        self.tiebaname = input("请输入需要爬取的贴吧名:")
        self.beginPage = int(input("请输入起始页："))
        self.endPage = int(input("请输入结束页："))
        self.url = "http://tieba.baidu.com/f?"
        self.headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) Ap
pleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36"}


    def loadPage(self, url):
```

```
        html = urllib.request.urlopen(url).read()
        content = etree.HTML(html)
        link_list = content.xpath('//div[@class="t_con cleafix"]/div[2]
/div[1]/div[1]/a/@href')
        for link in link_list:
            fulllink = "http://tieba.baidu.com" + link
            self.loadImage(fulllink)

    def loadImage(self, link):
        request = urllib.request.Request(link, headers=self.headers)
        html = urllib.request.urlopen(request).read()
        content = etree.HTML(html)
        link_list = content.xpath('//img[@class="BDE_Image"]/@src')
        for link in link_list:
            self.writeImage(link)

    def writeImage(self, link):
        request = urllib.request.Request(link, headers=self.headers)
        image = urllib.request.urlopen(request).read()
        # 取出连接后10 位做为文件名
        filename = link[-10:]
        with open(filename, "wb") as f:
            f.write(image)
        print("已经成功下载 " + filename)

    def tiebaSpider(self):
        for page in range(self.beginPage, self.endPage + 1):
            pn = (page - 1) * 50
            key = urllib.parse.urlencode({'pn': pn, "kw": self.tiebanam
e})
            fullurl = self.url + key
            self.loadPage(fullurl)
            print("谢谢使用")


if __name__ == "__main__":
    mySpider = Imagespider()
    mySpider.tiebaSpider()
```