第一部分 基础知识

第1章 预备知识

1.1 Python基础

1.1.1 推导式

1. 反映两个对象之间的映射关系

```
[i ** 2 for i in range(5)]  # 列表推导式
{i ** 2 for i in range(5)}  # 集合推导式
{i: i ** 2 for i in range(5)}  # 字典推导式
```

2. **生成器表达式**

当需要把列表等作为中间对象来进行操作时，考虑使用生成器节省内存空间

```
sum(i ** 2 for i in range(5))  # 生成器
```

3. **嵌套**

- 单个推导式存在多个迭代

```
# 多个迭代，按从左到右决定先后顺序，靠前的在外层
[(i, j) for i in ['桃', '星', '梅', '方'] for j in range(2, 6)]  # 结果是一个列表

# if在for前面
[(i, j) if j % 2 == 0 else None for i in ['桃', '星', '梅', '方'] for j in range(2, 6)]
```

- 推导式嵌套推导式

```
[[(i, j) for i in ['桃', '星', '梅', '方']] for j in range(2, 6)]  # 结果是列表内嵌列表
```

- 带`if`判断

```
L = [1, 2, 3, 4, 5, 6, 7]
[i if i <= 5 else 5 for i in L]  # [1, 2, 3, 4, 5, 5, 5]
[i for i in L if i <= 5]  # [1, 2, 3, 4, 5]
```

1.1.2 匿名函数

lambda后紧跟形参，只能使用表达式，不能赋值、循环与选择语句，但可以借助推导式和条件赋值来实现

```
lst = [[1, 2], [3, 4, 5], [6], [7, 8], [9]]
def func(x):
    flag = False
    for i in x:
        if i % 3 == 0:
            flag = True
    return flag

list(filter(func, lst))  # 包含3的整数倍的内层列表
list(filter(lambda x: sum(1 if i % 3 == 0 else 0 for i in x) > 0, lst))
```

1.1.3 打包函数

同时遍历两个迭代器相同位置的元素

```
l1 = list('abc')
l2 = ['apple', 'banana', 'cat']
res = list(zip(l1, l2))  # 打包
list(zip(*res))  # 解包

for i, j in enumerate(l1):
    print(i, j)

for i, j in zip(range(len(l1)), l1):
    print(i, j)
```

1.2 NumPy基础

1.2.1 NumPy数组的构造

常用：`np.array([1, 2])`

1. 等差数列

   `np.linspace()`、`np.arange()`

2. 特殊矩阵

   `np.zeros()`、`np.ones()`、`np.eye()`、`np.full()`

   `np.zeros_like()`、`np.ones_like()`、`np.full_like()`

3. 随机数组

   - 均匀分布**[low, high)**: `np.random.uniform(low=0, high=1, size=(2, 3))`
     - 均匀0-1分布**[0, 1)**: `np.random.rand(2, 3, 4)`
   - 正态分布$N[\mu, \sigma^2]$: `np.random.normal(loc=0, scale=1, size=(2, 3))`
     - 标准正态分布$N[0, 1]$: `np.random.randn(2, 3, 4)`

- 离散均匀分布整数数组: `np.random.randint(low=0, high=1, size=(2, 3))`
- 有放回抽样: `np.random.choice(a=range(5), size=(2, 3), p=[0.1, 0.2, 0.3, 0.2, 0.2])`
  - 无放回抽样: `np.random.choice(a=range(5), size=(2, 3), replace=False`
- 打散: `np.random.permutation([1, 2, 3])`
  - 等价于无放回抽样列表所有元素: `np.random.choice([1, 2, 3], 3, replace=False)`

4. 随机种子: `np.random.seed(7)`

1.2.2 NumPy数组的变形

1. 数组元素组织方式变化导致的变形
   - 维度交换

```
arr = np.arange(6).reshape(1, 2, 3)
np.transpose(a=arr, axes=(1, 2, 0))  # 原来的1维放到现在的0维，原来的2维放到1维

arr.T == np.transpose(2, 1, 0) == np.transpose(arr)  # 默认维度逆向交换

# 交换两个维度，np.transpose()需要写出所有维度
np.swapaxes(a=arr, axis1=1, axis2=0)
```

   - 维度变换

```
# reshape默认以行的顺序来读写，order='C'
arr = np.arange(6).reshape(1, 2, 3, order='F')  # 列优先

# 维度增加
arr = np.arange(6).reshape(2, 3)
expand_arr = np.expand_dims(a=arr, axis=(0, 1, 4))  # (1, 1, 2, 3, 1)
(arr.reshape(1, 1, 2, 3, 1) == expand_arr).all()
arr[np.newaxis, np.newaxis, :, :, np.newaxis] == expand_arr
# 维度缩减
np.squeeze(expand_arr, axis=(0, 1)).shape  # (2, 3, 1)，默认缩减所有=1的维度
```

2. 数组合并或拆分导致的变形
   - 合并`np.stack()`、`np.concatenate()`

```
# np.stack()，拼接的数组必须尺寸相同，且产生新的维度，axis决定新维度在哪产生，维度大小由拼接的数组数量决定
arr1 = np.arange(12).reshape(4, 3)
arr2 = np.arange(12, 24).reshape(4, 3)
np.stack([arr1, arr2], axis=0).shape  # (2, 4, 3)
np.stack([arr1, arr2], axis=1).shape  # (4, 2, 3)
np.stack([arr1, arr2], axis=2).shape  # (4, 3, 2)

# np.concatenate只需在拼接的维度上一样即可
np.concatenate([arr1, arr2], axis=2)  # AxisError: axis 2 is out of bounds for array of dimension 2
```

- 拆分`np.split()`

```
# np.split()，indices_or_sections为整数表示均分，为一维序列，表示沿着axis用索引切割
# indices_or_sections=[2, 3]表示：arr[:2]、arr[2:3]、arr[3:]
np.split(arr, indices_or_sections=[1, 2], axis=0)
```

- 重复`np.repeat()`

```
# np.repeat()，沿着axis对数组按照给定次数进行重复
arr = np.arange(6).reshape(2, 3)
np.repeat(a=arr, repeats=[2, 3], axis=0)
np.repeat(a=arr, repeats=[2, 3, 1], axis=1)  # repeats列表的长度必须与arr的轴的长度一致
```

1.2.3 NumPy数组的切片

```
# 输入切片，取子数组
arr = np.arange(24).reshape(4, 2, 3)
arr[0:2, 0:2, 0:2]  # 取2*2*2子数组

# 输入长度相同的列表，不是取子数组，而是取元素，输入值表示元素在各个维度的索引
arr[[0, 1], [0, 1]]  # 取出arr[0, 0]和arr[1, 1]
arr[[0, 1], [0, 1], [0, 1]]  # 取出arr[0, 0, 0]和arr[1, 1, 1]

# 输入布尔数组，保留某一维度的若干维数
arr[[True, False, True, False], :, :]

# 最后几个维度的:可以忽略
arr[[True, False, True, False], :, :] == arr[[True, False, True, False]]
arr[[0, 0, 0], [1, 1, 1], :] == arr[[0, 0, 0], [1, 1, 1]]

# 最初几个维度的:可以用...代替
arr[:, :, 0:2] == arr[..., 0:2]
```

1.2.4 广播机制

数组A的维度：$d_p^A \times \cdots \times d_1^A$

数组B的维度：$d_q^A \times \cdots \times d_1^A$

设$r = max(p, q)$

- 首先对数组维度小的数组补充维度：在数组**前面**补充，维数=1
- 对比两个数组，对维数=1的维度进行复制扩充，维数=另一个数组相同位置的维数
- 当相同位置的维数不相等，且任一维数都$\neq 1$，则报错

1. 标量和数组的广播

   当一个标量和数组进行运算时，标量会自动把大小扩充为数组大小，之后进行逐元素操作

2. 二维数组之间的广播

   除非其中的某个数组的维度是$m \times 1$或者$1 \times n$，扩充其具有1的维度为另一个数组对应维度的大小，否则报错

3. 一维数组与二维数组的广播

   当一维数组A_k与二维数组$B_{m,n}$操作时，等价于把一维数组视作$A_{1,k}$的二维数组，当$k! = n$且k, n都不是1时报错

```
np.ones(3) + np.ones((2,3))  # OK
np.ones(2) + np.ones((2,3))  # 报错
```

1.2.5 常用函数

1. **聚合函数**

   统计函数包括`max, min, mean, median, std, var, sum, quantile`，其中**分位数计算**是**全局方法**，因此只能通过`np.median()`和`np.quantile()`的方法调用

   如果数组中包含`nan`，结果也为`nan`。若要忽略`nan`进行计算，则使用**全局方法**`np.nanmax()`

2. **相关性计算**
   - 协方差：`np.cov(arr1, arr2)`
   - 相关性系数：`np.corrcoef(arr1, arr2)`
3. ufunc**函数**

   逐元素处理，**全局方法**

   `np.cos、sin、tan、arccos、arcsin、arctan、abs、sqrt、power、exp、log、log10、log2、ceil、floor()`

4. **逻辑函数**
   - **比较**：`<、>、<=、>=、!=、==`
   - **内置函数**：`isnan()、isinf()、all()、any()`
   - **逻辑运算符**：`~、&、|`
     - 优先级：`非not > 与and > 或or`
   - **填充函数**：`np.where(bool_arr, fill_arr_for_True, fill_arr_for_False)`

如果后两个待填充的值与第一个数组维度不匹配且符合广播条件，则会被广播，之后使用相应位置的值来填充

```
arr = np.arange(4).reshape(-1, 2)
np.where(arr > 0, arr.sum(0), arr.sum(1))
```

