Pandas 必知必会

Release 1.0

Datawhale, GYH

Nov 23, 2020

Contents

第 1 章　预备知识

1.1　Python 基础

1.1.1　列表推导式与条件赋值

在生成一个数字序列的时候，在 Python 中可以如下写出：

```
In [1]: L = []

In [2]: def my_func(x):
   ...:     return 2*x
   ...:

In [3]: for i in range(5):
   ...:     L.append(my_func(i))
   ...:

In [4]: L
Out[4]: [0, 2, 4, 6, 8]
```

事实上可以利用列表推导式进行写法上的简化：[* for i in *] 。其中，第一个 * 为映射函数，其输入为后面 i 指代的内容，第二个 * 表示迭代的对象。

```
In [5]: [my_func(i) for i in range(5)]
Out[5]: [0, 2, 4, 6, 8]
```

列表表达式还支持多层嵌套，如下面的例子中第一个 for 为外层循环，第二个为内层循环：

```
In [6]: [m+'_'+n for m in ['a', 'b'] for n in ['c', 'd']]
Out[6]: ['a_c', 'a_d', 'b_c', 'b_d']
```

除了列表推导式，另一个实用的语法糖是条件赋值，其形式为 value = a if condition else b ：

```
In [7]: value = 'cat' if 2>1 else 'dog'

In [8]: value
Out[8]: 'cat'
```

等价于如下的写法：

```
a, b = 'cat', 'dog'
condition = 2 > 1 # 此时为 True
if condition:
    value = a
else:
    value = b
```

下面举一个例子，截断列表中超过 5 的元素：

```
In [9]: L = [1, 2, 3, 4, 5, 6, 7]

In [10]: [i if i <= 5 else 5 for i in L]
Out[10]: [1, 2, 3, 4, 5, 5, 5]
```

1.1.2 匿名函数与 map 方法

有一些函数的定义具有清晰简单的映射关系，例如上面的 my_func 函数，这时候可以用匿名函数的方法简洁地表示：

```
In [11]: my_func = lambda x: 2*x

In [12]: my_func(3)
Out[12]: 6

In [13]: multi_para_func = lambda a, b: a + b

In [14]: multi_para_func(1, 2)
Out[14]: 3
```

但上面的用法其实违背了“匿名”的含义，事实上它往往在无需多处调用的场合进行使用，例如上面列表推导式中的例子，用户不关心函数的名字，只关心这种映射的关系：

```
In [15]: [(lambda x: 2*x)(i) for i in range(5)]
Out[15]: [0, 2, 4, 6, 8]
```

对于上述的这种列表推导式的匿名函数映射，Python 中提供了 map 函数来完成，它返回的是一个 map 对

象，需要通过 list 转为列表：

```
In [16]: list(map(lambda x: 2*x, range(5)))
Out[16]: [0, 2, 4, 6, 8]
```

对于多个输入值的函数映射，可以通过追加迭代对象实现：

```
In [17]: list(map(lambda x, y: str(x)+'_'+y, range(5), list('abcde')))
Out[17]: ['0_a', '1_b', '2_c', '3_d', '4_e']
```

1.1.3 zip 对象与 enumerate 方法

zip 函数能够把多个可迭代对象打包成一个元组构成的可迭代对象，它返回了一个 zip 对象，通过 tuple, list 可以得到相应的打包结果：

```
In [18]: L1, L2, L3 = list('abc'), list('def'), list('hij')

In [19]: list(zip(L1, L2, L3))
Out[19]: [('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j')]

In [20]: tuple(zip(L1, L2, L3))
Out[20]: (('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j'))
```

往往会在循环迭代的时候使用到 zip 函数：

```
In [21]: for i, j, k in zip(L1, L2, L3):
   ....:     print(i, j, k)
   ....:
a d h
b e i
c f j
```

enumerate 是一种特殊的打包，它可以在迭代时绑定迭代元素的遍历序号：

```
In [22]: L = list('abcd')

In [23]: for index, value in enumerate(L):
   ....:     print(index, value)
   ....:
0 a
1 b
2 c
3 d
```

用 zip 对象也能够简单地实现这个功能：

```
In [24]: for index, value in zip(range(len(L)), L):
   ....:     print(index, value)
   ....:
0 a
1 b
2 c
3 d
```

当需要对两个列表建立字典映射时，可以利用 zip 对象：

```
In [25]: dict(zip(L1, L2))
Out[25]: {'a': 'd', 'b': 'e', 'c': 'f'}
```

既然有了压缩函数，那么 Python 也提供了 * 操作符和 zip 联合使用来进行解压操作：

```
In [26]: zipped = list(zip(L1, L2, L3))

In [27]: zipped
Out[27]: [('a', 'd', 'h'), ('b', 'e', 'i'), ('c', 'f', 'j')]

In [28]: list(zip(*zipped)) # 三个元组分别对应原来的列表
Out[28]: [('a', 'b', 'c'), ('d', 'e', 'f'), ('h', 'i', 'j')]
```

1.2 Numpy 基础

1.2.1 np 数组的构造

最一般的方法是通过 array 来构造：

```
In [29]: import numpy as np

In [30]: np.array([1,2,3])
Out[30]: array([1, 2, 3])
```

下面讨论一些特殊数组的生成方式：

【a】等差序列：np.linspace, np.arange

```
In [31]: np.linspace(1,5,11) # 起始、终止（包含）、样本个数
Out[31]: array([1. , 1.4, 1.8, 2.2, 2.6, 3. , 3.4, 3.8, 4.2, 4.6, 5. ])
```

(continues on next page)

(continued from previous page)

```
In [32]: np.arange(1,5,2) # 起始、终止（不包含）、步长
Out[32]: array([1, 3])
```

【b】特殊矩阵：zeros, eye, full

```
In [33]: np.zeros((2,3)) # 传入元组表示各维度大小
Out[33]:
array([[0., 0., 0.],
       [0., 0., 0.]])

In [34]: np.eye(3) # 3*3 的单位矩阵
Out[34]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In [35]: np.eye(3, k=1) # 偏移主对角线 1 个单位的伪单位矩阵
Out[35]:
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])

In [36]: np.full((2,3), 10) # 元组传入大小，10 表示填充数值
Out[36]:
array([[10, 10, 10],
       [10, 10, 10]])

In [37]: np.full((2,3), [1,2,3]) # 通过传入列表填充每列的值
Out[37]:
array([[1, 2, 3],
       [1, 2, 3]])
```

【c】随机矩阵：np.random

最常用的随机生成函数为 rand, randn, randint, choice ，它们分别表示 0-1 均匀分布的随机数组、标准正态的随机数组、随机整数组和随机列表抽样：

```
In [38]: np.random.rand(3) # 生成服从 0-1 均匀分布的三个随机数
Out[38]: array([0.59565497, 0.5770432 , 0.51825326])

In [39]: np.random.rand(3, 3) # 注意这里传入的不是元组，每个维度大小分开输入
Out[39]:
array([[0.32408216, 0.34375498, 0.67285906],
       [0.38880074, 0.65984716, 0.14063781],
```

(continues on next page)

(continued from previous page)

```
       [0.38336702, 0.90235554, 0.2209778 ]])
```

对于服从区间 a 到 b 上的均匀分布可以如下生成：

```
In [40]: a, b = 5, 15

In [41]: (b - a) * np.random.rand(3) + a
Out[41]: array([14.6322828 , 13.14409776, 11.70739141])
```

randn 生成了 $N(\mathbf{0}, \mathbf{I})$ 的标准正态分布：

```
In [42]: np.random.randn(3)
Out[42]: array([ 1.07630581,  1.19924302, -0.67825676])

In [43]: np.random.randn(2, 2)
Out[43]:
array([[-2.47115591,  1.09418563],
       [-0.25876654,  0.02738611]])
```

对于服从方差为 σ^2 均值为 μ 的一元正态分布可以如下生成：

```
In [44]: sigma, mu = 2.5, 3

In [45]: mu + np.random.randn(3) * sigma
Out[45]: array([ 4.84305738, -0.46281697,  5.176269  ])
```

randint 可以指定生成随机整数的最小值最大值和维度大小：

```
In [46]: low, high, size = 5, 15, (2,2)

In [47]: np.random.randint(low, high, size)
Out[47]:
array([[10,  7],
       [11, 13]])
```

choice 可以从给定的列表中，以一定概率和方式抽取结果，当不指定概率时为均匀采样，默认抽取方式为有放回抽样：

```
In [48]: my_list = ['a', 'b', 'c', 'd']

In [49]: np.random.choice(my_list, 2, replace=False, p=[0.1, 0.7, 0.1 ,0.1])
Out[49]: array(['a', 'b'], dtype='<U1')
```

(continues on next page)

(continued from previous page)

```
In [50]: np.random.choice(my_list, (3,3))
Out[50]:
array([['a', 'd', 'd'],
       ['d', 'c', 'c'],
       ['b', 'a', 'c']], dtype='<U1')
```

当返回的元素个数与原列表相同时，等价于使用 permutation 函数，即打散原列表：

```
In [51]: np.random.permutation(my_list)
Out[51]: array(['a', 'c', 'd', 'b'], dtype='<U1')
```

最后，需要提到的是随机种子，它能够固定随机数的输出结果：

```
In [52]: np.random.seed(0)

In [53]: np.random.rand()
Out[53]: 0.5488135039273248

In [54]: np.random.seed(0)

In [55]: np.random.rand()
Out[55]: 0.5488135039273248
```

1.2.2 np 数组的变形与合并

【a】转置：T

```
In [56]: np.zeros((2,3)).T
Out[56]:
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

【b】合并操作：r_, c_

对于二维数组而言，r_ 和 c_ 分别表示上下合并和左右合并：

```
In [57]: np.r_[np.zeros((2,3)),np.zeros((2,3))]
Out[57]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

(continues on next page)

(continued from previous page)

```
In [58]: np.c_[np.zeros((2,3)),np.zeros((2,3))]
Out[58]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

一维数组和二维数组进行合并时，应当把其视作列向量，在长度匹配的情况下只能够使用左右合并的 c_ 操作：

```
In [59]: try:
   ....:     np.r_[np.array([0,0]),np.zeros((2,1))]
   ....: except Exception as e:
   ....:     Err_Msg = e
   ....:

In [60]: Err_Msg
Out[60]: ValueError('all the input arrays must have same number of dimensions, but the array at␣
↪index 0 has 1 dimension(s) and the array at index 1 has 2 dimension(s)')

In [61]: np.r_[np.array([0,0]),np.zeros(2)]
Out[61]: array([0., 0., 0., 0.])

In [62]: np.c_[np.array([0,0]),np.zeros((2,3))]
Out[62]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

【c】维度变换：reshape

reshape 能够帮助用户把原数组按照新的维度重新排列。在使用时有两种模式，分别为 C 模式和 F 模式，分别以逐行和逐列的顺序进行填充读取。

```
In [63]: target = np.arange(8).reshape(2,4)

In [64]: target
Out[64]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

In [65]: target.reshape((4,2), order='C') # 按照行读取和填充
Out[65]:
array([[0, 1],
       [2, 3],
       [4, 5],
```

(continues on next page)

(continued from previous page)

```
       [6, 7]])

In [66]: target.reshape((4,2), order='F') # 按照列读取和填充
Out[66]:
array([[0, 2],
       [4, 6],
       [1, 3],
       [5, 7]])
```

特别地，由于被调用数组的大小是确定的，reshape 允许有一个维度存在空缺，此时只需填充-1 即可：

```
In [67]: target.reshape((4,-1))
Out[67]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

下面将 n*1 大小的数组转为 1 维数组的操作是经常使用的：

```
In [68]: target = np.ones((3,1))

In [69]: target
Out[69]:
array([[1.],
       [1.],
       [1.]])

