### Python 数据分析包：pandas 基础

|  |  |  |  |
| --- | --- | --- | --- |
| **您的评价**: |  |  | [**收藏该经验**](javascript:void(0)) |

**阅读目录**

* [重新索引](http://www.open-open.com/lib/view/open1402477162868.html#_label0)
* [删除指定轴上的项](http://www.open-open.com/lib/view/open1402477162868.html#_label1)
* [索引和切片](http://www.open-open.com/lib/view/open1402477162868.html#_label2)
* [算术运算和数据对齐](http://www.open-open.com/lib/view/open1402477162868.html#_label3)
* [函数应用和映射](http://www.open-open.com/lib/view/open1402477162868.html#_label4)
* [排序和排名](http://www.open-open.com/lib/view/open1402477162868.html#_label5)
* [统计方法](http://www.open-open.com/lib/view/open1402477162868.html#_label6)
* [is(not)null](http://www.open-open.com/lib/view/open1402477162868.html#_label7)
* [dropna](http://www.open-open.com/lib/view/open1402477162868.html#_label8)
* [fillna](http://www.open-open.com/lib/view/open1402477162868.html#_label9)

pandas 是基于 Numpy 构建的含有更高级数据结构和工具的数据分析包

类似于 Numpy 的核心是 ndarray，pandas 也是围绕着 Series 和 DataFrame 两个核心数据结构展开的 。Series 和 DataFrame 分别对应于一维的序列和二维的表结构。pandas 约定俗成的导入方法如下：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2 | from pandas import Series,DataFrame  import pandas as pd |

# Series

Series 可以看做一个**定长的有序字典**。基本任意的一维数据都可以用来构造 Series 对象：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> s = Series([1,2,3.0,'abc'])  >>> s  0      1  1      2  2      3  3    abc  dtype: object |

虽然 dtype:object 可以包含多种基本数据类型，但总感觉会影响性能的样子，最好还是保持单纯的 dtype。

Series 对象包含两个主要的属性：index 和 values，分别为上例中左右两列。因为传给构造器的是一个列表，所以 index 的值是从 0 起递增的整数，如果传入的是一个类字典的键值对结构，就会生成 index-value 对应的 Series；或者在初始化的时候以关键字参数显式指定一个 index 对象：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> s = Series(data=[1,3,5,7],index = ['a','b','x','y'])  >>> s  a    1  b    3  x    5  y    7  dtype: int64  >>> s.index  Index(['a', 'b', 'x', 'y'], dtype='object')  >>> s.values  array([1, 3, 5, 7], dtype=int64) |

Series 对象的元素会严格依照给出的 index 构建，这意味着：如果 data 参数是有键值对的，那么只有 index 中含有的键会被使用；以及如果 data 中缺少响应的键，即使给出 NaN 值，这个键也会被添加。

注意 Series 的 index 和 values 的元素之间虽然存在对应关系，但这与字典的映射不同。index 和 values 实际仍为互相独立的 ndarray 数组，因此 Series 对象的性能完全 ok。

Series 这种使用键值对的数据结构最大的好处在于，Series 间进行算术运算时，index 会自动对齐。

另外，Series 对象和它的 index 都含有一个 name 属性：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> s.name = 'a\_series'  >>> s.index.name = 'the\_index'  >>> s  the\_index  a            1  b            3  x            5  y            7  Name: a\_series, dtype: int64 |

# DataFrame

DataFrame 是一个**表格**型的数据结构，它含有一组有序的列（类似于 index），每列可以是不同的值类型（不像 ndarray 只能有一个 dtype）。基本上可以把 DataFrame 看成是共享同一个 index 的 Series 的集合。

DataFrame 的构造方法与 Series 类似，只不过可以同时接受多条一维数据源，每一条都会成为单独的一列：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> data = {'state':['Ohino','Ohino','Ohino','Nevada','Nevada'],          'year':[2000,2001,2002,2001,2002],          'pop':[1.5,1.7,3.6,2.4,2.9]}  >>> df = DataFrame(data)  >>> df     pop   state  year  0  1.5   Ohino  2000  1  1.7   Ohino  2001  2  3.6   Ohino  2002  3  2.4  Nevada  2001  4  2.9  Nevada  2002    [5 rows x 3 columns] |

虽然参数 data 看起来是个字典，但字典的键并非充当 DataFrame 的 index 的角色，而是 Series 的 “name” 属性。这里生成的 index 仍是 “01234”。

**较**完整的 DataFrame 构造器参数为：DataFrame(data=None,index=None,coloumns=None)，columns 即 “name”：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> df = DataFrame(data,index=['one','two','three','four','five'],                 columns=['year','state','pop','debt'])  >>> df         year   state  pop debt  one    2000   Ohino  1.5  NaN  two    2001   Ohino  1.7  NaN  three  2002   Ohino  3.6  NaN  four   2001  Nevada  2.4  NaN  five   2002  Nevada  2.9  NaN    [5 rows x 4 columns] |

同样缺失值由 NaN 补上。看一下 index、columns 和 索引的类型：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> df.index  Index(['one', 'two', 'three', 'four', 'five'], dtype='object')  >>> df.columns  Index(['year', 'state', 'pop', 'debt'], dtype='object')  >>> type(df['debt'])  <class 'pandas.core.series.Series'> |

DataFrame 面向行和面向列的操作基本是平衡的，任意抽出一列都是 Series。

# 对象属性

### 重新索引

Series 对象的重新索引通过其 .reindex(index=None,\*\*kwargs) 方法实现。\*\*kwargs 中常用的参数有俩：method=None,fill\_value=np.NaN：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | ser = Series([4.5,7.2,-5.3,3.6],index=['d','b','a','c'])  >>> a = ['a','b','c','d','e']  >>> ser.reindex(a)  a   -5.3  b    7.2  c    3.6  d    4.5  e    NaN  dtype: float64  >>> ser.reindex(a,fill\_value=0)  a   -5.3  b    7.2  c    3.6  d    4.5  e    0.0  dtype: float64  >>> ser.reindex(a,method='ffill')  a   -5.3  b    7.2  c    3.6  d    4.5  e    4.5  dtype: float64  >>> ser.reindex(a,fill\_value=0,method='ffill')  a   -5.3  b    7.2  c    3.6  d    4.5  e    4.5  dtype: float64 |

.reindex() 方法会返回一个新对象，其 index 严格遵循给出的参数，method:{'backfill', 'bfill', 'pad', 'ffill', None} 参数用于指定插值（填充）方式，当没有给出时，自动用fill\_value 填充，默认为 NaN（ffill = pad，bfill = back fill，分别指插值时向前还是向后取值）

