Introducing Pandas Objects

Pandas对象简介

> At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's introduce these three fundamental Pandas data structures: the `Series`, `DataFrame`, and `Index`.

在最基本的层面上，Pandas的对象可以被认为是NumPy结构化数组的一个升级版本，它的行和列都可以使用标签指代，而不仅仅像NumPy那样只能使用整数的序号。随着本章的推进，你会学习到很多Pandas提供的工具、方法和功能，但是要学习它们都需要首先理解它的数据结构。因此，在这之前，让我们先来详细介绍三个Pandas数据结构的最基本概念：`Series`、`DataFrame`和`Index`。

> We will start our code sessions with the standard NumPy and Pandas imports:

在写其他代码前，我们先将NumPy和Pandas按照标准方式载入：

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

The Pandas Series Object

Pandas的Series对象

> A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

Pandas的`Series`是一个一维的带索引序号的数组。可以通过列表或数组进行创建：

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
        data
Out[2]: 0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64
```

> As we see in the output, the `Series` wraps both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

我们从结果看到，`Series`封装了一个值的序列（由列表指定）和一个索引序号的序列，我们可以分别通过`values`和`index`属性访问它们。`values`属性就是你已经很熟悉的NumPy数组：

```
In [3]: data.values
Out[3]: array([0.25, 0.5 , 0.75, 1.  ])
```

> The `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail momentarily.

`Index`是一个类似数组的对象，类型是`pd.Index`，我们很快会详细介绍它。

```
In [4]: data.index
Out[4]: RangeIndex(start=0, stop=4, step=1)
```

> Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

和NumPy一致，你可以通过Python的中括号加上相应的序号语法来访问数据值：

```
In [5]: data[1]
Out[5]: 0.5

In [6]: data[1:3]
Out[6]: 1    0.50
        2    0.75
        dtype: float64
```

> As we will see, though, the Pandas `Series` is much more general and flexible than the one-dimensional NumPy array that it emulates.

你将会看到，Pandas的`Series`会比它封装的一维NumPy数组通用和灵活很多。

`Series` as generalized NumPy array

`Series`作为通用的NumPy数组

> From what we've seen so far, it may look like the `Series` object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas `Series` has an *explicitly defined* index associated with the values.

目前为止，我们看到的`Series`对象和一维NumPy数组似乎是可以互换的概念。两者最基本的区别是索引序号的存在机制：NumPy数组的整数索引*隐式提供*的，而Pandas的`Series`的索引是*显式定义*的。

> This explicit index definition gives the `Series` object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

显式定义的索引提供了`Series`对象额外的能力。例如，索引值不需要一定是个整数，可以用任何需要的数据类型来定义索引。比方说，下面我们用字符串来作为索引：

```
In [7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data
Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

> And the item access works as expected:

然后元素可以通过相应的索引值来访问：

```
In [8]: data['b']
Out[8]: 0.5
```

> We can even use non-contiguous or non-sequential indices:

我们亦可以使用非连续的或非序列的索引值：

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=[2, 5, 3, 7])
        data
Out[9]: 2    0.25
        5    0.50
        3    0.75
        7    1.00
        dtype: float64

In [10]: data[5]
Out[10]: 0.5
```

Series as specialized dictionary

Series作为特殊的字典

> In this way, you can think of a Pandas `Series` a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a `Series` is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas `Series` makes it much more efficient than Python dictionaries for certain operations.

在这个层面上，你可以将Pandas的`Series`当成Python字典的一种特殊情形。Python中的字典可以将任意的关键字key和任意的值value对应起来，`Series`是一种能将特定类型的关键字key和特定类型的值value对应起来的字典。这种静态类型是很重要的：正如NumPy数组的静态类型能提供编译好的代码提升对Python列表或集合的操作性能一样，Pandas的`Series`能提供编译好的代码提升对Python字典的操作性能。

> The `Series`-as-dictionary analogy can be made even more clear by constructing a `Series` object directly from a Python dictionary:

用一个Python字典创建一个`Series`，更加方便理解`Series`作为一个字典的机制：

