Data Indexing and Selection

数据索引和选择

> In Chapter 2, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g., `arr[2, 1]`), slicing (e.g., `arr[:, 1:5]`), masking (e.g., `arr[arr > 0]`), fancy indexing (e.g., `arr[0, [1, 5]]`), and combinations thereof (e.g., `arr[:, [1, 5]]`). Here we'll look at similar means of accessing and modifying values in Pandas `Series` and `DataFrame` objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

在第二章，我们学习了使用NumPy工具在数组中获取，设置和修改元素或子数组的方法。这些方法包括索引（如`arr[2, 1]`），切片（如`arr[:, 1:5]`），遮盖（如`arr[arr>0]`），高级索引（如`arr[0, [1, 5]]`），以及上述的组合（如`arr[:, [1, 5]]`）。下面我们将介绍在Pandas中获取和修改`Series`和`DataFrame`对象的方法。如果你已经熟悉了NumPy的操作，那么Pandas的操作对你来说也很容易上手，只需要注意一些特别的地方。

> We'll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimesnional `DataFrame` object.

我们会从最简单的一维`Series`开始学习，然后再进入复杂一些的二维`DataFrame`对象。

Data Selection in Series

在Series中选择数据

> As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

我们上一节已经看到，`Series`对象在很多方面都表现的像一个一维NumPy数组，也同时在很多方面表现像是一个标准的Python字典。如果我们能将这两个基本概念记住，它们能帮助我们理解Series的数据索引和选择的方法。

Series as dictionary

将Series看成字典

> Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

像字典一样，`Series`对象提供了从关键字集合到值集合的映射：

In [1]:
```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```
Out[1]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [2]:
```
data['b']
```
Out[2]:
```
0.5
```

> We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

我们还可以使用标准Python字典的表达式和方法来检查Series的关键字和值：

In [3]:
```
'a' in data
```
Out[3]:
```
True
```

In [4]:
```
data.keys()
```
Out[4]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [5]:
```
list(data.items())
```
Out[5]:
```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

> `Series` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a `Series` by assigning to a new index value:

`Series`对象还可以使用字典操作进行修改。就像你可以给字典的一个新的关键字赋值一样，你可以新增一个index关键字来扩展`Series`。

In [6]:
```
data['e'] = 1.25
data
```
Out[6]:
```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

> This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

这样简便的修改对象的方法是一个有用的特性：虽然在底层Pandas会对内存分配和数据复制等进行操作，但是用户通常不需要担心这一点。

Series as one-dimensional array

将Series看成一维数组

> A `Series` builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

`Series`对象构建在字典一样的接口之上，并且提供了和NumPy数组一样的数据选择方式，即*切片*，*遮盖*和*高级索引*。请看下面的例子：

In [7]:
```
# 使用指定的索引值切片
data['a':'c']
```
Out[7]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [8]:
```
# 使用隐式整数索引值切片
data[0:2]
```
Out[8]:
```
a    0.25
b    0.50
dtype: float64
```

In [9]:
```
# 遮盖
data[(data > 0.3) & (data < 0.8)]
```
Out[9]:
```
b    0.50
c    0.75
dtype: float64
```

In [10]:
```
# 高级索引
data[['a', 'e']]
```
Out[10]:
```
a    0.25
e    1.25
dtype: float64
```

> Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

在上面的例子当中，切片可能是最容易让人误解的。首先看到使用指定的显式索引进行切片（例如`data['a':'c']`），结束位置的索引值是*包含*在切片里面的，然而，使用隐式索引进行切片（例如`data[0:2]`），结束位置的索引值是*不包含*在切片里面的。

Indexers: loc, iloc, and ix

索引符：loc，iloc 和 ix

> These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

仔细想一下，你会发现这样的切片和索引操作是会造成混乱的。例如，如果`Series`对象有显式的整数索引，那么`data[1]`的操作会使用显式索引，但是`data[1:3]`的操作会使用隐式索引。

In [11]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```
Out[11]:
```
1    a
3    b
5    c
dtype: object
```

In [12]:
```
# 使用的指定的显式索引
data[1]
```
Out[12]:
```
'a'
```

In [13]:
```
# 切片时使用的隐式索引
data[1:3]
```
Out[13]:
```
3    b
5    c
dtype: object
```

> Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series`.

因为存在上面看到的这种混乱，Pandas提供了一些特殊的*索引符*属性来明确指定使用哪种索引规则。这些索引符不是函数，而是用来访问`Series`数据的切片属性。

> First, the `loc` attribute allows indexing and slicing that always references the explicit index:

首先，`loc`属性允许用户永远使用显式索引来进行定位和切片：

In [14]:
```
data.loc[1]
```
Out[14]:
```
'a'
```

In [15]:
```
data.loc[1:3]
```
Out[15]:
```
1    a
3    b
dtype: object
```

> The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

`iloc`属性允许用户永远使用隐式索引来定位和切片：

In [16]:
```
data.iloc[1]
```
Out[16]:
```
'b'
```

In [17]:
```
data.iloc[1:3]
```
Out[17]:
```
3    b
5    c
dtype: object
```

> A third indexing attribute, `ix`, is a hybrid of the two, and for `Series` objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of `DataFrame` objects, which we will discuss in a moment.

第三个索引符属性`ix`，是两者的混合，对于`Series`对象来说，等同于标准的`[]`索引。`ix`索引符的意义会在`DataFrame`对象中体现出来，我们很快就会讨论到。

> One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Python编码的一大原则就有“明确含义优于隐含意义”。`loc`和`iloc`属性的明确含义使得它们对于维护干净和可读的代码方面非常有效；尤其是当使用显示整数索引的情况下，作者推荐坚持使用它们，既能保证代码的易读性，也能防止因为前面提到的混乱情况造成的难以发现的bug。

Data Selection in DataFrame

DataFrame的数据选择

> Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

回忆上一节，我们介绍过`DataFrame`表现得既像二维数组又像由共同的索引值组成的`Series`对象的字典。这个概念也能帮助你学习如何在`DataFrame`里面进行数据选择的方法。

DataFrame as a dictionary

将DataFrame当成字典

> The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let's return to our example of areas and populations of states:

首先我们将`DataFrame`看成是相关`Series`对象组成的字典。让我们回到之前那个美国州人口和面积的例子：

In [18]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```
Out[18]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

> The individual `Series` that make up the columns of the `DataFrame` can be accessed via dictionary-style indexing of the column name:

这个`DataFrame`中的列分别由两个独立的`Series`构成，它们可以使用字典方式的关键字进行访问：

In [19]:
```
data['area']
```
Out[19]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

> Equivalently, we can use attribute-style access with column names that are strings:

同样的，当列的名字是字符串时，我们也可以使用属性的方式访问：

In [20]:
```
data.area
```
Out[20]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

> This attribute-style column access actually accesses the exact same object as the dictionary-style access:

使用字典方式和使用属性方式访问的列对象是同一个：

In [21]:
```
data.area is data['area']
```
Out[21]:
```
True
```

> Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame`, this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the `"pop"` column:

虽然这是个有用的缩写方式，但是请记住属性表达式并不是通用的。例如，如果列名不是字符串，或者与`DataFrame`的方法名字发生冲突，属性表达式都没法使用。例如，`DataFrame`有`pop()`方法，因此，`data.pop`将会指向该方法而不是`"pop"`列：

In [22]:
```
data.pop is data['pop']
```
Out[22]:
```
False
```

> In particular, you should avoid the temptation to try column assignment via attribute (i.e., use `data['pop'] = z` rather than `data.pop = z`).

特别是应该避免使用属性表达式给列赋值（例如，应该使用`data['pop']=z`而不是`data.pop=z`）。

> Like with the `Series` objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

与`Series`对象一样，你也可以通过为一个新的关键字赋值来向`DataFrame`中添加新的列：

In [23]:
```
data['density'] = data['pop'] / data['area']
data
```
Out[23]:

	area	pop	density
California	423967	38332521	90.413926
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

> This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects; we'll dig into this further in Operating on Data in Pandas.

这里展示了使用直接的语法对多个`Series`对象按元素进行算术运算；我们会在在Pandas中操作数据一节中深入讨论。

DataFrame as two-dimensional array

将DataFrame看成二维数组

> As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

前面说到，我们也可以将`DataFrame`看成是一个扩展的二维数组。我们可以通过`values`属性查看`DataFrame`对象的底层数组：

In [24]:
```
data.values
```
Out[24]:
```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