- **截断函数**：`np.clip(arr, min, max)`

```
np.clip(arr, 1, 2)
# 等价于
arr1 = np.where(arr > 2, 2, arr)
np.where(arr1 < 1, 1, arr1)
```

5. **返回索引的函数**
   - 返回**非零**、**最大**、**最小**值所在的索引：`np.nonzero()、argmax()、argmin()`
6. **累计函数**
   - `cumprod`，`cumsum`分别表示累乘和累加函数，返回同长度的数组
   - `diff`表示和前一个元素做差，由于第一个元素为缺失值，因此在默认参数情况下，返回长度是原数组减1
7. **向量内积**

   `a.dot(b)`
8. **向量、矩阵范数**

   `np.linalg.norm()`

ord可选参数	矩阵范数	向量范数
None	Frobenius norm	2-norm
'fro'	Frobenius norm	/
'nuc'	nuclear norm	/
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	/	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	/	sum(abs(x)**ord)**(1./ord)

9. **矩阵乘法**

   `a @ b`

10. **卡方统计量**

    设矩阵$A_{m\times n}$，记$B_{ij}=\frac{(\sum_{i=1}^{m}A_{ij})\times(\sum_{j=1}^{n}A_{ij})}{\sum_{i=1}^{m}\sum_{j=1}^{n}A_{ij}}$，定义卡方值如下：$\chi^2=\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{(A_{ij}-B_{ij})^2}{B_{ij}}$

```
A = np.random.randint(10, 20, (8, 5))
B = A.sum(0) * A.sum(1)[:, None] / A.sum()
chi_val = ((A - B) ** 2 / B).sum()
```

第2章 pandas基础

2.1 文件的读取与写入

2.1.1 文件读取

一些常用的公共参数，`header=None`表示第一行不作为列名，`index_col`表示把某一列或几列作为索引，`usecols`表示读取列的集合，默认读取所有的列，`parse_dates`表示需要转化为时间的列，`nrows`表示读取的数据行数。这些参数在`read_csv、read_table、read_excel`里都可以使用

在读取`txt`文件时，经常遇到分隔符非空格的情况，`read_table`有一个**分割参数**`sep`，它使得用户可以自定义分割符号，进行`txt`数据的读取。参数`sep`中使用的是**正则表达式**，因此需要对诸如`|`等进行转义变成`\|`，否则无法读取到正确的结果。

2.1.2 数据写入

`pandas`中没有定义`to_table`函数，但是`to_csv`可以保存为`txt`文件，并且允许自定义分隔符，常用制表符`\t`分割。

如果想要把表格快速转换为`markdown`和`latex`语言，可以使用`to_markdown`和`to_latex`函数，此处需要安装`tabulate`包。

2.2 基本数据结构

2.2.1 Series

`Series`一般由四个部分组成，分别是序列的值`data`、索引`index`、存储类型`dtype`、序列的名字`name`。其中，索引也可以指定它的名字，默认为空。

```
s = pd.Series(data = [100, 'a', {'dic1':5}],
              index = pd.Index(['id1', 20, 'third'], name='my_idx'),  # 指定索引名
              dtype = 'object',
              name = 'my_name')
s.index.name = 'my_idx'  # 显式指定索引名
```

常用的`dtype`还有`int、float、string、category`

`object`代表了一种混合类型，目前`pandas`把纯字符串序列也默认认为是一种`object`类型的序列，但它也可以显式指定`string`类型存储

对于这些属性，可以通过 . 的方式来获取。`.values、index、dtype、name、shape`

2.2.2 DataFrame

`DataFrame`在`Series`的基础上增加了列索引，一个数据框可以由二维的`data`与行列索引来构造。

但一般而言，更多的时候会采用从列索引名到数据的映射来构造数据框，同时再加上行索引。

```
df = pd.DataFrame(data = {'col_0': [1,2,3],
                          'col_1':list('abc'),
                          'col_2': [1.2, 2.2, 3.2]},
                  index = ['row_0', 'row_0', 'row_0']  # 不报错
                  )
```

如果字典中'col_0'对应的是Series：

- Series与df索引一致，Series的值直接填入
- Series与df索引不一致，且Series索引不重复，当前df的行索引在Series未出现则填入nan值
- Series与df索引不一致，且Series索引重复，报错

在`DataFrame`中可以用`[col_name]`与`[col_list]`来取出相应的列与由多个列组成的表，结果分别为`Series`和`DataFrame`。如`df[col_0]`是**Series**，`df[[col_0]]`是**DataFrame**。

Series转换为DataFrame：`s.to_frame()`

对于df属性，可以通过 . 的方式来获取。`.values、index、columns、dtypes、shape`，与Series相比，没有`name`属性，多了`columns`属性，且dtype是复数带s

- 增加或修改一列时，直接使用`df[col_name]`
- 删除一列时，使用`drop()`。绝大多数方法都不会直接改变原df，而是返回一个拷贝

2.3 常用基本函数

2.3.1 汇总函数

`head, tail`函数分别表示返回表或者序列的前`n`行和后`n`行，其中`n`默认为5

`info, describe`分别返回表的信息概况和表中数值列对应的主要统计量。`info, describe`只能实现较少信息的展示，如果想要对一份数据集进行全面且有效的观察，特别是在列较多的情况下，推荐使用[pandas-profiling]包

2.3.2 特征统计函数

- 常见的是`sum, mean, median, var, std, max, min`
- `quantile, count, idxmax, idxmin`，它们分别返回的是分位数、非缺失值个数、最大/最小值对应的索引

上面这些所有的函数，由于操作后返回的是标量，所以又称为**聚合函数**，它们有一个公共参数`axis`，默认为0代表逐列聚合，如果设置为1则表示逐行聚合

2.3.3 频次函数

- 对序列`Series`使用`unique`和`nunique`可以分别得到其唯一值组成的列表和唯一值的个数
- 对序列`Series`使用`value_counts`可以得到唯一值和其对应出现的频数
- 如果想要观察`DataFrame`多个列组合的唯一值，可以使用`drop_duplicates`。其中的关键参数是`keep`，默认值`first`表示每个组合保留第一次出现的所在行，`last`表示保留最后一次出现的所在行，`False`表示把所有重复组合所在的行剔除
- `duplicated`和`drop_duplicates`的功能类似，但前者返回了是否为唯一值的布尔列表，其`keep`参数与后者一致

2.3.4 替换函数

一般而言，替换操作是针对某一个列进行的，因此下面的例子都以`Series`举例。

`pandas`中的替换函数可以归纳为三类：映射替换、逻辑替换、数值替换

- 在`replace`中，可以通过字典构造，或者传入两个列表来进行替换

```
df['Gender'].replace({'Female':0, 'Male':1})  # 字典
df['Gender'].replace(['Female', 'Male'], [0, 1])
```

- `replace`还有一种特殊的方向替换，指定`method`参数为`ffill`则为用前面一个最近的未被替换的值进行替换，`bfill`则使用后面最近的未被替换的值进行替换

```
s = pd.Series(['a', 1, 'b', 2, 1, 1, 'a'])
s.replace([1, 2], method='ffill')  # 1、2都被替换
```

- **逻辑替换**。包括了`where`和`mask`，这两个函数是完全对称的：`where`函数在传入条件为`False`的对应行进行替换，而`mask`在传入条件为`True`的对应行进行替换，当不指定替换值时，替换为缺失值。传入的条件只需是与被调用的`Series`索引一致的**布尔序列**即可
- **数值替换**。包含了`round, abs, clip`方法，它们分别表示按照给定精度四舍五入、取绝对值和截断

```
s.clip(0, 2)  # 分别表示上下截断边界
```

2.3.5 排序函数

1. **值排序**`sort_values`

   在排序中，经常遇到多列排序的问题，比如在体重相同的情况下，对身高进行排序，并且保持身高降序排列，体重升序排列

```
df_demo.sort_values(['Weight','Height'],ascending=[True,False])
```

2. **索引排序**`sort_index`

   索引排序的用法和值排序完全一致，只不过元素的值在索引中，此时需要用参数`level`指定索引层的名字或者层号。另外，需要注意的是字符串的排列顺序由字母顺序决定。

```
df_demo.sort_index(level=['Grade','Name'],ascending=[True,False])
```

3. **元素排序**`rank()`

   `Series.rank(ascending=True, pct=False, method='average')`

   `pct`是否返回元素对应的分位数

   `method`可能的参数：

   - 'min'/'max'：最小/大的可能排名
   - 'first'：先后排名
   - 'dense'：排名相差1

2.3.6 apply函数

`apply`方法常用于`DataFrame`的行迭代或者列迭代，`apply`的参数往往是一个以序列为输入的函数。

`apply`的自由是牺牲性能换来的，只有当不存在内置函数且迭代次数较少时，才使用`apply`

```
# 可以利用`lambda`表达式使得书写简洁，这里的`x`就指代被调用的`df_demo`表中逐个输入的序列
%timeit -n 100 -r 7 df_demo.apply(lambda x:x.mean())  # -r表示运行轮数(runs)，-n表示每轮运行次数(loops)
```

2.4 窗口

2.4.1 滑动窗口

要使用滑窗函数，就必须先要对一个序列使用`.rolling`得到滑窗对象，其最重要的参数为窗口大小`window`。需要注意的是窗口包含当前行所在的元素

- `min_periods`：参与计算的最小样本量，默认=window，必须满足`min_periods`<=`window`

类滑窗函数

`shift, diff, pct_change`是一组类滑窗函数，它们的公共参数为`periods=n`，默认为1。它们的功能可以用窗口大小为`n+1`的`rolling`方法等价代替

