信息的内容解析与提取

当我们可以使用自定义的爬虫程序访问到目标网页，并将他们以响应形式获取到本地的时候，往往会发现里面的信息仅有一部分是我们需要的，为了提取出这些有价值信息就需要对页面内容进行解析。

信息解析与提取的一般方法

1. 完整解析信息的标记形式，再提取关键信息；
    - 需要标记解析器；
    - 优点是解析准确；
    - 缺点是提取过程繁琐/速度慢。
2. 不解析全文，直接搜索信息；
    - 需要文本查找函数；
    - 优点是提取过程简洁，速度快；
    - 缺点是提取结果准确性和信息内容相关。
3. 适应性方法
    - 结合上述两种方法的方法。

有一些信息或数据是不需要内容解析的，例如图片、音频、视频等，而更常见的网页、文本或表格文件等是需要进行内容解析和提取的。需要内容解析和提取的信息可分为三类：

- 无结构的文本信息，例如txt文本；
- 半结构化的标记型文本信息，例如html网页、json数据、xml数据等；
- 结构化的信息，例如数据库文件、电子表格文件等；

针对不同类型的信息，有不同的信息解析方法。需要注意的是，文本信息解析不是指对文本全局的理解，而是指在文本中找到所需的内容，这是由于网络爬虫程序的目标是获取有价值的信息。

无结构文本信息的解析

从无结构的纯文本中匹配查找有价值信息的方法有：

- 利用正则表达式进行模式匹配

半结构标记型文本信息的解析

文本信息标记类型

文本信息大量存在于网络之中。无结构的文本信息往往是难以识别和理解的，对信息进行标记，使其具备一定的分类结构，可以有助于机器和人们理解信息。

国际上现今流行的文本信息标记形式有三种：

- XML，可扩展标记语言
- JSON，Javascript Object Notation，使用有类型的键值对表达信息。
  - 例如：{"name":"北京航空航天大学"}、{"value":1}、{"名字": ["王琪","王其梅"]}
  - 键值对可以嵌套使用
  - 方便之处是，JSON数据可以作为程序的一部分。
- YAML，YAML Ain't Markup Language，无类型的键值对表示信息。
  - 例如：

```
Name: -张三 -张优良
```

文本信息的解析

由于HTML、XML、JSON等半结构化信息的内容以字符形式表示，所以对这类信息的解析，一方面可以利用正则表达式进行模式匹配；另一方面可以利用半结构化文本中的标记或标签，通过对标签的匹配来查找所需的信息。

从半结构化的HTML中解析信息的方法主要有：

- 针对HMTL文档，有下列方法
  - 利用正则表达式进行模式匹配
  - 利用xpath进行HTML标签检索
  - 利用CSS选择器进行HTML标签检索
- 针对JSON数据
  - 利用JSON Path进行检索
  - 利用Python类型转换为json类
- 针对XML数据
  - 转化成Python类型（xmltodict）
  - XPath
  - CSS选择器
  - 正则表达式

结构化数据的解析

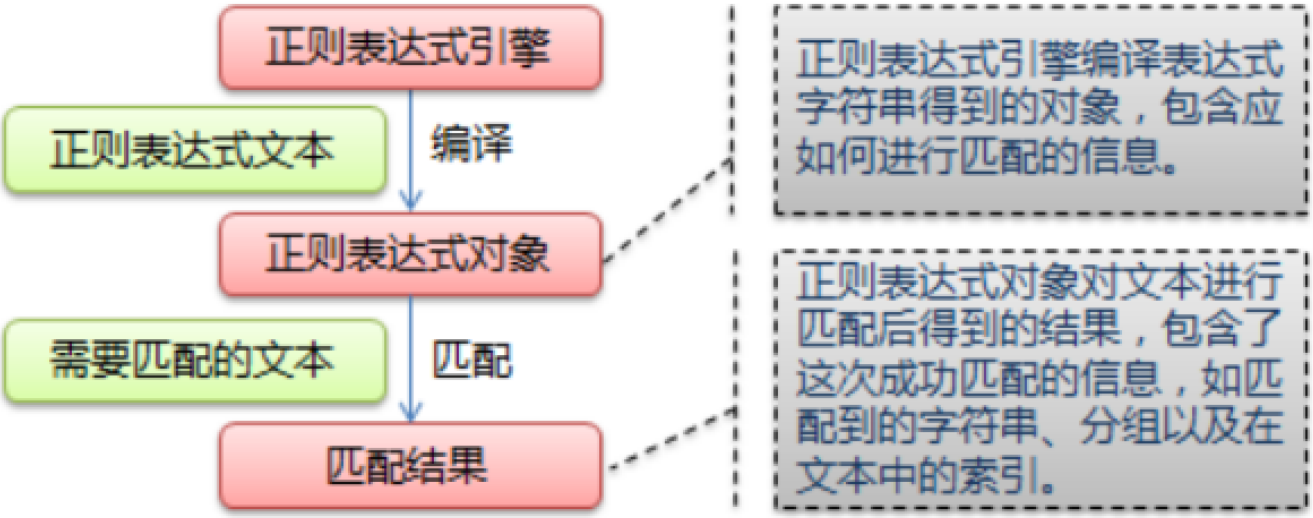
结构化数据，例如数据库文件、excel文件等，需要借助相应的API或python类库来解析。

