第八章 文本数据

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

一、str对象

1. str对象的设计意图

`str` 对象是定义在 `Index` 或 `Series` 上的属性，专门用于逐元素处理文本内容，其内部定义了大量方法，因此对一个序列进行文本处理，首先需要获取其 `str` 对象。在Python标准库中也有 `str` 模块，为了使用上的便利，有许多函数的用法 `pandas` 照搬了它的设计，例如字母转为大写的操作：

```
In [3]: var = 'abcd'

In [4]: str.upper(var) # Python内置str模块
Out[4]: 'ABCD'

In [5]: s = pd.Series(['abcd', 'efg', 'hi'])

In [6]: s.str
Out[6]: <pandas.core.strings.StringMethods at 0x2583db3cac8>

In [7]: s.str.upper() # pandas中str对象上的upper方法
Out[7]:
0    ABCD
1     EFG
2      HI
dtype: object
```

根据文档 `API` 材料，在 `pandas` 的50个 `str` 对象方法中，有31个是和标准库中的 `str` 模块方法同名且功能一致，这为批量处理序列提供了有力的工具。

2. []索引器

对于 `str` 对象而言，可理解为其对字符串进行了序列化的操作，例如在一般的字符串中，通过 `[]` 可以取出某个位置的元素：

```
In [8]: var[0]
Out[8]: 'a'
```

同时也能通过切片得到子串：

```
In [9]: var[-1: 0: -2]
Out[9]: 'db'
```

通过对 `str` 对象使用 `[]` 索引器，可以完成完全一致的功能，并且如果超出范围则返回缺失值：

```
In [10]: s.str[0]
Out[10]:
0    a
1    e
2    h
dtype: object

In [11]: s.str[-1: 0: -2]
Out[11]:
0    db
1     g
2     i
dtype: object

In [12]: s.str[2]
Out[12]:
0      c
1      g
2    NaN
dtype: object
```

3. string类型

在上一章提到，从 `pandas` 的 `1.0.0` 版本开始，引入了 `string` 类型，其引入的动机在于：原来所有的字符串类型都会以 `object` 类型的 `Series` 进行存储，但 `object` 类型只应当存储混合类型，例如同时存储浮点、字符串、字典、列表、自定义类型等，因此字符串有必要同数值型或 category 一样，具有自己的数据存放类型，从而引入了 `string` 类型。

总体上说，绝大多数对于 `object` 和 `string` 类型的序列使用 `str` 对象方法产生的结果是一致，但是在下面提到的两点上有较大差异：

首先，应当尽量保证每一个序列中的值都是字符串的情况下才使用 `str` 属性，但这并不是必须的，其必要条件是序列中至少有一个可迭代（Iterable）对象，包括但不限于字符串、字典、列表。对于一个可迭代对象， `string` 类型的 `str` 对象和 `object` 类型的 `str` 对象返回结果可能是不同的。

```
In [13]: s = pd.Series([{1: 'temp_1', 2: 'temp_2'}, ['a', 'b'], 0.5, 'my_string'])

In [14]: s.str[1]
Out[14]:
0    temp_1
1         b
2       NaN
3         y
dtype: object

In [15]: s.astype('string').str[1]
Out[15]:
0    1
1    '
2    .
3    y
dtype: string
```

除了最后一个字符串元素，前三个元素返回的值都不同，其原因在于当序列类型为 `object` 时，是对于每一个元素进行 `[]` 索引，因此对于字典而言，返回temp_1字符串，对于列表则返回第二个值，而第三个为不可迭代对象，返回缺失值，第四个是对字符串进行 `[]` 索引。而 `string` 类型的 `str` 对象先把整个元素转为字面意义的字符串，例如对于列表而言，第一个元素即“{“，而对于最后一个字符串元素而言，恰好转化前后的表示方法一致，因此结果和 `object` 类型一致。