DataFrame 对象的重新索引方法为：.reindex(index=None,columns=None,\*\*kwargs)。仅比 Series 多了一个可选的 columns 参数，用于给列索引。用法与上例类似，只不过插值方法 method 参数只能应用于**行**，即轴 0。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>> state = ['Texas','Utha','California']  >>> df.reindex(columns=state,method='ffill')      Texas  Utha  California  a      1   NaN           2  c      4   NaN           5  d      7   NaN           8    [3 rows x 3 columns]  >>> df.reindex(index=['a','b','c','d'],columns=state,method='ffill')     Texas  Utha  California  a      1   NaN           2  b      1   NaN           2  c      4   NaN           5  d      7   NaN           8    [4 rows x 3 columns] |

不过 fill\_value 依然对有效。聪明的小伙伴可能已经想到了，可不可以通过 df.T.reindex(index,method='\*\*').T 这样的方式来实现在列上的插值呢，答案是可行的。另外要注意，使用 reindex(index,method='\*\*') 的时候，index 必须是**单调**的，否则就会引发一个 ValueError: Must be monotonic for forward fill，比如上例中的最后一次调用，如果使用 index=['a','b','d','c'] 的话就不行。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 删除指定轴上的项

即删除 Series 的元素或 DataFrame 的某一行（列）的意思，通过对象的 .drop(labels, axis=0) 方法：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | >>> ser  d    4.5  b    7.2  a   -5.3  c    3.6  dtype: float64  >>> df     Ohio  Texas  California  a     0      1           2  c     3      4           5  d     6      7           8    [3 rows x 3 columns]  >>> ser.drop('c')  d    4.5  b    7.2  a   -5.3  dtype: float64  >>> df.drop('a')     Ohio  Texas  California  c     3      4           5  d     6      7           8    [2 rows x 3 columns]  >>> df.drop(['Ohio','Texas'],axis=1)     California  a           2  c           5  d           8    [3 rows x 1 columns] |

.drop() 返回的是一个新对象，元对象不会被改变。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 索引和切片

就像 Numpy，pandas 也支持通过 obj[::] 的方式进行索引和切片，以及通过布尔型数组进行过滤。

不过须要注意，因为 pandas 对象的 index 不限于整数，所以当使用**非整数**作为切片索引时，它是**末端包含**的。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | >>> foo  a    4.5  b    7.2  c   -5.3  d    3.6  dtype: float64  >>> bar  0    4.5  1    7.2  2   -5.3  3    3.6  dtype: float64  >>> foo[:2]  a    4.5  b    7.2  dtype: float64  >>> bar[:2]  0    4.5  1    7.2  dtype: float64  >>> foo[:'c']  a    4.5  b    7.2  c   -5.3  dtype: float64 |

这里 foo 和 bar 只有 index 不同——bar 的 index 是整数序列。可见当使用整数索引切片时，结果与 Python 列表或 Numpy 的默认状况相同；换成 'c' 这样的字符串索引时，结果就包含了这个边界元素。

另外一个特别之处在于 DataFrame 对象的索引方式，因为他有两个轴向（双重索引）。

可以这么理解：DataFrame 对象的标准切片语法为：.ix[::,::]。ix 对象可以接受两套切片，分别为行（axis=0）和列（axis=1）的方向：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> df     Ohio  Texas  California  a     0      1           2  c     3      4           5  d     6      7           8    [3 rows x 3 columns]  >>> df.ix[:2,:2]     Ohio  Texas  a     0      1  c     3      4    [2 rows x 2 columns]  >>> df.ix['a','Ohio']  0 |

而不使用 ix ，直接切的情况就特殊了：

* 索引时，选取的是列
* 切片时，选取的是行

这看起来有点不合逻辑，但作者解释说 “这种语法设定来源于实践”，我们信他。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> df['Ohio']  a    0  c    3  d    6  Name: Ohio, dtype: int32  >>> df[:'c']     Ohio  Texas  California  a     0      1           2  c     3      4           5    [2 rows x 3 columns]  >>> df[:2]     Ohio  Texas  California  a     0      1           2  c     3      4           5    [2 rows x 3 columns] |

使用布尔型数组的情况，注意行与列的不同切法（列切法的 : 不能省）：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | >>> df['Texas']>=4  a    False  c     True  d     True  Name: Texas, dtype: bool  >>> df[df['Texas']>=4]     Ohio  Texas  California  c     3      4           5  d     6      7           8    [2 rows x 3 columns]  >>> df.ix[:,df.ix['c']>=4]     Texas  California  a      1           2  c      4           5  d      7           8    [3 rows x 2 columns] |

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 算术运算和数据对齐

pandas 最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时，结果的索引取索引对的**并集**。自动的数据对齐在不重叠的索引处引入空值，默认为 NaN。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> foo = Series({'a':1,'b':2})  >>> foo  a    1  b    2  dtype: int64  >>> bar = Series({'b':3,'d':4})  >>> bar  b    3  d    4  dtype: int64  >>> foo + bar  a   NaN  b     5  d   NaN  dtype: float64 |

DataFrame 的对齐操作会同时发生在行和列上。

当不希望在运算结果中出现 NA 值时，可以使用前面 reindex 中提到过 fill\_value 参数，不过为了传递这个参数，就需要使用对象的方法，而不是操作符：df1.add(df2,fill\_value=0)。其他算术方法还有：sub(), div(), mul()。

Series 和 DataFrame 之间的算术运算涉及广播，暂时先不讲。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 函数应用和映射

Numpy 的 ufuncs（元素级数组方法）也可用于操作 pandas 对象。

当希望将函数应用到 DataFrame 对象的某一行或列时，可以使用 .apply(func, axis=0, args=(), \*\*kwds) 方法。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | f = lambda x:x.max()-x.min()  >>> df     Ohio  Texas  California  a     0      1           2  c     3      4           5  d     6      7           8    [3 rows x 3 columns]  >>> df.apply(f)  Ohio          6  Texas         6  California    6  dtype: int64  >>> df.apply(f,axis=1)  a    2  c    2  d    2  dtype: int64 |

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 排序和排名

Series 的 sort\_index(ascending=True) 方法可以对 index 进行排序操作，ascending 参数用于控制升序或降序，默认为升序。

若要按值对 Series 进行排序，当使用 .order() 方法，任何缺失值默认都会被放到 Series 的末尾。

在 DataFrame 上，.sort\_index(axis=0, by=None, ascending=True) 方法多了一个轴向的选择参数与一个 by 参数，by 参数的作用是针对某一（些）**列**进行排序（不能对行使用 by 参数）：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | >>> df.sort\_index(by='Ohio')     Ohio  Texas  California  a     0      1           2  c     3      4           5  d     6      7           8    [3 rows x 3 columns]  >>> df.sort\_index(by=['California','Texas'])     Ohio  Texas  California  a     0      1           2  c     3      4           5  d     6      7           8    [3 rows x 3 columns]  >>> df.sort\_index(axis=1)     California  Ohio  Texas  a           2     0      1  c           5     3      4  d           8     6      7    [3 rows x 3 columns] |

排名（Series.rank(method='average', ascending=True)）的作用与排序的不同之处在于，他会把对象的 values 替换成名次（从 1 到 n）。这时唯一的问题在于如何处理平级项，方法里的 method 参数就是起这个作用的，他有四个值可选：average, min, max, first。