- 针对各类数据库文件，可以借助相关的python库进行解析和读写
  - 对于Mysql数据库文件，可以使用pymysql库；
  - 对于Sqllite数据库文件，可以使用Sqlite3库；
  - 对于MS SqlServer数据库文件，可以使用pyodbc+pymssql库；
  - 对于Orange数据库文件，可以使用cx_oracle库；
  - ...
- 针对excel文件，可以借助下列python库进行内容解析和读写
  - xlwings：简单强大，可替代VBA；
  - openpyxl：简单易用，功能广泛；
  - pandas：数据处理功能强大；
  - win32com：还可以处理office其他类型文件;
  - Xlsxwriter：易于生成Excel文档；
  - DataNitro：内嵌于excel中，可替代VBA；
  - xlutils：结合xlrd/xlwt使用。

利用正则表达式实现文本信息提取

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”）。



正则表达式的特殊字符与匹配规则

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c
数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	
边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc
逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式	(abc){2} a(123|456)c	abcabc a456c

	中的|仅在该组中有效。		
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用'|'或后接数量词。	(?:abc){2}	abcabc
(?iLmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc2 abcabc

python中的正则表达式模块与应用

python3内置的re模块，包含了正则表达式的操作集。

re模块的一般使用步骤如下：

1. 编译正则表达式，即使用 compile() 函数将正则表达式的字符串形式编译为一个 Pattern 对象；
2. 对目标字符串进行匹配，即通过 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（Match 对象）；
3. 提取结果信息，即使用 Match 对象提供的属性和方法获得信息，还可根据需要进行其他的操作。

compile 函数

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

```
import re
pattern = re.compile(一个正则表达式)
```

成功编译并构造pattern对象后，就可以使用pattern对象方法查找、替换、统计目标字符串中与正则表达式匹配的子字符串了。pattern对象可调用的方法有：

- match 方法：从起始位置开始查找，一次匹配
- search
- fullmatch
- sub
- subn
- split
- purge
- template
- escape
- error
- findall
- finditer

事实上，这些函数不仅是对象可调用的，也是模块级的，即可以使用re模块直接调用。

match方法

match方法用于在字符串起始位置进行模式匹配，若匹配则返回Match对象，否则返回None。

> match(pattern, string, flags=0) method of re module Try to apply the pattern at the start of the string, returning a match object, or None if no match was found. match(string=None, pos=0, endpos=None, pattern=None) method of Pattern instance Matches zero or more characters at the beginning of the string.

下面举例说明：

In [1]:

```
"""re模块compile方法与match方法示例
"""
import re
# 目标字符串
text1 = 'Hello world, abcdefg, 1234567890,ABCDEFG.1+1=2'
text2 = 'abcdefg, 1234567890,ABCDEFG.1+1=2'
text3 = 'ABCDEFG'
# 定义正则表达式,下列可以用于匹配目标字串中的电子邮件
regexstr = 'abc'
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(regexstr)
# 保存结果为match对象
match = pattern.match(text1)
print(match)
print('--'*10)

match = pattern.match(text2)
print(match)
if match:
    print(match.group())
print('--'*10)
match = pattern.match(text1,pos=13)
if match:
    print(match.group())
print('--'*10)
pattern = re.compile(regexstr,flags=re.IGNORECASE)
match = pattern.match(text3)
if match:
    print(match.group())
```

```
None
--------------------
<_sre.SRE_Match object; span=(0, 3), match='abc'>
abc
--------------------
abc
--------------------
ABC
```

