Vectorized String Operations

向量化的字符串操作

> One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of *vectorized string operations* that become an essential piece of the type of munging required when working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

Python的一个强大的特点就是它能相对简单的处理和操作字符串数据。Pandas在此基础上提供了一整套*向量化字符串操作*，这成为了当我们处理（清洗）真实世界数据时非常关键的功能。在本节中，我们将对很多Pandas的字符串操作进行介绍，然后看它们在我们对从互联网采集到的非常不规范的数据集进行清洗时发挥的作用。

Introducing Pandas String Operations

Pandas字符串操作介绍

> We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

在前面章节中我们看到NumPy和Pandas的工具能向量化算术运算，让我们可以很容易和快速的对数组的元素进行相同的数学计算。例如：

In [1]:
```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
```

Out[1]:
```
array([ 4,  6, 10, 14, 22, 26])
```

> This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

这种*向量化*的操作能简化数组元素的操作语法：我们不再需要担心数组的大小和形状，只需要关注于需要进行的运算本身。对于字符串数组，NumPy没有提供这种简单的操作，因此你需要继续使用循环语法来处理：

In [2]:
```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

Out[2]:
```
['Peter', 'Paul', 'Mary', 'Guido']
```

> This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

这可能对于一些数据集来说足够了，但是对于含有缺失值的数据集来说就出问题了。例如：

In [3]:
```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-3b0264c38d59> in <module>
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

<ipython-input-3-3b0264c38d59> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

> Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

Pandas包含了前面说到的向量化的字符串操作，而且还能正确的处理缺失值，这可以通过Pandas的Series和Index对象的`str`属性来实现。例如，假设我们如下创建一个Pandas Series：

In [4]:
```
import pandas as pd
names = pd.Series(data)
names
```

Out[4]:
```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

> We can now call a single method that will capitalize all the entries, while skipping over any missing values:

我们现在可以调用一个方法就能将所有元素首字母变大写的功能，并能跳过缺失值：

In [5]:
```
names.str.capitalize()
```

Out[5]:
```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

> Using tab completion on this `str` attribute will list all the vectorized string methods available to Pandas.

在IPython中在`str`属性上使用制表符自动补全功能可以列出Pandas中支持的所有的向量化字符串操作。

Tables of Pandas String Methods

Pandas字符串方法列表

> If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

如果你已经很好理解了Python中的字符串操作，大多数的Pandas字符串操作语法是很直观的，因此简单的列一张表格说明所有可用的方法就足够理解了；我们在深入到一些细节之前可以先浏览这张表。本节的例子将会使用下面的一个姓名的Series对象：

In [6]:
```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods similar to Python string methods

类似Python的字符串方法

> Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

几乎所有Python内建的字符串方法都有Pandas的向量化版本。下面是Pandas的`str`属性中与Python内建字符串方法一致的方法：

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

> Notice that these have various return values. Some, like `lower()`, return a series of strings:

要提醒的是，这些方法与内建字符串方法可能有着不同的返回值，如`lower()`返回的是一个字符串的Series对象：

In [7]:
```
monte.str.lower()
```

Out[7]:
```
0    graham chapman
1       john cleese
2     terry gilliam
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

> But some others return numbers:

另外一些返回的是数字的Series对象：

In [8]:
```
monte.str.len()
```

Out[8]:
```
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

> Or Boolean values:

或布尔值的Series对象：

In [9]:
```
monte.str.startswith('T')
```

Out[9]:
```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

> Still others return lists or other compound values for each element:

还有一些会返回诸如列表那样的复合类型的Series对象：

In [10]:
```
monte.str.split()
```

Out[10]:
```
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

> We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

我们后面会讨论到如何操作这种列表组成的Series对象。

Methods using regular expressions

使用正则表达式的方法

> In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in `re` module:

除此之外，还有一些方法可以接受正则表达式来检查每个元素字符串是否匹配模式，它们遵从Python内建的`re`模块的API规范：

方法	描述
`match()`	在每个元素上调用`re.match()`方法，返回布尔类型Series
`extract()`	在每个元素上调用`re.match()`方法，返回匹配到模式的正则分组的Series
`findall()`	在每个元素上调用`re.findall()`方法
`replace()`	将匹配模式的字符串部分替换成其他字符串值
`contains()`	在每个元素上调用`re.search()`，返回布尔类型Series
`count()`	计算匹配到模式的次数
`split()`	等同于`str.split()`，但是能接受正则表达式参数
`rsplit()`	等同于`str.rsplit()`，但是能接受正则表达式参数

> With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

使用上面的方法，你可以执行很多有趣的操作。例如，我们可以通过匹配连续的一组字母的模式从姓名中提取出名字：

In [11]:
```
monte.str.extract('([A-Za-z]+)', expand=False)
```

Out[11]:
```
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
```

> Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (`^`) and end-of-string (`$`) regular expression characters:

或者我们可以执行更复杂的操作，如找出所有姓名中首字母和尾字母都是辅音字母的人，这里需要使用字符串开始位置（`^`）和字符串结束位置（`$`）正则表达式特殊符号：

In [12]:
```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

Out[12]:
```
0    [Graham Chapman]
1                  []
2     [Terry Gilliam]
3                  []
4       [Terry Jones]
5     [Michael Palin]
dtype: object
```

> The ability to concisely apply regular expressions across `Series` or `Dataframe` entries opens up many possibilities for analysis and cleaning of data.

这种在`Series`或`DataFrame`上简洁的应用正则表达式的特性，在清洗和分析数据任务中非常有用。

Miscellaneous methods

其他方法

> Finally, there are some miscellaneous methods that enable other convenient operations:

最后，下面是一些无法分类的其他方法但也是很方便的字符串功能：

方法	描述
`get()`	对每个元素使用索引值获取字符中的字符
`slice()`	对每个元素进行字符串切片
`slice_replace()`	将每个元素的字符串切片替换成另一个字符串值
`cat()`	将所有字符串元素连接成一个字符串
`repeat()`	对每个字符串元素进行重复操作
`normalize()`	返回字符串的unicode标准化结果
`pad()`	字符串对齐
`wrap()`	字符串换行
`join()`	字符串中字符的连接
`get_dummies()`	将字符串按照分隔符分割后形成一个二维的dummy DataFrame

Vectorized item access and slicing

向量化的索引和切片操作

> The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python's normal indexing syntax–for example, `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

`get()`和`slice()`操作，可以对每个字符串元素进行索引访问和切片的操作。例如，我们可以通过`str.slice(0, 3)`获取每个字符串元素的前三个字母。还需要说明的是，这个操作也可以通过Python标准的切片语法来完成，也就是说`df.str[:3]`等同于`df.str.slice(0, 3)`：

In [13]:
```
monte.str[:3]
```

Out[13]:
```
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

> Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar.

索引取值操作也是一样，`df.str[i]`等同于`df.str.get(i)`。

> These `get()` and `slice()` methods also let you access elements of arrays returned by `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

`get()`和`slice()`方法还能支持对`split()`返回的列表进行取值操作。例如我们使用`split()`和`get()`方法可以提取出每个人的姓：

In [14]:
```
monte.str.split().str.get(-1)
```

Out[14]:
```
0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

Indicator variables

指示器变量

> Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing some sort of coded indicator. For example, we might have a dataset that contains information in the form of codes, such as A="born in America," B="born in the United Kingdom," C="likes cheese," D="likes spam":

还有一个需要进行说明的方法是`get_dummies()`。这个方法在你的数据中含有某种编码的指示器的时候非常有用。例如，我们有一个数据集，里面有一个编码了的列，A代表“出生在美国”，B代表“出生在英国”，C代表“喜欢芝士”，D代表“喜欢肉罐头”：

In [15]:
```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
full_monte
```

Out[15]:

	name	info
0	Graham Chapman	B|C|D
1	John Cleese	B|D
2	Terry Gilliam	A|C
3	Eric Idle	B|D
4	Terry Jones	B|C
5	Michael Palin	B|C|D

> The `get_dummies()` routine lets you quickly split-out these indicator variables into a `DataFrame`:

`get_dummies()`方法能让你快速的将这些编码的指示器变量分解出来，并形成一个`DataFrame`：

In [16]:
```
full_monte['info'].str.get_dummies('|')
```

Out[16]:

	A	B	C	D
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

> With these operations as building blocks, you can construct an endless range of string processing procedures when cleaning your data.