- `shift`：取**向前**第`n`个元素的值，**n为负表示向后**

```
s.shift(n) == s.rolling(n + 1).apply(lambda x: list(x)[0])  # n为正
s.shift(n) == s[::-1].rolling(-n + 1).apply(lambda x: list(x)[0])[::-1]  # n为负
```

- `diff`：与向前第`n`个元素做差

```
s.diff(n) == s.rolling(n + 1).apply(lambda x:list(x)[-1]-list(x)[0])
```

- `pct_change`：与向前第`n`个元素相比计算增长率

```
s.pct_change() == s.rolling(2).apply(lambda x: list(x)[-1] / list(x)[0] - 1)
```

2.4.2 扩张窗口

扩张窗口又称累计窗口，可以理解为一个动态长度的窗口，其窗口的大小就是从序列开始处到具体操作的对应位置，其使用的聚合函数会作用于这些逐步扩张的窗口上。具体地说，设序列为a1, a2, a3, a4，则其每个位置对应的窗口即[a1]、[a1, a2]、[a1, a2, a3]、[a1, a2, a3, a4]

- `cummax`：`s.expanding().apply(lambda x: x.max())`
- `cumsum`：`s.expanding().apply(lambda x: x.sum())`
- `cumprod`：`s.expanding().apply(lambda x: x.prod())`

2.4.3 指数加权窗口

在扩张窗口中，可以使用各类函数进行历史的累计指标统计，但给窗口中的所有函数赋予了同样的权重。若给窗口中的元素赋予不同的权重，则此时为**指数加权窗口**。

最重要的参数是α，窗口权重为$\omega_i = (1-\alpha)^{t-i}$，序列第一个元素$x_0$距当前元素$x_t$最远，其权重最小：$\omega_0 = (1-\alpha)^t$

加权并归一化：

$$
\begin{aligned}
y_t &= \frac{\sum_{i=0}^{t} w_i x_{t-i}}{\sum_{i=0}^{t} w_i} \\
&= \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + \ldots + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + \ldots (1-\alpha)^t}
\end{aligned}
$$

```
s.ewm(alpha=0.2).mean()  # 调用ewm函数

def ewm_func(x, alpha=0.2):
    win = (1 - alpha) ** np.arange(x.shape[0])
    win = win[::-1]
    res = (win * x).sum() / win.sum()
s.expanding().apply(ewm_func)
```

第二部分 4类操作

第3章 索引

3.1 单级索引

3.1.1 DataFrame的列索引

列索引是最常见的索引形式，一般通过`[]`来实现。通过`[列名]`可以从`DataFrame`中取出相应的列，返回值为`Series`。

如果要取出多个列，则可以通过`[列名组成的列表]`，其返回值为一个`DataFrame`。

3.1.2 Series的行索引

1. 以字符串为索引的`Series`，索引可重复
    - 如果取出单个索引的对应元素，则可以使用`[item]`，若`Series`只**有单个值对应**，则返回这个**标量**值，如果有**多个值对应**，则返回一个`Series`
    - 如果取出多个索引的对应元素，则可以使用`[items的列表]`
    - 如果想要取出某两个索引之间的元素，并且这两个索引在整个索引中**唯一出现**，则**可以使用切片**，同时需要注意这里的切片会包含两个端点
    - 如果这两个索引在整个索引中**重复出现**，那么需要经过排序才能使用切片
2. 以整数为索引的`Series`，索引可重复
    - 和字符串一样，如果使用`[int]`或`[int_list]`，则可以取出对应索引元素的值
    - 如果使用整数切片，则会取出对应索引位置的值，注意这里的整数切片同`Python`中的切片一样**不包含右端点**

3.1.3 loc索引器

基于元素的`loc`索引器的一般形式是`loc[*, *]`，其中第一个`*`代表行的选择，第二个`*`代表列的选择，如果省略第二个位置写作`loc[*]`，这个`*`是指行的筛选。

`*`的位置一共有五类合法对象，分别是：单个元素、元素列表、元素切片、布尔列表以及函数

1. `*`**为单个元素**

   直接取出相应的行或列，如果该元素在索引中重复则结果为`DataFrame`，否则为`Series`

   也可以同时选择行和列，返回`Series`或标量

2. `*`**为元素列表**

   取出列表中所有元素值对应的行或列

3. `*`**为元素切片**

   字符串/整数索引，如果是**唯一值**的起点和终点字符，那么就可以使用切片，并且包含两个端点，如果**不唯一则报错**

4. `*`**为布尔列表**

   传入`loc`的布尔列表与`DataFrame`长度相同，且列表为`True`的位置所对应的行会被选中，`False`则会被剔除

5. `*`**为函数**

   这里的函数，必须以前面的四种合法形式之一为返回值，并且函数的输入值为`DataFrame`本身

```
def condition(x):
    condition_1_1 = x.School == 'Fudan University'
    condition_1_2 = x.Grade == 'Senior'
    condition_1_3 = x.Weight > 70
    condition_1 = condition_1_1 & condition_1_2 & condition_1_3
    condition_2_1 = x.School == 'Peking University'
    condition_2_2 = x.Grade == 'Senior'
    condition_2_3 = x.Weight > 80
    condition_2 = condition_2_1 & (~condition_2_2) & condition_2_3
    result = condition_1 | condition_2
    return result
df_demo.loc[condition]
```

- 由于函数无法返回如`start: end: step`的切片形式，故返回切片时要用`slice`对象进行包装

```
df_demo.loc[lambda x: slice('Gaojuan You', 'Gaoqiang Qian')]
```

- 在对表或者序列赋值时，应当在使用**一层索引器后直接进行赋值**操作，这样做是由于进行**多次索引后赋值是赋在临时返回的`copy`副本上**的，而没有真正修改元素，从而报出`SettingWithCopyWarning`警告

```
df_chain = pd.DataFrame([[0,0],[1,0],[-1,0]], columns=list('AB'))
df_chain
import warnings
with warnings.catch_warnings():
    warnings.filterwarnings('error')
    try:
        df_chain[df_chain.A!=0]['B'] = 1 # 使用方括号列索引后，再使用一次列索引，共使用2层索引
    except Warning as w:
        Warning_Msg = w
print(Warning_Msg)
df_chain  # 没有任何变化，因为赋值在临时副本上

df_chain.loc[df_chain.A!=0,'B'] = 1  # 只有一层索引
```

3.1.4 iloc索引器

`iloc`的使用与`loc`完全类似，只不过是针对位置进行筛选，在相应的`*`位置处一共也有五类合法对象，分别是：整数、整数列表、整数切片、布尔列表以及函数，函数的返回值必须是前面的四类合法对象中的一个，其输入同样也为`DataFrame`本身

- 与`loc`不同，`iloc`整数切片不包含结束端点
- 在使用布尔列表的时候要特别注意，不能传入`Series`而必须传入`Series`的`values`，否则会报错`ValueError: iLocation based boolean indexing cannot use an indexable as a mask`
- 当仅需索引单个元素时，可使用性能更好的`.at[low, col]`和`.iat[row_pos, col_pos]`

3.1.5 query()函数

1. 把字符串形式的查询表达式传入`query`方法来查询数据，其表达式的执行结果必须返回**布尔列表**。
2. 在`query`表达式中，帮用户注册了所有来自`DataFrame`的列名，所有属于该`Series`的方法都可以被调用，和正常的函数调用并没有区别。对于**含有空格的列名**，需要使用`` `col name` ``的方式进行引用。
3. 同时，在`query`中还注册了若干英语的字面用法，帮助提高可读性，例如：`or, and, or, in, not in`
4. 此外，在字符串中出现与列表的比较时，`==`和`!=`分别表示元素出现在列表和没有出现在列表，等价于`in`和`not in`
5. 对于`query`中的字符串，如果要引用外部变量，只需在变量名前加`@`符号

```
low, high =70, 80
df.query('(Weight >= @low) & (Weight <= @high)')
```

3.1.6 索引运算

筛选出两个表索引的交集/并集

1. $S_A.\text{intersection}(S_B) = S_A \cap S_B \Leftrightarrow \{x|x \in S_A \text{ and } x \in S_B\}$
2. $S_A.\text{union}(S_B) = S_A \cup S_B \Leftrightarrow \{x|x \in S_A \text{ or } x \in S_B\}$
3. $S_A.\text{difference}(S_B) = S_A - S_B \Leftrightarrow \{x|x \in S_A \text{ and } x \notin S_B\}$
4. $S_A.\text{symmetric\_difference}(S_B) = S_A \triangle S_B \Leftrightarrow \{x|x \in S_A \cup S_B - S_A \cap S_B\}$

由于索引的元素可能重复，而集合的元素要求互异，因此**先去重**，再计算

- 交集：`id1.intersection(id2)`
- 并集：`id1.union(id2)`
- 差集：`id1.difference(id2)`
- 异或集：`id1.symmetric_difference(id2)`

3.1.7 随机抽样

如果把`DataFrame`的每一行看作一个样本，或把每一列看作一个特征，再把整个`DataFrame`看作总体，想要对样本或特征进行随机抽样就可以用`sample`函数

`sample`函数中的主要参数为`n, axis, frac, replace, weights`，前三个分别是指抽样数量、抽样的方向（0为行、1为列）和抽样比例（0.3则为从总体中抽出30%的样本）。`replace`和`weights`分别是指是否放回和每个样本的抽样相对概率，当`replace = True`则表示有放回抽样。

```python
'''使用iloc实现sample函数'''
def sample(df, n=None, frac=None, replace=None, weights=None, random_state=None, axis=None):
    temp_df = df.copy()
    if n != None and frac != None:
        raise ValueError("n和frac只能存在一个")
    if n == None:
        n = int(df.shape[0] * frac)
    if isinstance(weights, list):
        weights = np.ones(df.shape[axis]) / df.shape[axis]
    idx = np.random.choice(range(df.shape[axis]), size=n, replace=replace, p=weights/weights.sum())
    return temp_df.iloc[:, idx] if axis else temp_df.iloc[idx]
```