除了对于某些对象的 `str` 序列化方法不同之外，两者另外的一个差别在于， `string` 类型是 `Nullable` 类型，但 `object` 不是。这意味着 `string` 类型的序列，如果调用的 `str` 方法返回值为整数 `Series` 和布尔 `Series` 时，其分别对应的 `dtype` 是 `Int` 和 `boolean` 的 `Nullable` 类型，而 `object` 类型则会分别返回 `int/float` 和 `bool/object` ，取决于缺失值的存在与否。同时，字符串的比较操作，也具有相似的特性，`string` 返回 `Nullable` 类型，但 `object` 不会。

```
In [16]: s = pd.Series(['a'])

In [17]: s.str.len()
Out[17]:
0    1
dtype: int64

In [18]: s.astype('string').str.len()
Out[18]:
0    1
dtype: Int64

In [19]: s == 'a'
Out[19]:
0    True
dtype: bool

In [20]: s.astype('string') == 'a'
Out[20]:
0    True
dtype: boolean

In [21]: s = pd.Series(['a', np.nan]) # 带有缺失值

In [22]: s.str.len()
Out[22]:
0    1.0
1    NaN
dtype: float64

In [23]: s.astype('string').str.len()
Out[23]:
0       1
1    <NA>
dtype: Int64

In [24]: s == 'a'
Out[24]:
0     True
1    False
dtype: bool

In [25]: s.astype('string') == 'a'
Out[25]:
0    True
1    <NA>
dtype: boolean
```

最后需要注意的是，对于全体元素为数值类型的序列，即使其类型为 `object` 或者 `category` 也不允许直接使用 `str` 属性。如果需要把数字当成 `string` 类型处理，可以使用 `astype` 强制转换为 `string` 类型的 `Series`：

```
In [26]: s = pd.Series([12, 345, 6789])

In [27]: s.astype('string').str[1]
Out[27]:
0    2
1    4
2    7
dtype: string
```

二、正则表达式基础

这一节的两个表格来自于 learn-regex-zh 这个关于正则表达式项目，其使用 `MIT` 开源许可协议。这里只是介绍正则表达式的基本用法，需要系统学习的读者可参考 正则表达式必知必会 一书。

1. 一般字符的匹配

正则表达式是一种按照某种正则模式，从左到右匹配字符串中内容的一种工具。对于一般的字符而言，它可以找到其所在的位置，这里为了演示便利，使用了 `python` 中 `re` 模块的 `findall` 函数来匹配所有出现过但不重叠的模式，第一个参数是正则表达式，第二个参数是待匹配的字符串。例如，在下面的字符串中找出 `apple`：

```
In [28]: import re

In [29]: re.findall('Apple', 'Apple! This Is an Apple!')
Out[29]: ['Apple', 'Apple']
```

2. 元字符基础

元字符	描述
.	匹配除换行符以外的任意字符。
[ ]	字符类，匹配方括号中包含的任意字符。
[^]	否定字符类，匹配方括号中不包含的任意字符
*	匹配前面的子表达式零次或多次
+	匹配前面的子表达式一次或多次
?	匹配前面的子表达式零次或一次
{n,m}	花括号，匹配前面字符至少 n 次，但是不超过 m 次。
(xyz)	字符组，按照确切的顺序匹配字符xyz。
|	分支结构，匹配符号之前的字符或后面的字符。
\	转义符，它可以还原元字符原来的含义
^	匹配行的开始
$	匹配行的结束

```
In [30]: re.findall('.', 'abc')
Out[30]: ['a', 'b', 'c']

In [31]: re.findall('[ac]', 'abc')
Out[31]: ['a', 'c']

In [32]: re.findall('[^ac]', 'abc')
Out[32]: ['b']

In [33]: re.findall('[ab]{2}', 'aaaabbbb') # {n}指匹配n次
Out[33]: ['aa', 'aa', 'bb', 'bb']

In [34]: re.findall('aaa|bbb', 'aaaabbbb')
Out[34]: ['aaa', 'bbb']

In [35]: re.findall('a\\?|a\*', 'aa?a*a')
Out[35]: ['a?', 'a*']

In [36]: re.findall('a?.', 'abaacadaae')
Out[36]: ['ab', 'aa', 'c', 'ad', 'aa', 'e']
```

