< 组合数据集：Concat 和 Append | 目录 | 聚合与分组 >

Open in Colab

Combining Datasets: Merge and Join

组合数据集：Merge 和 Join

> One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

Pandas提供的一个基本的特性就是它的高性能、内存中进行的联表和组合操作。如果你使用过数据库，你应该已经很熟悉相关的数据操作了。Pandas在这方面提供的主要接口是`pd.merge`函数，本节中我们会看到一些具体实践的例子。

> For convenience, we will start by redefining the `display()` functionality from the previous section:

为了方便起见，我们重新定义上一节中的`display()`类，用来展示多个数据集：

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

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Relational Algebra

关系代数

> The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

`pd.merge()`实现的是我们称为*关系代数*的一个子集，关系代数是一系列操作关系数据的规则的集合，它构成了大部分数据库的数学基础。关系代数的力量表现在它仅提出了几个基本的运算，这些基本运算成为了更多复杂运算的组成模块，能够应用到任何的数据集上。只要在数据库或者其他程序中实现了这些最基本的运算，那么绝大部分的复杂组合运算都可以在上面实现。

> Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of `Series` and `Dataframe` s. As we will see, these let you efficiently link data from different sources.

Pandas在`pd.merge()`函数中实现了一些上述所说的最基本的运算，`Series`和`DataFrame`的`join`方法也实现了这部分基本运算，你将会看到，这能让你很高效地从不同数据源组合数据。

Categories of Joins

联表的分类

> The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

`pd.merge()`函数实现了几种不同类型的联表：*一对一*、*多对一*和*多对多*。所有三种类型的联表都可以通过`pd.merge()`函数调用来实现；具体使用了哪种类型的联表取决于输入数据的格式。下面我们会展示一些简单的例子来说明三种联表类型，然后我们还会详细的讨论它们的选项。

One-to-one joins

一对一

> Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in Combining Datasets: Concat & Append. As a concrete example, consider the following two `DataFrames` which contain information on several employees in a company:

也许最简单的联表操作类型就是一对一连接，在很多方面，这种联表都和我们在组合数据集：Concat 和 Append中看到的按列进行数据集连接很相似。下面定义两个`DataFrame`含有公司的一些员工信息作为一个具体的例子来说明：

In [2]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

Out[2]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

> To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

要将这两个数据集组合成一个`DataFrame`，我们可以使用`pd.merge`函数：

In [3]:
```
df3 = pd.merge(df1, df2)
df3
```

Out[3]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

> The `pd.merge()` function recognizes that each `DataFrame` has an "employee" column, and automatically joins using this column as a key. The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the "employee" column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see the `left_index` and `right_index` keywords, discussed momentarily).

`pd.merge()`函数会自动识别每个`DataFrame`都有"employee"列，因此会自动按照这个列作为键对双方进行合并。合并的结果是一个新的`DataFrame`，其中的数据是两个输入数据集的联合。再注意到每个列的排列顺序在结果中并不一定保持了：在这个情况下，"employee"列的顺序在`df1`和`df2`中是不同的，而`pd.merge()`函数也正确的考虑到了这点。而且，要知道的是，合并的结果通常会丢弃了原本的行索引标签，除非在合并时制定了行索引（参见我们马上会讨论到的`left_index`和`right_index`参数）。

Many-to-one joins

多对一

> Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

多对一联表的情况发生在两个数据集的关键字列上的其中一个含有重复数据的时候。在这种多对一的情况下，结果的`DataFrame`会正确的保留那些重复的键值。看下面这个例子：

In [4]:
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

Out[4]:

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

> The resulting `DataFrame` has an aditional column with the "supervisor" information, where the information is repeated in one or more locations as required by the inputs.