Match对象

match对象是正则表达式匹配目标字符串后返回的结果对象。它可调用一下方法:

- 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))。

In [2]:

```
"""match方法示例
"""
import re

#
text = 'address123@example.com123中文'
#定义正则表达式,下列可以用于匹配目标字串中的电子邮件
regexstr = '\w+@\w+\.[a-z]+'
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(regexstr)
#
match = pattern.match(text)
if match:
    print(match.group())
    print(match.start())
    print(match.end())
    print(match.span())
```

```
address123@example.com
0
22
(0, 22)
```

search 方法

search 方法用于查找字符串的任何位置，它只返回从左至右第一个匹配的结果，而不是查找所有匹配的结果。

In [ ]:

```
"""search方法示例"""
import re

text = 'one12twothree34four'
#设置正则式查找第一个数字串
pattern = re.compile('\d+')
match = pattern.search(text)  # 这里如果使用 match 方法则不匹配
if match:
    print(match.group())
    print(match.span())
```

findall 与finditer 方法

上面的 match 和 search 方法都是一次性匹配，而有时需要获取目标字串中所有匹配的结果，这需要使用findall获finditer方法。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)。

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。

In [ ]:

```
"""findall 与finditer 方法"""
import re

text = 'one12twothree34four'
#设置正则式查找第一个数字串
pattern = re.compile('\d+')
matchlist = pattern.findall(text)  # 这里如果使用 match 方法则不匹配
if matchlist:
    print(matchlist)
else:
    print('None')

print('--'*10)

matchiter = pattern.finditer(text)
if matchiter:
    print(matchiter)
    for m in  matchiter:
        print('Match result: {} ,position: {}'.format(m.group(),m.span()))
else:
    print('None')
```

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。

In [ ]:

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

sub 方法

sub 方法用于替换。它的使用形式如下：

sub(repl, string[, count])

- 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。

In [ ]:

```
"""sub 方法示例"""
import re

# 设置模式：两个英文字符或数字的分组，中间用空格隔开
# \w = [A-Za-z0-9]
p = re.compile(r'(\w+) (\w+)')
s = 'hello 123, hello 456'

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

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
```

匹配中文

正则表达式不仅能匹配英文还可以匹配其他语言字符，例如匹配中文。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。

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

In [4]:

```
import re

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

print(result)
```

['你好', '世界']

贪婪模式与非贪婪模式

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

In [5]:

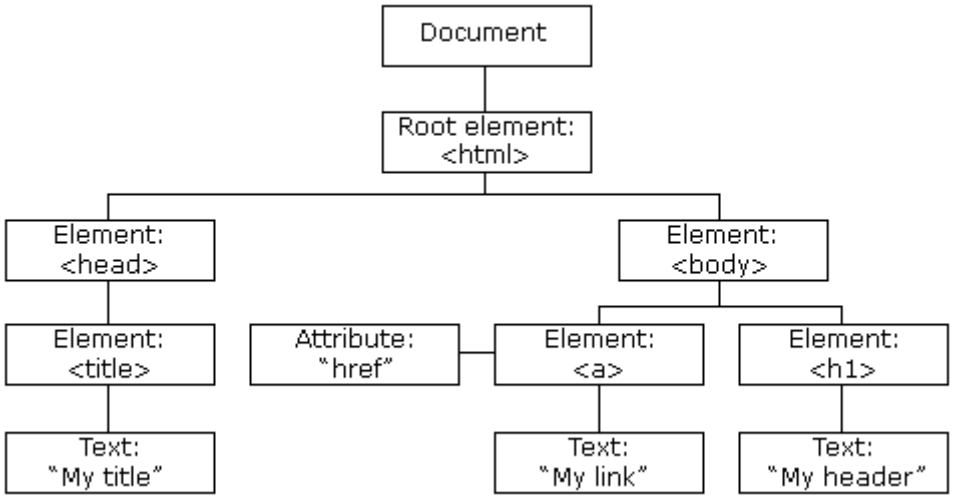
```
"""贪婪模式与非贪婪模式示例"""
import re

p = re.compile('ab*')
m = p.match('abbbbbbbbbbbbbbbccccccccccc')
print(m.group())

p = re.compile('ab*?')
m = p.match('abbbbbbbbbbbbbbbccccccccccc')
print(m.group())

t = 'aa<div>test1</div>bb<div>test2</div>cc'
p = re.compile('<div>.*</div>')
m = p.search(t)
print(m.group())

t = 'aa<div>test1</div>bb<div>test2</div>cc'
p = re.compile('<div>.*?</div>')
m = p.search(t)
print(m.group())
```

```
abbbbbbbbbbbbbbb
a
<div>test1</div>bb<div>test2</div>
<div>test1</div>
```