3. 简写字符集

此外，正则表达式中还有一类简写字符集，其等价于一组字符的集合：

简写	描述
\w	匹配所有字母、数字、下划线: [a-zA-Z0-9_]
\W	匹配非字母和数字的字符: [^\w]
\d	匹配数字: [0-9]

简写	描述
\D	匹配非数字: [^\d]
\s	匹配空格符: [\t\n\f\r\p{Z}]
\S	匹配非空格符: [^\s]
\B	匹配一组非空字符开头或结尾的位置，不代表具体字符

```
In [37]: re.findall('.s', 'Apple! This Is an Apple!')
Out[37]: ['is', 'Is']

In [38]: re.findall('\w{2}', '09 8? 7w c_ 9q p@')
Out[38]: ['09', '7w', 'c_', '9q']

In [39]: re.findall('\w\W\B', '09 8? 7w c_ 9q p@')
Out[39]: ['8?', 'p@']

In [40]: re.findall('.\s.', 'Constant dropping wears the stone.')
Out[40]: ['t d', 'g w', 's t', 'e s']

In [41]: re.findall('上海市(.{2,3}区)(.{2,3}路)(\d+号)',
   ....:            '上海市黄浦区方浜中路249号 上海市宝山区密山路5号')
   ....:
Out[41]: [('黄浦区', '方浜中路', '249号'), ('宝山区', '密山路', '5号')]
```

三、文本处理的五类操作

1. 拆分

`str.split` 能够把字符串的列进行拆分，其中第一个参数为正则表达式，可选参数包括从左到右的最大拆分次数 `n`，是否展开为多个列 `expand`。

```
In [42]: s = pd.Series(['上海市黄浦区方浜中路249号',
   ....:                '上海市宝山区密山路5号'])
   ....:

In [43]: s.str.split('[市区路]')
Out[43]:
0    [上海, 黄浦, 方浜中, 249号]
1       [上海, 宝山, 密山, 5号]
dtype: object

In [44]: s.str.split('[市区路]', n=2, expand=True)
Out[44]:
    0   1         2
0  上海  黄浦  方浜中路249号
1  上海  宝山       密山路5号
```

与其类似的函数是 `str.rsplit`，其区别在于使用 `n` 参数的时候是从右到左限制最大拆分次数：

```
In [45]: s.str.rsplit('[市区路]', n=2, expand=True)
Out[45]:
                0
0  上海市黄浦区方浜中路249号
1      上海市宝山区密山路5号
```

2. 合并

关于合并一共有两个函数，分别是 `str.join` 和 `str.cat`。 `str.join` 表示用某个连接符把 `Series` 中的字符串列表连接起来，如果列表中出现了字符串元素则返回缺失值：

```
In [46]: s = pd.Series([['a','b'], [1, 'a'], [['a', 'b'], 'c']])

In [47]: s.str.join('-')
Out[47]:
0    a-b
1    NaN
2    NaN
dtype: object
```

`str.cat` 用于合并两个序列，主要参数为连接符 `sep` 、连接形式 `join`` 以及缺失值替代符号 ``na_rep` ，其中连接形式默认为以索引为键的左连接。

```
In [48]: s1 = pd.Series(['a','b'])

In [49]: s2 = pd.Series(['cat','dog'])

In [50]: s1.str.cat(s2,sep='-')
Out[50]:
0    a-cat
1    b-dog
dtype: object

In [51]: s2.index = [1, 2]

In [52]: s1.str.cat(s2, sep='-', na_rep='?', join='outer')
Out[52]:
0      a-?
1    b-cat
2    ?-dog
dtype: object
```

3. 匹配

`str.contains` 返回了每个字符串是否包含正则模式的布尔序列：

```
In [53]: s = pd.Series(['my cat', 'he is fat', 'railway station'])

In [54]: s.str.contains('\s\wat')
Out[54]:
0     True
1     True
2    False
dtype: bool
```