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | >>> ser=Series([3,2,0,3],index=list('abcd'))  >>> ser  a    3  b    2  c    0  d    3  dtype: int64  >>> ser.rank()  a    3.5  b    2.0  c    1.0  d    3.5  dtype: float64  >>> ser.rank(method='min')  a    3  b    2  c    1  d    3  dtype: float64  >>> ser.rank(method='max')  a    4  b    2  c    1  d    4  dtype: float64  >>> ser.rank(method='first')  a    3  b    2  c    1  d    4  dtype: float64 |

注意在 ser[0]=ser[3] 这对平级项上，不同 method 参数表现出的不同名次。

DataFrame 的 .rank(axis=0, method='average', ascending=True) 方法多了个 axis 参数，可选择按行或列分别进行排名，暂时好像没有针对全部元素的排名方法。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### 统计方法

pandas 对象有一些统计方法。它们大部分都属于约简和汇总统计，用于从 Series 中提取单个值，或从 DataFrame 的行或列中提取一个 Series。

比如 DataFrame.mean(axis=0,skipna=True) 方法，当数据集中存在 NA 值时，这些值会被简单跳过，除非整个切片（行或列）全是 NA，如果不想这样，则可以通过 skipna=False 来禁用此功能：

[?](http://www.open-open.com/lib/view/open1402477162868.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | >>> df      one  two  a  1.40  NaN  b  7.10 -4.5  c   NaN  NaN  d  0.75 -1.3    [4 rows x 2 columns]  >>> df.mean()  one    3.083333  two   -2.900000  dtype: float64  >>> df.mean(axis=1)  a    1.400  b    1.300  c      NaN  d   -0.275  dtype: float64  >>> df.mean(axis=1,skipna=False)  a      NaN  b    1.300  c      NaN  d   -0.275  dtype: float64 |

其他常用的统计方法有：

|  |  |
| --- | --- |
| ######################## | \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| count | 非 NA 值的数量 |
| describe | 针对 Series 或 DF 的列计算汇总统计 |
| min , max | 最小值和最大值 |
| argmin , argmax | 最小值和最大值的索引位置（整数） |
| idxmin , idxmax | 最小值和最大值的索引值 |
| quantile | 样本分位数（0 到 1） |
| sum | 求和 |
| mean | 均值 |
| median | 中位数 |
| mad | 根据均值计算平均绝对离差 |
| var | 方差 |
| std | 标准差 |
| skew | 样本值的偏度（三阶矩） |
| kurt | 样本值的峰度（四阶矩） |
| cumsum | 样本值的累计和 |
| cummin , cummax | 样本值的累计最大值和累计最小值 |
| cumprod | 样本值的累计积 |
| diff | 计算一阶差分（对时间序列很有用） |
| pct\_change | 计算百分数变化 |

# 处理缺失数据

pandas 中 NA 的主要表现为 np.nan，另外 Python 内建的 None 也会被当做 NA 处理。

处理 NA 的方法有四种：dropna , fillna , isnull , notnull 。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### is(not)null

这一对方法对对象做元素级应用，然后返回一个布尔型数组，一般可用于布尔型索引。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### dropna

对于一个 Series，dropna 返回一个仅含非空数据和索引值的 Series。

问题在于对 DataFrame 的处理方式，因为一旦 drop 的话，至少要丢掉一行（列）。这里的解决方式与前面类似，还是通过一个额外的参数：dropna(axis=0, how='any', thresh=None) ，how 参数可选的值为 any 或者 all。all 仅在切片元素全为 NA 时才抛弃该行(列)。另外一个有趣的参数是 thresh，该参数的类型为整数，它的作用是，比如 thresh=3，会在一行中至少有 3 个**非 NA** 值时将其保留。

[回到顶部](http://www.open-open.com/lib/view/open1402477162868.html#_labelTop)

### fillna

fillna(value=None, method=None, axis=0) 中的 value 参数除了基本类型外，还可以使用字典，这样可以实现对不同的列填充不同的值。method 的用法与前面 .reindex() 方法相同，这里不再赘述。

# inplace 参数

前面有个点一直没讲，结果整篇示例写下来发现还挺重要的。就是 Series 和 DataFrame 对象的方法中，凡是会对数组作出修改并返回一个新数组的，往往都有一个 replace=False 的可选参数。如果手动设定为 True，那么原数组就可以被替换。

# 2. pandas入门

在本书的剩下部分，pandas将是我们最敢兴趣的主要库。它包含高级的数据结构和精巧的工具，使得在Python中处理数据非常快速和简单。pandas建造在NumPy之上，它使得以NumPy为中心的应用很容易使用。

作为一点儿背景，早在2008年，我任职于AQR（一个量化投资管理公司）开始构建pandas。当时，我有一组不同的需求，但对于我不能有一个单一的工具来很好的解决：

\* 支持自动或明确的数据对齐的带有标签轴的数据结构。这可以防止由数据不对齐引起的常见错误，并可以处理不同来源的不同索引数据。

\* 整合的时间序列功能。

\* 以相同的数据结构来处理时间序列和非时间序列。

\* 支持传递元数据（坐标轴标签）的算术运算和缩减。

\* 灵活处理丢失数据。

\* 在常用的基于数据的数据库（例如基于SQL）中的合并和其它关系操作。

我想要在一个地方能够做上面的所有的事情，最好是在一个非常适合于通用软件开发的语言中。Python是一个很好的候选，但是在那个时候没有一个完整的数据结构和工具的集合来提供这些功能。

在过去的四年里，pandas出乎我的意料，已经成熟到一个非常大的库，可以解决非常广泛的数据处理问题。虽然它的使用范围扩大了，但并没有抛弃我最初所渴望的简单和易使用性。我希望，通过阅读本书后，你会想我一样的发现它是一个必不可少的工具。

在本书的剩余部分，我对pandas使用下面的导入惯例：

In [1]: from pandas import Series, DataFrame

In [2]: import pandas as pd

## 2.1. pandas数据结构入门

为了开始使用pandas，你需要熟悉它的两个重要的数据结构： Series 和 DataFrame 。虽然它们不是没一个问题的通用解决方案，但提供了一个坚实的，易于使用的大多数应用程序的基础。

### 2.1.1. Series

Series是一个一维的类似的数组对象，包含一个数组的数据（任何NumPy的数据类型）和一个与数组关联的数据标签，被叫做 索引 。最简单的Series是由一个数组的数据构成：

In [4]: obj = Series([4, 7, -5, 3])

In [5]: obj

Out[5]:

0 4

1 7

2 -5

3 3

Seriers的交互式显示的字符窜表示形式是索引在左边，值在右边。因为我们没有给数据指定索引，一个包含整数0到 N-1 （这里N是数据的长度）的默认索引被创建。 你可以分别的通过它的 values 和 index 属性来获取Series的数组表示和索引对象：

In [6]: obj.values

Out[6]: array([ 4, 7, -5, 3])

In [7]: obj.index

Out[7]: Int64Index([0, 1, 2, 3])