In [70]: target.reshape(-1)
Out[70]: array([1., 1., 1.])
```

1.2.3 np 数组的切片与索引

数组的切片模式支持使用 slice 类型的 start:end:step 切片，还可以直接传入列表指定某个维度的索引进行切片：

```
In [71]: target = np.arange(9).reshape(3,3)

In [72]: target
Out[72]:
array([[0, 1, 2],
       [3, 4, 5],
```

(continues on next page)

(continued from previous page)

```
       [6, 7, 8]])

In [73]: target[:-1, [0,2]]
Out[73]:
array([[0, 2],
       [3, 5]])
```

此外，还可以利用 np.ix_ 在对应的维度上使用布尔索引，但此时不能使用 slice 切片：

```
In [74]: target[np.ix_([True, False, True], [True, False, True])]
Out[74]:
array([[0, 2],
       [6, 8]])

In [75]: target[np.ix_([1,2], [True, False, True])]
Out[75]:
array([[3, 5],
       [6, 8]])
```

当数组维度为 1 维时，可以直接进行布尔索引，而无需 np.ix_ ：

```
In [76]: new = target.reshape(-1)

In [77]: new[new%2==0]
Out[77]: array([0, 2, 4, 6, 8])
```

1.2.4 常用函数

为了简单起见，这里假设下述函数输入的数组都是一维的。

【a】where

where 是一种条件函数，可以指定满足条件与不满足条件位置对应的填充值：

```
In [78]: a = np.array([-1,1,-1,0])

In [79]: np.where(a>0, a, 5) # 对应位置为 True 时填充 a 对应元素，否则填充 5
Out[79]: array([5, 1, 5, 5])
```

【b】nonzero, argmax, argmin

这三个函数返回的都是索引，nonzero 返回非零数的索引，argmax, argmin 分别返回最大和最小数的索引：

```
In [80]: a = np.array([-2,-5,0,1,3,-1])

In [81]: np.nonzero(a)
Out[81]: (array([0, 1, 3, 4, 5], dtype=int64),)

In [82]: a.argmax()
Out[82]: 4

In [83]: a.argmin()
Out[83]: 1
```

【c】any, all

any 指当序列至少 存在一个 True 或非零元素时返回 True ， 否则返回 False

all 指当序列元素 全为 True 或非零元素时返回 True ， 否则返回 False

```
In [84]: a = np.array([0,1])

In [85]: a.any()
Out[85]: True

In [86]: a.all()
Out[86]: False
```

【d】cumprod, cumsum, diff

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

```
In [87]: a = np.array([1,2,3])

In [88]: a.cumprod()
Out[88]: array([1, 2, 6], dtype=int32)

In [89]: a.cumsum()
Out[89]: array([1, 3, 6], dtype=int32)

In [90]: np.diff(a)
Out[90]: array([1, 1])
```

【e】统计函数

常用的统计函数包括 max, min, mean, median, std, var, sum, quantile ， 其中分位数计算是全局方法，因此不能通过 array.quantile 的方法调用：

```
In [91]: target = np.arange(5)

In [92]: target
Out[92]: array([0, 1, 2, 3, 4])

In [93]: target.max()
Out[93]: 4

In [94]: np.quantile(target, 0.5) # 0.5 分位数
Out[94]: 2.0
```

但是对于含有缺失值的数组，它们返回的结果也是缺失值，如果需要略过缺失值，必须使用 nan* 类型的函数，上述的几个统计函数都有对应的 nan* 函数。

```
In [95]: target = np.array([1, 2, np.nan])

In [96]: target
Out[96]: array([ 1.,  2., nan])

In [97]: target.max()
Out[97]: nan

In [98]: np.nanmax(target)
Out[98]: 2.0

In [99]: np.nanquantile(target, 0.5)
Out[99]: 1.5
```

对于协方差和相关系数分别可以利用 cov, corrcoef 如下计算：

```
In [100]: target1 = np.array([1,3,5,9])

In [101]: target2 = np.array([1,5,3,-9])

In [102]: np.cov(target1, target2)
Out[102]:
array([[ 11.66666667, -16.66666667],
       [-16.66666667,  38.66666667]])

In [103]: np.corrcoef(target1, target2)
Out[103]:
array([[ 1.        , -0.78470603],
       [-0.78470603,  1.        ]])
```

最后，需要说明二维 Numpy 数组中统计函数的 axis 参数，它能够进行某一个维度下的统计特征计算，当

axis=0 时结果为列的统计指标，当 axis=1 时结果为行的统计指标：

```
In [104]: target = np.arange(1,10).reshape(3,-1)

In [105]: target
Out[105]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [106]: target.sum(0)
Out[106]: array([12, 15, 18])

In [107]: target.sum(1)
Out[107]: array([ 6, 15, 24])
```

1.2.5 广播机制

广播机制用于处理两个不同维度数组之间的操作，这里只讨论不超过两维的数组广播机制。

【a】标量和数组的操作

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：

```
In [108]: res = 3 * np.ones((2,2)) + 1

In [109]: res
Out[109]:
array([[4., 4.],
       [4., 4.]])

In [110]: res = 1 / res

In [111]: res
Out[111]:
array([[0.25, 0.25],
       [0.25, 0.25]])
```

【b】二维数组之间的操作

当两个数组维度完全一致时，使用对应元素的操作，否则会报错，除非其中的某个数组的维度是 $m \times 1$ 或者 $1 \times n$，那么会扩充其具有 1 的维度为另一个数组对应维度的大小。例如，1×2 数组和 3×2 数组做逐元素运算时会把第一个数组扩充为 3×2，扩充时的对应数值进行赋值。但是，需要注意的是，如果第一个数组的维度是 1×3，那么由于在第二维上的大小不匹配且不为 1，此时报错。

```
In [112]: res = np.ones((3,2))

In [113]: res
Out[113]:
array([[1., 1.],
       [1., 1.],
       [1., 1.]])

In [114]: res * np.array([[2,3]]) # 扩充第一维度为 3
Out[114]:
array([[2., 3.],
       [2., 3.],
       [2., 3.]])

In [115]: res * np.array([[2],[3],[4]]) # 扩充第二维度为 2
Out[115]:
array([[2., 2.],
       [3., 3.],
       [4., 4.]])

In [116]: res * np.array([[2]]) # 等价于两次扩充
Out[116]:
array([[2., 2.],
       [2., 2.],
       [2., 2.]])
```

【c】一维数组与二维数组的操作

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，使用的广播法则与【b】中一致，当 $k! = n$ 且 k, n 都不是 1 时报错。

```
In [117]: np.ones(3) + np.ones((2,3))
Out[117]:
array([[2., 2., 2.],
       [2., 2., 2.]])

In [118]: np.ones(3) + np.ones((2,1))
Out[118]:
array([[2., 2., 2.],
       [2., 2., 2.]])

In [119]: np.ones(1) + np.ones((2,3))
Out[119]:
array([[2., 2., 2.],
       [2., 2., 2.]])
```

1.2.6 向量与矩阵的计算

【a】向量内积：dot

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i} a_i b_i$$

```
In [120]: a = np.array([1,2,3])

In [121]: b = np.array([1,3,5])

In [122]: a.dot(b)
Out[122]: 22
```

【b】向量范数和矩阵范数：np.linalg.norm

在矩阵范数的计算中，最重要的是 ord 参数，可选值如下：

ord	norm for matrices	norm for vectors
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)

```
In [123]: martix_target =  np.arange(4).reshape(-1,2)

In [124]: martix_target
Out[124]:
array([[0, 1],
       [2, 3]])

In [125]: np.linalg.norm(martix_target, 'fro')
Out[125]: 3.7416573867739413

In [126]: np.linalg.norm(martix_target, np.inf)
Out[126]: 5.0
```

(continues on next page)

(continued from previous page)

```
In [127]: np.linalg.norm(martix_target, 2)
Out[127]: 3.702459173643833
```

```
In [128]: vector_target =  np.arange(4)

In [129]: vector_target
Out[129]: array([0, 1, 2, 3])

In [130]: np.linalg.norm(vector_target, np.inf)
Out[130]: 3.0

In [131]: np.linalg.norm(vector_target, 2)
Out[131]: 3.7416573867739413

In [132]: np.linalg.norm(vector_target, 3)
Out[132]: 3.3019272488946263
```

【c】矩阵乘法：@

$$[\mathbf{A}_{\mathrm{m\times p}}\mathbf{B}_{\mathrm{p\times n}}]_{\mathrm{ij}} = \sum_{\mathrm{k}=1}^{\mathrm{p}} \mathbf{A}_{\mathrm{ik}}\mathbf{B}_{\mathrm{kj}}$$

```
In [133]: a = np.arange(4).reshape(-1,2)

In [134]: a
Out[134]:
array([[0, 1],
       [2, 3]])

In [135]: b = np.arange(-4,0).reshape(-1,2)

In [136]: b
Out[136]:
array([[-4, -3],
       [-2, -1]])

In [137]: a@b
Out[137]:
array([[ -2,  -1],
       [-14,  -9]])
```

1.3 练习

1.3.1 Ex1：利用列表推导式写矩阵乘法

一般的矩阵乘法根据公式，可以由三重循环写出：

```
In [138]: M1 = np.random.rand(2,3)

In [139]: M2 = np.random.rand(3,4)

In [140]: res = np.empty((M1.shape[0],M2.shape[1]))

In [141]: for i in range(M1.shape[0]):
   .....:     for j in range(M2.shape[1]):
   .....:         item = 0
   .....:         for k in range(M1.shape[1]):
   .....:             item += M1[i][k] * M2[k][j]
   .....:         res[i][j] = item
   .....:

In [142]: ((M1@M2 - res) < 1e-15).all() # 排除数值误差
Out[142]: True
```

请将其改写为列表推导式的形式。

1.3.2 Ex2：更新矩阵

设矩阵 $A_{m\times n}$ ，现在对 A 中的每一个元素进行更新生成矩阵 B ，更新方法是 $B_{ij} = A_{ij}\sum_{k=1}^{n}\frac{1}{A_{ik}}$ ，例如下面的矩阵为 A ，则 $B_{2,2} = 5\times(\frac{1}{4}+\frac{1}{5}+\frac{1}{6}) = \frac{37}{12}$ ，请利用 Numpy 高效实现。

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

1.3.3 Ex3：卡方统计量

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

$$\chi^2=\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{(A_{ij}-B_{ij})^2}{B_{ij}}$$

请利用 Numpy 对给定的矩阵 A 计算 χ^2 。

```
In [143]: np.random.seed(0)

In [144]: A = np.random.randint(10, 20, (8, 5))
```

1.3.4 Ex4：改进矩阵计算的性能

设 Z 为 $m\times n$ 的矩阵，B 和 U 分别是 $m\times p$ 和 $p\times n$ 的矩阵，B_i 为 B 的第 i 行，U_j 为 U 的第 j 列，下面定义 $R=\sum_{i=1}^{m}\sum_{j=1}^{n}\|B_i-U_j\|_2^2Z_{ij}$ ，其中 $\|\mathbf{a}\|_2^2$ 表示向量 $\mathbf{a}$ 的分量平方和 $\sum_i a_i^2$ 。

现有某人根据如下给定的样例数据计算 R 的值，请充分利用 Numpy 中的函数，基于此问题改进这段代码的性能。

```
In [145]: np.random.seed(0)

In [146]: m, n, p = 100, 80, 50

In [147]: B = np.random.randint(0, 2, (m, p))

In [148]: U = np.random.randint(0, 2, (p, n))

In [149]: Z = np.random.randint(0, 2, (m, n))
```

```
In [150]: def solution(B=B, U=U, Z=Z):
   .....:     L_res = []
   .....:     for i in range(m):
   .....:         for j in range(n):
   .....:             norm_value = ((B[i]-U[:,j])**2).sum()
   .....:             L_res.append(norm_value*Z[i][j])
   .....:     return sum(L_res)
   .....:

In [151]: solution(B, U, Z)
Out[151]: 100566
```

1.3.5 Ex5：连续整数的最大长度

输入一个整数的 Numpy 数组，返回其中递增连续整数子数组的最大长度，正向是指递增方向。例如，输入 [1,2,5,6,7]，[5,6,7] 为具有最大长度的连续整数子数组，因此输出 3；输入 [3,2,1,2,3,4,6]，[1,2,3,4] 为具有最大长度的连续整数子数组，因此输出 4。请充分利用 Numpy 的内置函数完成。（提示：考虑使用 nonzero, diff 函数）

第 2 章　pandas 基础

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

In [2]: import pandas as pd
```

在开始学习前，请保证 pandas 的版本号不低于如下所示的版本，否则请务必升级！

```
In [3]: pd.__version__
Out[3]: '1.1.3'
```

2.1　文件的读取和写入

2.1.1　文件读取

pandas 可以读取的文件格式有很多，这里主要介绍读取 csv, excel, txt 文件。

```
In [4]: df_csv = pd.read_csv('data/my_csv.csv')

In [5]: df_csv
Out[5]:
   col1 col2  col3    col4      col5
0     2    a   1.4   apple  2020/1/1
1     3    b   3.4  banana  2020/1/2
2     6    c   2.5  orange  2020/1/5
3     5    d   3.2   lemon  2020/1/7

In [6]: df_txt = pd.read_table('data/my_table.txt')

In [7]: df_txt
Out[7]:
   col1 col2  col3             col4
0     2    a   1.4   apple 2020/1/1
```

(continues on next page)

(continued from previous page)

```
1     3    b   3.4  banana 2020/1/2
2     6    c   2.5  orange 2020/1/5
3     5    d   3.2   lemon 2020/1/7

In [8]: df_excel = pd.read_excel('data/my_excel.xlsx')

In [9]: df_excel
Out[9]:
   col1 col2  col3    col4      col5
0     2    a   1.4   apple  2020/1/1
1     3    b   3.4  banana  2020/1/2
2     6    c   2.5  orange  2020/1/5
3     5    d   3.2   lemon  2020/1/7
```

这里有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，索引的内容将会在第三章进行详述，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，关于时间序列的有关内容将在第十章讲解，nrows 表示读取的数据行数。上面这些参数在上述的三个函数里都可以使用。

```
In [10]: pd.read_table('data/my_table.txt', header=None)
Out[10]:
      0     1     2               3
0  col1  col2  col3            col4
1     2     a   1.4   apple 2020/1/1
2     3     b   3.4  banana 2020/1/2
3     6     c   2.5  orange 2020/1/5
4     5     d   3.2   lemon 2020/1/7

In [11]: pd.read_csv('data/my_csv.csv', index_col=['col1', 'col2'])
Out[11]:
           col3    col4      col5
col1 col2
2    a      1.4   apple  2020/1/1
3    b      3.4  banana  2020/1/2
6    c      2.5  orange  2020/1/5
5    d      3.2   lemon  2020/1/7

In [12]: pd.read_table('data/my_table.txt', usecols=['col1', 'col2'])
Out[12]:
   col1 col2
0     2    a
1     3    b
2     6    c
```