```
In [14]: population_dict = {'California': 38332521,
                            'Texas': 26448193,
                            'New York': 19651127,
                            'Florida': 19552860,
                            'Illinois': 12882135}
         population = pd.Series(population_dict)
         population
Out[14]: California    38332521
         Texas         26448193
         New York      19651127
         Florida       19552860
         Illinois      12882135
         dtype: int64
```

> By default, a `Series` will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

默认情况下，`Series`会以排序关键字的方式创建一个字典。然后就可以使用Python标准的字典语法获取值：

```
In [15]: population['California']
Out[15]: 38332521
```

> Unlike a dictionary, though, the `Series` also supports array-style operations such as slicing:

下面这个操作是字典所不具有的，`Series`还支持按照数组方式的操作来对字典进行切片：

```
In [16]: population['California':'Illinois']
Out[16]: California    38332521
         Texas         26448193
         New York      19651127
         Florida       19552860
         Illinois      12882135
         dtype: int64
```

> We'll discuss some of the quirks of Pandas indexing and slicing in Data Indexing and Selection.

我们会在数据索引和选择中更详细介绍Pandas索引和切片操作。

Constructing Series objects

构建Series对象

> We've already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following:

我们已经看到几种构建Pandas的`Series`对象的方法；其语法基础都是下面的构造方法：

```
>>> pd.Series(data, index=index)
```

> where `index` is an optional argument, and `data` can be one of many entities.

其中的`index`是一个可选的参数，而`data`可以使很多种的数据集合。

> For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

例如，`data`可以是一个列表或NumPy数组，在这种情况下`index`默认是一个整数序列：

```
In [17]: pd.Series([2, 4, 6])
Out[17]: 0    2
         1    4
         2    6
         dtype: int64
```

> `data` can be a scalar, which is repeated to fill the specified index:

`data`可以是一个标量，这种情况下标量的值会填充到整个序列的index中：

```
In [18]: pd.Series(5, index=[100, 200, 300])
Out[18]: 100    5
         200    5
         300    5
         dtype: int64
```

> `data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

`data`可以是一个字典，这种情况下`index`默认是一个排序的关键字key序列：

```
In [19]: pd.Series({2:'a', 1:'b', 3:'c'})
Out[19]: 2    a
         1    b
         3    c
         dtype: object
```

> In each case, the index can be explicitly set if a different result is preferred:

每种情况下，index都可以作为额外的明确指定索引的方式，结果也会依据index参数而发生变化：

```
In [20]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
Out[20]: 3    c
         2    a
         dtype: object
```

> Notice that in this case, the `Series` is populated only with the explicitly identified keys.

上例表明，结果中包含的数据仅是index明确指定部分。

The Pandas DataFrame Object

Pandas的DaraFrame对象

> The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` object discussed in the previous section, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

Pandas的另一个基础数据结构是`DataFrame`。就像刚才介绍的`Series`一样，`DataFrame`既可以被当成是一种更通用的NumPy数组，也可以被当成是一种特殊的Python字典。下面来分别看看。

DataFrame as a generalized NumPy array

DataFrame作为一种通用的NumPy数组

> If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by "aligned" we mean that they share the same index.

如果说`Series`是带有灵活索引的通用一维数组的话，那么`DataFrame`就是带有灵活的行索引和列索引的通用二维数组。你也可以将`DataFrame`想象成一系列的`Series`对象堆叠在一起，所谓的堆叠实际上指的是这些`Series`拥有相同的索引值序列。

> To demonstrate this, let's first construct a new `Series` listing the area of each of the five states discussed in the previous section:

下面我们构建一个新的`Series`存储着美国5个州面积（和上面的州人口例子一致）来说明这一点：

```
In [21]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                      'Florida': 170312, 'Illinois': 149995}
         area = pd.Series(area_dict)
         area
Out[21]: California    423967
         Texas         695662
         New York      141297
         Florida       170312
         Illinois      149995
         dtype: int64
```

> Now that we have this along with the `population` Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

现在我们就有了两个`Series`，一个人口和一个面积，我们可以再使用一个字典来创建一个二维的对象来存储两个序列的数据：