3.2 多级索引

3.2.1 多级索引及其表结构

- 行索引和列索引都是`MultiIndex`类型，只不过索引中的一个元素是**元组**而不是单层索引中的标量。
- 索引的名字和值属性分别可以通过`names`和`values`获得。
- 如果想要得到**某一层的索引**，则需要通过`get_level_values`获得
- 对于索引而言，无论是单层还是多层，用户都无法通过`index_obj[0] = item`的方式来修改元素。

```python
ind[0] = 'Gaopeng Yang'  # TypeError: Index does not support mutable operations
ind[0] = ('A', 'Female')  # TypeError: Index does not support mutable operations
```

- 多层索引，不能通过`index_name[0] = new_name`的方式来修改名字（多层索引名是FrozenList类型，单层索引名是标量）

```python
ind.names[0] = 'school'  # TypeError: 'FrozenList' does not support mutable operations.
ind.names = ['school', 'gender']  # 整体赋值，OK
```

3.2.2 多级索引中的loc索引器

对索引元组整体进行切片

- 由于多级索引中的单个元素以**元组**为单位，因此之前在第一节介绍的`loc`和`iloc`方法完全可以照搬，只需把标量的位置替换成对应的元组。当传入元组列表或单个元组或返回前二者的函数时，需要先进行**索引排序**以避免性能警告`PerformanceWarning: indexing past lexsort depth may impact performance.`
- 当使用**切片**时需要注意，在单级索引中只要切片端点元素是唯一的，那么就可以进行切片，但在多级索引中，无论元组在索引中是否重复出现，都**必须经过排序**才能使用切片，否则报错。`UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (0)'`

- 在多级索引中的元组有一种特殊的用法，可以对多层的元素进行交叉组合后索引，但同时需要指定`loc`的列，全选则用`:`表示。其中，每一层需要选中的元素用列表存放，传入`loc`的形式为`[(level_0_list, level_1_list), cols]`

对索引元组单层进行切片

需要引入`IndexSlice`对象，`Slice`对象一共有两种形式，第一种为`loc[idx[*,*]]`型，第二种为`loc[idx[*,*],idx[*,*]]`型。

定义：`idx = pd.IndexSlice`

1. **`loc[idx[*,*]]`型**

   不能进行多层分别切片，前一个`*`表示行的选择，后一个`*`表示列的选择，与单纯的`loc`是类似的。也支持布尔序列的索引

```
df_ex.loc[idx['C':, ('D', 'f'):]]
df_ex.loc[idx[:'A', lambda x:x.sum()>0]] # 列和大于0
```

2. **`loc[idx[*,*],idx[*,*]]`型**

   能够分层进行切片，前一个`idx`指代的是行索引，后一个是列索引。但需要注意的是，此时不支持使用函数

```
df_ex.loc[idx[:'A', 'b':], idx['E':, 'e':]]  # OK

try:
    df_ex.loc[idx[:'A', lambda x: 'b'], idx['E':, 'e':]]  # KeyError(<function __main__.<lambda>(x)>)
except Exception as e:
    Err_Msg = e
Err_Msg
```

3.2.3 多级索引的构造

除了使用`set_index`之外，常用的有`from_tuples, from_arrays, from_product`三种方法，它们都是`pd.MultiIndex`对象下的函数。

1. **`from_tuples`**指根据传入由元组组成的列表进行构造

```
my_tuple = [('a','cat'),('a','dog'),('b','cat'),('b','dog')]
pd.MultiIndex.from_tuples(my_tuple, names=['First','Second'])
```

2. **`from_arrays`**指根据传入列表中，对应层的列表进行构造

```
my_array = [list('aabb'), ['cat', 'dog']*2]
pd.MultiIndex.from_arrays(my_array, names=['First','Second'])
```

3. **`from_product`**指根据给定多个列表的笛卡尔积进行构造

```
my_list1 = ['a','b']
my_list2 = ['cat','dog']
pd.MultiIndex.from_product([my_list1, my_list2], names=['First','Second'])
```

3.3 常用索引方法

3.3.1 索引层的交换和删除

行或列索引**内部的交换**由`swaplevel`和`reorder_levels`完成，前者只能交换两个层，而后者可以交换任意层，两者都可以指定交换的是轴是哪一个，即行索引或列索引。

```
df_ex.swaplevel(0,2,axis=1)  # 列索引的第一层和第三层交换
df_ex.reorder_levels([2,0,1],axis=0)  # 列表数字指代原来索引中的层，原来的第2层变成第0层
```

若想要删除某一层的索引，可以使用`droplevel`方法

```
df_ex.droplevel([0,1],axis=0)
```

3.3.2 索引属性的修改

1. 通过`rename_axis`可以对**索引层的名字**进行修改，常用的修改方式是传入字典的映射。传入参数也可以是函数，其输入值就是索引层名字

   ```
   df_ex.rename_axis(index={'Upper':'Changed_row'}, columns=
   {'Other':'Changed_Col'})  # 字典
   df_ex.rename_axis(index=lambda s: 'Changed_row' if s == 'Upper' else s)  #
   函数
   ```

2. 通过`rename`可以对**索引的值**进行修改，如果是多级索引需要指定修改的层号`level`。传入参数也可以是函数，其输入值就是索引元素

   ```
   df_ex.rename(columns={'cat':'not_cat'}, level=2)
   df_ex.rename(index=lambda x:str.upper(x), level=2)
   ```

3. 替换某一层的整个索引元素
   - 迭代器

     ```
     df_ex.rename(index=list('abcdefgh'), level=2)  # TypeError: 'list'
     object is not callable
     new_values = iter(list('abcdefgh'))
     df_ex.rename(index=lambda x: next(new_values), level=2)  # OK
     ```

4. 修改索引的某个值
   - 单层索引

     先取出索引的`values`属性，再给对得到的列表进行修改，最后再对`index`对象重新赋值

```
df_demo.index.values[0] = 'Gaopeng Yang'
```

- 多层索引

  定义在`Index`上的`map`方法，与前面`rename`方法是类似的，只不过它传入的不是整个层（列），而是直接传入索引的元组（行）

```
new_idx = df_temp.index.map(lambda x: (x[0], x[1], x[2].upper()))  # 传入的是索引元组
df_temp.index = new_idx

new_idx = df_temp.index.map(lambda x: (x[0]+'-'+x[1]+'-'+x[2]))  # 对多级索引压缩成单层索引
df_temp.index = new_idx

new_idx = df_temp.index.map(lambda x:tuple(x.split('-')))  # 对压缩的单层索引展开成多级索引
df_temp.index = new_idx

# 索引的names即使全是None，长度也是固定的，修改后的索引names长度应该保持一致
new_idx = df_temp.index.map(lambda x: (x[0]+'-'+x[2], x[1]))  # ValueError: Length of names must match number of levels in MultiIndex.
df_temp.index = df_temp.index.map(lambda x: (x[0]+'-'+x[2], x[1], x[2]))
df_temp.droplevel(-1, axis=0)  # 3层行索引，合并成2层
```

3.3.3 索引的设置与重置

1. 索引的设置可以使用`set_index`完成，这里的主要参数是`append`，表示是否来保留原来的索引，直接把新设定的添加到原索引的内层
2. 如果想要添加索引的列没有出现在`df`其中，那么可以直接在参数中传入相应的`Series`

```
my_index = pd.Series(list('WXYZ'), name='D')
df_new = df_new.set_index(['A', my_index])
```

3. `reset_index`是`set_index`的逆函数，其主要参数是`drop`，表示是否要把去掉的索引层丢弃，而不是添加到列中。如果重置了所有的索引，那么`pandas`会直接重新生成一个默认索引

3.3.4 索引的对齐

1. 在某些场合下，需要对索引做一些扩充或者剔除，更具体地要求是给定一个新的索引，把原表中相应的索引对应元素填充到新索引构成的表中，原来表中的数据和新表中会根据索引自动对齐。

```
df_reindex.reindex(index=['1001','1002','1003','1004'],
                   columns=['Weight','Gender'])
```

2. 还有一个与`reindex`功能类似的函数是`reindex_like`，其功能是仿照传入的表索引来进行被调用表索引的变形。

```
df_existed = pd.DataFrame(index=['1001','1002','1003','1004'],
                          columns=['Weight','Gender'])
df_reindex.reindex_like(df_existed)
```

第4章 分组

4.1 分组模式及其对象

4.1.1 分组的一般形式

分组操作，必须明确三个要素：分组依据、数据来源、操作及其返回结果

`df.groupby(分组依据)[数据来源].使用操作`

4.1.2 分组依据的本质

分组的依据来自于数据来源组合的unique值

4.1.3 groupby对象

`gb = df.groupby(['School', 'Grade'])`

1. `gb.ngroups`，通过`ngroups`属性，可以得到分组个数
2. `gb.groups`，通过`groups`属性，可以返回从组名映射到组索引列表的字典
3. `gb.size`，统计每个组的元素个数
4. `gb.get_group(组名)`，直接获取所在组对应的行

4.2 聚合函数

4.2.1 内置聚合函数

`max/min/mean/median/count/all/any/idxmax/idxmin/mad/nunique/skew/quantile/sum/std/var/sem/size/prod()`