(continues on next page)

(continued from previous page)

```
3     5    d

In [13]: pd.read_csv('data/my_csv.csv', parse_dates=['col5'])
Out[13]:
   col1 col2  col3    col4       col5
0     2    a   1.4   apple 2020-01-01
1     3    b   3.4  banana 2020-01-02
2     6    c   2.5  orange 2020-01-05
3     5    d   3.2   lemon 2020-01-07

In [14]: pd.read_excel('data/my_excel.xlsx', nrows=2)
Out[14]:
   col1 col2  col3    col4      col5
0     2    a   1.4   apple  2020/1/1
1     3    b   3.4  banana  2020/1/2
```

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。例如，下面的读取的表以 |||| 为分割：

```
In [15]: pd.read_table('data/my_table_special_sep.txt')
Out[15]:
              col1 |||| col2
0  TS |||| This is an apple.
1    GQ |||| My name is Bob.
2         WT |||| Well done!
3    PT |||| May I help you?
```

上面的结果显然不是理想的，这时可以使用 sep ，同时需要指定引擎为 python ：

```
In [16]: pd.read_table('data/my_table_special_sep.txt',
   ....:               sep=' \|\|\|\| ', engine='python')
   ....:
Out[16]:
  col1               col2
0   TS  This is an apple.
1   GQ    My name is Bob.
2   WT         Well done!
3   PT    May I help you?
```

sep 是正则参数

在使用 read_table 的时候需要注意，参数 sep 中使用的是正则表达式，因此需要对 | 进行转义变成 \| ，否则无法读取到正确的结果。有关正则表达式的基本内容可以参考第八章或者其他相

关资料。

2.1.2 数据写入

一般在数据写入中，最常用的操作是把 index 设置为 False ，特别当索引没有特殊意义的时候，这样的行为能把索引在保存的时候去除。

```
In [17]: df_csv.to_csv('data/my_csv_saved.csv', index=False)

In [18]: df_excel.to_excel('data/my_excel_saved.xlsx', index=False)
```

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：

```
In [19]: df_txt.to_csv('data/my_txt_saved.txt', sep='\t', index=False)
```

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。

```
In [20]: print(df_csv.to_markdown())
|    |   col1 | col2   |   col3 | col4   | col5     |
|---:|-------:|:-------|-------:|:-------|:---------|
|  0 |      2 | a      |    1.4 | apple  | 2020/1/1 |
|  1 |      3 | b      |    3.4 | banana | 2020/1/2 |
|  2 |      6 | c      |    2.5 | orange | 2020/1/5 |
|  3 |      5 | d      |    3.2 | lemon  | 2020/1/7 |

In [21]: print(df_csv.to_latex())
\begin{tabular}{lrlrll}
\toprule
{} &  col1 & col2 &  col3 &    col4 &      col5 \\
\midrule
0 &     2 &    a &   1.4 &   apple &  2020/1/1 \\
1 &     3 &    b &   3.4 &  banana &  2020/1/2 \\
2 &     6 &    c &   2.5 &  orange &  2020/1/5 \\
3 &     5 &    d &   3.2 &   lemon &  2020/1/7 \\
\bottomrule
\end{tabular}
```

2.2 基本数据结构

pandas 中具有两种基本的数据存储结构，存储一维 values 的 Series 和存储二维 values 的 DataFrame ，在这两种结构上定义了很多的属性和方法。

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。

```
In [22]: s = pd.Series(data = [100, 'a', {'dic1':5}],
   ....:               index = pd.Index(['id1', 20, 'third'], name='my_idx'),
   ....:               dtype = 'object',
   ....:               name = 'my_name')
   ....:

In [23]: s
Out[23]:
my_idx
id1               100
20                  a
third    {'dic1': 5}
Name: my_name, dtype: object
```

object 类型

object 代表了一种混合类型，正如上面的例子中存储了整数、字符串以及 Python 的字典数据结构。此外，目前 pandas 把纯字符串序列也默认认为是一种 object 类型的序列，但它也可以用 string 类型存储，文本序列的内容会在第八章中讨论。

对于这些属性，可以通过 . 的方式来获取：

```
In [24]: s.values
Out[24]: array([100, 'a', {'dic1': 5}], dtype=object)

In [25]: s.index
Out[25]: Index(['id1', 20, 'third'], dtype='object', name='my_idx')

In [26]: s.dtype
Out[26]: dtype('O')
```

(continues on next page)

(continued from previous page)

```
In [27]: s.name
Out[27]: 'my_name'
```

利用 .shape 可以获取序列的长度：

```
In [28]: s.shape
Out[28]: (3,)
```

索引是 pandas 中最重要的概念之一，它将在第三章中被详细地讨论。如果想要取出单个索引对应的值，可以通过 [index_item] 可以取出。

```
In [29]: s['third']
Out[29]: {'dic1': 5}
```

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：

```
In [30]: data = [[1, 'a', 1.2], [2, 'b', 2.2], [3, 'c', 3.2]]

In [31]: df = pd.DataFrame(data = data,
   ....:                   index = ['row_%d'%i for i in range(3)],
   ....:                   columns=['col_0', 'col_1', 'col_2'])
   ....:

In [32]: df
Out[32]:
       col_0 col_1  col_2
row_0      1     a    1.2
row_1      2     b    2.2
row_2      3     c    3.2
```

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：

```
In [33]: df = pd.DataFrame(data = {'col_0': [1,2,3], 'col_1':list('abc'),
   ....:                           'col_2': [1.2, 2.2, 3.2]},
   ....:                   index = ['row_%d'%i for i in range(3)])
   ....:

In [34]: df
Out[34]:
       col_0 col_1  col_2
```

(continues on next page)

(continued from previous page)

```
row_0      1     a    1.2
row_1      2     b    2.2
row_2      3     c    3.2
```

由于这种映射关系，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 ：

```
In [35]: df['col_0']
Out[35]:
row_0    1
row_1    2
row_2    3
Name: col_0, dtype: int64

In [36]: df[['col_0', 'col_1']]
Out[36]:
       col_0 col_1
row_0      1     a
row_1      2     b
row_2      3     c
```

与 Series 类似，在数据框中同样可以取出相应的属性：

```
In [37]: df.values
Out[37]:
array([[1, 'a', 1.2],
       [2, 'b', 2.2],
       [3, 'c', 3.2]], dtype=object)

In [38]: df.index
Out[38]: Index(['row_0', 'row_1', 'row_2'], dtype='object')

In [39]: df.columns
Out[39]: Index(['col_0', 'col_1', 'col_2'], dtype='object')

In [40]: df.dtypes # 返回的是值为相应列数据类型的 Series
Out[40]:
col_0      int64
col_1     object
col_2    float64
dtype: object

In [41]: df.shape
Out[41]: (3, 3)
```

通过 .T 可以把 DataFrame 进行转置：

```
In [42]: df.T
Out[42]:
      row_0 row_1 row_2
col_0     1     2     3
col_1     a     b     c
col_2   1.2   2.2   3.2
```

2.3 常用基本函数

为了进行举例说明，在接下来的部分和其余章节都将会使用一份 learn_pandas.csv 的虚拟数据集，它记录了四所学校学生的体测个人信息。

```
In [43]: df = pd.read_csv('data/learn_pandas.csv')

In [44]: df.columns
Out[44]:
Index(['School', 'Grade', 'Name', 'Gender', 'Height', 'Weight', 'Transfer',
       'Test_Number', 'Test_Date', 'Time_Record'],
      dtype='object')
```

上述列名依次代表学校、年级、姓名、性别、身高、体重、是否为转系生、体测场次、测试时间、1000 米成绩，本章只需使用其中的前七列。

```
In [45]: df = df[df.columns[:7]]
```

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：

```
In [46]: df.head(2)
Out[46]:
                          School     Grade            Name  Gender  Height  Weight Transfer
0  Shanghai Jiao Tong University  Freshman    Gaopeng Yang  Female   158.9    46.0        N
1              Peking University  Freshman  Changqiang You    Male   166.5    70.0        N

In [47]: df.tail(3)
Out[47]:
                            School      Grade            Name  Gender  Height  Weight Transfer
197  Shanghai Jiao Tong University     Senior  Chengqiang Chu  Female   153.9    45.0        N
```

(continues on next page)

(continued from previous page)

```
198  Shanghai Jiao Tong University     Senior   Chengmei Shen    Male   175.3    71.0        N
199            Tsinghua University  Sophomore     Chunpeng Lv    Male   155.7    51.0        N
```

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：

```
In [48]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   School    200 non-null    object
 1   Grade     200 non-null    object
 2   Name      200 non-null    object
 3   Gender    200 non-null    object
 4   Height    183 non-null    float64
 5   Weight    189 non-null    float64
 6   Transfer  188 non-null    object
dtypes: float64(2), object(5)
memory usage: 11.1+ KB

In [49]: df.describe()
Out[49]:
           Height      Weight
count  183.000000  189.000000
mean   163.218033   55.015873
std      8.608879   12.824294
min    145.400000   34.000000
25%    157.150000   46.000000
50%    161.900000   51.000000
75%    167.500000   65.000000
max    193.900000   89.000000
```

更全面的数据汇总

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，它将在第十一章被再次提到。

2.3.2 特征统计函数

在 Series 和 DataFrame 上定义了许多统计函数，最常见的是 sum, mean, median, var, std, max, min 。例如，选出身高和体重列进行演示：

```
In [50]: df_demo = df[['Height', 'Weight']]

In [51]: df_demo.mean()
Out[51]:
Height    163.218033
Weight     55.015873
dtype: float64

In [52]: df_demo.max()
Out[52]:
Height    193.9
Weight     89.0
dtype: float64
```

此外，需要介绍的是 quantile, count, idxmax 这三个函数，它们分别返回的是分位数、非缺失值个数、最大值对应的索引：

```
In [53]: df_demo.quantile(0.75)
Out[53]:
Height    167.5
Weight     65.0
Name: 0.75, dtype: float64

In [54]: df_demo.count()
Out[54]:
Height    183
Weight    189
dtype: int64

In [55]: df_demo.idxmax() # idxmin 是对应的函数
Out[55]:
Height    193
Weight      2
dtype: int64
```

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：

```
In [56]: df_demo.mean(axis=1).head() # 在这个数据集上体重和身高的均值并没有意义
```

(continues on next page)

(continued from previous page)

```
Out[56]:
0    102.45
1    118.25
2    138.95
3     41.00
4    124.00
dtype: float64
```

2.3.3 唯一值函数

对序列使用 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：

```
In [57]: df['School'].unique()
Out[57]:
array(['Shanghai Jiao Tong University', 'Peking University',
       'Fudan University', 'Tsinghua University'], dtype=object)

In [58]: df['School'].nunique()
Out[58]: 4
```

value_counts 可以得到唯一值和其对应出现的频数：

```
In [59]: df['School'].value_counts()
Out[59]:
Tsinghua University              69
Shanghai Jiao Tong University    57
Fudan University                 40
Peking University                34
Name: School, dtype: int64
```

如果想要观察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。

```
In [60]: df_demo = df[['Gender','Transfer','Name']]

In [61]: df_demo.drop_duplicates(['Gender', 'Transfer'])
Out[61]:
    Gender Transfer            Name
0   Female        N    Gaopeng Yang
1     Male        N  Changqiang You
12  Female      NaN        Peng You
```

(continues on next page)

(continued from previous page)

```
21    Male      NaN   Xiaopeng Shen
36    Male        Y    Xiaojuan Qin
43  Female        Y      Gaoli Feng

In [62]: df_demo.drop_duplicates(['Gender', 'Transfer'], keep='last')
Out[62]:
     Gender Transfer            Name
147    Male      NaN        Juan You
150    Male        Y   Chengpeng You
169  Female        Y   Chengquan Qin
194  Female      NaN     Yanmei Qian
197  Female        N  Chengqiang Chu
199    Male        N     Chunpeng Lv

In [63]: df_demo.drop_duplicates(['Name', 'Gender'],
   ....:                         keep=False).head() # 保留只出现过一次的性别和姓名组合
   ....:
Out[63]:
   Gender Transfer            Name
0  Female        N    Gaopeng Yang
1    Male        N  Changqiang You
2    Male        N         Mei Sun
4    Male        N     Gaojuan You
5  Female        N     Xiaoli Qian

In [64]: df['School'].drop_duplicates() # 在 Series 上也可以使用
Out[64]:
0    Shanghai Jiao Tong University
1                Peking University
3                 Fudan University
5              Tsinghua University
Name: School, dtype: object
```

此外，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。其返回的序列，把重复元素设为 True ，否则为 False 。drop_duplicates 等价于把 duplicated 为 True 的对应行剔除。

```
In [65]: df_demo.duplicated(['Gender', 'Transfer']).head()
Out[65]:
0    False
1    False
2     True
3     True
```

(continues on next page)