`str.startswith` 和 `str.endswith` 返回了每个字符串以给定模式为开始和结束的布尔序列，它们都不支持正则表达式：

```
In [55]: s.str.startswith('my')
Out[55]:
0     True
1    False
2    False
dtype: bool

In [56]: s.str.endswith('t')
Out[56]:
0     True
1     True
2    False
dtype: bool
```

如果需要用正则表达式来检测开始或结束字符串的模式，可以使用 `str.match` ，其返回了每个字符串起始处是否符合给定正则模式的布尔序列：

```
In [57]: s.str.match('m|h')
Out[57]:
0     True
1     True
2    False
dtype: bool

In [58]: s.str[::-1].str.match('ta[f|g]|n') # 反转后匹配
Out[58]:
0    False
1     True
2     True
dtype: bool
```

当然，这些也能通过在 `str.contains` 的正则中使用 `^` 和 `$` 来实现：

```
In [59]: s.str.contains('^[m|h]')
Out[59]:
0     True
1     True
2    False
dtype: bool

In [60]: s.str.contains('[f|g]at|n$')
Out[60]:
0    False
1     True
2     True
dtype: bool
```

除了上述返回值为布尔的匹配之外，还有一种返回索引的匹配函数，即 `str.find` 与 `str.rfind`，其分别返回从左到右和从右到左第一次匹配的位置的索引，未找到则返回-1。需要注意的是这两个函数不支持正则匹配，只能用于字符子串的匹配：

```
In [61]: s = pd.Series(['This is an apple. That is not an apple.'])

In [62]: s.str.find('apple')
Out[62]:
0    11
dtype: int64

In [63]: s.str.rfind('apple')
Out[63]:
0    33
dtype: int64
```

4. 替换

`str.replace` 和 `replace` 并不是一个函数，在使用字符串替换时应当使用前者。

```
In [64]: s = pd.Series(['a_1_b','c_?'])

In [65]: s.str.replace('\d|\?', 'new')
Out[65]:
0    a_new_b
1      c_new
dtype: object
```

当需要对不同部分进行有差别的替换时，可以利用 子组 的方法，并且此时可以通过传入自定义的替换函数来分别进行处理，注意 `group(k)` 代表匹配到的第 `k` 个子组（圆括号之间的内容）：

```
In [66]: s = pd.Series(['上海市黄浦区方浜中路249号',
    ....:               '上海市宝山区密山路5号',
    ....:               '北京市昌平区北农路2号'])
    ....:

In [67]: pat = '(\w+市)(\w+区)(\w+路)(\d+号)'

In [68]: city = {'上海市': 'Shanghai', '北京市': 'Beijing'}

In [69]: district = {'昌平区': 'CP District',
    ....:               '黄浦区': 'HP District',
    ....:               '宝山区': 'BS District'}
    ....:

In [70]: road = {'方浜中路': 'Mid Fangbin Road',
    ....:          '密山路': 'Mishan Road',
    ....:          '北农路': 'Beinong Road'}
    ....:

In [71]: def my_func(m):
    ....:     str_city = city[m.group(1)]
    ....:     str_district = district[m.group(2)]
    ....:     str_road = road[m.group(3)]
    ....:     str_no = 'No. ' + m.group(4)[:-1]
    ....:     return ' '.join([str_city,
    ....:                      str_district,
    ....:                      str_road,
    ....:                      str_no])
    ....:

In [72]: s.str.replace(pat, my_func)
Out[72]:
0    Shanghai HP District Mid Fangbin Road No. 249
1           Shanghai BS District Mishan Road No. 5
2           Beijing CP District Beinong Road No. 2
dtype: object
```

