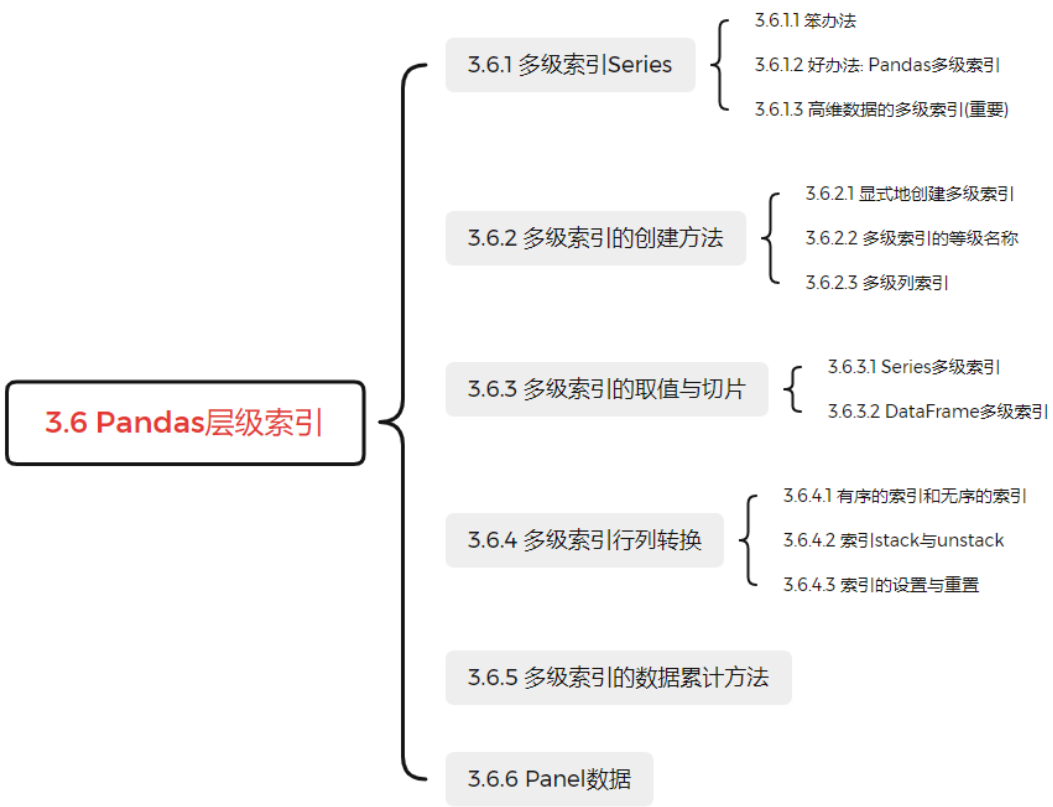
# 3.6 Pandas层级索引

## 思维导图



当目前为止, 接触的都是一维数据和二维数据, 用 Pandas 的 `Series` 和 `DataFrame` 对象就可以存储. 但也经常会遇到存储多维数据的需求, **数据索引超过一两个键**. 为此, Pandas 提供了 `Panel` 和 `Panel4D` 对象解决三维数据与四维数据(详情请参见 3.7 节). 而在实践中, 更直观的形式是通过**层级索引**(hierarchical indexing, 也被称为**多级索引**, multi-indexing)配合多个有不同**等级**(level)的一级索引一起使用, 这样就可以将高维数组转换成类似一维 `Series` 和二维 `DataFrame` 对象的形式.

> 多级索引：在一个轴axis上有多个(两个以上)的索引，能够以低维度形式来表示高维度的数据。单级索引是Index对象，多级索引是`MultiIndex`对象。

在这一节中, 将介绍创建 `MultiIndex` 对象的方法, 多级索引数据的取值、切片和统计值的计算, 以及普通索引与层级索引的转换方法. 首先导入 Pandas 和 NumPy:

[1]:

```
import pandas as pd
import numpy as np
```

# 3.6.1 多级索引Series

让看看如何用一维的 `Series` 对象表示二维数据——用一系列包含特征与数值的数据点来简单演示.