(continued from previous page)

```
4     True
dtype: bool

In [66]: df['School'].duplicated().head() # 在 Series 上也可以使用
Out[66]:
0    False
1    False
2     True
3    False
4     True
Name: School, dtype: bool
```

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。其中映射替换包含 replace 方法、第八章中的 str.replace 方法以及第九章中的 cat.codes 方法，此处介绍 replace 的用法。

在 replace 中，可以通过字典构造，或者传入两个列表来进行替换：

```
In [67]: df['Gender'].replace({'Female':0, 'Male':1}).head()
Out[67]:
0    0
1    1
2    1
3    0
4    1
Name: Gender, dtype: int64

In [68]: df['Gender'].replace(['Female', 'Male'], [0, 1]).head()
Out[68]:
0    0
1    1
2    1
3    0
4    1
Name: Gender, dtype: int64
```

另外，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。从下面的例子可以看到，它们的结果是不同的：

```
In [69]: s = pd.Series(['a', 1, 'b', 2, 1, 1, 'a'])
```

(continues on next page)

(continued from previous page)

```
In [70]: s.replace([1, 2], method='ffill')
Out[70]:
0    a
1    a
2    b
3    b
4    b
5    b
6    a
dtype: object

In [71]: s.replace([1, 2], method='bfill')
Out[71]:
0    a
1    b
2    b
3    a
4    a
5    a
6    a
dtype: object
```

正则替换请使用 str.replace

> 虽然对于 replace 而言可以使用正则替换，但是当前版本下对于 string 类型的正则替换还存在 bug ，因此如有此需求，请选择 str.replace 进行替换操作，具体的方式将在第八章中讲解。

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。

```
In [72]: s = pd.Series([-1, 1.2345, 100, -50])

In [73]: s.where(s<0)
Out[73]:
0    -1.0
1     NaN
2     NaN
3   -50.0
dtype: float64

In [74]: s.where(s<0, 100)
```

(continues on next page)

(continued from previous page)

```
Out[74]:
0     -1.0
1    100.0
2    100.0
3    -50.0
dtype: float64

In [75]: s.mask(s<0)
Out[75]:
0         NaN
1      1.2345
2    100.0000
3         NaN
dtype: float64

In [76]: s.mask(s<0, -50)
Out[76]:
0    -50.0000
1      1.2345
2    100.0000
3    -50.0000
dtype: float64
```

需要注意的是，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：

```
In [77]: s_condition= pd.Series([True,False,False,True],index=s.index)

In [78]: s.mask(s_condition, -50)
Out[78]:
0    -50.0000
1      1.2345
2    100.0000
3    -50.0000
dtype: float64
```

数值替换包含了 round, abs, clip 方法，它们分别表示取整、取绝对值和截断：

```
In [79]: s = pd.Series([-1, 1.2345, 100, -50])

In [80]: s.round(2)
Out[80]:
0     -1.00
1      1.23
```

(continues on next page)

(continued from previous page)

```
2    100.00
3    -50.00
dtype: float64

In [81]: s.abs()
Out[81]:
0      1.0000
1      1.2345
2    100.0000
3     50.0000
dtype: float64

In [82]: s.clip(0, 2) # 前两个数分别表示上下截断边界
Out[82]:
0    0.0000
1    1.2345
2    2.0000
3    0.0000
dtype: float64
```

练一练

在 clip 中，超过边界的只能截断为边界值，如果要把超出边界的替换为自定义的值，应当如何做？

2.3.5 排序函数

排序共有两种方式，其一为值排序，其二为索引排序，对应的函数是 sort_values 和 sort_index 。

为了演示排序函数，下面先利用 set_index 方法把年级和姓名两列作为索引，多级索引的内容和索引设置的方法将在第三章进行详细讲解。

```
In [83]: df_demo = df[['Grade', 'Name', 'Height',
   ....:               'Weight']].set_index(['Grade','Name'])
   ....:
```

对身高进行排序，默认参数 ascending=True 为升序：

```
In [84]: df_demo.sort_values('Height').head()
Out[84]:
                      Height  Weight
Grade     Name
```

(continues on next page)

(continued from previous page)

```
Junior    Xiaoli Chu      145.4    34.0
Senior    Gaomei Lv       147.3    34.0
Sophomore Peng Han        147.8    34.0
Senior    Changli Lv      148.7    41.0
Sophomore Changjuan You   150.5    40.0
```

```
In [85]: df_demo.sort_values('Height', ascending=False).head()
Out[85]:
                        Height  Weight
Grade     Name
Senior    Xiaoqiang Qin   193.9    79.0
          Mei Sun         188.9    89.0
          Gaoli Zhao      186.5    83.0
Freshman  Qiang Han       185.3    87.0
Senior    Qiang Zheng     183.9    87.0
```

在排序中，进场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：

```
In [86]: df_demo.sort_values(['Weight','Height'],ascending=[True,False]).head()
Out[86]:
                       Height  Weight
Grade     Name
Sophomore Peng Han      147.8    34.0
Senior    Gaomei Lv     147.3    34.0
Junior    Xiaoli Chu    145.4    34.0
Sophomore Qiang Zhou    150.5    36.0
Freshman  Yanqiang Xu   152.4    38.0
```

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，此时需要指定索引层的名字或者层号，用参数 level 表示。另外，需要注意的是字符串的排列顺序由字母顺序决定。

```
In [87]: df_demo.sort_index(level=['Grade','Name'],ascending=[True,False]).head()
Out[87]:
                        Height  Weight
Grade     Name
Freshman  Yanquan Wang    163.5    55.0
          Yanqiang Xu     152.4    38.0
          Yanqiang Feng   162.3    51.0
          Yanpeng Lv        NaN    65.0
          Yanli Zhang     165.1    52.0
```

2.3.6 apply 方法

apply 方法常用于 DataFrame 的行迭代或者列迭代，它的 axis 含义与第 2 小节中的统计聚合函数一致，apply 的参数往往是一个以序列为输入的函数。例如对于 .mean() ，使用 apply 可以如下地写出：

```
In [88]: df_demo = df[['Height', 'Weight']]

In [89]: def my_mean(x):
   ....:     res = x.mean()
   ....:     return res
   ....:

In [90]: df_demo.apply(my_mean)
Out[90]:
Height    163.218033
Weight     55.015873
dtype: float64
```

同样的，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：

```
In [91]: df_demo.apply(lambda x:x.mean())
Out[91]:
Height    163.218033
Weight     55.015873
dtype: float64
```

若指定 axis=1 ，那么每次传入函数的就是行元素组成的 Series ，其结果与之前的逐行均值结果一致。

```
In [92]: df_demo.apply(lambda x:x.mean(), axis=1).head()
Out[92]:
0    102.45
1    118.25
2    138.95
3     41.00
4    124.00
dtype: float64
```

这里再举一个例子：mad 函数返回的是一个序列中偏离该序列均值的绝对值大小的均值，例如序列 1,3,7,10 中，均值为 5.25，每一个元素偏离的绝对值为 4.25,2.25,1.75,4.75，这个偏离序列的均值为 3.25。现在利用 apply 计算升高和体重的 mad 指标：

```
In [93]: df_demo.apply(lambda x:(x-x.mean()).abs().mean())
Out[93]:
Height     6.707229
```

(continues on next page)

(continued from previous page)

```
Weight    10.391870
dtype: float64
```

这与使用内置的 mad 函数计算结果一致：

```
In [94]: df_demo.mad()
Out[94]:
Height     6.707229
Weight    10.391870
dtype: float64
```

谨慎使用 apply

得益于传入自定义函数的处理，apply 的自由度很高，但这是以性能为代价的。一般而言，使用 pandas 的内置函数处理和 apply 来处理同一个任务，其速度会相差较多，因此只有在确实存在自定义需求的情境下才考虑使用 apply 。

2.4 窗口对象

pandas 中有 3 类窗口，分别是滑动窗口 rolling 、扩张窗口 expanding 以及指数加权窗口 ewm 。需要说明的是，以日期偏置为窗口大小的滑动窗口将在第十章讨论，指数加权窗口见本章练习。

2.4.1 滑窗对象

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 。

```
In [95]: s = pd.Series([1,2,3,4,5])

In [96]: roller = s.rolling(window = 3)

In [97]: roller
Out[97]: Rolling [window=3,center=False,axis=0]
```

在得到了滑窗对象后，能够使用相应的聚合函数进行计算，需要注意的是窗口包含当前行所在的元素，例如在第四个位置进行均值运算时，应当计算 (2+3+4)/3，而不是 (1+2+3)/3：

```
In [98]: roller.mean()
Out[98]:
0    NaN
```

(continues on next page)

(continued from previous page)

```
1    NaN
2    2.0
3    3.0
4    4.0
dtype: float64

In [99]: roller.sum()
Out[99]:
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

对于滑动相关系数或滑动协方差的计算，可以如下写出：

```
In [100]: s2 = pd.Series([1,2,6,16,30])

In [101]: roller.cov(s2)
Out[101]:
0     NaN
1     NaN
2     2.5
3     7.0
4    12.0
dtype: float64

In [102]: roller.corr(s2)
Out[102]:
0         NaN
1         NaN
2    0.944911
3    0.970725
4    0.995402
dtype: float64
```

此外，还支持使用 apply 传入自定义函数，其传入值是对应窗口的 Series ，例如上述的均值函数可以等效表示：

```
In [103]: roller.apply(lambda x:x.mean())
Out[103]:
0    NaN
```

(continues on next page)

(continued from previous page)

```
1    NaN
2    2.0
3    3.0
4    4.0
dtype: float64
```

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，分别表示取向前第 n 个元素的值、与向前第 n 个元素做差（与 Numpy 中不同，后者表示 n 阶差分）、与向前第 n 个元素相比计算增长率。这里的 n 可以为负，表示反方向的类似操作。

```
In [104]: s = pd.Series([1,3,6,10,15])

In [105]: s.shift(2)
Out[105]:
0    NaN
1    NaN
2    1.0
3    3.0
4    6.0
dtype: float64

In [106]: s.diff(3)
Out[106]:
0     NaN
1     NaN
2     NaN
3     9.0
4    12.0
dtype: float64

In [107]: s.pct_change()
Out[107]:
0         NaN
1    2.000000
2    1.000000
3    0.666667
4    0.500000
dtype: float64

In [108]: s.shift(-1)
Out[108]:
0     3.0
1     6.0
```

(continues on next page)

(continued from previous page)

```
2    10.0
3    15.0
4     NaN
dtype: float64

In [109]: s.diff(-2)
Out[109]:
0   -5.0
1   -7.0
2   -9.0
3    NaN
4    NaN
dtype: float64
```

将其视作类滑窗函数的原因是，它们的功能可以用窗口大小为 n+1 的 rolling 方法等价代替：

```
In [110]: s.rolling(3).apply(lambda x:list(x)[0]) # s.shift(2)
Out[110]:
0    NaN
1    NaN
2    1.0
3    3.0
4    6.0
dtype: float64

In [111]: s.rolling(4).apply(lambda x:list(x)[-1]-list(x)[0]) # s.diff(3)
Out[111]:
0     NaN
1     NaN
2     NaN
3     9.0
4    12.0
dtype: float64

In [112]: def my_pct(x):
   .....:     L = list(x)
   .....:     return L[-1]/L[0]-1
   .....:

In [113]: s.rolling(2).apply(my_pct) # s.pct_change()
Out[113]:
0         NaN
1    2.000000
```

(continues on next page)

(continued from previous page)

```
2    1.000000
3    0.666667
4    0.500000
dtype: float64
```

练一练

rolling 对象的默认窗口方向都是向前的，某些情况下用户需要向后的窗口，例如对 1,2,3 设定向后窗口为 2 的 sum 操作，结果为 3,5,NaN，此时应该如何实现向后的滑窗操作？（提示：使用 shift ）

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]。

```
In [114]: s = pd.Series([1, 3, 6, 10])

In [115]: s.expanding().mean()
Out[115]:
0    1.000000
1    2.000000
2    3.333333
3    5.000000
dtype: float64
```

练一练

cummax, cumsum, cumprod 函数是典型的类扩张窗口函数，请使用 expanding 对象依次实现它们。

2.5 练习

2.5.1 Ex1：口袋妖怪数据集

现有一份口袋妖怪的数据集，下面进行一些背景说明：

- # 代表全国图鉴编号，不同行存在相同数字则表示为该妖怪的不同状态
- 妖怪具有单属性和双属性两种，对于单属性的妖怪，Type 2 为缺失值
- Total, HP, Attack, Defense, Sp. Atk, Sp. Def, Speed 分别代表种族值、体力、物攻、防御、特攻、特防、速度，其中种族值为后 6 项之和

```
In [116]: df = pd.read_csv('data/pokemon.csv')