通常，需要创建一个带有索引来确定没一个数据点的Series：

In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [9]: obj2

Out[9]:

d 4

b 7

a -5

c 3

In [10]: obj2.index

Out[10]: Index([d, b, a, c], dtype=object)

与正规的NumPy数组相比，你可以使用索引里的值来选择一个单一值或一个值集：

In [11]: obj2['a']

Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]

Out[13]:

c 3

a -5

d 6

NumPy数组操作，例如通过一个布尔数组过滤，纯量乘法，或使用数学函数，将会保持索引和值间的关联：

In [14]: obj2

Out[14]:

d 6

b 7

a -5

c 3

In [15]: obj2[obj2 > 0] In [16]: obj2 \* 2 In [17]: np.exp(obj2)

Out[15]: Out[16]: Out[17]:

d 6 d 12 d 403.428793

b 7 b 14 b 1096.633158

c 3 a -10 a 0.006738

c 6 c 20.085537

另一种思考的方式是，Series是一个定长的，有序的字典，因为它把索引和值映射起来了。它可以适用于许多期望一个字典的函数：

In [18]: 'b' in obj2

Out[18]: True

In [19]: 'e' in obj2

Out[19]: False

如果你有一些数据在一个Python字典中，你可以通过传递字典来从这些数据创建一个Series：

In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3

Out[22]:

Ohio 35000

Oregon 16000

Texas 71000

Utah 5000113

只传递一个字典的时候，结果Series中的索引将是排序后的字典的建。

In [23]: states = [‘California’, ‘Ohio’, ‘Oregon’, ‘Texas’] In [24]: obj4 = Series(sdata, index=states) In [25]: obj4 Out[25]: California NaN Ohio 35000 Oregon 16000 Texas 71000

在这种情况下， **sdata** 中的3个值被放在了合适的位置，但因为没有发现对应于 **‘California’** 的值，就出现了 **NaN** （不是一个数），这在pandas中被用来标记数据缺失或 NA 值。我使用“missing”或“NA”来表示数度丢失。在pandas中用函数 **isnull** 和 **notnull** 来检测数据丢失：

In [26]: pd.isnull(obj4) In [27]: pd.notnull(obj4)

Out[26]: Out[27]:

California True California False

Ohio False Ohio True

Oregon False Oregon True

Texas False Texas True

Series也提供了这些函数的实例方法：

In [28]: obj4.isnull()

Out[28]:

California True

Ohio False

Oregon False

Texas False

有关数据丢失的更详细的讨论将在本章的后面进行。

在许多应用中Series的一个重要功能是在算术用算中它会自动对齐不同索引的数据：

In [29]: obj3 In [30]: obj4

Out[29]: Out[30]:

Ohio 35000 California NaN

Oregon 16000 Ohio 35000

Texas 71000 Oregon 16000

Utah 5000 Texas 71000

In [31]: obj3 + obj4

Out[31]:

California NaN

Ohio 70000

Oregon 32000

Texas 142000

Utah NaN

数据对齐被安排为一个独立的话题。

Series对象本身和它的索引都有一个 **name** 属性，它和pandas的其它一些关键功能整合在一起：

In [32]: obj4.name = 'population'

In [33]: obj4.index.name = 'state'

In [34]: obj4

Out[34]:

state

California NaN

Ohio 35000

Oregon 16000

Texas 71000

Name: population

Series的索引可以通过赋值就地更改：

In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj

Out[36]:

Bob 4

Steve 7

Jeff -5

Ryan 3

### 2.1.2. DataFrame

一个Datarame表示一个表格，类似电子表格的数据结构，包含一个经过排序的列表集，它们没一个都可以有不同的类型值（数字，字符串，布尔等等）。Datarame有行和列的索引；它可以被看作是一个Series的字典（每个Series共享一个索引）。与其它你以前使用过的（如 **R** 的 **data.frame** )类似Datarame的结构相比，在DataFrame里的面向行和面向列的操作大致是对称的。在底层，数据是作为一个或多个二维数组存储的，而不是列表，字典，或其它一维的数组集合。DataDrame内部的精确细节已超出了本书的范围。

因为DataFrame在内部把数据存储为一个二维数组的格式，因此你可以采用分层索引以表格格式来表示高维的数据。分层索引是后面章节的一个主题，并且是pandas中许多更先进的数据处理功能的关键因素。

有很多方法来构建一个DataFrame，但最常用的一个是用一个相等长度列表的字典或NumPy数组：

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],

'year': [2000, 2001, 2002, 2001, 2002],

'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

frame = DataFrame(data)

由此产生的DataFrame和Series一样，它的索引会自动分配，并且对列进行了排序：

In [38]: frame

Out[38]:

pop state year

0 1.5 Ohio 2000

1 1.7 Ohio 2001

2 3.6 Ohio 2002

3 2.4 Nevada 2001

4 2.9 Nevada 2002

如果你设定了一个列的顺序，DataFrame的列将会精确的按照你所传递的顺序排列：

In [39]: DataFrame(data, columns=['year', 'state', 'pop'])

Out[39]:

year state pop

0 2000 Ohio 1.5

1 2001 Ohio 1.7

2 2002 Ohio 3.6

3 2001 Nevada 2.4

4 2002 Nevada 2.9

和Series一样，如果你传递了一个行，但不包括在 **data** 中，在结果中它会表示为NA值：

In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],

....: index=['one', 'two', 'three', 'four', 'five'])

In [41]: frame2

Out[41]:

year state pop debt

one 2000 Ohio 1.5 NaN

two 2001 Ohio 1.7 NaN

three 2002 Ohio 3.6 NaN

four 2001 Nevada 2.4 NaN

five 2002 Nevada 2.9 NaN

In [42]: frame2.columns

Out[42]: Index([year, state, pop, debt], dtype=object)

和Series一样，在DataFrame中的一列可以通过字典记法或属性来检索：

In [43]: frame2['state'] In [44]: frame2.year

Out[43]: Out[44]:

one Ohio one 2000

two Ohio two 2001

three Ohio three 2002

four Nevada four 2001

five Nevada five 2002

Name: state Name: year

注意，返回的Series包含和DataFrame相同的索引，并它们的 **name** 属性也被正确的设置了。

行也可以使用一些方法通过位置或名字来检索，例如 **ix** 索引成员（field）（更多的将在后面介绍）：

In [45]: frame2.ix['three']

Out[45]:

year 2002

state Ohio

pop 3.6

debt NaN

Name: three

列可以通过赋值来修改。例如，空的 **‘debt’** 列可以通过一个纯量或一个数组来赋值：

In [46]: frame2['debt'] = 16.5

In [47]: frame2

Out[47]:

year state pop debt

one 2000 Ohio 1.5 16.5

two 2001 Ohio 1.7 16.5

three 2002 Ohio 3.6 16.5

four 2001 Nevada 2.4 16.5

five 2002 Nevada 2.9 16.5

In [48]: frame2['debt'] = np.arange(5.)