4.2.2 agg函数

1. 使用多个函数

   用列表的形式把内置聚合函数对应的字符串传入。此时的列索引为多级索引，第一层为数据源，第二层为使用的聚合方法

   ```
   gb.agg(['sum', 'idxmax', 'skew'])
   ```

2. 对特定的列使用特定的聚合函数

   通过构造字典传入`agg`中实现，其中字典以列名为键，以聚合字符串或字符串列表为值

   ```
   gb.agg({'Height':['mean','max'], 'Weight':'count'})
   ```

3. 使用自定义函数

   传入函数的参数是之前数据源中的列，**逐列进行计算，不能对多列数据同时处理**。只需保证**返回值是标量**即可

```
gb.agg(lambda x: x.mean() - x.min())
```

4. 聚合结果重命名

   将上述函数的位置改写成元组，元组的第一个元素为新的名字，第二个位置为内置聚合函数/自定义函数

```
gb.agg([('range', lambda x: x.max() - x.min()), ('my_sum', 'sum')])
```

   另外需要注意，使用对一个或者多个列使用单个聚合的时候，重命名需要加括号，否则就不知道是新的名字还是手误输错的内置函数字符串

```
gb.agg([('my_sum', 'sum')])
```

4.3 变换和过滤

4.3.1 变换函数transform

变换函数的返回值为**同长度的序列**，最常用的内置变换函数是累计函数：`cumcount/cumsum/cumprod/cummax/cummin`

`cumcount`：按照元素的取值进行连续编号

当用自定义变换时需要使用`transform`方法，其传入值为数据源的序列，与`agg`的传入类型是一致的，其最后的返回结果是行列索引与数据源一致的`DataFrame`。

1. `groupby`对象不能使用多个变换函数
2. 无法像`agg`一样，通过传入字典，对特定的列使用特定的变换函数

```
def transform_helper(x):  # x代表一列，Series
    if x.name == 'A':  # Series传入函数时，自带名字
        return x + 1
    elif x.name == 'B':
        return x - 1

df.groupby('C').transform(transform_helper)
```

3. `transform`只能返回同长度的序列，但事实上还可以返回一个标量，这会使得结果被广播到其所在的整个组，这种标量广播的技巧在特征工程中是非常常见的。

```
gb.transform('mean')  # 传入返回标量的函数也是可以的
```

4.3.2 组索引与过滤filter

过滤在分组中是对于组的过滤，而索引中无论是布尔列表还是元素列表或者位置列表，本质上都是对于行的筛选

组过滤作为行过滤的推广，指的是如果对一个组的全体所在行进行统计的结果返回`True`则会被保留，`False`则该组会被过滤，最后把所有未被过滤的组其对应的所在行拼接起来作为`DataFrame`返回。

在`groupby`对象中，使用`filter`方法进行组的筛选，其中自定义函数的输入参数为数据源构成的`DataFrame`本身，只需保证自定义函数的返回为布尔值即可。

- 从概念上说，索引功能是组过滤功能的子集，可以使用`filter`函数完成`loc[...]`的功能

```
df.groupby(df.index).filter(lambda x: x.index[0] == item)  # 必须返回bool值
df.groupby(df.index).filter(lambda x: x.index[0] in item_list)
```

4.4 跨列分组

逐列处理：`agg`

返回序列：`transform`

过滤操作：`filter`

多列操作：`apply`

`apply`的自定义函数传入参数与`filter`完全一致，只不过后者只允许返回布尔值，而`apply`可返回标量、Series、DataFrame

1. 返回标量

   结果得到的是`Series`，索引与`agg`的结果一致

```
gb.apply(lambda x: 0)
```

2. 返回Series

   得到的是`DataFrame`，行索引与标量情况一致，列索引为`Series`的索引

```
gb.apply(lambda x: pd.Series([0,0],index=['a','b']))  # df的columns为['a', 'b']
```

3. 返回DataFrame

   得到的是`DataFrame`，行索引最内层在每个组原先`agg`的结果索引上，再加一层返回的`DataFrame`行索引，同时分组结果`DataFrame`的列索引和返回的`DataFrame`列索引一致。

```
gb.apply(lambda x: pd.DataFrame(np.ones((2,2)), index = ['a','b'], columns=pd.Index([('w','x'),('y','z')])))  # df最内层行索引为['a', 'b'], columns为
```

第5章 变形

5.1 长宽表的变形

长宽表是对元素序列而言的，如果元素序列以某个列的形式存在，则称该表是关于这个元素序列的**长表**，如果以列索引的形式存在，则称为**宽表**。

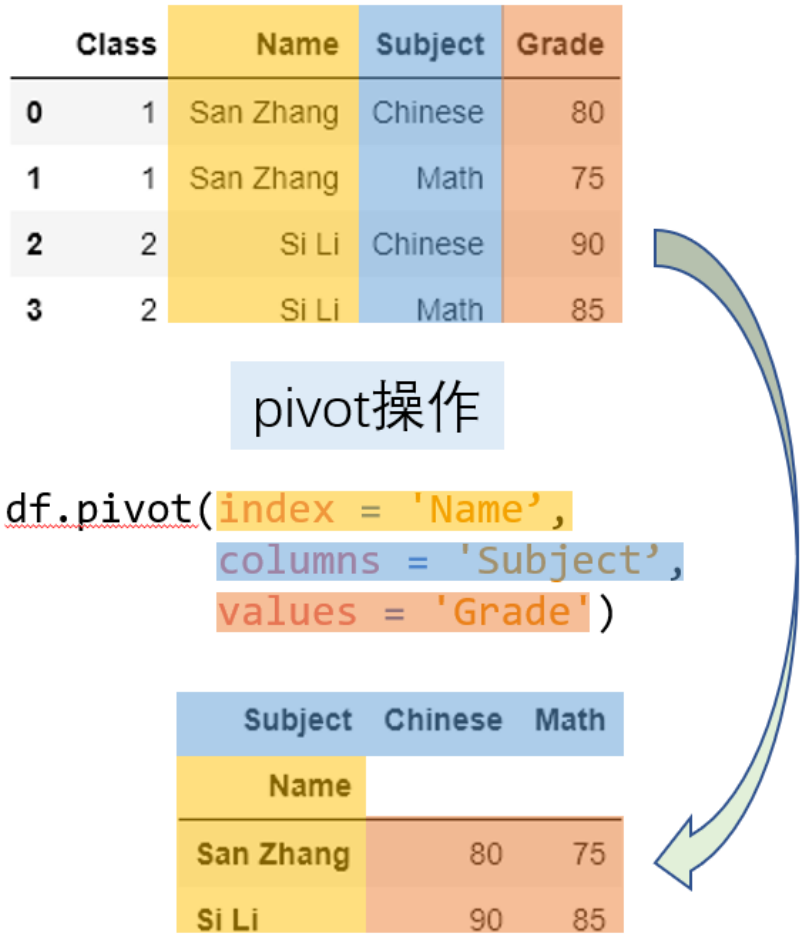
```
pd.DataFrame({'Gender':['F','F','M','M'], 'Height':[163, 160, 175, 180]})  # 性别，单独是一个列，长表
pd.DataFrame({'Height: F':[163, 160], 'Height: M':[175, 180]})  # 性别以列索引存在，宽表
```

5.1.1 长表的透视变形

1. **`pivot`**

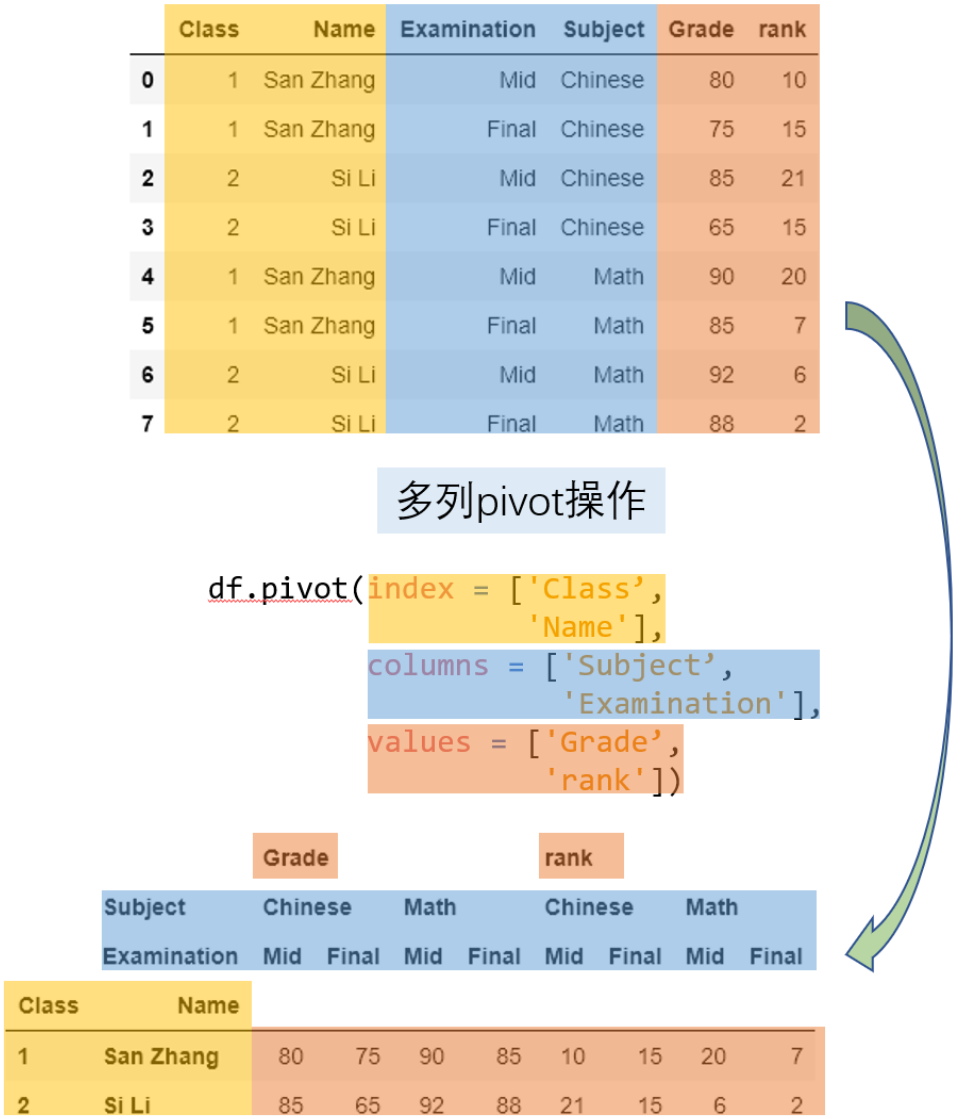
   长表到宽表的转换，可以用`pivot`实现

   对于长变宽操作而言，最重要的有三个要素，分别是**变形后的行索引**、需要**转到列索引的列**，以及这些**列和行索引对应的数值**，它们分别对应了`pivot`方法中的`index, columns, values`参数。