结果的`DataFrame`多了一列`supervisor`，上面的数据也是按照`group`的重复情况进行重复的。

Many-to-many joins

多对多

> Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a `DataFrame` showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

多对多联表在概念上有一点混乱，但实际上良好定义了的。如果左右的数据集在关键字列上都有重复数据，那么结果就是一个多对多的组合。当然用一个具体的例子来说明是很有帮助的。比如下面的数据集`df5`存储的是一个岗位和其对应的技能。进行了多对多联表后，我们可以获得每个员工对应的技能表：

In [5]:
```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

Out[5]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

> These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

这三种类型的连接可以和Pandas的其他工具联合使用，来实现很强大的功能。但是在实践中，数据集极少好像我们上面的例子那样干净。在接下来的部分，我们会介绍`pd.merge()`提供的一些参数，能让你精细的对连接操作进行调整。

Specification of the Merge Key

指定合并关键字

> We've already seen the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

上面我们看到`pd.merge()`的默认行为：它会在两个输入数据集中寻找一个或多个相同的列名，然后使用这（些）列作为合并的关键字。然而，通常情况下，列名并不会这么匹配，`pd.merge()`提供了一系列的参数来处理这种情况。

The `on` keyword

`on` 关键字参数

> Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

最简单的，你可以使用`on`关键字参数明确指定合并使用的关键字列名，参数可以是一个列名或者一个列名的列表：

In [6]:
```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

Out[6]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

> This option works only if both the left and right `DataFrame` s have the specified column name.

该参数仅在左右两个`DataFrame`都含有相同的指定列名的情况下有效。

The `left_on` and `right_on` keywords

`left_on` 和 `right_on` 关键字参数

> At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

在你希望使用不同列名来合并两个数据集的情况下；例如，我们有一个数据集，在它里面员工姓名的列名不是"employee"而是"name"。在这种情况下，我们可以使用`left_on`和`right_on`关键字来分别指定两个列的名字：

In [7]:
```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
```

Out[7]:

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

> The result has a redundant column that we can drop if desired–for example, by using the `drop()` method of `DataFrame` s:

结果中有一个冗余的列，我们可以将改列移除，例如使用`DataFrame`的`drop()`方法：

In [8]:
```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

Out[8]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The `left_index` and `right_index` keywords

`left_index` 和 `right_index` 关键字参数

> Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

有时候，你不是需要按列进行合并，而是需要按照行索引进行合并。例如，将`df1`和`df2`数据集修改为如下情况：

In [9]:
```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

Out[9]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

> You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

通过指定`left_index`和`right_index`标志参数，你可以将两个数据集按照行索引进行合并：

In [10]:
```
display('df1a', 'df2a',
        "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

Out[10]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

> For convenience, `DataFrame` s implement the `join()` method, which performs a merge that defaults to joining on indices:

为了方便，`DataFrame`实现了`join()`方法，默认按照行索引合并数据集：

In [11]:
```
display('df1a', 'df2a', 'df1a.join(df2a)')
```

Out[11]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

df1a.join(df2a)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

> If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

如果需要混合的进行行或列的合并，你可以通过混合指定`left_index`和`right_on`参数或者`left_on`和`right_index`参数来实现：

In [12]:
```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
```

Out[12]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1a, df3, left_index=True, right_on='name')

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

> All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the "Merge, Join, and Concatenate" section of the Pandas documentation.

所有上面的参数都能应用到多重行索引和/或多重列上；这个接口的定义是非常直观的。需要了解更多的信息，参见Pandas在线文档"Merge、join和Concatenate"章节。

Specifying Set Arithmetic for Joins

指定合并的集合算术运算

> In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

在上面的例子中，我们都忽略了在进行数据集合并时一个重要的内容：合并时所使用的集合算术运算类型。这部分内容对于当一个数据集的键值在另一个数据集中不存在时很有意义。看下例：

In [13]:
```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

Out[13]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7)

	name	food	drink
0	Mary	bread	wine

> Here we have merged two datasets that have only a single "name" entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to `"inner"`:

上面我们合并的两个数据集在关键字列上只有一个"name"数据是共同的：Mary。默认情况下，结果会包含两个集合的*交集*；这被称为*内连接*。我们显式的指定`how`关键字参数，它的默认值是`"inner"`：

In [14]:
```
pd.merge(df6, df7, how='inner')
```

Out[14]:

	name	food	drink
0	Mary	bread	wine

> Other options for the `how` keyword are `'outer'`, `'left'`, and `'right'`. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

`how`参数的其他选项包括`'outer'`、`'left'`和`'right'`。*外连接*outer会返回两个集合的并集，并将缺失的数据填充为Pandas的NA值：