```
In [22]: states = pd.DataFrame({'population': population,
                                'area': area})
         states
Out[22]:
```

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

> Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

`DataFrame`对象也像`Series`一样有着`index`属性，包括所有的数据的索引标签：

```
In [23]: states.index
Out[23]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

> Additionally, the `DataFrame` has a `columns` attribute, which is an `Index` object holding the column labels:

因为上面的`DataFrame`是二维的，因此它额外含有一个`columns`属性，同样也是一个`Index`对象，存储这所有列的标签：

```
In [24]: states.columns
Out[24]: Index(['population', 'area'], dtype='object')
```

> Thus the `DataFrame` can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

因此`DataFrame`也可以被看成是二维NumPy数组的通用形式，它的行和列都带有通用的索引序列用来访问数据。

DataFrame as specialized dictionary

DataFrame作为特殊的字典

> Similarly, we can also think of a `DataFrame` as a specialization of a dictionary. Where a dictionary maps a key to a value, a `DataFrame` maps a column name to a `Series` of column data. For example, asking for the `'area'` attribute returns the `Series` object containing the areas we saw earlier:

类似`Series`，我们也可以将`DataFrame`看成是一种特殊的字典。普通的字典将一个关键字key映射成一个值value，而`DataFrame`将一个列标签映射成一个`Series`对象，里面含有整列的数据。例如，访问`area`属性会返回一个`Series`对象包含前面我们放入的面积数据：

```
In [25]: states['area']
Out[25]: California    423967
         Texas         695662
         New York      141297
         Florida       170312
         Illinois      149995
         Name: area, dtype: int64
```

> Notice the potential point of confusion here: in a two-dimesnional NumPy array, `data[0]` will return the first *row*. For a `DataFrame`, `data['col0']` will return the first *column*. Because of this, it is probably better to think about `DataFrame`s as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing `DataFrame`s in Data Indexing and Selection.

这里要注意一下容易混淆的地方：NumPy的二维数组中，`data[0]`会返回第一行数据，而在`DataFrame`中，`data['col0']`会返回第一列数据。正因为此，最好还是将`DataFrame`当成是一个特殊的字典而不是通用的二维数组。我们会在数据的索引和选择一节中详细讨论更多更灵活的索引操作。

Constructing DataFrame objects

构建DataFrame对象

> A Pandas `DataFrame` can be constructed in a variety of ways. Here we'll give several examples.

Pandas中的`DataFrame`可以有多种方法进行构建。下面我们介绍几个方式。

From a single Series object

从单个**Series**对象构建

> A `DataFrame` is a collection of `Series` objects, and a single-column `DataFrame` can be constructed from a single `Series`:

`DataFrame`是`Series`对象的集合，因此单列的`DataFrame`可以从单个的`Series`对象创建：

```
In [26]: pd.DataFrame(population, columns=['population'])
Out[26]:
```

	population
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

From a list of dicts

从字典的列表构建

> Any list of dictionaries can be made into a `DataFrame`. We'll use a simple list comprehension to create some data:

任何字典的列表都可以用来创建`DataFrame`，我们使用一个简单的列表解析表达式来创建一个DataFrame：

```
In [27]: data = [{'a': i, 'b': 2 * i}
                 for i in range(3)]
         pd.DataFrame(data)
Out[27]:
```

	a	b
0	0	0
1	1	2
2	2	4

> Even if some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values:

甚至在某些关键字对应的值在字典中不存在的情况下，Pandas会自动将它们填充为`NaN`（非数字）值：

```
In [28]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
Out[28]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

From a dictionary of Series objects

从**Series**对象的字典构建

> As we saw before, a `DataFrame` can be constructed from a dictionary of `Series` objects as well:

我们之前看到`DataFrame`可以从一个`Series`对象构成的字典中创建：