In [49]: frame2

Out[49]:

year state pop debt

one 2000 Ohio 1.5 0

two 2001 Ohio 1.7 1

three 2002 Ohio 3.6 2

four 2001 Nevada 2.4 3

five 2002 Nevada 2.9 4

通过列表或数组给一列赋值时，所赋的值的长度必须和DataFrame的长度相匹配。如果你使用Series来赋值，它会代替在DataFrame中精确匹配的索引的值，并在说有的空洞插入丢失数据：

In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [51]: frame2['debt'] = val

In [52]: frame2

Out[52]:

year state pop debt

one 2000 Ohio 1.5 NaN

two 2001 Ohio 1.7 -1.2

three 2002 Ohio 3.6 NaN

four 2001 Nevada 2.4 -1.5

five 2002 Nevada 2.9 -1.7

给一个不存在的列赋值，将会创建一个新的列。 像字典一样 **del** 关键字将会删除列：

In [53]: frame2['eastern'] = frame2.state == 'Ohio'

In [54]: frame2

Out[54]:

year state pop debt eastern

one 2000 Ohio 1.5 NaN True

two 2001 Ohio 1.7 -1.2 True

three 2002 Ohio 3.6 NaN True

four 2001 Nevada 2.4 -1.5 False

five 2002 Nevada 2.9 -1.7 False

In [55]: del frame2['eastern']

In [56]: frame2.columns

Out[56]: Index([year, state, pop, debt], dtype=object)

索引DataFrame时返回的列是底层数据的一个视窗，而不是一个拷贝。因此，任何在Series上的就地修改都会影响DataFrame。列可以使用Series的 **copy** 函数来显式的拷贝。

另一种通用的数据形式是一个嵌套的字典的字典格式：

In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},

....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

如果被传递到DataFrame，它的外部键会被解释为列索引，内部键会被解释为行索引：

In [58]: frame3 = DataFrame(pop)

In [59]: frame3

Out[59]:

Nevada Ohio

2000 NaN 1.5

2001 2.4 1.7

2002 2.9 3.6

当然，你总是可以对结果转置：

In [60]: frame3.T

Out[60]:

2000 2001 2002

Nevada NaN 2.4 2.9

Ohio 1.5 1.7 3.6

内部字典的键被结合并排序来形成结果的索引。如果指定了一个特定的索引，就不是这样的了：

In [61]: DataFrame(pop, index=[2001, 2002, 2003])

Out[61]:

Nevada Ohio

2001 2.4 1.7

2002 2.9 3.6

2003 NaN NaN

Series的字典也以相同的方式来处理：

In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],

....: 'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)

Out[63]:

Nevada Ohio

2000 NaN 1.5

2001 2.4 1.7

你可以传递到DataFrame构造器的东西的完整清单，见[表格5-1](http://pda.readthedocs.org/en/latest/chp5.html#id2)。

如果一个DataFrame的 **index** 和 **columns** 有它们的 **name** ，也会被显示出来：

In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3

Out[65]:

state Nevada Ohio

year

2000 NaN 1.5

2001 2.4 1.7

2002 2.9 3.6

像Series一样， **values** 属性返回一个包含在DataFrame中的数据的二维ndarray：

In [66]: frame3.values

Out[66]:

array([[ nan, 1.5],

[ 2.4, 1.7],

[ 2.9, 3.6]])

如果DataFrame的列有不同的dtypes，返回值数组将会给所有的列选择一个合适的dtyps：

In [67]: frame2.values

Out[67]:

array([[2000, Ohio, 1.5, nan],

[2001, Ohio, 1.7, -1.2],

[2002, Ohio, 3.6, nan],

[2001, Nevada, 2.4, -1.5],

[2002, Nevada, 2.9, -1.7]], dtype=object)

| 可能的传递到DataFrame的构造器 | |
| --- | --- |
| **二维ndarray** | **一个数据矩阵，有可选的行标和列标** |
| 数组，列表或元组的字典 | 每一个序列成为DataFrame中的一列。所有的序列必须有相同的长度。 |
| NumPy的结构/记录数组 | 和“数组字典”一样处理 |
| Series的字典 | 每一个值成为一列。如果没有明显的传递索引，将结合每一个Series的索引来形成结果的行索引。 |
| 字典的字典 | 每一个内部的字典成为一列。和“Series的字典”一样，结合键值来形成行索引。 |
| 字典或Series的列表 | 每一项成为DataFrame中的一列。结合字典键或Series索引形成DataFrame的列标。 |
| 列表或元组的列表 | 和“二维ndarray”一样处理 |
| 另一个DataFrame | DataFrame的索引将被使用，除非传递另外一个 |
| NumPy伪装数组（MaskedArray） | 除了蒙蔽值在DataFrame中成为NA/丢失数据之外，其它的和“二维ndarray”一样 |

### 2.1.3. 索引对象

pandas的索引对象用来保存坐标轴标签和其它元数据（如坐标轴名或名称）。构建一个Series或DataFrame时任何数组或其它序列标签在内部转化为索引：

In [68]: obj = Series(range(3), index=['a', 'b', 'c'])

In [69]: index = obj.index

In [70]: index

Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]

Out[71]: Index([b, c], dtype=object)

索引对象是不可变的，因此不能由用户改变：

In [72]: index[1] = 'd'

---------------------------------------------------------------------------

Exception Traceback (most recent call last)

<ipython-input-72-676fdeb26a68> in <module>()

----> 1 index[1] = 'd'

/Users/wesm/code/pandas/pandas/core/index.pyc in \_\_setitem\_\_(self, key, value)

302 def \_\_setitem\_\_(self, key, value):

303 """Disable the setting of values."""

--> 304 raise Exception(str(self.\_\_class\_\_) + ' object is immutable')

305

306 def \_\_getitem\_\_(self, key):

Exception: <class 'pandas.core.index.Index'> object is immutable

索引对象的不可变性非常重要，这样它可以在数据结构中结构中安全的共享：

In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index

Out[75]: True

[表格5-2](http://pda.readthedocs.org/en/latest/chp5.html#id4) 是库中内建的索引类清单。通过一些开发努力，索引可以被子类化，来实现特定坐标轴索引功能。

多数用户不必要知道许多索引对象的知识，但是它们仍然是pandas数据模型的重要部分。

| pandas中的主要索引对象 | |
| --- | --- |
| **Index** | **最通用的索引对象，使用Python对象的NumPy数组来表示坐标轴标签。** |
| Int64Index | 对整形值的特化索引。 |
| MultiIndex | “分层”索引对象，表示单个轴的多层次的索引。可以被认为是类似的元组的数组。 |
| DatetimeIndex | 存储纳秒时间戳（使用NumPy的datetime64 dtyppe来表示）。 |
| PeriodIndex | 对周期数据（时间间隔的）的特化索引。 |