   利用`pivot`进行变形操作需要满足**唯一性**的要求，即由于在新表中的行列索引对应了唯一的`value`，因此原表中的`index`和`columns`对应的**行列组合的value值必须唯一**，不然不能确定到底要填入哪个值。

pivot相关的三个参数允许被设置为列表，这也意味着会返回多级索引。根据唯一性原则，新表的行索引等价于对index中的多列使用drop_duplicates，而列索引的长度为values中的元素个数乘以columns`的唯一组合数量。



	Class	Name	Examination	Subject	Grade	rank
0	1	San Zhang	Mid	Chinese	80	10
1	1	San Zhang	Final	Chinese	75	15
2	2	Si Li	Mid	Chinese	85	21
3	2	Si Li	Final	Chinese	65	15
4	1	San Zhang	Mid	Math	90	20
5	1	San Zhang	Final	Math	85	7
6	2	Si Li	Mid	Math	92	6
7	2	Si Li	Final	Math	88	2

		Grade				rank			
Subject		Chinese		Math		Chinese		Math	
Examination		Mid	Final	Mid	Final	Mid	Final	Mid	Final
Class	Name								
1	San Zhang	80	75	90	85	10	15	20	7
2	Si Li	85	65	92	88	21	15	6	2

2. **pivot_table**

pivot的使用依赖于唯一性条件，那如果不满足唯一性条件，那么必须通过聚合操作使得相同行列组合对应的多个值变为一个值。例如，张三和李四都参加了两次语文考试和数学考试，按照学院规定，最后的成绩是两次考试分数的平均值，此时就无法通过pivot函数来完成。

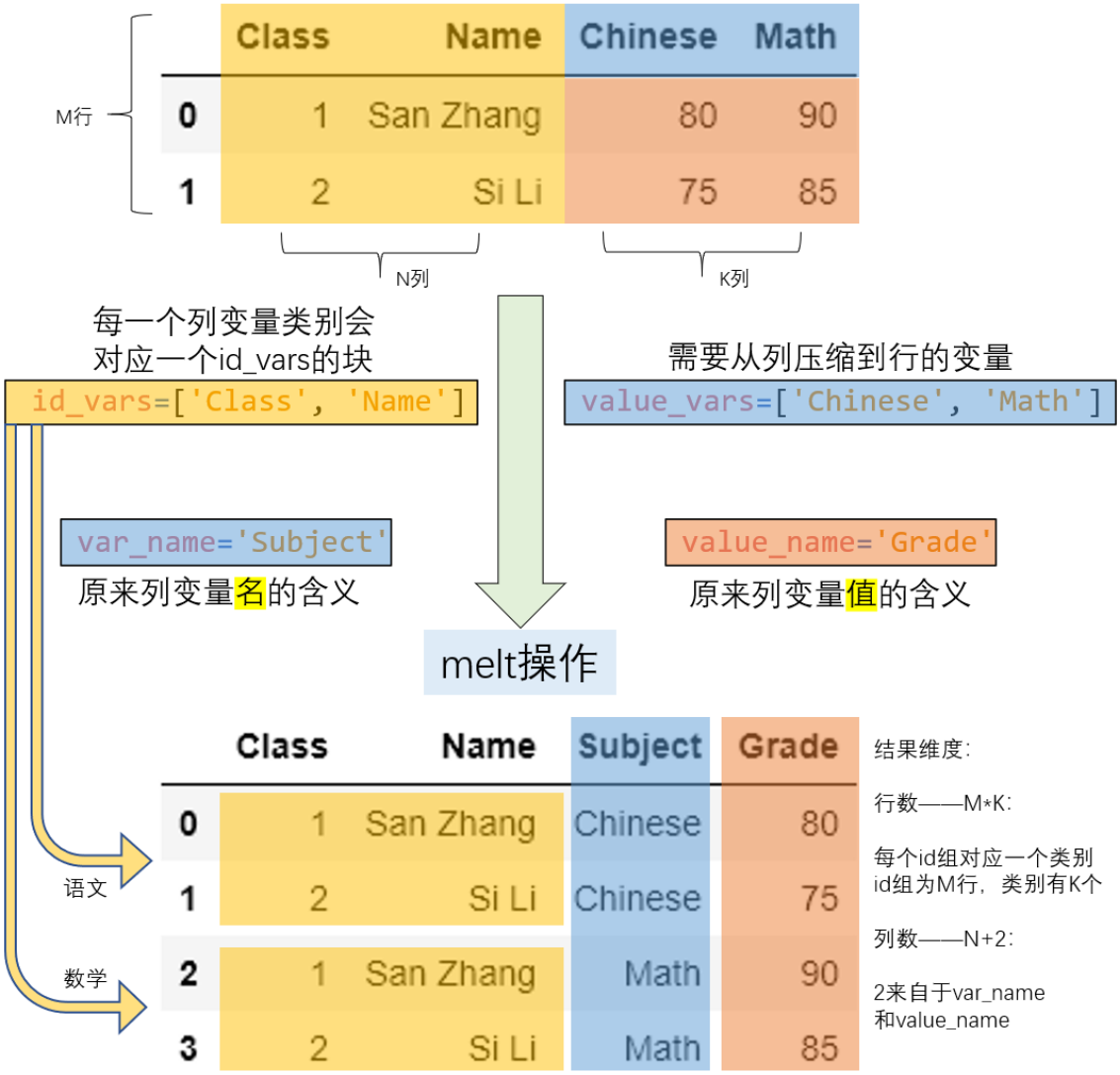
pivot_table中提供aggfunc参数来进行聚合操作。此外，pivot_table具有边际汇总的功能，可以通过设置margins=True来实现，其中边际的聚合方式与aggfunc中给出的聚合方法一致。

```
df.pivot_table(index = 'Name',
               columns = 'Subject',
               values = 'Grade',
               aggfunc='mean',
               margins=True)
```

5.1.2 宽表的逆透视变形

1. 通过相应的逆操作`melt`，把宽表转为长表

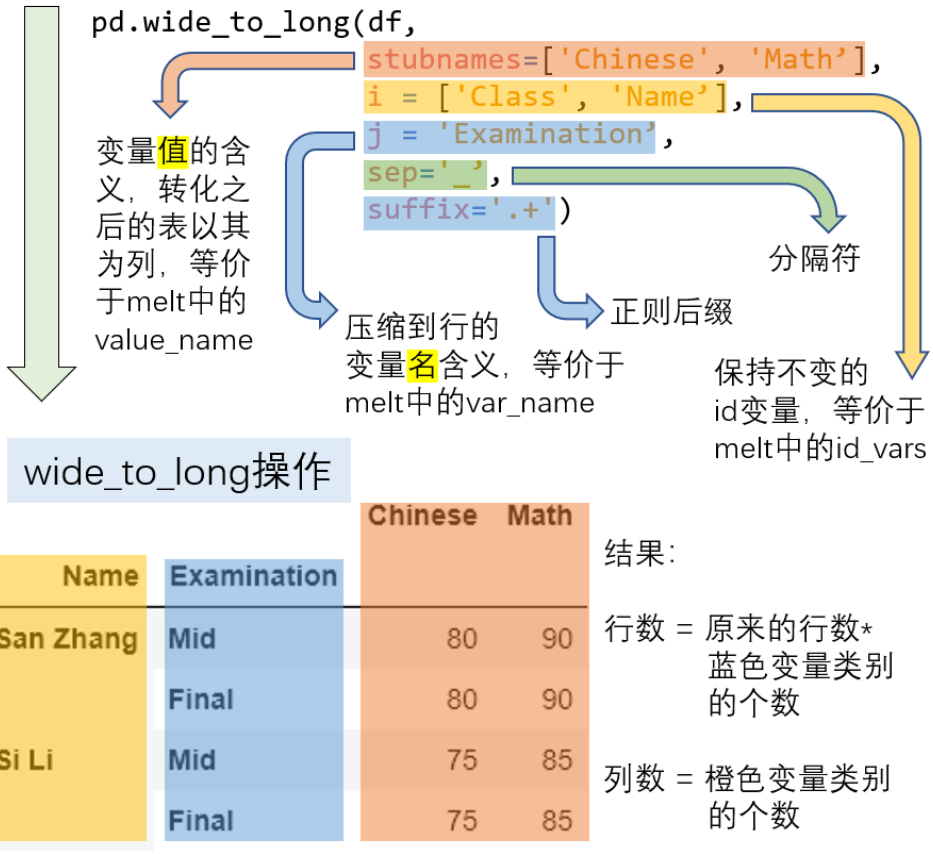
```
df_melted = df.melt(id_vars = ['Class', 'Name'],
                    value_vars = ['Chinese', 'Math'],
                    var_name = 'Subject',
                    value_name = 'Grade')
```