## 3.6.1.1 笨办法

假设想要分析美国各州在两个不同年份的数据. 如果用前面介绍的 Pandas 工具来处理, 那么可能会用一个 Python 元组来表示索引:

[3]:

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
print(pop)
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

通过**元组构成的多级索引**, 可以直接在 `Series` 上取值或用切片查询数据:

[4]:

```
print(pop[('California', 2000):('Texas', 2000)])
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

但是这么做很不方便. 假如想要选择所有 2000 年的数据, 那么就得用一些比较复杂的(可能也比较慢的)清理方法了:

[5]:

```
print(pop[[i    for i in pop.index if i[1] == 2000]])
(California, 2000)    33871648
(New York, 2000)      18976457
(Texas, 2000)         20851820
dtype: int64
```

这么做虽然也能得到需要的结果, 但是与 Pandas 令人爱不释手的切片语法相比, 这种方法确实不够简洁(在处理较大的数据时也不够高效).

## 3.6.1.2 好办法: Pandas多级索引

好在 Pandas 提供了更好的解决方案. 用元组表示索引其实是多级索引的基础, Pandas 的 `MultiIndex` 类型提供了更丰富的操作方法. 可以用元组创建一个多级索引, 如下所示:

[7]:

```
print(pop)
print('--------pd.MultiIndex.from_tuples(index)-----------')
index = pd.MultiIndex.from_tuples(index)
print(index)
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
--------pd.MultiIndex.from_tuples(index)-----------
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

会发现 `MultiIndex` 里面有一个 `levels` 属性表示索引的**等级** (索引的维度) ——这样做可以将州名和年份作为每个数据点的不同**标签**.

> 总维度= 数据(列)维度 + 索引(行)维度。比如上述例子，数据维度为1, 索引维度为2，那么总维度为3

如果将前面创建的 `pop` 的索引重置(`reindex`)为 `MultiIndex`, 就会看到层级索引:

[10]:

```
pop = pop.reindex(index)
print(pop)
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

其中前两列表示 `Series` 的多级索引值(二维), 第三列是数据(一维). **会发现有些行仿佛缺失了第一列数据——这其实是多级索引的表现形式, 每个空格与上面的索引相同.** 现在可以直接用第二个索引获取 2010 年的全部数据, 与 Pandas 的多维切片查询用法一致:

[11]:

```
print(pop[:, 2010])
#print(pop['Texas'])
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

结果是单索引的数组, 正是需要的. 与之前的元组索引相比, 多级索引的语法更简洁. (操作也更方便! )下面继续介绍层级索引的取值操作方法.

## 3.6.1.3 高维数据的多级索引(重要)

可能已经注意到, 其实完全可以用一个带行列索引的简单 `DataFrame` 代替前面的多级索引. 其实 Pandas 已经实现了类似的功能. `unstack()` 方法可以快速将一个多级索引的 `Series` 转化为普通索引的 `DataFrame`。

1. 行列索引转化， `.unstack()` 数据升维，索引降维(最内层索引)。而 `.stack()` 数据降维，索引升维(最内层索引)。
2. 每列的元素总个数 = 各维度行索引数的乘积。 6 = 2 * 3
3. 元素总个数 = 各维度行索引数*列索引数。 6 = 2 * 3 * 1

[13]:

```
print(pop)
print('---pop.unstack()----')
pop_df = pop.unstack()
print(pop_df)
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
---pop.unstack()----
                2000      2010
California  33871648  37253956
New York    18976457  19378102
Texas       20851820  25145561
```

当然了, 也有 `stack()` 方法实现相反的效果:

[14]:

```
print(pop_df)
print('---pop.stack()----')
print(pop_df.stack())
                2000      2010
California  33871648  37253956
New York    18976457  19378102
```

```
Texas       20851820  25145561
---pop.stack()----
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

可能会纠结于为什么要费时间研究层级索引. 其实理由很简单: 如果可以用含多级索引的一维 `Series` 数据表示二维数据, 那么就可以用 `Series` 或 `DataFrame` 表示三维甚至更高维度的数据. 多级索引每增加一级, 就表示数据增加一维, 利用这一特点就可以轻松表示任意维度的数据了. 假如要增加一列显示每一年各州的人口统计指标(例如 18 岁以下的人口), 那么对于这种带有 `MultiIndex` 的对象, 增加一列就像 `DataFrame` 的操作一样简单:

[15]:

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
print(pop_df)
print('---pop.stack()----')
print(pop_df.stack())
                     total  under18
California 2000  33871648  9267089
           2010  37253956  9284094
New York   2000  18976457  4687374
           2010  19378102  4318033
Texas      2000  20851820  5906301
           2010  25145561  6879014
---pop.stack()----
California  2000  total      33871648
                  under18     9267089
            2010  total      37253956
                  under18     9284094
New York    2000  total      18976457
                  under18     4687374
            2010  total      19378102
                  under18     4318033
Texas       2000  total      20851820
                  under18     5906301
            2010  total      25145561
                  under18     6879014
dtype: int64
```

另外, 所有在 3.4 节介绍过的通用函数和其他功能也同样适用于层级索引. 可以计算上面数据中 18 岁以下的人口占总人口的比例:

[16]:

```
f_u18 = pop_df['under18'] / pop_df['total']
print(f_u18)
print('------f_u18.unstack()-------')
print(f_u18.unstack())
```

```
California  2000    0.273594
            2010    0.249211
New York    2000    0.247010
            2010    0.222831
Texas       2000    0.283251
            2010    0.273568
dtype: float64
------f_u18.unstack()-------
                2000      2010
California  0.273594  0.249211
New York    0.247010  0.222831
Texas       0.283251  0.273568
```

同样, 也可以快速浏览和操作高维数据.

## 3.6.2 多级索引的创建方法

为 `Series` 或 `DataFrame` 创建多级索引最直接的办法就是将 `index` 参数设置为至少二维的索引数组, 如下所示:

[83]:

```
df = pd.DataFrame(np.random.rand(4, 2),  # 4*2 = 8
                  index=[['English', 'English', 'Chinese', 'Chinese'],
                         ['like','dislike','like','dislike']],  # 4 + 4
                  columns=['girl', 'boy'])
print(df)
print('---MultiIndex.from_product()笛卡尔积外积----')
df2 = pd.DataFrame(np.random.rand(4, 2),
                   index=pd.MultiIndex.from_product([['English','Chinese'],
                                                     ['like','dislike']]), # 2*2
= 4
                   columns= ['girl', 'boy']) # 2
print(df2)    # 创建多级 行 索引
                      girl       boy
English like     0.705040  0.611248
        dislike  0.661523  0.745367
Chinese like     0.749331  0.339376
        dislike  0.124760  0.102350
---MultiIndex.from_product()----
                      girl       boy
English like     0.247010  0.956254
        dislike  0.468136  0.120788
Chinese like     0.188142  0.398446
        dislike  0.229750  0.071471
```

`MultiIndex` 的创建工作将在后台完成. 同理, 如果把将元组作为键的字典传递给 Pandas, Pandas 也会默认转换为 `MultiIndex`:

[84]:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
```

```
        ('New York', 2010): 19378102}
print(pd.Series(data))
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
New York    2000    18976457
            2010    19378102
dtype: int64
```

但是有时候显式地创建 `MultiIndex` 也是很有用的, 下面来介绍一些创建方法.

## 3.6.2.1 显式地创建多级索引

可以用 `pd.MultiIndex` 中的类方法更加灵活地构建多级索引. 下面介绍4种常见的方式:

1. 通过一个有不同等级的若干简单数组组成的列表来构建 `MultiIndex`:

[85]:

```
print(pd.MultiIndex.from_arrays([ ['a', 'a', 'b', 'b'], [1, 2, 1, 2] ]))
MultiIndex(levels=[['a', 'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