In [15]:
```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

Out[15]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

> The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

*左连接*left和*右连接*right返回的结果是包括所有的左边或右边集合。例如：

In [16]:
```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
```

Out[16]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='left')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

> The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.
>
> All of these options can be applied straightforwardly to any of the preceding join types.

结果中的行与左集合保持一致。使用`how='right'`结果会和右集合保持一致。

所有这些集合运算类型可以和前面的连接类型组合使用。

Overlapping Column Names: The `suffixes` Keyword

列名冲突：`suffixes`关键字参数

> Finally, you may end up in a case where your two input `DataFrame` s have conflicting column names. Consider this example:

最后，你可能会碰到一种情况两个输入`DataFrame`有着冲突的列名。例如：

In [17]:
```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

Out[17]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name")

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

> Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

因为结果可能会有两个相同的列名，发生冲突，merge函数会自动为这两个列添加`_x`和`_y`后缀，使得输出结果每个列名称唯一。如果默认的后缀不是你希望的，可以使用`suffixes`关键字参数为输出列添加自定义的后缀：

In [18]:
```
display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
```

Out[18]:

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

> These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

这些后缀可以应用在所有的连接方式中，也可以在多个列冲突时使用。

> For more information on these patterns, see Aggregation and Grouping where we dive a bit deeper into relational algebra. Also see the Pandas "Merge, Join and Concatenate" documentation for further discussion of these topics.

需要了解更多知识，参见聚合与分组，我们会更加深入的介绍关系代数。也可以参见Pandas在线"Merge, Join 和 Concatenate"文档学习更多内容。

Example: US States Data

例子：美国州数据

> Merge and join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at http://github.com/jakevdp/data-USstates/:

合并及联表操作在你处理多个不同数据来源时会经常出现。下面我们使用美国州及其人口数据作为例子来进行更加直观的说明。这些数据文件可以在http://github.com/jakevdp/data-USstates/ 中找到：

In [19]:
```
# 如果你没有数据文件，可以使用下面的命令下载它们
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-population.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-areas.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-abbrevs.csv
```

> Let's take a look at the three datasets, using the Pandas `read_csv()` function:

下面我们来载入三个相关的数据文件，使用Pandas的`read_csv()`函数：

In [20]:
```
pop = pd.read_csv('data/state-population.csv')
areas = pd.read_csv('data/state-areas.csv')
abbrevs = pd.read_csv('data/state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')
```

Out[20]:

pop.head()

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

areas.head()

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

abbrevs.head()

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

> Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to find the result.

有了数据之后，假如我们需要计算一个相对非常直接的结果：根据美国各州2010年人口密度进行排名。很显然我们有相关的数据，但是我们需要合并数据集才能找到结果。

> We'll start with a many-to-one merge that will give us the full state name within the population `DataFrame`. We want to merge based on the `state/region` column of `pop`, and the `abbreviation` column of `abbrevs`. We'll use `how='outer'` to make sure no data is thrown away due to mismatched labels.

我们先进行一个多对一的合并，将州全名和人口数据合并在一个`DataFrame`中。我们希望合并基于`pop`数据集的`state/region`列以及`abbreviation`数据集的`abbrevs`列。使用`how='outer'`来保证合并过程中不会因为不匹配的标签而丢失任何数据。

In [21]:
```
merged = pd.merge(pop, abbrevs, how='outer',
                  left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # 移除冗余的列
merged.head()
```

Out[21]:

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

> Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

让我们检查结果中是否有不匹配的情况，通过在数据集中寻找空值来查看：

In [22]:
```
merged.isnull().any()
```

Out[22]:
```
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

> Some of the `population` info is null; let's figure out which these are!

一些人口`population`数据是空的；再来看看是哪些。

In [23]:
```
merged[merged['population'].isnull()].head()
```

Out[23]:

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

> It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

发现所有空的人口数据都是2000年前波多黎各的；这可能因为数据来源本来就没有这些数据造成的。

> More importantly, we see also that some of the new `state` entries are also null, which means that there was no corresponding entry in the `abbrevs` key! Let's figure out which regions lack this match:

更重要的是，我们发现一些新的州`state`的数据也是空的，这意味着`abbrevs`列中不存在这些州的简称。再看看是哪些州有这种情况：

In [24]:
```
merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