2. `melt`方法中，在列索引中被压缩的一组值对应的列元素只能代表同一层次的含义，即`values_name`。现在如果列中包含了交叉类别，这种情况下就要使用`wide_to_long`函数来完成。

```
pd.wide_to_long(df,
                stubnames=['Chinese', 'Math'],
                i = ['Class', 'Name'],  # 等价于melt中的id_vars
                j='Examination',  # 等价于melt中的var_name
                sep='_',  # 分隔符，之前的元素保留在列索引上，之后的元素用suffix正在表达式来捕获
                suffix='.+')  # 至少匹配一个除换行符以外的字符
```

	Class	Name	Chinese_Mid	Math_Mid	Chinese_Final	Math_Final
0	1	San Zhang	80	90	80	90
1	2	Si Li	75	85	75	85



Class	Name	Examination	Chinese	Math
1	San Zhang	Mid	80	90
		Final	80	90
2	Si Li	Mid	75	85
		Final	75	85

结果：

行数 = 原来的行数* 蓝色变量类别的个数

列数 = 橙色变量类别的个数

5.2 其他变形方法

5.2.1 索引变形

`swaplevel`或者`reorder_levels`进行索引内部的层交换，行列索引之间的交换，行索引转换为列索引：索引透视`unstack()`，列索引转换为行索引：索引逆透视`stack()`。

1. `unstack`的主要参数是移动的层号，默认转化最内层，移动到列索引的最内层，同时支持同时转化多个层。

   在`unstack`中必须保证被转为列索引的行索引层和被保留的行索引层构成的组合是唯一的。

2. `stack`的作用就是把列索引的层压入行索引，其他类似。

3. 把行索引全部压入列索引/列索引全部压入行索引，都得到一个`Series`，只是索引顺序不一样。另外，`stack()`会把结果中的`nan`所在的行删掉，而`unstack()`会保留`nan`

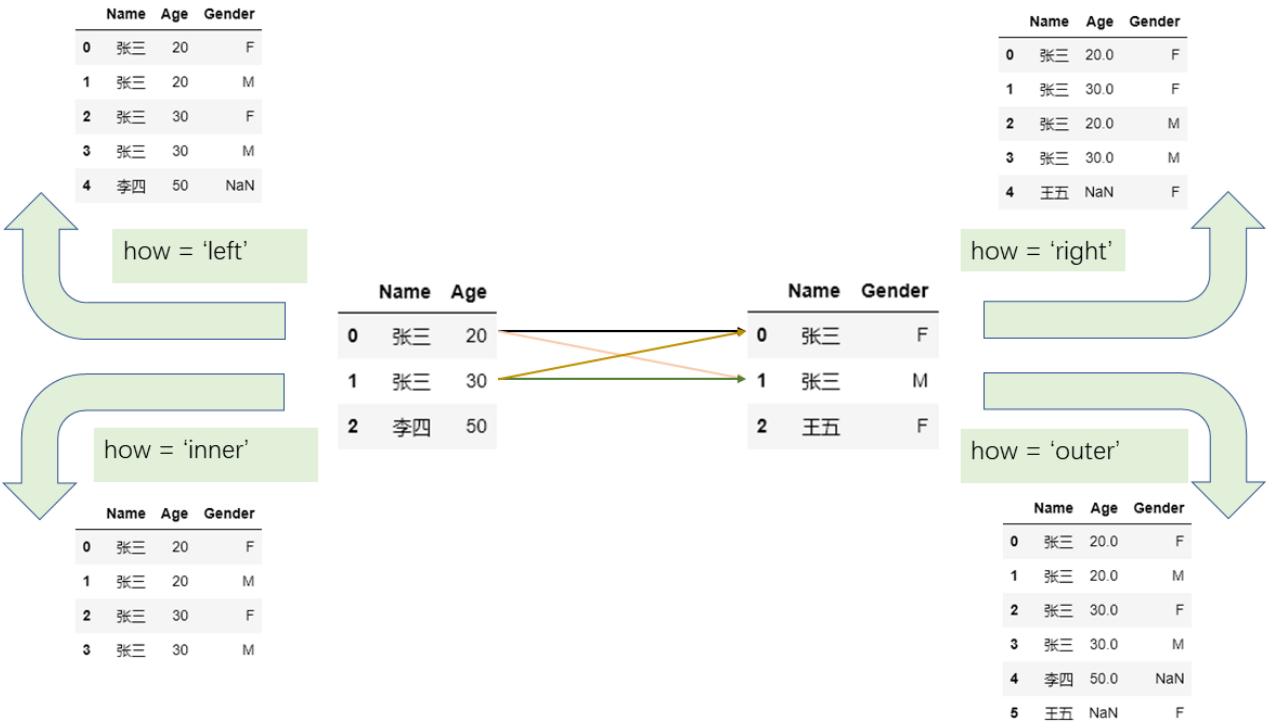
5.2.2 扩张变形

1. `df.explode()`函数能够对某一列的元素进行纵向的展开，被展开的单元格必须存储`list, tuple, Series, np.ndarray`中的一种类型。
2. `pd.get_dummies()`是用于特征构建的重要函数之一，其作用是把类别特征转为指示变量。

第6章 连接

6.1 关系连接

6.1.1 关系连接的基本概念



	Name	Age	Gender
0	张三	20	F
1	张三	20	M
2	张三	30	F
3	张三	30	M
4	李四	50	NaN

	Name	Age
0	张三	20
1	张三	30
2	李四	50

	Name	Gender
0	张三	F
1	张三	M
2	王五	F

	Name	Age	Gender
0	张三	20.0	F
1	张三	30.0	F
2	张三	20.0	M
3	张三	30.0	M
4	王五	NaN	F

	Name	Age	Gender
0	张三	20	F
1	张三	20	M
2	张三	30	F
3	张三	30	M

	Name	Age	Gender
0	张三	20.0	F
1	张三	20.0	M
2	张三	30.0	F
3	张三	30.0	M
4	李四	50.0	NaN
5	王五	NaN	F

6.1.2 列连接

通过几列值的组合进行连接，这种基于值的连接在`pandas`中可以由`merge`函数实现。

- 如果两个表中想要连接的列不具备相同的列名，可以通过`left_on`和`right_on`指定
- 如果两个表中的列出现了重复的列名，那么可以通过`suffixes`参数指定
- 在进行基于唯一性的连接下，如果键不是唯一的，那么结果就会产生问题。想要保证唯一性，除了用`duplicated`检查是否重复外，`merge`中也提供了`validate`参数来检查连接的唯一性模式。这里共有三种模式，即一对一连接`1:1`，一对多连接`1:m`，多对一连接`m:1`连接，第一个是指左右表的键都是唯一的，后面两个分别指左表键唯一和右表键唯一

6.1.3 索引连接

索引连接，就是把索引当作键，因此这和值连接本质上没有区别，`pandas`中利用`join`函数来处理索引连接，它的参数选择要少于`merge`。

- 除了必须的`on`和`how`之外，可以对重复的列指定左右后缀`lsuffix`和`rsuffix`。其中，`on`参数指索引名，单层索引时省略参数表示按照当前索引连接
- 想要进行类似于`merge`中以多列为键的操作的时候，`join`需要使用多级索引

从pandas的1.2.0版本开始，`merge()`和`join()`支持一种新的连接方式——“交叉连接”，`how='cross'`，将两个DataFrame按行进行笛卡尔积的组合，一个m行的df和n行的df检查连接，得到$m \times n$行的DataFrame。

6.2 其他连接

6.2.1 方向连接

1. **表与表的合并**

   有时候用户并不关心以哪一列为键来合并，只是希望把两个表或者多个表按照纵向或者横向拼接，可使用`concat`函数。

   方向连接与关系连接最大的区别之一在于，**方向连接应保证被连接表在连接方向上的索引是一一对应的**（不是要求索引不重复，当两张表所有元素的值和位置都一一对应，即使索引重复也可以）。

   一一对应：左边的索引在右边的索引中最多出现一次，同样，右边的索引也最多在左边索引中出现一次。

```
df_a = pd.dataFrame({'语': [80, 95, 70], '英': [90, 92, 80]}, index=['张三', '李四', '王五'])
df_b = pd.dataFrame({'数': [85, 75, 75]}, index=['李四', '张三', '王五'])

df_a.index = ['张三', '张三', '王五']  # 打破一一对应原则
pd.concat([df_a, df_b], axis=1)  # InvalidIndexError: Reundexing only valid with uniquely valued Index object

df_b.index = df_a.index  # OK，一一对应，班上恰好有多个"张三"

```

   - 在`concat`中，最常用的有三个参数，它们是`axis, join, keys`，分别表示拼接方向，连接形式，以及在新表中指示来自于哪一张旧表的名字
   - 虽然说`concat`是处理关系型合并的函数，但是它仍然是关于索引进行连接的。拼接会根据行/列索引对齐，默认状态下`join=outer`，表示保留所有的行/列，并将不存在的值设为缺失；`join=inner`，表示保留两个表都出现过的行/列。

     因此，当确认要使用多表直接的方向合并时，尤其是横向的合并，可以先用`reset_index`方法恢复默认整数索引再进行合并，防止出现由索引的误对齐和重复索引的笛卡尔积带来的错误结果。
   - `keys`参数的使用场景在于多个表合并后，用户仍然想要知道新表中的数据来自于哪个原表，这时可以通过`keys`参数，新增一层行/列索引，产生多级索引进行标记。