1. 通过包含多个索引值的元组构成的列表创建`MultiIndex`:

[86]:

```
print(pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)]))
MultiIndex(levels=[['a', 'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

1. 通过两个索引的笛卡尔积(Cartesian product)创建 `MultiIndex`:

[87]:

```
print(pd.MultiIndex.from_product([['a', 'b'], [1, 2]]))
MultiIndex(levels=[['a', 'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

1. 提供 `levels`(包含每个等级的索引值列表的列表)和 `labels`(包含每个索引值标签列表的列表)创建 `MultiIndex`:

[88]:

```
print(pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                    codes=[[0, 0, 1, 1], [0, 1, 0, 1]]))
MultiIndex(levels=[['a', 'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

在创建 `Series` 或 `DataFrame` 时, 可以将这些对象作为 `index` 参数, 或者通过 `reindex` 方法更新 `Series` 或 `DataFrame` 的索引.

### 3.6.2.2 多级索引的等级名称

给 `MultiIndex` 的等级加上名称会为一些操作提供便利. 可以在前面任何一个 `MultiIndex` 构造器中通过 `names` 参数设置等级名称, 也可以在创建之后通过索引的 `names` 属性来修改名称:

[89]:

```
print(pop)
print('------pop.index.names------')
pop.index.names = ['state', 'year']
print(pop)
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
------pop.index.names------
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

在处理复杂的数据时, 为等级设置名称是管理多个索引值的好办法.

### 3.6.2.3 多级列索引

每个 `DataFrame` 的行与列都是对称的, 也就是说既然有多级行索引, 那么同样可以有多级列索引. 让通过一份医学报告的模拟数据来演示:

[17]:

```
# 多级行列索引
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit']) # 2*2
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type']) # 3*2

print(index)
print(columns)
# 模拟数据
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# 创建DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
MultiIndex(levels=[[2013, 2014], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['year', 'visit'])
```

```
MultiIndex(levels=[['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
           names=['subject', 'type'])
```

[17]:

| | subject | Bob | Guido | Sue | | | |
|---|---|---|---|---|---|---|---|
| | type | HR | Temp | HR | Temp | HR | Temp |
| year | visit | | | | | | |
| 2013 | 1 | 39.0 | 37.5 | 34.0 | 36.3 | 39.0 | 36.0 |
| | 2 | 32.0 | 38.2 | 33.0 | 36.8 | 32.0 | 37.0 |
| 2014 | 1 | 34.0 | 38.3 | 37.0 | 36.9 | 39.0 | 37.6 |
| | 2 | 56.0 | 36.1 | 17.0 | 37.9 | 43.0 | 36.6 |

多级行列索引的创建**非常**简单. 上面创建了一个简易的四维数据, 四个维度分别为被检查人的姓名、检查项目、检查年份和检查次数. 可以在列索引的第一级查询姓名, 从而获取包含一个人(例如 Guido)全部检查信息的 `DataFrame`:

[91]:

```
health_data['Guido']
```

[91]:

| | type | HR | Temp |
|---|---|---|---|
| year | visit | | |
| 2013 | 1 | 24.0 | 36.6 |
| | 2 | 49.0 | 35.9 |
| 2014 | 1 | 34.0 | 36.5 |
| | 2 | 26.0 | 35.9 |

如果想获取包含多种标签的数据, 需要通过对多个维度(姓名、国家、城市等标签)的多次查询才能实现, 这时使用多级行列索引进行查询会非常方便.

# 3.6.3 多级索引的取值与切片

对 `MultiIndex` 的取值和切片操作很直观, 可以直接把索引看成额外增加的维度. 先来介绍 `Series` 多级索引的取值与切片方法, 再介绍 `DataFrame` 的用法.