除了类似于阵列，索引也有类似固定大小集合一样的功能：

In [76]: frame3

Out[76]:

state Nevada Ohio

year

2000 NaN 1.5

2001 2.4 1.7

2002 2.9 3.6

In [77]: 'Ohio' in frame3.columns

Out[77]: True

In [78]: 2003 in frame3.index

Out[78]: False

每个索引都有许多关于集合逻辑的方法和属性，且能够解决它所包含的数据的常见问题。这些都总结在[表格5-3](http://pda.readthedocs.org/en/latest/chp5.html#id5) 中。

| 索引方法和属性 | |
| --- | --- |
| **append** | **链接额外的索引对象，产生一个新的索引** |
| diff | 计算索引的差集 |
| intersection | 计算交集 |
| union | 计算并集 |
| isin | 计算出一个布尔数组表示每一个值是否包含在所传递的集合里 |
| delete | 计算删除位置i的元素的索引 |
| drop | 计算删除所传递的值后的索引 |
| insert | 计算在位置i插入元素后的索引 |
| is\_monotonic | 返回True，如果每一个元素都比它前面的元素大或相等 |
| is\_unique | 返回True，如果索引没有重复的值 |
| unique | 计算索引的唯一值数组 |

## 2.2. 重要的功能

在本节中，我将带你穿过Series或DataFrame所包含的数据的基础结构的相互关系。在接下来的章节中，将要更深入的探究使用pandas进行数据分析和处理的主题。本书并不想要作为一个关于pandas库的详尽的文档；反而我将注意力集中在最重要的特性上，让不常见（也就是，比较深奥）的东西，你去自己探索。

### 2.2.1. 重新索引

pandas对象的一个关键的方法是 **reindex** ，意味着使数据符合一个新的索引来构造一个新的对象。来看一下下面一个简单的例子：

In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [80]: obj

Out[80]:

d 4.5

b 7.2

a -5.3

c 3.6

在Series上调用 **reindex** 重排数据，使得它符合新的索引，如果那个索引的值不存在就引入缺失数据值：

In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [82]: obj2

Out[82]:

a -5.3

b 7.2

c 3.6

d 4.5

e NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill\_value=0)

Out[83]:

a -5.3

b 7.2

c 3.6

d 4.5

e 0.0

为了对时间序列这样的数据排序，当重建索引的时候可能想要对值进行内插或填充。 **method** 选项可以是你做到这一点，使用一个如 **ffill** 的方法来向前填充值：

In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')

Out[85]:

0 blue

1 blue

2 purple

3 purple

4 yellow

5 yellow

[表格5-4](http://pda.readthedocs.org/en/latest/chp5.html#id8) 是可用的 **method** 选项的清单。在此，内差比正向和反向填充更复杂。

| reindex 的 method（内插）选项 | |
| --- | --- |
| **参数** | **描述** |
| ffill或pad | 前向（或进位）填充 |
| bfill或backfill | 后向（或进位）填充 |

对于DataFrame， **reindex** 可以改变（行）索引，列或两者。当只传入一个序列时，结果中的行被重新索引了：

In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],

....: columns=['Ohio', 'Texas', 'California'])

In [87]: frame

Out[87]:

Ohio Texas California

a 0 1 2

c 3 4 5

d 6 7 8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [89]: frame2

Out[89]:

Ohio Texas California

a 0 1 2

b NaN NaN NaN

c 3 4 5

d 6 7 8

使用 **columns** 关键字可以是列重新索引：

In [90]: states = ['Texas', 'Utah', 'California']

In [91]: frame.reindex(columns=states)

Out[91]:

Texas Utah California

a 1 NaN 2

c 4 NaN 5

d 7 NaN 8

一次可以对两个重新索引，可是插值只在行侧（0坐标轴）进行：

In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',

....: columns=states)

Out[92]:

Texas Utah California

a 1 NaN 2

b 1 NaN 2

c 4 NaN 5

d 7 NaN 8

正如你将看到的，使用带标签索引的 **ix** 可以把重新索引做的更简单：

In [93]: frame.ix[['a', 'b', 'c', 'd'], states]

Out[93]:

Texas Utah California

a 1 NaN 2

b NaN NaN NaN

c 4 NaN 5

d 7 NaN 8

| reindex 函数的参数 | |
| --- | --- |
| **index** | **作为索引的新序列。可以是索引实例或任何类似序列的Python数据结构。一个索引被完全使用，没有任何拷贝。** |
| method | 插值（填充）方法，见[表格5-4](http://pda.readthedocs.org/en/latest/chp5.html#id8)的选项 |
| fill\_value | 代替重新索引时引入的缺失数据值 |
| limit | 当前向或后向填充时，最大的填充间隙 |
| level | 在多层索引上匹配简单索引，否则选择一个子集 |
| copy | 如果新索引与就的相等则底层数据不会拷贝。默认为True(即始终拷贝） |

## 2.3. 从一个坐标轴删除条目

从坐标轴删除一个多或多个条目是很容易的，如果你有一个索引数组或列表且没有这些条目，但是这可能需要一点修改和集合逻辑。 **drop** 方法将会返回一个新的对象并从坐标轴中删除指定的一个或多个值：

In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [95]: new\_obj = obj.drop('c')

In [96]: new\_obj

Out[96]:

a 0

b 1

d 3

e 4

In [97]: obj.drop(['d', 'c'])

Out[97]:

a 0

b 1

e 4

对于DataFrame，可以从任何坐标轴删除索引值：

In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),

....: index=['Ohio', 'Colorado', 'Utah', 'New York'],

....: columns=['one', 'two', 'three', 'four'])

In [99]: data.drop(['Colorado', 'Ohio'])

Out[99]:

one two three four

Utah 8 9 10 11

New York 12 13 14 15

In [100]: data.drop('two', axis=1) In [101]: data.drop(['two', 'four'], axis=1)

Out[100]: Out[101]:

one three four one three

Ohio 0 2 3 Ohio 0 2

Colorado 4 6 7 Colorado 4 6

Utah 8 10 11 Utah 8 10

New York 12 14 15 New York 12 14

### 2.3.1. 索引，挑选和过滤

Series索引( **obj[...]** )的工作原理类似与NumPy索引，除了可以使用Series的索引值，也可以仅使用整数来索引。下面是关于这一点的一些例子：

In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [103]: obj['b'] In [104]: obj[1]

Out[103]: 1.0 Out[104]: 1.0

In [105]: obj[2:4] In [106]: obj[['b', 'a', 'd']]

Out[105]: Out[106]:

c 2 b 1

d 3 a 0

d 3

In [107]: obj[[1, 3]] In [108]: obj[obj < 2]

Out[107]: Out[108]:

b 1 a 0

d 3 b 1

使用标签来切片和正常的Python切片并不一样，它会把结束点也包括在内：

In [109]: obj['b':'c']

Out[109]:

b 1

c 2

使用这些函数来复制，其工作方法和你想象的一样：

In [110]: obj['b':'c'] = 5

In [111]: obj

Out[111]:

a 0

b 5

c 5

d 3

正如上面你所见到的，索引DataFrame来检索一个或多个列，可以使用一个单一值或一个序列：

In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),

.....: index=['Ohio', 'Colorado', 'Utah', 'New York'],

.....: columns=['one', 'two', 'three', 'four'])