In [117]: df.head(3)
Out[117]:
   #       Name Type 1  Type 2  Total  HP  Attack  Defense  Sp. Atk  Sp. Def  Speed
0  1  Bulbasaur  Grass  Poison    318  45      49       49       65       65     45
1  2    Ivysaur  Grass  Poison    405  60      62       63       80       80     60
2  3   Venusaur  Grass  Poison    525  80      82       83      100      100     80
```

1. 对 HP, Attack, Defense, Sp. Atk, Sp. Def, Speed 进行加总，验证是否为 Total 值。
2. 对于 # 重复的妖怪只保留第一条记录，解决以下问题：

(a) 求第一属性的种类数量和前三多数量对应的种类

(b) 求第一属性和第二属性的组合种类

(c) 求尚未出现过的属性组合

3. 按照下述要求，构造 Series：

(a) 取出物攻，超过 120 的替换为 high，不足 50 的替换为 low，否则设为 mid

(b) 取出第一属性，分别用 replace 和 apply 替换所有字母为大写

(c) 求每个妖怪六项能力的离差，即所有能力中偏离中位数最大的值，添加到 df 并从大到小排序

2.5.2 Ex2：指数加权窗口

1. 作为扩张窗口的 ewm 窗口

在扩张窗口中，用户可以使用各类函数进行历史的累计指标统计，但这些内置的统计函数往往把窗口中的所有元素赋予了同样的权重。事实上，可以给出不同的权重来赋给窗口中的元素，指数加权窗口就是这样一种特殊的扩张窗口。

其中，最重要的参数是 alpha ，它决定了默认情况下的窗口权重为 $w_i = (1-\alpha)^i, i \in \{0, 1, ..., t\}$ ，其中 $i = t$ 表示当前元素，$i = 0$ 表示序列的第一个元素。

从权重公式可以看出，离开当前值越远则权重越小，若记原序列为 x ，更新后的当前元素为 y_t ，此时通过加权公式归一化后可知：

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

对于 Series 而言，可以用 ewm 对象如下计算指数平滑后的序列：

```
In [118]: np.random.seed(0)

In [119]: s = pd.Series(np.random.randint(-1,2,30).cumsum())

In [120]: s.head()
Out[120]:
0   -1
1   -1
2   -2
3   -2
4   -2
dtype: int32

In [121]: s.ewm(alpha=0.2).mean().head()
Out[121]:
0   -1.000000
1   -1.000000
2   -1.409836
3   -1.609756
4   -1.725845
dtype: float64
```

请用 expanding 窗口实现。

2. 作为滑动窗口的 ewm 窗口

从第 1 问中可以看到，ewm 作为一种扩张窗口的特例，只能从序列的第一个元素开始加权。现在希望给定一个限制窗口 n ，只对包含自身最近的 n 个窗口进行滑动加权平滑。请根据滑窗函数，给出新的 w_i 与 y_t 的更新公式，并通过 rolling 窗口实现这一功能。

第 3 章　索引

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

In [2]: import pandas as pd
```

3.1　索引器

3.1.1　表的列索引

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 ，例如从表中取出姓名一列：

```
In [3]: df = pd.read_csv('data/learn_pandas.csv',
   ...:                  usecols = ['School', 'Grade', 'Name', 'Gender',
   ...:                             'Weight', 'Transfer'])
   ...:

In [4]: df['Name'].head()
Out[4]:
0      Gaopeng Yang
1    Changqiang You
2          Mei Sun
3      Xiaojuan Sun
4       Gaojuan You
Name: Name, dtype: object
```

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 ，例如从表中取出性别和姓名两列：

```
In [5]: df[['Gender', 'Name']].head()
Out[5]:
```

(continues on next page)

(continued from previous page)

```
   Gender            Name
0  Female    Gaopeng Yang
1    Male  Changqiang You
2    Male         Mei Sun
3  Female    Xiaojuan Sun
4    Male     Gaojuan You
```

此外，若要取出单列，且列名中不包含空格，则可以用 . 列名 取出，这和 [列名] 是等价的：

```
In [6]: df.Name.head()
Out[6]:
0      Gaopeng Yang
1    Changqiang You
2           Mei Sun
3      Xiaojuan Sun
4       Gaojuan You
Name: Name, dtype: object
```

3.1.2 序列的行索引

【a】以字符串为索引的 Series

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：

```
In [7]: s = pd.Series([1, 2, 3, 4, 5, 6],
   ...:                index=['a', 'b', 'a', 'a', 'a', 'c'])
   ...:

In [8]: s['a']
Out[8]:
a    1
a    3
a    4
a    5
dtype: int64

In [9]: s['b']
Out[9]: 2
```

如果取出多个索引的对应元素，则可以使用 [items 的列表] ：

```
In [10]: s[['c', 'b']]
Out[10]:
c    6
b    2
dtype: int64
```

如果想要取出某两个索引之间的元素，并且这两个索引是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：

```
In [11]: s['c': 'b': -2]
Out[11]:
c    6
a    4
b    2
dtype: int64
```

【b】以整数为索引的 Series

在使用数据的读入函数时，如果不特别指定所对应的列作为索引，那么会生成从 0 开始的整数索引作为默认索引。当然，任意一组符合长度要求的整数都可以作为索引。

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：

```
In [12]: s = pd.Series(['a', 'b', 'c', 'd', 'e', 'f'],
   ....:                index=[1, 3, 1, 2, 5, 4])
   ....:

In [13]: s[1]
Out[13]:
1    a
1    c
dtype: object

In [14]: s[[2,3]]
Out[14]:
2    d
3    b
dtype: object
```

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：

```
In [15]: s[1:-1:2]
Out[15]:
3    b
```

(continues on next page)

(continued from previous page)

```
2    d
dtype: object
```

关于索引类型的说明

> 如果不想陷入麻烦，那么请不要把纯浮点以及任何混合类型（字符串、整数、浮点类型等的混合）作为索引，否则可能会在具体的操作时报错或者返回非预期的结果，并且在实际的数据分析中也不存在这样做的动机。

3.1.3 loc 索引器

前面讲到了对 DataFrame 的列进行选取，下面要讨论其行的选取。对于表而言，有两种索引器，一种是基于元素的 loc 索引器，另一种是基于 位置的 iloc 索引器。

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。其中，* 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，下面将依次说明。

为了演示相应操作，先利用 set_index 方法把 Name 列设为索引，关于该函数的其他用法将在多级索引一章介绍。

```
In [16]: df_demo = df.set_index('Name')

In [17]: df_demo.head()
Out[17]:
                                        School      Grade  Gender  Weight Transfer
Name
Gaopeng Yang    Shanghai Jiao Tong University   Freshman  Female    46.0        N
Changqiang You              Peking University   Freshman    Male    70.0        N
Mei Sun         Shanghai Jiao Tong University     Senior    Male    89.0        N
Xiaojuan Sun                 Fudan University  Sophomore  Female    41.0        N
Gaojuan You                  Fudan University  Sophomore    Male    74.0        N
```

【a】* 为单个元素

此时，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 ：

```
In [18]: df_demo.loc['Qiang Sun'] # 多个人叫此名字
Out[18]:
                               School      Grade  Gender  Weight Transfer
Name
```

(continues on next page)

(continued from previous page)

```
Qiang Sun            Tsinghua University     Junior  Female    53.0        N
Qiang Sun            Tsinghua University  Sophomore  Female    40.0        N
Qiang Sun  Shanghai Jiao Tong University     Junior  Female     NaN        N

In [19]: df_demo.loc['Quan Zhao'] # 名字唯一
Out[19]:
School      Shanghai Jiao Tong University
Grade                              Junior
Gender                             Female
Weight                                 53
Transfer                                N
Name: Quan Zhao, dtype: object
```

也可以同时选择行和列：

```
In [20]: df_demo.loc['Qiang Sun', 'School'] # 返回 Series
Out[20]:
Name
Qiang Sun              Tsinghua University
Qiang Sun              Tsinghua University
Qiang Sun    Shanghai Jiao Tong University
Name: School, dtype: object

In [21]: df_demo.loc['Quan Zhao', 'School'] # 返回单个元素
Out[21]: 'Shanghai Jiao Tong University'
```

【b】* 为元素列表

此时，取出列表中所有元素值对应的行或列：

```
In [22]: df_demo.loc[['Qiang Sun','Quan Zhao'], ['School','Gender']]
Out[22]:
                                School  Gender
Name
Qiang Sun            Tsinghua University  Female
Qiang Sun            Tsinghua University  Female
Qiang Sun  Shanghai Jiao Tong University  Female
Quan Zhao  Shanghai Jiao Tong University  Female
```

【c】* 为切片

之前的 Series 使用字符串索引时提到，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：

```
In [23]: df_demo.loc['Gaojuan You':'Gaoqiang Qian', 'School':'Gender']
Out[23]:
                                    School      Grade  Gender
Name
Gaojuan You                 Fudan University  Sophomore    Male
Xiaoli Qian              Tsinghua University   Freshman  Female
Qiang Chu      Shanghai Jiao Tong University   Freshman  Female
Gaoqiang Qian            Tsinghua University     Junior  Female
```

需要注意的是，如果 DataFrame 使用整数索引，其使用整数切片的时候和上面字符串索引的要求一致，都是 元素切片，包含端点且起点、终点不允许有重复值。

```
In [24]: df_loc_slice_demo = df_demo.copy()

In [25]: df_loc_slice_demo.index = range(df_demo.shape[0],0,-1)

In [26]: df_loc_slice_demo.loc[5:3]
Out[26]:
                         School   Grade  Gender  Weight Transfer
5               Fudan University  Junior  Female    46.0        N
4            Tsinghua University  Senior  Female    50.0        N
3  Shanghai Jiao Tong University  Senior  Female    45.0        N

In [27]: df_loc_slice_demo.loc[3:5] # 没有返回，说明不是整数位置切片
Out[27]:
Empty DataFrame
Columns: [School, Grade, Gender, Weight, Transfer]
Index: []
```

【d】* 为布尔列表

在实际的数据处理中，根据条件来筛选行是极其常见的，此处传入 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。

例如，选出体重超过 70kg 的学生：

```
In [28]: df_demo.loc[df_demo.Weight>70].head()
Out[28]:
                                    School      Grade Gender  Weight Transfer
Name
Mei Sun        Shanghai Jiao Tong University     Senior   Male    89.0        N
Gaojuan You                 Fudan University  Sophomore   Male    74.0        N
Xiaopeng Zhou  Shanghai Jiao Tong University   Freshman   Male    74.0        N
Xiaofeng Sun             Tsinghua University     Senior   Male    71.0        N
Qiang Zheng    Shanghai Jiao Tong University     Senior   Male    87.0        N
```

前面所提到的传入元素列表，也可以通过 isin 方法返回的布尔列表等价写出，例如选出所有大一和大四的同学信息：

```
In [29]: df_demo.loc[df_demo.Grade.isin(['Freshman', 'Senior'])].head()
Out[29]:
                                    School     Grade  Gender  Weight Transfer
Name
Gaopeng Yang    Shanghai Jiao Tong University  Freshman  Female    46.0        N
Changqiang You             Peking University  Freshman    Male    70.0        N
Mei Sun         Shanghai Jiao Tong University    Senior    Male    89.0        N
Xiaoli Qian               Tsinghua University  Freshman  Female    51.0        N
Qiang Chu       Shanghai Jiao Tong University  Freshman  Female    52.0        N
```

对于复合条件而言，可以用 |（或），&（且），~（取反） 的组合来实现，例如选出复旦大学中体重超过 70kg 的大四学生，或者北大男生中体重超过 80kg 的非大四的学生：

```
In [30]: condition_1_1 = df_demo.School == 'Fudan University'

In [31]: condition_1_2 = df_demo.Grade == 'Senior'

In [32]: condition_1_3 = df_demo.Weight > 70

In [33]: condition_1 = condition_1_1 & condition_1_2 & condition_1_3

In [34]: condition_2_1 = df_demo.School == 'Peking University'

In [35]: condition_2_2 = df_demo.Grade == 'Senior'

In [36]: condition_2_3 = df_demo.Weight > 80

In [37]: condition_2 = condition_2_1 & (~condition_2_2) & condition_2_3

In [38]: df_demo.loc[condition_1 | condition_2]
Out[38]:
                           School     Grade Gender  Weight Transfer
Name
Qiang Han       Peking University  Freshman   Male    87.0        N
Chengpeng Zhou   Fudan University    Senior   Male    81.0        N
Changpeng Zhao  Peking University  Freshman   Male    83.0        N
Chengpeng Qian   Fudan University    Senior   Male    73.0        Y
```

练一练

select_dtypes 是一个实用函数，它能够从表中选出相应类型的列，若要选出所有数值型的列，只

需使用 .select_dtypes('number') ，请利用布尔列表选择的方法结合 DataFrame 的 dtypes 属性在 learn_pandas 数据集上实现这个功能。

【e】* 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。假设仍然是上述复合条件筛选的例子，可以把逻辑写入一个函数中再返回，需要注意的是函数的形式参数 x 本质上即为 df_demo ：

```
In [39]: 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
   ....:

In [40]: df_demo.loc[condition]
Out[40]:
                          School     Grade Gender  Weight Transfer
Name
Qiang Han       Peking University  Freshman   Male    87.0        N
Chengpeng Zhou   Fudan University    Senior   Male    81.0        N
Changpeng Zhao  Peking University  Freshman   Male    83.0        N
Chengpeng Qian   Fudan University    Senior   Male    73.0        Y
```

此外，还支持使用 lambda 表达式，其返回值也同样必须是先前提到的四种形式之一：

```
In [41]: df_demo.loc[lambda x:'Quan Zhao', lambda x:'Gender']
Out[41]: 'Female'
```