## 3.6.3.1 Series多级索引

看看下面由各州历年人口数量创建的多级索引 `Series`:

[18]:

```
print(pop)
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

可以通过对多个级别索引值获取单个元素:

[93]:

```
print(pop['California', 2000])
33871648
```

`MultiIndex` 也支持**局部取值**(partial indexing), 即**只取索引的某一个层级, 由外层到内层**. 假如只取最高级的索引, 那么会把内层的索引保留, 获得的结果是一个新的 `Series`, 未被选中的低层索引值会被保留:

[23]:

```
print(pop['Texas'])
print(pop[:, 2000])
2000    20851820
2010    25145561
dtype: int64
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

类似的还有局部切片, 不过要求 `MultiIndex` 是按顺序排列的(就像将在 3.6.4 节介绍的那样):

[24]:

```
print(pop.loc['California':'New York'])
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
dtype: int64
```

如果索引已经排序, 那么可以用**低层级的索引取值, 高层索引需要使用:切片**:

[96]:

```
print(pop[:, 2000])
state
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

其他取值与数据选择的方法(详情请参见 3.3 节)也都起作用. 下面的例子是通过布尔掩码选择数据:

[97]:

```
print(pop[pop > 22000000])
state       year
California  2000    33871648
            2010    37253956
Texas       2010    25145561
dtype: int64
```

也可以用花哨的索引选择数据:

[98]:

```
print(pop[ ['California', 'Texas'] ])
state       year
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
dtype: int64
```

## 3.6.3.2 DataFrame多级索引

`DataFrame` 多级索引的用法与 `Series` 类似. 还用之前的体检报告数据来演示:

[99]:

```
health_data
```

[99]:

| | subject | Bob | Guido | Sue | | | |
|---|---|---|---|---|---|---|---|
| | type | HR | Temp | HR | Temp | HR | Temp |
| year | visit | | | | | | |
| 2013 | 1 | 30.0 | 36.3 | 24.0 | 36.6 | 28.0 | 38.7 |
| | 2 | 50.0 | 36.6 | 49.0 | 35.9 | 36.0 | 35.7 |
| 2014 | 1 | 40.0 | 36.9 | 34.0 | 36.5 | 49.0 | 37.2 |
| | 2 | 47.0 | 37.3 | 26.0 | 35.9 | 30.0 | 38.0 |

由于 `DataFrame` 的**基本索引是列索引**, 因此 `Series` 中多级索引的用法到了 `DataFrame` 中就应用在列上了. 例如, 可以通过简单的操作获取 Guido 的心率数据:

[100]:

```
print(health_data['Guido', 'HR'])
year  visit
2013  1        24.0
      2        49.0
2014  1        34.0
      2        26.0
Name: (Guido, HR), dtype: float64
```

与单索引类似, 在 3.3 节介绍的 `loc`、`iloc` 和 `ix` 索引器都可以使用, 例如:

[101]:

```
health_data.iloc[:2, :2]
```

[101]:

| | **subject** | **Bob** | |
|---|---|---|---|
| | type | HR | Temp |
| year | visit | | |
| 2013 | 1 | 30.0 | 36.3 |
| 2 | 50.0 | 36.6 | |

虽然这些索引器将多维数据当作二维数据处理, 但是在 `loc` 和 `iloc` 中可以传递多个层级的索引元组, 例如:

[102]:

```
# 行索引 和 列索引
print(health_data.loc[:, ('Bob', 'HR')])
year  visit
2013  1        30.0
      2        50.0
2014  1        40.0
      2        47.0
Name: (Bob, HR), dtype: float64
```

这种索引元组的用法不是很方便, 如果在元组中使用切片还会导致语法错误:

[103]:

```
# health_data.loc[(:, 1), (:, 'HR')]
```

虽然可以用 Python 内置的 `slice()` 函数获取想要的切片, 但是还有一种更好的办法, 就是使用 `IndexSlice` 对象. Pandas 专门用它解决这类问题, 例如:

[104]:

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

[104]:

| | subject | Bob | Guido | Sue |
|---|---|---|---|---|
| | type | HR | HR | HR |
| year | visit | | | |
| 2013 | 1 | 30.0 | 24.0 | 28.0 |
| 2014 | 1 | 40.0 | 34.0 | 49.0 |

和带多级索引的 `Series` 和 `DataFrame` 进行数据交互的方法有很多, 但就像本书中的诸多工具一样, 若想掌握, 最好的办法就是使用它们!

- 基本索引 采用列索引方式;
- 索引器 采用行列索引方式； health_data.iloc[:2, :2]
- 索引器 对行列多级索引会出错，索引引入IndexSlice对象。

# 3.6.4 多级索引行列转换

使用多级索引的关键是掌握有效数据转换的方法. Pandas 提供了许多操作, 可以让数据在内容保持不变的同时, 按照需要进行行列转换. 之前用一个简短的例子演示过 `stack()` 和 `unstack()` 的用法, 但其实还有许多合理控制层级行列索引的方法, 让来一探究竟.

## 3.6.4.1 有序的索引和无序的索引

在前面的内容里, 曾经简单提过多级索引排序, 这里需要详细介绍一下. **如果 `MultiIndex` 不是有序的索引, 那么大多数切片操作都会失败.** 首先创建一个**不按字典顺序(lexographically)排列的**多级索引 `Series`:

[25]:

```
index = pd.MultiIndex.from_product([ ['a', 'c', 'b'], [1, 2] ])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
print(data)
```

```
char  int
a     1      0.606461
      2      0.725862
c     1      0.341187
      2      0.307212
b     1      0.308410
      2      0.420743
dtype: float64
```

如果想对索引使用局部切片, 那么错误就会出现:

[26]:

```
try:
    data['a':'b']
except KeyError as e:
    print(type(e))
    print(e)
```

```
<class 'pandas.errors.UnsortedIndexError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

尽管从错误信息里面看不出具体的细节, 但问题是出在 `MultiIndex` 无序排列上. 局部切片和许多其他相似的操作都要求 `MultiIndex` 的各级索引是有序的(即按照字典顺序由 A 至 Z). 为此, Pandas 提供了许多便捷的操作完成排序, 如 `sort_index()` 和 `sortlevel()` 方法. 用最简单的 `sort_index()` 方法来演示:

[28]:

```
data = data.sort_index()
print(data)
char  int
a     1      0.606461
      2      0.725862
b     1      0.308410
      2      0.420743
c     1      0.341187
      2      0.307212
dtype: float64
```

索引排序之后, 局部切片就可以正常使用了:

[29]:

```
print(data['a':'b'])
char  int
a     1      0.606461
      2      0.725862
b     1      0.308410
      2      0.420743
dtype: float64
```

## 3.6.4.2 索引stack与unstack

前文曾提过, 可以将一个多级索引数据集转换成简单的二维形式, 可以通过 `level` 参数设置转换的索引层级:

[109]:

```
print(pop)
print('-----unstack(level=0)---- ')
print(pop.unstack(level=0))
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
-----unstack(level=0)----
state  California  New York     Texas
year
2000     33871648  18976457  20851820
2010     37253956  19378102  25145561
```

[110]:

```
print(pop)
print('-----unstack(level=1)---- ')
print(pop.unstack(level=1))
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
-----unstack(level=1)----
year            2000      2010
state
California  33871648  37253956
New York    18976457  19378102
Texas       20851820  25145561
```

`unstack()` 是 `stack()` 的逆操作, 同时使用这两种方法让数据保持不变:

[111]:

```
print(pop.unstack().stack())
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

## 3.6.4.3 索引的设置与重置

层级数据维度转换的另一种方法是行列标签转换, 可以通过 `reset_index` 方法实现. 如果在上面的人口数据 `Series` 中使用该方法, 则会生成一个列标签中包含之前行索引标签 **state** 和 **year** 的 `DataFrame`. 也可以用数据的 `name` 属性为列设置名称:

[30]:

```
print(pop)
print('-----pop.reset_index---- ')
pop_flat = pop.reset_index(name='population')
print(pop_flat)
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
-----pop.reset_index----
      level_0  level_1  population
0  California     2000    33871648
1  California     2010    37253956
2    New York     2000    18976457
```