2. **序列与表的合并**

   利用`concat`可以实现多个表之间的方向拼接，如果想要把一个序列追加到表的行末或者列末，则可以分别使用`append`和`assign`方法

   - 在`append`中，如果原表是默认整数序列的索引，那么可以使用`ignore_index=True`对新序列对应的索引自动标号，否则必须对`Series`指定`name`属性

```
s = pd.Series(['Wu Wang', 21], index = df1.columns)
df1.append(s)  # TypeError: Can only append a Series if ignore_index=True or if the Series has a name
```

- 对于`assign`而言，虽然可以利用其添加新的列，但一般通过`df['new_col'] = ...`的形式就可以等价地添加新列。同时，使用`[]`修改的缺点是它会直接在原表上进行改动，而`assign`返回的是一个**临时副本**

```
df1.assign(Grade=s)  # 传入的参数名即为新列名，支持一次拼接多个列
```

6.2.2 比较与组合

- `compare`是在`1.1.0`后引入的新函数，它能够比较两个表或者序列的不同处并将其汇总展示。如果相同则会被填充为缺失值`NaN`，如果想要完整显示表中所有元素的比较情况，可以设置`keep_shape=True`。

```
df1.compare(df2, keep_shape=True)
```

- `combine`函数能够让两张表按照一定的规则进行组合，在进行规则比较时会自动进行列索引的对齐。对于传入的函数而言，每一次操作中输入的参数是来自两个表的同名`Series`，依次传入的列是两个表列名的并集，在某个表中并不存在的列会变成全空的序列，并且来自第一个表的序列索引会被`reindex`成两个索引的并集

  设置`overtwrite`参数为`False`可以保留被调用表中未出现在传入的参数表中的列，而不会设置未缺失值

```
def choose_min(s1, s2):
    s2 = s2.reindex_like(s1)
    res = s1.where(s1<s2, s2)
    res = res.mask(s1.isna()) # isna表示是否为缺失值，返回布尔序列
    return res
df1 = pd.DataFrame({'A':[1,2], 'B':[3,4], 'C':[5,6]})
df2 = pd.DataFrame({'B':[5,6], 'C':[7,8], 'D':[9,10]}, index=[1,2])
df1.combine(df2, choose_min)

df1.combine(df2, choose_min, overwrite=False)

df1.combine(df2, lambda s1, s2: s1.where(s1 > s2.max(), s2))  # s1的列元素>s2相应列元素的最大值，取s1，否则取s2
```

- 除了`combine`之外，`pandas`中还有一个`combine_first`方法，其功能是在对两张表组合时，若第二张表中的值在第一张表中对应索引位置的值不是缺失状态，那么就使用第一张表的值填充。

第三部分 4类数据

第7章 缺失数据

7.1 缺失值的统计和删除

7.1.1 缺失信息的统计

- 缺失数据可以使用`isna`或`isnull`（两个函数没有区别）来查看每个单元格是否缺失，结合`mean`可以计算出每列缺失值的比例
- 如果想要查看某一列缺失或者非缺失的行，可以利用`Series`上的`isna`或者`notna`进行布尔索引
- 如果想要同时对几个列，检索出全部为缺失或者至少有一个缺失或者没有缺失的行，可以使用`isna, notna`和`any, all`的组合

7.1.2 缺失信息的删除

- `dropna`的主要参数为轴方向`axis`（默认为0，即删除行）、删除方式`how`、删除的非缺失值个数阈值`thresh`（非缺失值没有达到这个数量的相应维度会被删除）、备选的删除子集`subset`，其中`how`主要有`any`和`all`两种参数可以选择
- 不用`dropna`，使用布尔索引同样是可行的

7.2 缺失值的填充和插值

7.2.1 利用fillna()进行填充

在`fillna`中有三个参数是常用的：`value, method, limit`。

- 其中，`value`为填充值，可以是标量，也可以是索引到元素的字典映射；

```
Series.fillna({'a': 100, 'd': 200})  # a、d代表索引，通过索引映射填充不同的值
```

- `method`为填充方法，有用前面的元素填充`ffill`和用后面的元素填充`bfill`两种类型；
- `limit`参数表示连续缺失情况下，缺失值的最大填充次数

7.2.2 插值函数

在关于`interpolate`函数的文档描述中，列举了许多插值法，包括了大量`Scipy`中的方法。

对于`interpolate`而言，除了插值方法（默认为`linear`线性插值）之外，有与`fillna`类似的两个常用参数，一个是控制方向的`limit_direction`，另一个是控制最大连续缺失值插值个数的`limit`。其中，限制插值的方向默认为`forward`，这与`fillna`的`method`中的`ffill`是类似的，若想要后向限制插值或者双向限制插值可以指定为`backward`或`both`。

- 线性插值

```
s = pd.Series([np.nan, np.nan, 1, np.nan, np.nan, np.nan, 2, np.nan,
np.nan])
s.interpolate(limit_direction='backward', limit=1)
```

- 近邻插值

  缺失值的元素和离它最近的非缺失值元素一样。

```
s.interpolate('nearest')
```

- 索引插值

  根据索引大小进行线性插值，这种方法对于时间戳索引也是可以使用的。

```
s = pd.Series([0,np.nan,10], index=pd.to_datetime(['20200101', '20200102', '20200111']))
s.interpolate(method='index')  # 和索引有关的线性插值，计算相应索引大小对应的值
```

在`interpolate()`中如果选用`polynomial`的插值方法，并指定阶数`order`，它内部调用的是`scipy.interpolate.interp1d(*,*,kind=order)`，这个函数内部调用的是`make_interp_spline()`方法，实现的是基样条插值算法(basic spline)，而不是所谓的多项式插值，更不是类似于`numpy`中的`polyfit()`多项式拟合插值；

而当选用`spline`方法时，`pandas`调用的是`scipy.interpolate.UnivariateSpline`，实现的是平滑样条(smoothing spline)算法，参数`order`的范围一般是1-5。

7.3 Nullable类型

7.3.1 缺失记号及其缺陷

- 在`python`中的缺失值用`None`表示，该元素除了等于自己本身之外，与其他任何元素不相等。
- 在`numpy`中利用`np.nan`来表示缺失值，该元素除了不和其他任何元素相等之外，和自身的比较结果也返回`False`。

  在对缺失序列或表格的元素进行比较操作的时候，`np.nan`的对应位置会返回`False`，但是在使用`equals()`函数进行两张表或两个序列的相同性检验时，会自动跳过两侧表都是缺失值的位置，直接返回`True`。
- 在时间序列的对象中，`pandas`利用`pd.NaT`来指代缺失值。

  `NaT`问题的根源来自于`np.nan`的本身是一种浮点类型，而如果浮点和时间类型混合存储，如果不设计新的内置缺失类型来处理，就会变成含糊不清的`object`类型。

  由于`np.nan`的浮点性质，如果在一个整数的`Series`中出现缺失，那么其类型会转变为`float64`；而如果在一个布尔类型的序列中出现缺失，那么其类型就会转为`object`而不是`bool`。
- `pandas`尝试设计了一种新的缺失类型`pd.NA`以及三种`Nullable`序列类型来应对这些缺陷，它们分别是`Int, boolean`和`string`。在这三种序列中，返回的结果会尽可能地成为`Nullable`的类型。

```
pd.Series([np.nan, 1], dtype = 'Int64')  # <NA>
```

7.3.2 Nullable类型的性质

1. 在上述三个`Nullable`类型中存储缺失值，都会转为`pandas`内置的`pd.NA`
2. 对于`boolean`类型的序列而言，其和`bool`序列的行为主要有两点区别：
   - 带有缺失的布尔列表无法进行索引器中的选择，而`boolean`会把缺失值看作`False`

```python
s = pd.Series(['a', 'b'])
s_bool = pd.Series([True, np.nan])
s_boolean = pd.Series([True, np.nan]).astype('boolean')
# s[s_bool] # 报错
s[s_boolean]
```

- 在进行逻辑运算时，`bool`类型在缺失处返回的永远是`False`，而`boolean`会根据逻辑运算是否能确定唯一结果来返回相应的值。
  - `True | pd.NA`中无论缺失值为什么值，必然返回`True`；
  - `False | pd.NA`中的结果会根据缺失值取值的不同而变化，此时返回`pd.NA`；
  - `False & pd.NA`中无论缺失值为什么值，必然返回`False`。

在实际数据处理时，先通过`convert_dtypes`转为`Nullable`类型。

7.3.3 缺失数据的计算和分组

- 当调用函数`sum, prod`使用加法和乘法，以及使用累计函数时，会自动跳过缺失值所处的位置。
- 当进行单个标量运算的时候，除了`np.nan ** 0`和`1 ** np.nan`这两种情况为确定的值之外，所有运算结果全为缺失（`pd.NA`的行为与此一致），并且`np.nan`在比较操作时一定返回`False`，而`pd.NA`返回`pd.NA`。
- `diff, pct_change`这两个函数虽然功能相似，但是对于缺失的处理不同，前者凡是参与缺失计算的部分全部设为了缺失值，而后者缺失值位置会沿用前一个非缺失值，导致 0% 的变化率。
- 对于`groupby`，`get_dummies`，缺失可以作为一个类别处理。

```python
df_nan.groupby('category', dropna=False)['value'].mean()
pd.get_dummies(df_nan.category, dummy_na=True)
```

- 对于`groupby`对象，`groups`和`ngroups`不同
  - 单列分组时，两者都剔除缺失值
  - 多列分组时，`groups`会包括含缺失值的组，`ngroups`只会计算不含缺失值的组的数量

第8章 文本数据

第9章 分类数据

第10章 时间序列数据

第四部分 进阶实践

第11章 数据观测

第12章 特征工程

第13章 性能优化

附录 CheatSheet

功能	函数	备注