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：

```
In [42]: df_demo.loc[lambda x: slice('Gaojuan You', 'Gaoqiang Qian')]
Out[42]:
                                  School      Grade  Gender  Weight Transfer
Name
Gaojuan You                Fudan University  Sophomore    Male    74.0        N
Xiaoli Qian              Tsinghua University   Freshman  Female    51.0        N
```

(continues on next page)

(continued from previous page)

```
Qiang Chu      Shanghai Jiao Tong University   Freshman  Female    52.0        N
Gaoqiang Qian            Tsinghua University     Junior  Female    50.0        N
```

最后需要指出的是，对于 Series 也可以使用 loc 索引，其遵循的原则与 DataFrame 中用于行筛选的 loc[*] 完全一致，此处不再赘述。

不要使用链式赋值

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。例如，下面给出的例子：

```
In [43]: df_chain = pd.DataFrame([[0,0],[1,0],[-1,0]], columns=list('AB'))

In [44]: df_chain
Out[44]:
   A  B
0  0  0
1  1  0
2 -1  0

In [45]: import warnings

In [46]: 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，再使用点的列索引
   ....:     except Warning as w:
   ....:         Warning_Msg = w
   ....:

In [47]: print(Warning_Msg)

A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/
↪indexing.html#returning-a-view-versus-a-copy

In [48]: df_chain
Out[48]:
   A  B
```

(continues on next page)

(continued from previous page)

```
0  0  0
1  1  0
2 -1  0

In [49]: df_chain.loc[df_chain.A!=0,'B'] = 1

In [50]: df_chain
Out[50]:
   A  B
0  0  0
1  1  1
2 -1  1
```

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。

```
In [51]: df_demo.iloc[1, 1] # 第二行第二列
Out[51]: 'Freshman'

In [52]: df_demo.iloc[[0, 1], [0, 1]] # 前两行前两列
Out[52]:
                                      School     Grade
Name
Gaopeng Yang    Shanghai Jiao Tong University  Freshman
Changqiang You              Peking University  Freshman

In [53]: df_demo.iloc[1: 4, 2:4] # 切片不包含结束端点
Out[53]:
                Gender  Weight
Name
Changqiang You    Male    70.0
Mei Sun           Male    89.0
Xiaojuan Sun    Female    41.0

In [54]: df_demo.iloc[lambda x: slice(1, 4)] # 传入切片为返回值的函数
Out[54]:
                                      School      Grade  Gender  Weight Transfer
Name
Changqiang You              Peking University   Freshman    Male    70.0        N
```

(continues on next page)

(continued from previous page)

```
Mei Sun          Shanghai Jiao Tong University     Senior    Male    89.0        N
Xiaojuan Sun                  Fudan University  Sophomore  Female    41.0        N
```

在使用布尔列表的时候要特别注意，不能传入 Series 而必须传入序列的 values，否则会报错。因此，在使用布尔筛选的时候还是应当优先考虑 loc 的方式。

例如，选出体重超过 80kg 的学生：

```
In [55]: df_demo.iloc[(df_demo.Weight>80).values].head()
Out[55]:
                                      School      Grade Gender  Weight Transfer
Name
Mei Sun          Shanghai Jiao Tong University     Senior   Male    89.0        N
Qiang Zheng      Shanghai Jiao Tong University     Senior   Male    87.0        N
Qiang Han                   Peking University   Freshman   Male    87.0        N
Chengpeng Zhou               Fudan University     Senior   Male    81.0        N
Feng Han         Shanghai Jiao Tong University  Sophomore   Male    82.0        N
```

对 Series 而言同样也可以通过 iloc 返回相应位置的值或子序列：

```
In [56]: df_demo.School.iloc[1]
Out[56]: 'Peking University'

In [57]: df_demo.School.iloc[1:5:2]
Out[57]:
Name
Changqiang You    Peking University
Xiaojuan Sun       Fudan University
Name: School, dtype: object
```

3.1.5 query 方法

在 pandas 中，支持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。在进行复杂索引时，由于这种检索方式无需像普通方法一样重复使用 DataFrame 的名字来引用列名，一般而言会使代码长度在不降低可读性的前提下有所减少。

例如，将 loc 一节中的复合条件查询例子可以如下改写：

```
In [58]: df.query('((School == "Fudan University")&'
   ....:          ' (Grade == "Senior")&'
   ....:          ' (Weight > 70))|'
   ....:          '((School == "Peking University")&'
   ....:          ' (Grade != "Senior")&'
```

(continues on next page)

(continued from previous page)

```
   ....:           ' (Weight > 80))')
   ....:
Out[58]:
                School     Grade            Name Gender  Weight Transfer
38   Peking University  Freshman       Qiang Han   Male    87.0        N
66    Fudan University    Senior  Chengpeng Zhou   Male    81.0        N
99   Peking University  Freshman  Changpeng Zhao   Male    83.0        N
131   Fudan University    Senior  Chengpeng Qian   Male    73.0        Y
```

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，例如查询体重超过均值的学生：

```
In [59]: df.query('Weight > Weight.mean()').head()
Out[59]:
                           School      Grade            Name  Gender  Weight Transfer
1              Peking University   Freshman  Changqiang You    Male    70.0        N
2   Shanghai Jiao Tong University     Senior         Mei Sun    Male    89.0        N
4               Fudan University  Sophomore     Gaojuan You    Male    74.0        N
10  Shanghai Jiao Tong University   Freshman   Xiaopeng Zhou    Male    74.0        N
14           Tsinghua University     Senior    Xiaomei Zhou  Female    57.0        N
```

query 中引用带空格的列名

对于含有空格的列名，需要使用 `col name` 的方式进行引用。

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, is in, not in 。例如，筛选出男生中不是大一大二的学生：

```
In [60]: df.query('(Grade not in ["Freshman", "Sophomore"]) and'
   ....:          '(Gender == "Male")').head()
   ....:
Out[60]:
                           School   Grade           Name Gender  Weight Transfer
2   Shanghai Jiao Tong University  Senior        Mei Sun   Male    89.0        N
16           Tsinghua University  Junior  Xiaoqiang Qin   Male    68.0        N
17           Tsinghua University  Junior      Peng Wang   Male    65.0        N
18           Tsinghua University  Senior   Xiaofeng Sun   Male    71.0        N
21  Shanghai Jiao Tong University  Senior  Xiaopeng Shen   Male    62.0      NaN
```

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 is in 和 not in，例如查询所有大三和大四的学生：

```
In [61]: df.query('Grade == ["Junior", "Senior"]').head()
Out[61]:
                          School   Grade           Name  Gender  Weight Transfer
2   Shanghai Jiao Tong University  Senior        Mei Sun    Male    89.0        N
7              Tsinghua University  Junior  Gaoqiang Qian  Female    50.0        N
9                Peking University  Junior        Juan Xu  Female     NaN        N
11             Tsinghua University  Junior    Xiaoquan Lv  Female    43.0        N
12  Shanghai Jiao Tong University  Senior       Peng You  Female    48.0      NaN
```

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。例如，取出体重位于 70kg 到 80kg 之间的学生：

```
In [62]: low, high =70, 80

In [63]: df.query('Weight.between(@low, @high)').head()
Out[63]:
                          School      Grade            Name Gender  Weight Transfer
1                Peking University   Freshman  Changqiang You   Male    70.0        N
4                 Fudan University  Sophomore     Gaojuan You   Male    74.0        N
10  Shanghai Jiao Tong University   Freshman   Xiaopeng Zhou   Male    74.0        N
18             Tsinghua University     Senior    Xiaofeng Sun   Male    71.0        N
35               Peking University   Freshman      Gaoli Zhao   Male    78.0        N
```

3.1.6 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。有时在拿到大型数据集后，想要对统计特征进行计算来了解数据的大致分布，但是这很费时间。同时，由于许多统计特征在等概率不放回的简单随机抽样条件下，是总体统计特征的无偏估计，比如样本均值和总体均值，那么就可以先从整张表中抽出一部分来做近似估计。

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。例如，对下面构造的 df_sample 以 value 值的相对大小为抽样概率进行有放回抽样，抽样数量为 3。

```
In [64]: df_sample = pd.DataFrame({'id': list('abcde'),
   ....:                           'value': [1, 2, 3, 4, 90]})
   ....:

In [65]: df_sample
Out[65]:
  id  value
```

(continues on next page)

(continued from previous page)

```
0  a      1
1  b      2
2  c      3
3  d      4
4  e     90

In [66]: df_sample.sample(3, replace = True, weights = df_sample.value)
Out[66]:
  id  value
1  b      2
3  d      4
4  e     90
```

3.2 多级索引

3.2.1 多级索引及其表的结构

为了更加清晰地说明具有多级索引的 DataFrame 结构，下面新构造一张表，读者可以忽略这里的构造方法，它们将会在第 4 小节被更详细地讲解。

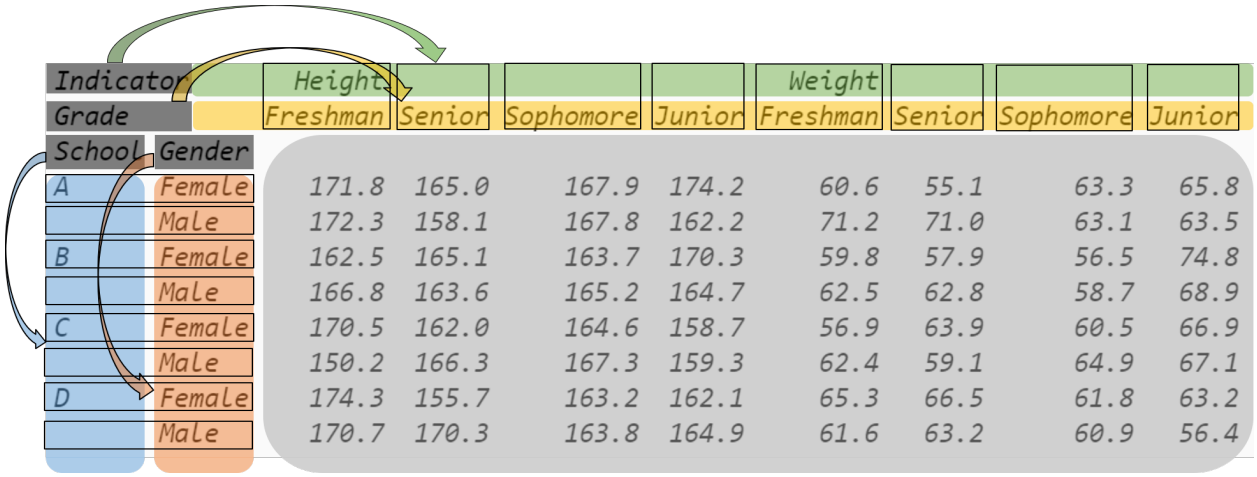
```
In [67]: np.random.seed(0)

In [68]: multi_index = pd.MultiIndex.from_product([list('ABCD'),
   ....:                df.Gender.unique()], names=('School', 'Gender'))
   ....:

In [69]: multi_column = pd.MultiIndex.from_product([['Height', 'Weight'],
   ....:                 df.Grade.unique()], names=('Indicator', 'Grade'))
   ....:

In [70]: df_multi = pd.DataFrame(np.c_[(np.random.randn(8,4)*5 + 163).tolist(),
   ....:                               (np.random.randn(8,4)*5 + 65).tolist()],
   ....:                          index = multi_index,
   ....:                          columns = multi_column).round(1)
   ....:

In [71]: df_multi
Out[71]:
Indicator       Height                            Weight
Grade         Freshman Senior Sophomore Junior Freshman Senior Sophomore Junior
School Gender
```

(continues on next page)

(continued from previous page)

A	Female	171.8	165.0	167.9	174.2	60.6	55.1	63.3	65.8
	Male	172.3	158.1	167.8	162.2	71.2	71.0	63.1	63.5
B	Female	162.5	165.1	163.7	170.3	59.8	57.9	56.5	74.8
	Male	166.8	163.6	165.2	164.7	62.5	62.8	58.7	68.9
C	Female	170.5	162.0	164.6	158.7	56.9	63.9	60.5	66.9
	Male	150.2	166.3	167.3	159.3	62.4	59.1	64.9	67.1
D	Female	174.3	155.7	163.2	162.1	65.3	66.5	61.8	63.2
	Male	170.7	170.3	163.8	164.9	61.6	63.2	60.9	56.4

下图通过颜色区分，标记了 DataFrame 的结构。与单层索引的表一样，具备元素值、行索引和列索引三个部分。其中，这里的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。例如，行索引的第四个元素为 ("B", "Male") ，列索引的第二个元素为 ("Height", "Senior") ，这里需要注意，外层连续出现相同的值时，第一次之后出现的会被隐藏显示，使结果的可读性增强。



与单层索引类似，MultiIndex 也具有名字属性，图中的 School 和 Gender 分别对应了表的第一层和第二层行索引的名字，Indicator 和 Grade 分别对应了第一层和第二层列索引的名字。