这里的数字标识并不直观，可以使用 命名子组 更加清晰地写出子组代表的含义：

```
In [73]: pat = '(?P<市名>\w+市)(?P<区名>\w+区)(?P<路名>\w+路)(?P<编号>\d+号)'

In [74]: def my_func(m):
    ....:     str_city = city[m.group('市名')]
    ....:     str_district = district[m.group('区名')]
    ....:     str_road = road[m.group('路名')]
    ....:     str_no = 'No. ' + m.group('编号')[:-1]
    ....:     return ' '.join([str_city,
    ....:                      str_district,
    ....:                      str_road,
    ....:                      str_no])
    ....:

In [75]: s.str.replace(pat, my_func)
Out[75]:
0    Shanghai HP District Mid Fangbin Road No. 249
1           Shanghai BS District Mishan Road No. 5
2           Beijing CP District Beinong Road No. 2
dtype: object
```

这里虽然看起来有些繁杂，但是实际数据处理中对应的替换，一般都会通过代码来获取数据从而构造字典映射，在具体写法上会简洁的多。

5. 提取

提取既可以认为是一种返回具体元素（而不是布尔值或元素对应的索引位置）的匹配操作，也可以认为是一种特殊的拆分操作。前面提到的 `str.split` 例子中会把分隔符去除，这并不是用户想要的效果，这时候就可以用 `str.extract` 进行提取：

```
In [76]: pat = '(\w+市)(\w+区)(\w+路)(\d+号)'

In [77]: s.str.extract(pat)
Out[77]:
     0    1     2      3
0  上海市  黄浦区  方浜中路  249号
1  上海市  宝山区   密山路    5号
2  北京市  昌平区   北农路    2号
```

通过子组的命名，可以直接对新生成 `DataFrame` 的列命名：

```
In [78]: pat = '(?P<市名>\w+市)(?P<区名>\w+区)(?P<路名>\w+路)(?P<编号>\d+号)'

In [79]: s.str.extract(pat)
Out[79]:
    市名    区名     路名     编号
0  上海市  黄浦区  方浜中路  249号
1  上海市  宝山区   密山路    5号
2  北京市  昌平区   北农路    2号
```

str.extractall 不同于 str.extract 只匹配一次，它会把所有符合条件的模式全部匹配出来，如果存在多个结果，则以多级索引的方式存储：

```
In [80]: s = pd.Series(['A135T15,A26S5','B674S2,B25T6'], index = ['my_A','my_B'])

In [81]: pat = '[A|B](\d+)[T|S](\d+)'

In [82]: s.str.extractall(pat)
Out[82]:
              0   1
    match
my_A 0      135  15
     1       26   5
my_B 0      674   2
     1       25   6

In [83]: pat_with_name = '[A|B](?P<name1>\d+)[T|S](?P<name2>\d+)'

In [84]: s.str.extractall(pat_with_name)
Out[84]:
           name1 name2
    match
my_A 0       135    15
     1        26     5
my_B 0       674     2
     1        25     6
```

str.findall 的功能类似于 str.extractall ，区别在于前者把结果存入列表中，而后者处理为多级索引，每个行只对应一组匹配，而不是把所有匹配组合构成列表。

```
In [85]: s.str.findall(pat)
Out[85]:
my_A    [(135, 15), (26, 5)]
my_B     [(674, 2), (25, 6)]
dtype: object
```

四、常用字符串函数

除了上述介绍的五类字符串操作有关的函数之外， str 对象上还定义了一些实用的其他方法，在此进行介绍：

1. 字母型函数

upper, lower, title, capitalize, swapcase 这五个函数主要用于字母的大小写转化，从下面的例子中就容易领会其功能：

```
In [86]: s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])

In [87]: s.str.upper()
Out[87]:
0                 LOWER
1             CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object

In [88]: s.str.lower()
Out[88]:
0                 lower
1             capitals
2    this is a sentence
3              swapcase
dtype: object

In [89]: s.str.title()
Out[89]:
0                 Lower
1             Capitals
2    This Is A Sentence
3              Swapcase
dtype: object

In [90]: s.str.capitalize()
Out[90]:
0                 Lower
1             Capitals
2    This is a sentence
3              Swapcase
dtype: object

In [91]: s.str.swapcase()
Out[91]:
0                 LOWER
1             capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

2. 数值型函数

这里着重需要介绍的是 `pd.to_numeric` 方法，它虽然不是 `str` 对象上的方法，但是能够对字符格式的数值进行快速转换和筛选。其主要参数包括 `errors` 和 `downcast` 分别代表了非数值的处理模式和转换类型。其中，对于不能转换为数值的有三种 `errors` 选项， `raise, coerce, ignore` 分别表示直接报错、设为缺失以及保持原来的字符串。