Out[24]:
```
array(['PR', 'USA'], dtype=object)
```

> We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling in appropriate entries:

从上面的结果很容易发现：人口数据集中包括波多黎各（PR）和全美国（USA）的数据，而州简称数据集中却没有这两者数据。通过填充相应的数据可以很快解决这个问题：

In [25]:
```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
```

Out[25]:
```
state/region    False
ages            False
year            False
population       True
state           False
dtype: bool
```

> No more nulls in the `state` column: we're all set!

`state`列没有空值了：我们准备好了。

> Now we can merge the result with the area data using a similar procedure. Examining our results, we will want to join on the `state` column in both:

下面我们可以将上面的结果数据集和面积数据集进行合并。研究两个数据集发现，我们需要在`state`列上进行数据集合并操作：

In [26]:
```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

Out[26]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

> Again, let's check for nulls to see if there were any mismatches:

再一次，我们检查一次空值，来看是否存在不匹配的情况：

In [27]:
```
final.isnull().any()
```

Out[27]:
```
state/region     False
ages             False
year             False
population        True
state            False
area (sq. mi)     True
dtype: bool
```

> There are nulls in the `area` column; we can take a look to see which regions were ignored here:

面积`area`列有空值；我们看看是哪里出现的：

In [28]:
```
final['state'][final['area (sq. mi)'].isnull()].unique()
```

Out[28]:
```
array(['United States'], dtype=object)
```

> We see that our `areas` `DataFrame` does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population density of the entire United States is not relevant to our current discussion:

结果显示面积数据集不包括整个美国的面积。我们可以为这个空值插入正确的值（使用所有州的面积数据之和），但是这个例子中我们只需要简单地移除空值数据即可，因为全美国的人口密度数据与我们前面的问题无关：

In [29]:
```
final.dropna(inplace=True)
final.head()
```

Out[29]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

> Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (this requires the `numexpr` package to be installed; see [High-Performance Pandas: `eval()` and `query()`]):

现在我们需要数据都已经准备好了。要回答前面那个问题，首先要选择出2010年相应的部分数据集以及不分年龄的全体人口数。我们使用`query()`函数来快速完成这项任务（这需要安装`numexpr`包，参见[高性能Pandas: `eval()` 和 `query()`]）：

In [30]:
```
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

Out[30]:

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

> Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result:

下面我们可以计算人口密度并排序输出了。我们现将数据集按照`state`进行重新索引，然后计算结果：

In [31]:
```
data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']
```

In [32]:
```
density.sort_values(ascending=False, inplace=True)
density.head()
```

Out[32]:
```
state
District of Columbia    8898.897059
Puerto Rico             1058.665149
New Jersey              1009.253268
Rhode Island             681.339159
Connecticut              645.600649
dtype: float64
```

> The result is a ranking of US states plus Washington, DC, and Puerto Rico in order of their 2010 population density, in residents per square mile. We can see that by far the densest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

结果是美国州根据2010年人口密度的排名，包括华盛顿特区和波多黎各，数据是每平方英里的居住人数。结果显示人口密度最稠密的地区是华盛顿特区（表中的the District of Columbia）；在其他的州中，人口密度最大的是新泽西。

> We can also check the end of the list:

我们也可以查看结果的最后部分：

In [33]:
```
density.tail()
```

Out[33]:
```
state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype: float64
```

> We see that the least dense state, by far, is Alaska, averaging slightly over one resident per square mile.

结果显示密度最小的州，阿拉斯加，平均每平方英里略大于1个居民。

> This type of messy data merging is a common task when trying to answer questions using real-world data sources. I hope that this example has given you an idea of the ways you can combine tools we've covered in order to gain insight from your data!

当使用真实世界数据回答这种问题的时候，这种数据集的合并是很常见的任务。作者希望这个例子能为你展示了Pandas数据集合并的工具的使用，并能在你的数据集中应用这些方法。

< 组合数据集：Concat 和 Append | 目录 | 聚合与分组 >

Open in Colab