利用XPATH实现XML和HTML文本信息提取

XML指可扩展标记语言,被设计用来传输和存储数据。有关XML的详细介绍，可参考W3School官方文档: http://www.w3school.com.cn/xml/ (http://www.w3school.com.cn/xml/) HTML指的是超文本标记语言 (Hyper Text Markup Language),是WWW上用于编写网页的主要工具，详细信息请参考http://www.w3school.com.cn/htm (http://www.w3school.com.cn/htm) XML和HTML都是一种标记语言 (markup language)，使用标记标签来描述数据，这些标签可用于查找和定位数据。

HTML DOM 定义了访问和操作 HTML 文档的标准方法，以树结构方式表达 HTML 文档。



XML的节点关系

下面是xml文档的一个例子：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

    <book>
      <title lang="eng">Harry Potter</title>
      <price>29.99</price>
    </book>

    <book>
      <title lang="eng">Learning XML</title>
      <price>39.95</price>
    </book>

</bookstore>
```

- 父（Parent）每个元素以及属性都有一个父，上例中book 元素是 title、price 元素的父结点；
- 子（Children）元素节点可有零个、一个或多个子，上例中title、price 元素都是 book 元素的子结点；
- 同胞（Sibling）拥有相同的父的节点，上例中title、price 为同胞结点；

- 先辈（Ancestor）某节点的父、父的父等等，上例中title、price 的先辈结点是bookstore。
- 后代（Descendant）某个节点的子，子的子，上例中bookstore的后代结点为book、title、price。

XPath语法

XPATH

XPath (XML Path Language) 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。细节可以参考W3School官方文档：http://www.w3school.com.cn/xpath/index.asp (http://www.w3school.com.cn/xpath/index.asp)

XPath 使用路径表达式来选取 XML 文档中的节点或节点集。节点是通过沿着路径 (path) 或者步 (steps) 来选取的。

XPath 开发工具有：

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

使用XPATH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。这些路径表达式与文件系统中的表达式非常相似。

下面列出了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

表达式	描述
.	选取当前节点
..	选取当前节点的父节点
@	选取属性

"实例" 在下面的表格中，我们已列出了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

谓语

XPATH中的谓语（Predicates）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。 "实例"

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。

路径表达式	结果
/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。

XPATH通配符

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。

"实例" 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。

“|”运算符

通过在路径表达式中使用“|”运算符，您可以选取若干个路径。

"实例" 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
//book/title | //book/price	选取 book 元素的所有 title 和 price 元素。
//title | //price	选取文档中的所有 title 和 price 元素。
/bookstore/book/title | //price	选取属于 bookstore 元素的 book 元素的所有 title 元素，以及文档中所有的 price 元素。

以上是有关XPATH的部分语法，通过XPATH可以简单定位XML中的信息。如果能将其他标签语言文本转换为XML文本，XPATH就可用于该文本的选取。

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。

使用前需要安装：pip install lxml

In [6]:

```
"""lxml解析xml文档的使用示例"""
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>
     </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.decode('utf-8'))
# 注意，lxml会自动补齐缺少的</li>闭合标签
```

```
<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 it
em</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。

利用lxml读取文件

除了直接读取字符串，lxml还支持从文件里读取内容。

In [ ]:

```
"""lxml读取xml文档的示例"""
from lxml import etree

# 读取外部文件axmldoc.xml

html = etree.parse('./axmldoc.xml')
result = etree.tostring(html, pretty_print=True)

print(result.decode('utf-8'))
```