```
3    New York      2010    19378102
4       Texas      2000    20851820
5       Texas      2010    25145561
```

在解决实际问题的时候, 如果能将类似这样的原始输入数据的列直接转换成 `MultiIndex`, 通常将大有裨益. 其实可以通过 `DataFrame` 的 `set_index` 方法实现, 返回结果就会是一个带多级索引的 `DataFrame`:

[117]:

```
pop_flat.set_index(['state', 'year'])
print(pop_flat)
        state  year  population
0  California  2000    33871648
1  California  2010    37253956
2    New York  2000    18976457
3    New York  2010    19378102
4       Texas  2000    20851820
5       Texas  2010    25145561
```

在实践中, 我发现用这种重建索引的方法处理数据集非常好用.

## 3.6.5 多级索引的数据累计方法

前面已经介绍过一些 Pandas 自带的数据累计方法, 比如 `mean()`、`sum()` 和 `max()`. 而对于层级索引数据, 可以设置参数 `level` 实现对数据子集的累计操作. 再一次以体检数据为例:

[31]:

```
health_data
```

[31]:

| | subject | Bob | | Guido | Sue | | |
|---|---|---|---|---|---|---|---|
| | type | HR | Temp | HR | Temp | HR | Temp |
| year | visit | | | | | | |
| 2013 | 1 | 39.0 | 37.5 | 34.0 | 36.3 | 39.0 | 36.0 |
| | 2 | 32.0 | 38.2 | 33.0 | 36.8 | 32.0 | 37.0 |
| 2014 | 1 | 34.0 | 38.3 | 37.0 | 36.9 | 39.0 | 37.6 |
| | 2 | 56.0 | 36.1 | 17.0 | 37.9 | 43.0 | 36.6 |

如果需要计算每一年各项指标的平均值, 那么可以将参数 `level` 设置为索引 `year`:

[32]:

```
data_mean = health_data.mean(level='year')
data_mean
```

[32]:

| subject | Bob | | Guido | | Sue | |
|---|---|---|---|---|---|---|
| type | HR | Temp | HR | Temp | HR | Temp |
| year | | | | | | |
| 2013 | 35.5 | 37.85 | 33.5 | 36.55 | 35.5 | 36.5 |
| 2014 | 45.0 | 37.20 | 27.0 | 37.40 | 41.0 | 37.1 |

如果再设置 `axis` 参数, 就可以对列索引进行类似的累计操作了:

[33]:

```
data_mean.mean(axis=1, level='type')
```

[33]:

| type | HR | Temp |
|---|---|---|
| year | | |
| 2013 | 34.833333 | 36.966667 |
| 2014 | 37.666667 | 37.233333 |

通过这两行数据, 就可以获取每一年所有人的平均心率和体温了. 这种语法其实就是 `GroupBy` 功能的快捷方式, 将在 3.9 节详细介绍. 尽管这只是一个简单的示例, 但是其原理和实际工作中遇到的情况类似.

## 3.6.6 Panel数据

这里还有一些 Pandas 的基本数据结构没有介绍到, 包括 `pd.Panel` 对象和 `pd.Panel4D` 对象. 这两种数据结构可以分别看成是(一维数组) `Series` 和(二维数组) `DataFrame` 的三维与四维形式. 如果熟悉 `Series` 和 `DataFrame` 的使用方法, 那么 `Panel` 和 `Panel4D` 使用起来也会很简单, `ix`、`loc` 和 `iloc` 索引器(详情请参见 3.3 节)在高维数据结构上的用法更是完全相同.

但是本书并不打算进一步介绍这两种数据结构, 我个人认为多级索引在大多数情况下都是更实用、更直观的高维数据形式. 另外, `Panel` 采用密集数据存储形式, 而多级索引采用稀疏数据存储形式. 在解决许多真实的数据集时, 随着维度的不断增加, 密集数据存储形式的效率将越来越低. 但是这类数据结构对一些有特殊需求的应用还是有用的. 如果想对 `Panel` 与 `Panel4D` 数据结构有更多的认识, 请参见 3.14 节.