```
In [29]: pd.DataFrame({'population': population,
                       'area': area})
Out[29]:
```

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

From a two-dimensional NumPy array

从一个二维**NumPy**数组构建

> Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

在给定一个二维NumPy数组的情况下，我们指定其相应的列和行的索引序列来构建一个`DataFrame`。如果行或列的index没有指定，默认会使用一个整数索引序列来指定：

```
In [30]: pd.DataFrame(np.random.rand(3, 2),
                      columns=['foo', 'bar'],
                      index=['a', 'b', 'c'])
Out[30]:
```

	foo	bar
a	0.435638	0.153130
b	0.070155	0.671968
c	0.974456	0.358945

From a NumPy structured array

从**NumPy**结构化数组构建

> We covered structured arrays in Structured Data: NumPy's Structured Arrays. A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

上一章最后一节我们介绍了结构化数组（参见结构化数据：NumPy结构化数组）。Pandas的`DataFrame`对象与结构化数组非常接近，因此可以直接从后者构建：

```
In [31]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
         A
Out[31]: array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])

In [32]: pd.DataFrame(A)
Out[32]:
```

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

The Pandas Index Object

Pandas的Index对象

> We have seen here that both the `Series` and `DataFrame` objects contain an explicit *index* that lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multi-set, as `Index` objects may contain repeated values). Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an `Index` from a list of integers:

前面内容介绍的`Series`和`DataFrame`对象都包含着一个显式定义的*索引index*对象，它的作用就是让你快速访问和修改数据。`Index`对象是一个很有趣的数据结构，它可以被当成*不可变的数组*或者*排序的集合*（严格来说是多数据集合，因为`Index`允许包含重复的值）。这两种看法在对`Index`对象进行操作时会产生一些很有趣的结果。先以一个简单的例子来说明，我们从整数列表构建一个`Index`对象：

```
In [33]: ind = pd.Index([2, 3, 5, 7, 11])
         ind
Out[33]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

Index作为不可变数组

> The `Index` in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

`Index`很多的操作都像一个数组。例如，我们可以使用标准的Python索引语法来获得值和切片：

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

In [35]: ind[::2]
Out[35]: Int64Index([2, 5, 11], dtype='int64')
```

> `Index` objects also have many of the attributes familiar from NumPy arrays:

`Index`对象也有很多你熟悉的NumPy数组属性：

```
In [36]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
         5 (5,) 1 int64
```

> One difference between `Index` objects and NumPy arrays is that indices are immutable–that is, they cannot be modified via the normal means:

NumPy数组和`Index`对象的最大区别是你无法改变`Index`的元素值，它们是不可变的：

```
In [37]: ind[1] = 0

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-906a9fa1424c> in <module>
----> 1 ind[1] = 0

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in __setitem__(self, key, value)
   3936
   3937     def __setitem__(self, key, value):
-> 3938         raise TypeError("Index does not support mutable operations")
   3939
   3940     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

> This immutability makes it safer to share indices between multiple `DataFrame`s and arrays, without the potential for side effects from inadvertent index modification.

这种不变性能在多个`DataFrame`之间共享索引时提供一种安全性，避免因为疏忽造成的索引修改和其他的副作用。

Index as ordered set

Index作为排序集合

> Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The `Index` object follows many of the conventions used by Python's built-in `set` data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

Pandas对象被设计成能够满足跨数据集进行操作，例如连接多个数据集查找或操作数据，这很大程度依赖于集合运算。`Index`对象遵循Python内建的`set`数据结构的运算法则，因此并集、交集、差集和其他的集合操作也可以按照熟悉的方式进行：

```
In [38]: indA = pd.Index([1, 3, 5, 7, 9])
         indB = pd.Index([2, 3, 5, 7, 11])

In [39]: indA & indB  # 交集
Out[39]: Int64Index([3, 5, 7], dtype='int64')

In [40]: indA | indB  # 并集
Out[40]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [41]: indA ^ indB  # 互斥差集
Out[41]: Int64Index([1, 2, 9, 11], dtype='int64')
```

> These operations may also be accessed via object methods, for example `indA.intersection(indB)`.

这些操作也可以通过对象的方法来实现，例如`indA.intersection(indB)`。