XPath选取信息实践

1. 获取所有的 li 标签
2. 获取 li 标签的所有 class属性
3. 继续获取 li 标签下href为 link1.html 的 a 标签
4. 获取li 标签下的所有span 标签
5. 获取 li 标签下的 a 标签里的所有 class
6. 获取最后一个li 的 a 的 href
7. 获取倒数第二个元素的内容
8. 获取 class 值为 bold 的标签名

In [7]:

```
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"><span class="bold">third i
tem</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>
'''
html = etree.HTML(text)

# 1. 获取所有的 li 标签
result = html.xpath('//li')
print(result)

# 2. 获取 li 标签的所有 class属性
result = html.xpath('//li/@class')
print(result)

# 3. 获取 li 标签下href为 link1.html 的 a 标签
result = html.xpath('//li/a[@href="link1.html"]')
print(result)
# 4. 获取li 标签下的所有span 标签
result = html.xpath('//li//span')
print(result)
# 5. 获取 li 标签下的 a 标签里的所有 class
result = html.xpath('//li/a//@class')
print(result.pop())
# 6. 获取最后一个li 的 a  的 href
result = html.xpath('//li[last()]/a/@href')
print(result)
# 7. 获取倒数第二个元素的内容
result = html.xpath('//li[last()-1]/a/text()')
print(result)
```

```
# 8. 获取 class 值为 bold 的标签名
result = html.xpath('//*[@class="bold"]')
print(result[0].tag)
```

```
[<Element li at 0x1f71e892dc8>, <Element li at 0x1f71e892e08>, <
Element li at 0x1f71e892e48>, <Element li at 0x1f71e892e88>, <El
ement li at 0x1f71e892ec8>]
['item-0', 'item-1', 'item-inactive', 'item-1', 'item-0']
[<Element a at 0x1f71e8a4088>]
[<Element span at 0x1f71e892dc8>]
['bold']
['link5.html']
['fourth item']
span
```

应用BeautifulSoup库解析网页内容

简介与安装

官方地址：https://www.crummy.com/software/BeautifulSoup/ (https://www.crummy.com/software/BeautifulSoup/) 中文文档：https://beautifulsoup.readthedocs.io/zh_CN/v4.4.0/ (https://beautifulsoup.readthedocs.io/zh_CN/v4.4.0/)

Beautiful Soup 是一个可以从HTML或XML文件中提取数据的Python库，它能够通过你喜欢的parser实现文档导航、查找、修改文档的parser tree。Beautiful Soup会帮你节省数小时甚至数天的工作时间.

安装过程很简单，在安装python与pip工具后，运行下面语句：

```
pip install beautifulsoup4
```

使用lxml parser解析HTML并提取内容

首先看一个简单应用BeautifulSoup解析网页的例子。在这个例子中，BeautifulSoup会依据HTML文档文本建立对象。

In [9]:

```
"""BeautifulSoup示例"""
from bs4 import BeautifulSoup

htmlDoc = """
<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>
"""
#指定parser为html.parser
bs = BeautifulSoup(htmlDoc,'html.parser')
print(type(bs))
print(bs)
```

```
<class 'bs4.BeautifulSoup'>

<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 sister
s; and their names were
<a class="sister" href="http://example.com/elsie" id="link1">Els
ie</a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lac
ie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Ti
llie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
</body></html>
```

上面的例子中，建立了一个BeautifulSoup对象html，并指定了文档解析器为“html.parser”，类似的parser有lxml库的HTML parser、XMLparser，以及html5lib库的parser，这些parser都可以通过“lxml”、“lxml-xml”、“html5lib”等字符串形式引用。注意，事先应安装lxml库或html5lib库。

BeautifulSoup中常见对象

使用BeautifulSoup解析文档，首先需要建立BeautifulSoup对象，这是一个复杂的树形对象，它有大量用于查找和修改文档的方法。常用的对象有：

- BeautifulSoup对象：表示的是一个文档的全部内容，它与Tag对象很类似；
- 标签对象Tag：对象与XML或HTML原生文档中的tag（标签）相同。Tag有很多方法和属性,最重要的属性是：
  - name：Tag名，例如body、a；
  - attributes：Tag的属性；
  - 事实上，对于xml或html文档对象，也是Tag对象的特殊类型。
- 可遍历的字符串对象NavigableString，这类字符串通常被包围在一些标签中，通过标签对象.string属性访问；
  - 使用标签对象的string属性获得；
  - 可以跨越多个标签层次；
- 注释对象Comment：一个特殊类型的 NavigableString 对象。