有了上述的这些向量化字符串方法，你可以在清洗数据时构建无穷无尽的字符串处理流程。

> We won't dive further into these methods here, but I encourage you to read through "Working with Text Data" in the Pandas online documentation, or to refer to the resources listed in Further Resources.

在这里我们不在深入介绍每个方法，作者推荐你去阅读Pandas的在线文档处理文本数据，或者参考更多资源中的其他资料。

Example: Recipe Database

例子：菜谱数据库

> These vectorized string operations become most useful in the process of cleaning up messy, real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the Web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

上述介绍的这些向量化字符串操作是我们对不规范的真实世界数据进行清洗的最有效工具。下面我们将使用网络中收集的一个菜谱数据库作为例子来总体说明。我们的目标是将这些菜谱数据解析成配方的列表，这样我们就能很快速的根据我们手头的材料找到相应配方的菜谱。

> The scripts used to compile this can be found at https://github.com/fictivekin/openrecipes, and the link to the current version of the database is found there as well.

用来收集菜谱的脚本可以在https://github.com/fictivekin/openrecipes 这里找到，最新版本的菜谱数据库的连接也在这个页面上。

> As of Spring 2016, this database is about 30 MB, and can be downloaded and unzipped with these commands:

到2016年春天，这个数据库有大约30MB大小，可以通过下面的命令下载以及解压：

译者注：open recipes数据库在2017年后已经无法使用原作者的地址进行下载，参见Issue#179。使用新的地址可以正确下载数据库内容。下述shell代码也使用新地址进行了修改。

In [17]:
```
# !curl -O https://s3.amazonaws.com/openrecipes/20170107-061401-recipeitems.json.gz
# !gunzip 20170107-061401-recipeitems.json.gz
```

> The database is in JSON format, so we will try `pd.read_json` to read it:

数据库是JSON格式，因此我们需要使用`pd.read_json`方法来读取它：

In [18]:
```
try:
    recipes = pd.read_json('data/20170107-061401-recipeitems.json')
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Trailing data
```

> Oops! We get a `ValueError` mentioning that there is "trailing data." Searching for the text of this error on the Internet, it seems that it's due to using a file in which *each line* is itself a valid JSON, but the full file is not. Let's check if this interpretation is true:

喔噢，这里会产生一个`ValueError`指出有冗余的数据。通过在网上搜索这个错误信息，我们得到原因是这个文件*每一行*都是一个正确的JSON，但是整个文件不是正确的JSON格式。我们来验证一下：

In [19]:
```
with open('data/20170107-061401-recipeitems.json') as f:
    line = f.readline()
pd.read_json(line).shape
```

Out[19]:
```
(2, 12)
```

> Yes, apparently each line is a valid JSON, so we'll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with `pd.read_json`:

通过读取文件一行我们验证了我们的想法，现在我们需要将这些正确的JSON行合并在一起。实现这个目标的一种方式就是我们手动将所有的行合并成一个JSON Array，然后将这个JSON Array的字符串传递到`pd.read_json`来进行解析：

In [20]:
```
# 将每一行JSON对象合并成一整个JSON Array
with open('data/20170107-061401-recipeitems.json', 'r') as f:
    # 提取每一行
    data = (line.strip() for line in f)
    # 每两个对象之间用,分隔，最后首尾加上中括号表示数组
    data_json = "[{0}]".format(','.join(data))
# 将结果字符串作为JSON格式读取到Pandas中
recipes = pd.read_json(data_json)
```