> With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

有了这个基本概念之后，很多熟悉的数组操作都可以应用在`DataFrame`对象上。例如，我们可以将`DataFrame`的行和列交换，也就是矩阵的倒置：

In [25]:
```
data.T
```
Out[25]:

	California	Texas	New York	Florida	Illinois
area	4.239670e+05	6.956620e+05	1.412970e+05	1.703120e+05	1.499950e+05
pop	3.833252e+07	2.644819e+07	1.965113e+07	1.955286e+07	1.288214e+07
density	9.041393e+01	3.801874e+01	1.390767e+02	1.148061e+02	8.588376e+01

> When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

当我们需要对`DataFrame`对象进行索引时，因为列所具有的字典索引方式，我们无法简单地按照NumPy数组的方式来处理。比方说传递一个索引值来获取一行：

In [26]:
```
data.values[0]
```
Out[26]:
```
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

> and passing a single "index" to a `DataFrame` accesses a column:

传递一个索引值来获取一个列：

In [27]:
```
data['area']
```
Out[27]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

> Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

因此对于数组方式的索引方式，我们需要使用另一种方法。Pandas仍然使用`loc`、`iloc`和`ix`索引符来进行操作。当你使用`iloc`时，这就是使用隐式索引，Pandas会把`DataFrame`当成底层的NumPy数组来处理，但行和列的索引值还是会保留在结果中：

In [28]:
```
data.iloc[:3, :2]
```
Out[28]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

> Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

类似的，使用`loc`索引符时，我们使用的是明确指定的显示索引：

In [29]:
```
data.loc[:'Illinois', :'pop']
```
Out[29]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

> The `ix` indexer allows a hybrid of these two approaches:

`ix`索引符是上两种方式的混合体：

译者注：ix已经在新版的Pandas中已经被抛弃了，因此会有一个警告，也说明读者应该慎用这个属性。

In [30]:
```
data.ix[:3, :'pop']
```
```
/home/wangy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  """Entry point for launching an IPython kernel.
```
Out[30]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

> Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects.

请牢记对于整型的索引来说，`ix`同样也会产生之前在`Series`中阐述的那种混乱情况。

> Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

然后，任何NumPy中熟悉的操作都可以在上面的索引符中使用。例如，`loc`索引符中我们可以结合遮盖和高级索引模式：

In [31]:
```
data.loc[data.density > 100, ['pop', 'density']]
```
Out[31]:

	pop	density
New York	19651127	139.076746
Florida	19552860	114.806121

> Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

上面的索引方式可以用来设置或修改数据；这可以通过你已经熟悉的NumPy的标准方式来进行：

In [32]:
```
data.iloc[0, 2] = 90
data
```
Out[32]:

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

> To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

为了锻炼你操作Pandas数据的熟练度，作者建议花些时间构建一个简单的`DataFrame`对象，然后在上面运用索引、切片、遮盖和高级索引等各种操作。

Additional indexing conventions

额外索引规则

> There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

除了上面介绍的，还有一些额外的索引规则在实践中也很有用处。首先*索引*是针对列的，而切片是针对行的：

In [33]:
```
data['Florida':'Illinois']
```
Out[33]:

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

> Such slices can also refer to rows by number rather than by index:

这样的切片操作也可以通过行的序号来索引：

In [34]:
```
data[1:3]
```
Out[34]:

	area	pop	density
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746

> Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

类似的，直接的遮盖操作也是对行的操作而不是对列的操作：

In [35]:
```
data[data.density > 100]
```
Out[35]:

	area	pop	density
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121

> These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

上面两个规则与NumPy数组语法保持一致，然而他们和Pandas风格可能并不完全一致，但是它们在实践中还是很有用的。