如果你有一定的HTML文档知识，你会发现这些对象对于处理HTML文档细节时很有用。下面我们举几个例子来加深理解。

In [11]:

```
"""BeautifulSoup示例"""
from bs4 import BeautifulSoup

htmlDoc = """
<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>
<div><!-- This is a comment --></div>
</body>
</html>
"""
# 将文档映射为BeautifulSoup对象，映射时使用指定parser为html.parser
bsoup = BeautifulSoup(htmlDoc,'html.parser')
print('---'*10)
# 利用Tag对象获取标签及内部信息
print(bsoup.head)
print(bsoup.title)
print(bsoup.body)

# 获取第一个a标签
print('---'*10)
tagA = bsoup.a
print(type(tagA))
print(tagA)
# 获取所有a标签
print('---'*10)
print(bsoup.findAll(name=bsoup.a.name))
print('---'*10)
#可操作字符串NavigableString对象
print(bsoup.p.string)
```

```
print(type(bsoup.p.string))

#Comments对象
print(bsoup.div.string)
print(type(bsoup.div.string))
```

```
------------------------------
<head><title>The Dormouse's story</title></head>
<title>The Dormouse's story</title>
<body>
<p class="title"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sister
s; and their names were
    <a class="sister" href="http://example.com/elsie" id="link
1">Elsie</a>,
    <a class="sister" href="http://example.com/lacie" id="link
2">Lacie</a> and
    <a class="sister" href="http://example.com/tillie" id="link
3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
<div><!-- This is a comment --></div>
</body>
------------------------------
<class 'bs4.element.Tag'>
<a class="sister" href="http://example.com/elsie" id="link1">Els
ie</a>
------------------------------
[<a class="sister" href="http://example.com/elsie" id="link1">El
sie</a>, <a class="sister" href="http://example.com/lacie" id="l
ink2">Lacie</a>, <a class="sister" href="http://example.com/till
ie" id="link3">Tillie</a>]
------------------------------
The Dormouse's story
<class 'bs4.element.NavigableString'>
 This is a comment
<class 'bs4.element.Comment'>
```

从上面的例子可以明确，成功构建BeautifulSoup对象后，文档内的标签都可以被视为一个Tag对象来处理。HTML文档被映射为BeautifulSoup对象后，里面所有的标签都可以用Tag对象来访问。

标签属性的获取

Tag对象的name、attrs等属性能够令我们十分便捷的获取相关信息。如果存在多个name属性一样的标签，Tag对象会返回从上到下的第一个同名标签。 例如：Tag对象的attrs属性可以获取该标签中的所有属性列表。

HMTL中存在一些多值属性，最常见的就是class属性（用于定义CSS类），类似的属性有：

> class，rel, rev, accept-charset, headers, and accesskey等

In [ ]:

```
"""BeautifulSoup示例"""
from bs4 import BeautifulSoup

htmlDoc = """
<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>
<div><!-- This is a comment --></div>
</body>
</html>
"""
# 将文档映射为BeautifulSoup对象，映射时使用指定parser为html.parser
bsoup = BeautifulSoup(htmlDoc,'html.parser')
tagA = bsoup.a
print(tagA.name)
print(tagA.attrs)
print(tagA.text)
print('---'*10)
print(tagA.parent)
print('---'*10)
```

文档标签的遍历

在BeautifulSoup中遍历各个Tag对象（标签）时有三种方式，相对应的遍历方法有：

- 下行遍历（从根到叶）
  - .contents：子节点的列表，将tag所有儿子结点存入列表；

    - .children：子节点的迭代类型，与.contents类似，用于循环遍历子节点；
    - .deseendants：子孙结点的迭代类型，包含所有子孙节点，用于循环遍历。
- 上行遍历（从叶到根）
    - .parent：节点的父亲标签；
    - .parents：节点先辈标签的迭代类型，用于循环遍历先辈节点。
- 平行遍历（同一父亲节点下的兄弟之间遍历）
    - .next_sibling：返回按照HTML文本顺序的下一个平行节点标签；
    - .previous_sibling：返回按照HTML文本顺序的上一个平行节点标签；
    - .next_sibling：迭代类型，返回按照HTML文本顺序的后续所有平行标签；
    - .previous_siblings：迭代类型，返回按照HTML文本顺序的前续所有平行节点的标签。