In [21]:
```
recipes.shape
```

Out[21]:
```
(173278, 17)
```

> We see there are nearly 200,000 recipes, and 17 columns. Let's take a look at one row to see what we have:

从形状可知有接近20万个菜谱，每个菜谱有17列数据。看看其中的一行：

In [22]:
```
recipes.iloc[0]
```

Out[22]:
```
_id                                {'$oid': '5160756b96cc62079cc2db15'}
cookTime                                                           PT30M
creator                                                              NaN
dateModified                                                         NaN
datePublished                                                 2013-03-11
description           Late Saturday afternoon, after Marlboro Man ha...
image                 http://static.thepioneerwoman.com/cooking/file...
ingredients           Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
name                                   Drop Biscuits and Sausage Gravy
prepTime                                                           PT10M
recipeCategory                                                       NaN
recipeInstructions                                                   NaN
recipeYield                                                           12
source                                                   thepioneerwoman
totalTime                                                            NaN
ts                                          {'$date': 1365276011104}
url                   http://thepioneerwoman.com/cooking/2013/03/dro...
Name: 0, dtype: object
```

> There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the Web. In particular, the ingredient list is in string format; we're going to have to carefully extract the information we're interested in. Let's start by taking a closer look at the ingredients:

结果中有很多的数据，但大多数列的数据都是混乱不堪的，正如所有从网络中爬取的数据一样。特别注意到，配方列表是一个字符串的格式，因此我们需要特别小心的在我们感兴趣的列中进行数据提取操作。让我们大致看一些配方列的基本情况：

In [23]:
```
recipes.ingredients.str.len().describe()
```

Out[23]:
```
count    173278.000000
mean        244.617926
std         146.705285
min           0.000000
25%         147.000000
50%         221.000000
75%         314.000000
max        9067.000000
Name: ingredients, dtype: float64
```

> The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10,000 characters!

配方的列表平均有250个字符长，最短的是0个字符，而最长的能达到接近10000个字符。

> Just out of curiousity, let's see which recipe has the longest ingredient list:

为了满足一下好奇心，让我们看看那个菜谱有着最长的配方列表：

译者注：在Series上使用`argmax()`方法或者`np.argmax(series)`函数会出现下面的警告，原因后续Pandas版本会修改它们的行为，目前版本这两种方法返回的是最大值所在的行，后续会修改为返回行号的最大值。建议使用`Series.idxmax()`方法替代。译者后续增加了使用`idxmax()`方法的版本。

In [24]:
```
recipes.name[np.argmax(recipes.ingredients.str.len())]
```

```
/home/wangy/anaconda3/lib/python3.7/site-packages/numpy/core/fromnumeric.py:56: FutureWarning:
The current behaviour of 'Series.argmax' is deprecated, use 'idxmax'
instead.
The behavior of 'argmax' will be corrected to return the positional
maximum in the future. For now, use 'series.values.argmax' or
'np.argmax(np.array(values))' to get the position of the maximum
row.
  return getattr(obj, method)(*args, **kwds)
```

Out[24]:
```
'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp; Cream Cheese Frosting and M
arzipan Carrots'
```

In [25]:
```
recipes.name[recipes.ingredients.str.len().idxmax()]
```

Out[25]:
```
'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp; Cream Cheese Frosting and M
arzipan Carrots'
```

> That certainly looks like an involved recipe.

这个菜谱看起来就很复杂的样子。

> We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

我们可以继续研究一下这个数据集，例如，看看有多少种早餐的菜谱：

In [26]:
```
recipes.description.str.contains('[Bb]reakfast').sum()
```

Out[26]:
```
3524
```

> Or how many of the recipes list cinnamon as an ingredient:

或者多少中菜谱中用到了肉桂做原料：

In [27]:
```
recipes.ingredients.str.contains('[Cc]innamon').sum()
```

Out[27]:
```
10526
```

> We could even look to see whether any recipes misspell the ingredient as "cinamon":