```
In [92]: s = pd.Series(['1', '2.2', '2e', '??', '-2.1', '0'])

In [93]: pd.to_numeric(s, errors='ignore')
Out[93]:
0      1
1    2.2
2     2e
3     ??
4   -2.1
5      0
dtype: object

In [94]: pd.to_numeric(s, errors='coerce')
Out[94]:
0    1.0
1    2.2
2    NaN
3    NaN
4   -2.1
5    0.0
dtype: float64
```

在数据清洗时，可以利用 `coerce` 的设定，快速查看非数值型的行：

```
In [95]: s[pd.to_numeric(s, errors='coerce').isna()]
Out[95]:
2    2e
3    ??
dtype: object
```

3. 统计型函数

`count` 和 `len` 的作用分别是返回出现正则模式的次数和字符串的长度：

```
In [96]: s = pd.Series(['cat rat fat at', 'get feed sheet heat'])

In [97]: s.str.count('[r|f]at|ee')
Out[97]:
0    2
1    2
dtype: int64

In [98]: s.str.len()
Out[98]:
0    14
1    19
dtype: int64
```

4. 格式型函数

格式型函数主要分为两类，第一种是除空型，第二种时填充型。其中，第一类函数一共有三种，它们分别是 `strip, rstrip, lstrip`，分别代表去除两侧空格、右侧空格和左侧空格。这些函数在数据清洗时是有用的，特别是列名含有非法空格的时候。

```
In [99]: my_index = pd.Index([' col1', 'col2 ', ' col3 '])

In [100]: my_index.str.strip().str.len()
Out[100]: Int64Index([4, 4, 4], dtype='int64')

In [101]: my_index.str.rstrip().str.len()
Out[101]: Int64Index([5, 4, 5], dtype='int64')

In [102]: my_index.str.lstrip().str.len()
Out[102]: Int64Index([4, 5, 5], dtype='int64')
```

对于填充型函数而言，`pad` 是最灵活的，它可以选定字符串长度、填充的方向和填充内容：

```
In [103]: s = pd.Series(['a','b','c'])

In [104]: s.str.pad(5,'left','*')
Out[104]:
0    ****a
1    ****b
2    ****c
dtype: object

In [105]: s.str.pad(5,'right','*')
Out[105]:
0    a****
1    b****
2    c****
dtype: object

In [106]: s.str.pad(5,'both','*')
Out[106]:
0    **a**
1    **b**
2    **c**
dtype: object
```

上述的三种情况可以分别用 `rjust, ljust, center` 来等效完成，需要注意 `ljust` 是指右侧填充而不是左侧填充：

```
In [107]: s.str.rjust(5, '*')
Out[107]:
0    ****a
1    ****b
2    ****c
dtype: object

In [108]: s.str.ljust(5, '*')
Out[108]:
0    a****
1    b****
2    c****
dtype: object

In [109]: s.str.center(5, '*')
Out[109]:
0    **a**
1    **b**
2    **c**
dtype: object
```

在读取 `excel` 文件时，经常会出现数字前补0的需求，例如证券代码读入的时候会把”000007”作为数值7来处理， `pandas` 中除了可以使用上面的左侧填充函数进行操作之外，还可用 `zfill` 来实现。

```
In [110]: s = pd.Series([7, 155, 303000]).astype('string')

In [111]: s.str.pad(6,'left','0')
Out[111]:
0    000007
1    000155
2    303000
dtype: string

In [112]: s.str.rjust(6,'0')
Out[112]:
0    000007
1    000155
2    303000
dtype: string

In [113]: s.str.zfill(6)
Out[113]:
0    000007
1    000155
2    303000
dtype: string
```