In [113]: data

Out[113]:

one two three four

Ohio 0 1 2 3

Colorado 4 5 6 7

Utah 8 9 10 11

New York 12 13 14 15

In [114]: data['two'] In [115]: data[['three', 'one']]

Out[114]: Out[115]:

Ohio 1 three one

Colorado 5 Ohio 2 0

Utah 9 Colorado 6 4

New York 13 Utah 10 8

Name: two New York 14 12

像这样的索引有一些特殊的情况。首先，可以通过切片或一个布尔数组来选择行：

In [116]: data[:2] In [117]: data[data['three'] > 5]

Out[116]: Out[117]:

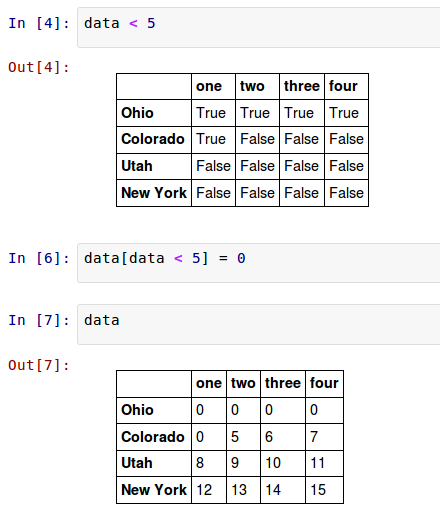
one two three four one two three four

Ohio 0 1 2 3 Colorado 4 5 6 7

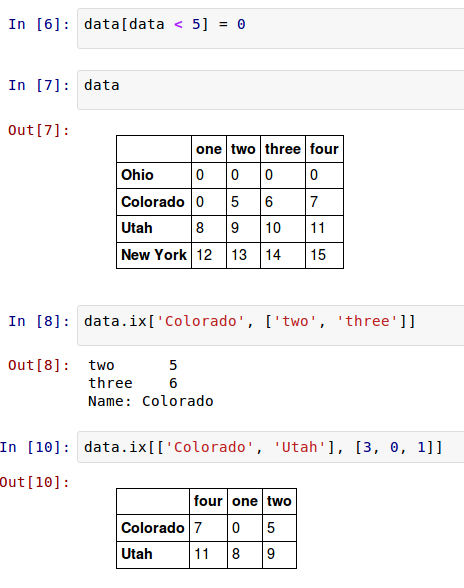
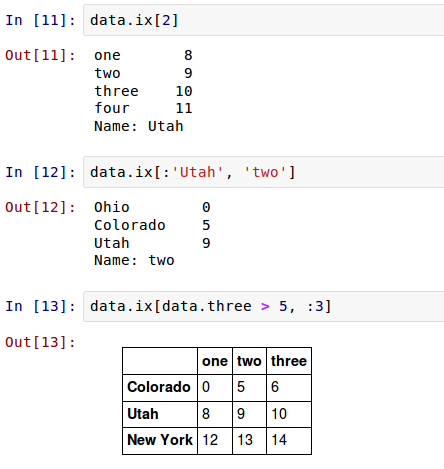
Colorado 4 5 6 7 Utah 8 9 10 11

New York 12 13 14 15

对一些读者来说这似乎不一致，但出现这种语法除了实用并没有其它什么。另一种用法是在索引中使用一个布尔DataFrame，例如通过纯量比较产生的：



这旨在在这种情况下使得DataFrame的语法更像一个ndarry。为了使DataFrame可以在行上进行标签索引，我将介绍特殊的索引字段 **ix** 。这使你可以从DataFrame选择一个行和列的子集，使用像NumPy的记法再加上轴标签。正如我早先提到的，这也是一种不是很冗长的重新索引的方法：

因此，有很多方法来选择和重排包含在pandas对象中的数据。对于DataFrame， [表格5-6](http://pda.readthedocs.org/en/latest/chp5.html#id11) 是这些方法的简短概要。稍后你将接触到分层索引，那时你会有一些额外的选项。

在设计pandas时，我觉得不得不敲下 **frame[:, col]** 来选择一列，是非常冗余的（且易出错的），因此列选择是最常见的操作之一。因此，我做了这个设计权衡，把所有的富标签索引引入到 **ix** 。

| **obj[val]** | **从DataFrame选择单一列或连续列。特殊情况下的便利：布尔数组（过滤行），切片（行切片），或布尔DataFrame（根据一些标准来设置值）。** |
| --- | --- |
| obj.ix[val] | 从DataFrame的行集选择单行 |
| obj.ix[:, val] | 从列集选择单列 |
| obj.ix[val1, val2] | 选择行和列 |
| reindex 方法 | 转换一个或多个轴到新的索引 |
| xs 方法 | 通过标签选择单行或单列到一个Series |
| icol, irow 方法 | 通过整数位置，分别的选择单行或单列到一个Series |
| get\_value, set\_value 方法 | 通过行和列标选择一个单值 |

### 2.3.2. 算术和数据对其

pandas的最重要的特性之一是在具有不同索引的对象间进行算术运算的行为。当把对象加起来时，如果有任何的索引对不相同的话，在结果中将会把各自的索引联合起来。让我们看一个简单的例子：