索引的名字和值属性分别可以通过 names 和 values 获得：

```
In [72]: df_multi.index.names
Out[72]: FrozenList(['School', 'Gender'])

In [73]: df_multi.columns.names
Out[73]: FrozenList(['Indicator', 'Grade'])

In [74]: df_multi.index.values
Out[74]:
array([('A', 'Female'), ('A', 'Male'), ('B', 'Female'), ('B', 'Male'),
       ('C', 'Female'), ('C', 'Male'), ('D', 'Female'), ('D', 'Male')],
      dtype=object)
```

(continues on next page)

(continued from previous page)

```
In [75]: df_multi.columns.values
Out[75]:
array([('Height', 'Freshman'), ('Height', 'Senior'),
       ('Height', 'Sophomore'), ('Height', 'Junior'),
       ('Weight', 'Freshman'), ('Weight', 'Senior'),
       ('Weight', 'Sophomore'), ('Weight', 'Junior')], dtype=object)
```

如果想要得到某一层的索引，则需要通过 get_level_values 获得：

```
In [76]: df_multi.index.get_level_values(0)
Out[76]: Index(['A', 'A', 'B', 'B', 'C', 'C', 'D', 'D'], dtype='object', name='School')
```

但对于索引而言，无论是单层的还是多层的，用户都无法通过 index_obj[0] = item 的方式来修改元素，也不能通过 index_name[0] = new_name 的方式来修改名字，关于如何修改这些属性的话题将在第三节被讨论。

3.2.2 多级索引中的 loc 索引器

熟悉了结构后，现在回到原表，将学校和年级设为索引，此时的行为多级索引，列为单级索引，由于默认状态的列索引不含名字，因此对应于刚刚图中 Indicator 和 Grade 的索引名位置是空缺的。

```
In [77]: df_multi = df.set_index(['School', 'Grade'])

In [78]: df_multi.head()
Out[78]:
                                                Name  Gender  Weight Transfer
School                          Grade
Shanghai Jiao Tong University Freshman     Gaopeng Yang  Female    46.0        N
Peking University             Freshman   Changqiang You    Male    70.0        N
Shanghai Jiao Tong University Senior            Mei Sun    Male    89.0        N
Fudan University              Sophomore    Xiaojuan Sun  Female    41.0        N
                              Sophomore     Gaojuan You    Male    74.0        N
```

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，不过在索引前最好对 MultiIndex 进行排序以避免性能警告：

```
In [79]: df_multi = df_multi.sort_index()

In [80]: df_multi.loc[('Fudan University', 'Junior')].head()
Out[80]:
                                Name  Gender  Weight Transfer
School           Grade
Fudan University Junior      Yanli You  Female    48.0        N
```

(continues on next page)

(continued from previous page)

```
                Junior  Chunqiang Chu    Male    72.0        N
                Junior   Changfeng Lv    Male    76.0        N
                Junior     Yanjuan Lv  Female    49.0      NaN
                Junior  Gaoqiang Zhou  Female    43.0        N

In [81]: df_multi.loc[[('Fudan University', 'Senior'),
   ....:               ('Shanghai Jiao Tong University', 'Freshman')]].head()
   ....:
Out[81]:
                                  Name  Gender  Weight Transfer
School           Grade
Fudan University Senior  Chengpeng Zheng  Female    38.0        N
                 Senior        Feng Zhou  Female    47.0        N
                 Senior        Gaomei Lv  Female    34.0        N
                 Senior        Chunli Lv  Female    56.0        N
                 Senior   Chengpeng Zhou    Male    81.0        N

In [82]: df_multi.loc[df_multi.Weight > 70].head() # 布尔列表也是可用的
Out[82]:
                                   Name Gender  Weight Transfer
School           Grade
Fudan University Freshman       Feng Wang   Male    74.0        N
                 Junior     Chunqiang Chu   Male    72.0        N
                 Junior      Changfeng Lv   Male    76.0        N
                 Senior    Chengpeng Zhou   Male    81.0        N
                 Senior    Chengpeng Qian   Male    73.0        Y

In [83]: df_multi.loc[lambda x:('Fudan University','Junior')].head()
Out[83]:
                                 Name  Gender  Weight Transfer
School           Grade
Fudan University Junior      Yanli You  Female    48.0        N
                 Junior  Chunqiang Chu    Male    72.0        N
                 Junior   Changfeng Lv    Male    76.0        N
                 Junior     Yanjuan Lv  Female    49.0      NaN
                 Junior  Gaoqiang Zhou  Female    43.0        N
```

练一练

与单层索引类似，若存在重复元素，则不能使用切片，请去除重复索引后给出一个元素切片的例子。

此外，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] 。例如，想要得到所有北大和复旦的大二大三学生，可以如下写出：

```
In [84]: res = df_multi.loc[(['Peking University', 'Fudan University'],
   ....:                     ['Sophomore', 'Junior']), :]
   ....:

In [85]: res.head()
Out[85]:
                                  Name  Gender  Weight Transfer
School            Grade
Peking University Sophomore   Changmei Xu  Female    43.0        N
                  Sophomore  Xiaopeng Qin    Male     NaN        N
                  Sophomore        Mei Xu  Female    39.0        N
                  Sophomore   Xiaoli Zhou  Female    55.0        N
                  Sophomore      Peng Han  Female    34.0      NaN

In [86]: res.shape
Out[86]: (33, 4)
```

下面的语句和上面类似，但仍然传入的是元素（这里为元组）的列表，它们的意义是不同的，表示的是选出北大的大三学生和复旦的大二学生：

```
In [87]: res = df_multi.loc[[('Peking University', 'Junior'),
   ....:                     ('Fudan University', 'Sophomore')]]
   ....:

In [88]: res.head()
Out[88]:
                                  Name  Gender  Weight Transfer
School            Grade
Peking University Junior        Juan Xu  Female     NaN        N
                  Junior  Changjuan You  Female    47.0        N
                  Junior       Gaoli Xu  Female    48.0        N
                  Junior   Gaoquan Zhou    Male    70.0        N
                  Junior      Qiang You  Female    56.0        N

In [89]: res.shape
Out[89]: (16, 4)
```

3.2.3 IndexSlice 对象

前面介绍的方法，即使在索引不重复的时候，也只能对元组整体进行切片，而不能对每层进行切片，也不允许将切片和布尔列表混合使用，引入 IndexSlice 对象就能解决这个问题。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，下面将进行介绍。为了方便演示，下面构造一个 索引不重复的 DataFrame：

```
In [90]: np.random.seed(0)

In [91]: L1,L2 = ['A','B','C'],['a','b','c']

In [92]: mul_index1 = pd.MultiIndex.from_product([L1,L2],names=('Upper', 'Lower'))

In [93]: L3,L4 = ['D','E','F'],['d','e','f']

In [94]: mul_index2 = pd.MultiIndex.from_product([L3,L4],names=('Big', 'Small'))

In [95]: df_ex = pd.DataFrame(np.random.randint(-9,10,(9,9)),
   ....:                      index=mul_index1,
   ....:                      columns=mul_index2)
   ....:

In [96]: df_ex
Out[96]:
Big          D        E        F
Small        d  e  f  d  e  f  d  e  f
Upper Lower
A     a      3  6 -9 -6 -6 -2  0  9 -5
      b     -3  3 -8 -3 -2  5  8 -4  4
      c     -1  0  7 -4  6  6 -9  9 -6
B     a      8  5 -2 -9 -8  0 -9  1 -6
      b      2  9 -7 -9 -9 -5 -4 -3 -1
      c      8  6 -5  0  1 -8 -8 -2  0
C     a     -6 -3  2  5  9 -9  5 -6  3
      b      1  2 -5 -3 -5  6 -6  3 -5
      c     -1  5  6 -6  6  4  7  8 -4
```

为了使用 silce 对象，先要进行定义：

```
In [97]: idx = pd.IndexSlice
```

【a】loc[idx[*,*]] 型

这种情况并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：

```
In [98]: df_ex.loc[idx['C':, ('D', 'f'):]]
Out[98]:
Big          D  E        F
Small        f  d  e  f  d  e  f
Upper Lower
C     a      2  5  9 -9  5 -6  3
      b     -5 -3 -5  6 -6  3 -5
      c      6 -6  6  4  7  8 -4
```

另外，也支持布尔序列的索引：

```
In [99]: df_ex.loc[idx[:'A', lambda x:x.sum()>0]] # 列和大于 0
Out[99]:
Big          D     F
Small        d  e  e
Upper Lower
A     a      3  6  9
      b     -3  3 -4
      c     -1  0  9
```

【b】loc[idx[*,*],idx[*,*]] 型

这种情况能够分层进行切片，前一个 idx 指代的是行索引，后一个是列索引。

```
In [100]: df_ex.loc[idx[:'A', 'b':], idx['E':, 'e':]]
Out[100]:
Big          E     F
Small        e  f  e  f
Upper Lower
A     b     -2  5 -4  4
      c      6  6  9 -6
```

3.2.4 多级索引的构造

前面提到了多级索引表的结构和切片，那么除了使用 set_index 之外，如何自己构造多级索引呢？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。

from_tuples 指根据传入由元组组成的列表进行构造：

```
In [101]: my_tuple = [('a','cat'),('a','dog'),('b','cat'),('b','dog')]

In [102]: pd.MultiIndex.from_tuples(my_tuple, names=['First','Second'])
Out[102]:
MultiIndex([('a', 'cat'),
```

(continues on next page)

(continued from previous page)

```
            ('a', 'dog'),
            ('b', 'cat'),
            ('b', 'dog')],
           names=['First', 'Second'])
```

from_arrays 指根据传入列表中，对应层的列表进行构造：

```
In [103]: my_array = [list('aabb'), ['cat', 'dog']*2]

In [104]: pd.MultiIndex.from_arrays(my_array, names=['First','Second'])
Out[104]:
MultiIndex([('a', 'cat'),
            ('a', 'dog'),
            ('b', 'cat'),
            ('b', 'dog')],
           names=['First', 'Second'])
```

from_product 指根据给定多个列表的笛卡尔积进行构造：

```
In [105]: my_list1 = ['a','b']

In [106]: my_list2 = ['cat','dog']

In [107]: pd.MultiIndex.from_product([my_list1,
   .....:                             my_list2],
   .....:                            names=['First','Second'])
   .....:
Out[107]:
MultiIndex([('a', 'cat'),
            ('a', 'dog'),
            ('b', 'cat'),
            ('b', 'dog')],
           names=['First', 'Second'])
```

3.3 索引的常用方法

3.3.1 索引层的交换和删除

为了方便理解交换的过程，这里构造一个三级索引的例子：

```
In [108]: np.random.seed(0)

In [109]: L1,L2,L3 = ['A','B'],['a','b'],['alpha','beta']

In [110]: mul_index1 = pd.MultiIndex.from_product([L1,L2,L3],
   .....:                 names=('Upper', 'Lower','Extra'))
   .....:

In [111]: L4,L5,L6 = ['C','D'],['c','d'],['cat','dog']

In [112]: mul_index2 = pd.MultiIndex.from_product([L4,L5,L6],
   .....:                 names=('Big', 'Small', 'Other'))
   .....:

In [113]: df_ex = pd.DataFrame(np.random.randint(-9,10,(8,8)),
   .....:                           index=mul_index1,
   .....:                           columns=mul_index2)
   .....:

In [114]: df_ex
Out[114]:
Big                 C               D
Small               c       d       c       d
Other             cat dog cat dog cat dog cat dog
Upper Lower Extra
A     a     alpha   3   6  -9  -6  -6  -2   0   9
            beta   -5  -3   3  -8  -3  -2   5   8
      b     alpha  -4   4  -1   0   7  -4   6   6
            beta   -9   9  -6   8   5  -2  -9  -8
B     a     alpha   0  -9   1  -6   2   9  -7  -9
            beta   -9  -5  -4  -3  -1   8   6  -5
      b     alpha   0   1  -8  -8  -2   0  -6  -3
            beta    2   5   9  -9   5  -6   3   1
```

索引层的交换由 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：

```
In [115]: df_ex.swaplevel(0,2,axis=1).head() # 列索引的第一层和第三层交换
Out[115]:
Other             cat dog cat dog cat dog cat dog
Small               c   c   d   d   c   c   d   d
Big                 C   C   C   C   D   D   D   D
Upper Lower Extra
A     a     alpha   3   6  -9  -6  -6  -2   0   9
```

(continues on next page)

(continued from previous page)

```
            beta   -5  -3   3  -8  -3  -2   5   8
      b     alpha  -4   4  -1   0   7  -4   6   6
            beta   -9   9  -6   8   5  -2  -9  -8
B     a     alpha   0  -9   1  -6   2   9  -7  -9

In [116]: df_ex.reorder_levels([2,0,1],axis=0).head() # 列表数字指代原来索引中的层
Out[116]:
Big                 C               D
Small               c       d       c       d
Other             cat dog cat dog cat dog cat dog
Extra Upper Lower
alpha A     a       3   6  -9  -6  -6  -2   0   9
beta  A     a      -5  -3   3  -8  -3  -2   5   8
alpha A     b      -4   4  -1   0   7  -4   6   6
beta  A     b      -9   9  -6   8   5  -2  -9  -8
alpha B     a       0  -9   1  -6   2   9  -7  -9
```