我们甚至可以找到有多少种菜谱将肉桂拼写成了“肉挂”（cinamon)：

In [28]:
```
recipes.ingredients.str.contains('[Cc]inamon').sum()
```

Out[28]:
```
11
```

> This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

这些类型的基础数据分析工作，都可以通过Pandas的字符串工具进行并获得结果。这正是Python在数据科学领域优于其他语言的地方。

A simple recipe recommender

一个简单的菜单推荐器

> Let's go a bit further, and start working on a simple recipe recommendation system: given a list of ingredients, find a recipe that uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

我们再深入的研究一点，来试试实现一个简单的菜谱推荐系统：给定一系列的原材料组成的配方，找到应用了所有原料的菜谱。虽然看起来很容易，实际上这个任务的难点在于数据的异构性：即无法找到一个简单的操作，能从每一行中提取出干净的原料列表。因此我们来做个弊：我们构建一个很通用的原料列表，然后在每个菜谱的配方中搜索它们是否存在。为简单起见，首先从草药和香料开始：

In [29]:
```
spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
              'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

> We can then build a Boolean `DataFrame` consisting of True and False values, indicating whether this ingredient appears in the list:

然后就能创建一个布尔`DataFrame`，显示上述列表的原料是否在每个菜谱中存在：

In [30]:
```
import re
spice_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spice, re.IGNORECASE))
                             for spice in spice_list))
spice_df.head()
```

Out[30]:

	salt	pepper	oregano	sage	parsley	rosemary	tarragon	thyme	paprika	cumin
0	False	False	False	True	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	True	True	False	False	False	False	False	False	False	True
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False

> Now, as an example, let's say we'd like to find a recipe that uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of `DataFrame`s, discussed in High-Performance Pandas: `eval()` and `query()`:

有了这个布尔类型的数据集，比方说如果我们想要找到一个配方使用了香菜、辣椒和龙蒿，那么我们就可以很方便的使用`DataFrame`的`query()`方法来找到这些配方（`query`方法详见高性能Pandas: eval() 和 query()）：

In [31]:
```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
```

Out[31]:
```
10
```

> We find only 10 recipes with this combination; let's use the index returned by this selection to discover the names of the recipes that have this combination:

这种原料组合的菜谱只有10种；我们可以用结果的行索引将这些菜谱的名称筛选出来：

In [32]:
```
recipes.name[selection.index]
```

Out[32]:
```
2069      All cremat with a Little Gem, dandelion and wa...
74964                         Lobster with Thermidor butter
93768      Burton's Southern Fried Chicken with White Gravy
113926                     Mijo's Slow Cooker Shredded Beef
137686                     Asparagus Soup with Poached Eggs
140530                                 Fried Oyster Po’boys
158475                Lamb shank tagine with herb tabbouleh
158486                 Southern fried chicken in buttermilk
163175              Fried Chicken Sliders with Pickles + Slaw
165243                        Bar Tartine Cauliflower Salad
Name: name, dtype: object
```

> Now that we have narrowed down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what we'd like to cook for dinner.

现在我们就已经将可选择的菜谱范围缩小了将近两万倍，我们可以很容易的对晚餐的菜谱做出最后的选择。

Going further with recipes

更加深入的分析菜谱数据库

> Hopefully this example has given you a bit of a flavor (ba-dum!) for the types of data cleaning operations that are efficiently enabled by Pandas string methods. Of course, building a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately, the wide variety of formats used makes this a relatively time-consuming process. This points to the truism that in data science, cleaning and munging of real-world data often comprises the majority of the work, and Pandas provides the tools that can help you do this efficiently.

希望上面的推荐器例子能够让你认识到了使用Pandas的字符串方法我们可以对数据进行异常方便的清洗操作。当然如果希望构建一个成熟的菜谱推荐系统的话，需要比上例*复杂的多*的技巧和工程。将每个菜谱中的原料配方提取出来变成一个列表会是其中很重要的一环；不幸的是，因为数据格式的多样性，这项任务会相对很耗时。这阐述了数据科学中的一个事实，那就是清洗和预处理真实世界的数据是这个领域非常主要的工作之一，Pandas提供了一些工具能帮助你很有效率的完成它。