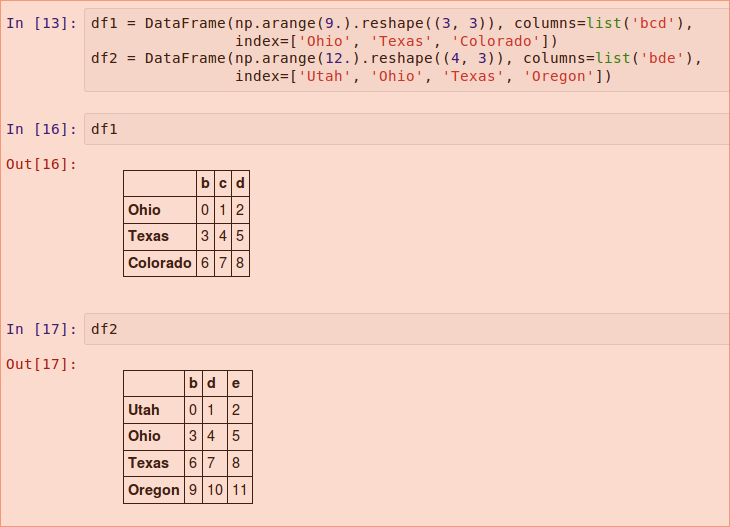
.. image:: \_static/126.png

把它们加起来生成：

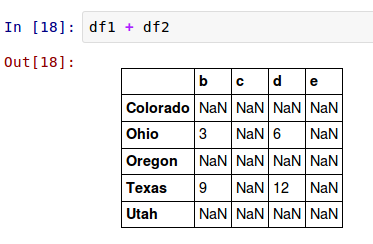
..image:: \_static/130.png

内部数据对其，在索引不重合的地方引入了NA值。数据缺失在算术运算中会传播。

对于DataFrame，对其在行和列上都表现的很好：

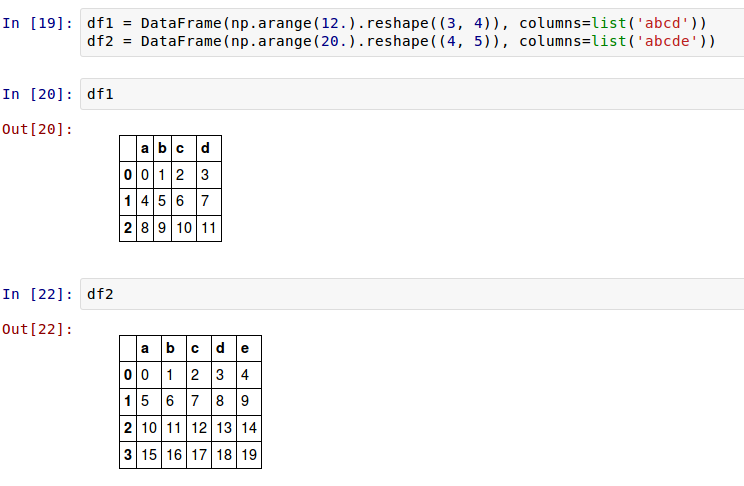


把这些加起来返回一个DataFrame，它的索引和列是每一个DataFrame对应的索引和列的联合：

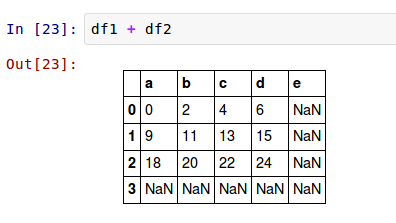


#### 2.3.2.1. 带填充值的算术方法

在不同索引对象间的算术运算，当一个轴标签在另一个对象中找不到时，你可能想要填充一个特定的值，如0：



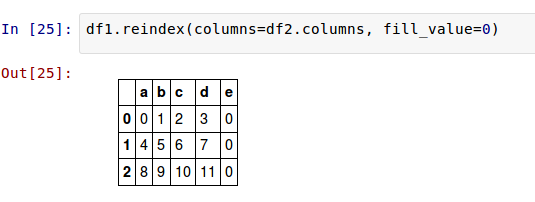
把它们加起来导致在不重合的位置出现NA值：



在 **df1** 上使用 **add** 方法，我把 **df2** 传递给它并给 **fill\_value** 赋了一个参数：

.. image:: \_static/141.png

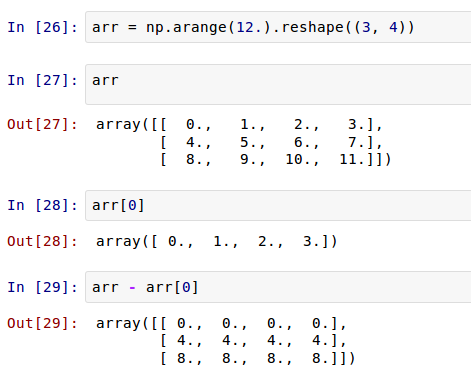
相关的，当你重新索引Series或DataFrame时，你可以设定一个不同的填充值：



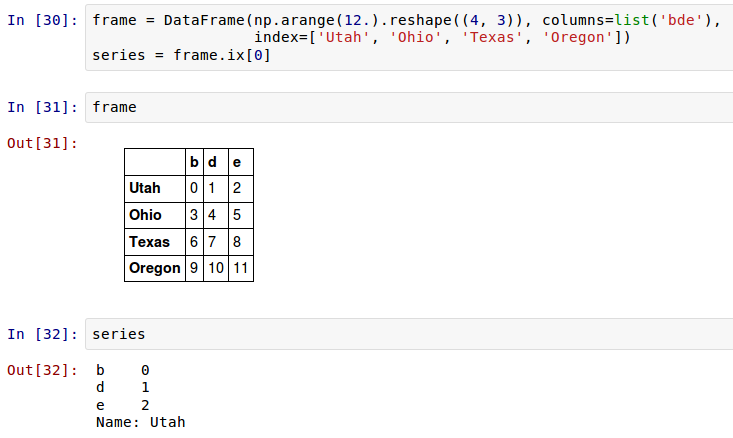
| 灵活的算术方法 | |
| --- | --- |
| **add** | **加法(+)** |
| sub | 减法(-) |
| div | 除法(/) |
| mul | 乘法(\*) |

#### 2.3.2.2. DataFrame 和 Series 间的操作

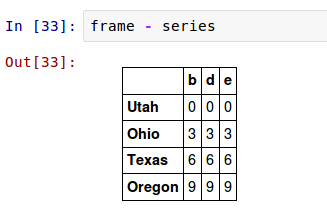
与NumPy数组一样，很好的定义了DataFrame和Series间的算术操作。首先，作为一个激发性的例子，考虑一个二维数组和它的一个行间的差分：



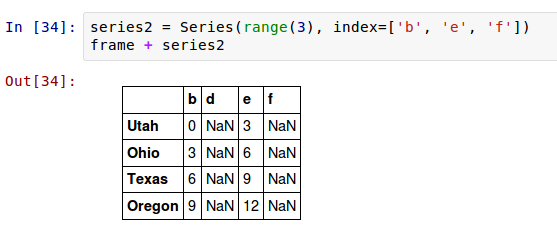
这被称为 广播 (broadcasting)，在[第12章](http://pda.readthedocs.org/en/latest/chp12.html#chp12index)将会对此进行更详细的解释。在一个DataFrame和一个Series间的操作是类似的：



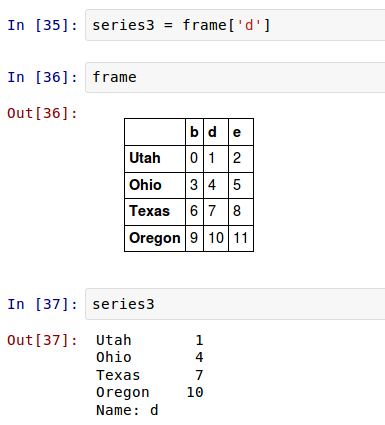
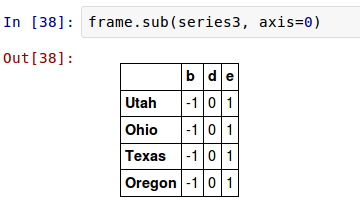
默认的，DataFrame和Series间的算术运算Series的索引将匹配DataFrame的列，并在行上扩展：



如果一个索引值在DataFrame的列和Series的索引里都找不着，对象将会从它们的联合重建索引：



如果想在行上而不是列上进行扩展，你要使用一个算术方法。例如：

你所传递的坐标值是将要匹配的 坐标 。在这种情况下，我们的意思是匹配DataFrame的行，并进行扩展。

### 2.3.3. 函数应用和映射