In [ ]:

```
"""BeautifulSoup文档标签遍历示例"""
from bs4 import BeautifulSoup

htmlDoc = """
<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>
<div><!-- This is a comment --></div>
</body>
</html>
"""
# 将文档映射为BeautifulSoup对象，映射时使用指定parser为html.parser
bsoup = BeautifulSoup(htmlDoc,'html.parser')
print(bsoup.head)

# 用于下行遍历的几个属性
print(bsoup.head.contents)
print('---'*10)
print(bsoup.body.contents)
print('---'*10)
print( bsoup.body.children)
print([_ for _ in bsoup.body.children])
print('---'*10)
print(bsoup.body.descendants)
print([_.name for _ in bsoup.body.descendants])

# 用于上行遍历的属性
print('---'*20)
print(bsoup.b)
print('---'*10)
```

```
print(bsoup.b.parent)
print('---'*10)
print([_.name for _ in bsoup.b.parents])

# 用于平行遍历的属性
print('---'*20)
print(bsoup.a)
print(bsoup.a.nextsibling)
print(bsoup.a.previous_sibling)
print([_.name for _ in bsoup.a.next_siblings])
print([ _.name for _ in bsoup.a.previous_siblings])
```

BeautifulSoup的其他辅助功能

- 文档结构展现的优化

prettify()函数可以更好的展现HTML或XML结构，有利于人们浏览文档。

In [ ]:

```
"""BeautifulSoup文档prettify示例"""
from bs4 import BeautifulSoup

htmlDoc = """
<html><head><title>The Dormouse's story</title></head><body><p class="title"><b>Th
e Dormouse's story</b></p><p class="story">Once upon a time there were three littl
e sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a><a href="htt
p://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>and they l
ived at the bottom of a well.</p><p class="story">...</p><div><!-- This is a comme
nt --></div></body></html>
"""
# 将文档映射为BeautifulSoup对象，映射时使用指定parser为html.parser
bsoup = BeautifulSoup(htmlDoc,'html.parser')
print(bsoup.prettify())
```

BeautifulSoup文本内容查找方法

BeautifulSoup中有多个内容查找方法：

- find()
- find_all(name, attrs, recursive, string, **kwargs)
  - 功能：查找所有复合条件的标签，返回一个列类型；
  - name: 对标签名称的检索字符串；
-

In [ ]:

```
"""BeautifulSoup内容查找示例"""
from bs4 import BeautifulSoup
import re

htmlDoc = """
<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="brother" 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>
<div><!-- This is a comment --></div>
</body>
</html>
"""

bsoup = BeautifulSoup(htmlDoc,'html.parser')

# 以下例子给出了使用find_all函数，以name值为条件查找标签的方法
print([ (_.name,_.string) for _ in bsoup.find_all('a')])
print('---'*10)

print([ (_.name,_.string) for _ in bsoup.find_all(['a','b'])])
print('---'*10)

print([ (_.name,_.string) for _ in bsoup.find_all(True)])
print('---'*10)

print([ (_.name,_.string) for _ in bsoup.find_all(re.compile('b',re.I))])
print('---'*10)

# 以下例子给出了使用find_all函数，以name值、attrs值为条件查找标签的方法
print([ (_.name,_.string) for _ in bsoup.find_all('a','sister')])
```

```
print('---'*10)

# 以下例子给出了使用find_all函数，以id值为条件查找标签的方法
print([ (_.name,_.string) for _ in bsoup.find_all(id = re.compile('link',re.I))])
print('---'*10)
```

实例

实例1 爬取大学排名

上海交通大学设计了一个“最好大学网”，上面列出了当前的大学排名。我们要设计爬虫程序，爬取大学排名信息。

爬虫功能要求：

- 输入：大学排名URL链接
- 输出：大学排名信息的屏幕输出（排名，大学名称，总分）
- 工具：python3、requests、beautifulsoup

程序设计思路：

1. 研究大学排名网站网页URL
2. 设计fetchUrl函数，尝试获取页面；
3. 设计parseHtml函数，解析内容；
4. 设计output函数，组织列表形式输出；
5. 使用main函数调用程序。

In [ ]:

```
"""请同学们自行实现"""
```