轴之间的索引交换

这里只涉及行或列索引内部的交换，不同方向索引之间的交换将在第五章中被讨论。

若想要删除某一层的索引，可以使用 droplevel 方法：

```
In [117]: df_ex.droplevel(1,axis=1)
Out[117]:
Big                 C               D
Other             cat dog cat dog cat dog cat dog
Upper Lower Extra
A     a     alpha   3   6  -9  -6  -6  -2   0   9
            beta   -5  -3   3  -8  -3  -2   5   8
      b     alpha  -4   4  -1   0   7  -4   6   6
            beta   -9   9  -6   8   5  -2  -9  -8
B     a     alpha   0  -9   1  -6   2   9  -7  -9
            beta   -9  -5  -4  -3  -1   8   6  -5
      b     alpha   0   1  -8  -8  -2   0  -6  -3
            beta    2   5   9  -9   5  -6   3   1

In [118]: df_ex.droplevel([0,1],axis=0)
Out[118]:
Big     C               D
Small   c       d       c       d
Other cat dog cat dog cat dog cat dog
```

(continues on next page)

(continued from previous page)

```
Extra
alpha   3   6  -9  -6  -6  -2   0   9
beta   -5  -3   3  -8  -3  -2   5   8
alpha  -4   4  -1   0   7  -4   6   6
beta   -9   9  -6   8   5  -2  -9  -8
alpha   0  -9   1  -6   2   9  -7  -9
beta   -9  -5  -4  -3  -1   8   6  -5
alpha   0   1  -8  -8  -2   0  -6  -3
beta    2   5   9  -9   5  -6   3   1
```

3.3.2 索引属性的修改

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：

```
In [119]: df_ex.rename_axis(index={'Upper':'Changed_row'},
   .....:                   columns={'Other':'Changed_Col'}).head()
   .....:
Out[119]:
Big                         C               D
Small                       c       d       c       d
Changed_Col               cat dog cat dog cat dog cat dog
Changed_row Lower Extra
A           a     alpha   3   6  -9  -6  -6  -2   0   9
                  beta   -5  -3   3  -8  -3  -2   5   8
            b     alpha  -4   4  -1   0   7  -4   6   6
                  beta   -9   9  -6   8   5  -2  -9  -8
B           a     alpha   0  -9   1  -6   2   9  -7  -9
```

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 ：

```
In [120]: df_ex.rename(columns={'cat':'not_cat'},
   .....:              level=2).head()
   .....:
Out[120]:
Big                       C                       D
Small                     c           d           c           d
Other               not_cat dog not_cat dog not_cat dog not_cat dog
Upper Lower Extra
A     a     alpha       3   6      -9  -6      -6  -2       0   9
            beta       -5  -3       3  -8      -3  -2       5   8
      b     alpha      -4   4      -1   0       7  -4       6   6
            beta       -9   9      -6   8       5  -2      -9  -8
B     a     alpha       0  -9       1  -6       2   9      -7  -9
```

传入参数也可以是函数，其输入值就是索引元素：

```
In [121]: df_ex.rename(index=lambda x:str.upper(x),
   .....:              level=2).head()
   .....:
Out[121]:
Big                 C              D
Small               c       d      c       d
Other             cat dog cat dog cat dog cat dog
Upper Lower Extra
A     a     ALPHA   3   6  -9  -6  -6  -2   0   9
            BETA   -5  -3   3  -8  -3  -2   5   8
      b     ALPHA  -4   4  -1   0   7  -4   6   6
            BETA   -9   9  -6   8   5  -2  -9  -8
B     a     ALPHA   0  -9   1  -6   2   9  -7  -9
```

练一练

尝试在 rename_axis 中使用函数完成与例子中一样的功能。

对于整个索引的元素替换，可以利用迭代器实现：

```
In [122]: new_values = iter(list('abcdefgh'))

In [123]: df_ex.rename(index=lambda x:next(new_values),
   .....:              level=2)
   .....:
Out[123]:
Big                 C              D
Small               c       d      c       d
Other             cat dog cat dog cat dog cat dog
Upper Lower Extra
A     a     a       3   6  -9  -6  -6  -2   0   9
            b      -5  -3   3  -8  -3  -2   5   8
      b     c      -4   4  -1   0   7  -4   6   6
            d      -9   9  -6   8   5  -2  -9  -8
B     a     e       0  -9   1  -6   2   9  -7  -9
            f      -9  -5  -4  -3  -1   8   6  -5
      b     g       0   1  -8  -8  -2   0  -6  -3
            h       2   5   9  -9   5  -6   3   1
```

若想要对某个位置的元素进行修改，在单层索引时容易实现，即先取出索引的 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。但是如果是多级索引的话就有些麻烦，一个解决的方案是先把某一层索引临时转为表的元素，然后再进行修改，最后重新设定为索引，下面一节将介绍这些操作。

另外一个需要介绍的函数是 map ，它是定义在 Index 上的方法，与前面 rename 方法中层的函数式用法是类似的，只不过它传入的不是层的标量值，而是直接传入索引的元组，这为用户进行跨层的修改提供了遍历。例如，可以等价地写出上面的字符串转大写的操作：

```
In [124]: df_temp = df_ex.copy()

In [125]: new_idx = df_temp.index.map(lambda x: (x[0],
   .....:                                        x[1],
   .....:                                        str.upper(x[2])))
   .....:

In [126]: df_temp.index = new_idx

In [127]: df_temp.head()
Out[127]:
Big                  C               D
Small                c       d       c       d
Other              cat dog cat dog cat dog cat dog
Upper Lower Extra
A     a     ALPHA    3   6  -9  -6  -6  -2   0   9
            BETA    -5  -3   3  -8  -3  -2   5   8
      b     ALPHA   -4   4  -1   0   7  -4   6   6
            BETA    -9   9  -6   8   5  -2  -9  -8
B     a     ALPHA    0  -9   1  -6   2   9  -7  -9
```

关于 map 的另一个使用方法是对多级索引的压缩，这在第四章和第五章的一些操作中是有用的：

```
In [128]: df_temp = df_ex.copy()

In [129]: new_idx = df_temp.index.map(lambda x: (x[0]+'-'+
   .....:                                        x[1]+'-'+
   .....:                                        x[2]))
   .....:

In [130]: df_temp.index = new_idx

In [131]: df_temp.head() # 单层索引
Out[131]:
Big          C               D
Small        c       d       c       d
Other      cat dog cat dog cat dog cat dog
A-a-alpha    3   6  -9  -6  -6  -2   0   9
A-a-beta    -5  -3   3  -8  -3  -2   5   8
A-b-alpha   -4   4  -1   0   7  -4   6   6
A-b-beta    -9   9  -6   8   5  -2  -9  -8
```

(continues on next page)

(continued from previous page)

```
B-a-alpha   0  -9   1  -6   2   9  -7  -9
```

同时，也可以反向地展开：

```
In [132]: new_idx = df_temp.index.map(lambda x:tuple(x.split('-')))

In [133]: df_temp.index = new_idx

In [134]: df_temp.head() # 三层索引
Out[134]:
Big         C               D
Small       c       d       c       d
Other     cat dog cat dog cat dog cat dog
A a alpha   3   6  -9  -6  -6  -2   0   9
    beta   -5  -3   3  -8  -3  -2   5   8
  b alpha  -4   4  -1   0   7  -4   6   6
    beta   -9   9  -6   8   5  -2  -9  -8
B a alpha   0  -9   1  -6   2   9  -7  -9
```

3.3.3 索引的设置与重置

为了说明本节的函数，下面构造一个新表：

```
In [135]: df_new = pd.DataFrame({'A':list('aacd'),
   .....:                        'B':list('PQRT'),
   .....:                        'C':[1,2,3,4]})
   .....:
```

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：

```
In [136]: df_new.set_index('A')
Out[136]:
   B  C
A
a  P  1
a  Q  2
c  R  3
d  T  4

In [137]: df_new.set_index('A', append=True)
Out[137]:
```

(continues on next page)

(continued from previous page)

```
    B  C
  A
0 a  P  1
1 a  Q  2
2 c  R  3
3 d  T  4
```

可以同时指定多个列作为索引：

```
In [138]: df_new.set_index(['A', 'B'])
Out[138]:
     C
A B
a P  1
  Q  2
c R  3
d T  4
```

如果想要添加索引的列没有出现再其中，那么可以直接在参数中传入相应的 Series ：

```
In [139]: my_index = pd.Series(list('WXYZ'), name='D')

In [140]: df_new = df_new.set_index(['A', my_index])

In [141]: df_new
Out[141]:
     B  C
A D
a W  P  1
  X  Q  2
c Y  R  3
d Z  T  4
```

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：

```
In [142]: df_new.reset_index(['D'])
Out[142]:
   D  B  C
A
a  W  P  1
a  X  Q  2
c  Y  R  3
```

(continues on next page)

(continued from previous page)

```
d  Z  T  4

In [143]: df_new.reset_index(['D'], drop=True)
Out[143]:
   B  C
A
a  P  1
a  Q  2
c  R  3
d  T  4
```

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：

```
In [144]: df_new.reset_index()
Out[144]:
   A  D  B  C
0  a  W  P  1
1  a  X  Q  2
2  c  Y  R  3
3  d  Z  T  4
```

3.3.4 索引的变形

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。例如，下面的表中给出了员工信息，需要重新制作一张新的表，要求增加一名员工的同时去掉身高列并增加性别列：

```
In [145]: df_reindex = pd.DataFrame({"Weight":[60,70,80],
   .....:                            "Height":[176,180,179]},
   .....:                            index=['1001','1003','1002'])
   .....:

In [146]: df_reindex
Out[146]:
      Weight  Height
1001      60     176
1003      70     180
1002      80     179

In [147]: df_reindex.reindex(index=['1001','1002','1003','1004'],
   .....:                    columns=['Weight','Gender'])
   .....:
```

(continues on next page)

(continued from previous page)

```
Out[147]:
      Weight  Gender
1001    60.0     NaN
1002    80.0     NaN
1003    70.0     NaN
1004     NaN     NaN
```

这种需求常出现在时间序列索引的时间点填充以及 ID 编号的扩充。另外，需要注意的是原来表中的数据和新表中会根据索引自动对其，例如原先的 1002 号位置在 1003 号之后，而新表中相反，那么 reindex 中会根据元素对其，与位置无关。

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的索引来进行被调用表索引的变形。例如，现在以及存在一张表具备了目标索引的条件，那么上述功能可以如下等价地写出：

```
In [148]: df_existed = pd.DataFrame(index=['1001','1002','1003','1004'],
   .....:                           columns=['Weight','Gender'])
   .....:

In [149]: df_reindex.reindex_like(df_existed)
Out[149]:
      Weight  Gender
1001    60.0     NaN
1002    80.0     NaN
1003    70.0     NaN
1004     NaN     NaN
```

3.4 索引运算

3.4.1 集合的运算法则

经常会有一种利用集合运算来取出符合条件行的需求，例如有两张表 A 和 B ，它们的索引都是员工编号，现在需要筛选出两表索引交集的所有员工信息，此时通过 Index 上的运算操作就很容易实现。

不过在此之前，不妨先复习一下常见的四种集合运算：

$$\mathrm{S_A.intersection(S_B)} = \mathrm{S_A} \cap \mathrm{S_B} \Leftrightarrow \{\mathrm{x|x} \in \mathrm{S_A}\ \mathrm{and}\ \mathrm{x} \in \mathrm{S_B}\}$$

$$\mathrm{S_A.union(S_B)} = \mathrm{S_A} \cup \mathrm{S_B} \Leftrightarrow \{\mathrm{x|x} \in \mathrm{S_A}\ \mathrm{or}\ \mathrm{x} \in \mathrm{S_B}\}$$

$$\mathrm{S_A.difference(S_B)} = \mathrm{S_A} - \mathrm{S_B} \Leftrightarrow \{\mathrm{x|x} \in \mathrm{S_A}\ \mathrm{and}\ \mathrm{x} \notin \mathrm{S_B}\}$$

$$\mathrm{S_A.symmetric\_difference(S_B)} = \mathrm{S_A} \triangle \mathrm{S_B} \Leftrightarrow \{\mathrm{x|x} \in \mathrm{S_A} \cup \mathrm{S_B} - \mathrm{S_A} \cap \mathrm{S_B}\}$$

3.4.2 一般的索引运算

由于集合的元素是互异的，但是索引中可能有相同的元素，先用 unique 去重后再进行运算。下面构造两张最为简单的示例表进行演示：

```
In [150]: df_set_1 = pd.DataFrame([[0,1],[1,2],[3,4]],
   .....:                         index = pd.Index(['a','b','a'],name='id1'))
   .....:

In [151]: df_set_2 = pd.DataFrame([[4,5],[2,6],[7,1]],
   .....:                         index = pd.Index(['b','b','c'],name='id2'))
   .....:

In [152]: id1, id2 = df_set_1.index.unique(), df_set_2.index.unique()

In [153]: id1.intersection(id2)
Out[153]: Index(['b'], dtype='object')

In [154]: id1.union(id2)
Out[154]: Index(['a', 'b', 'c'], dtype='object')

In [155]: id1.difference(id2)
Out[155]: Index(['a'], dtype='object')

In [156]: id1.symmetric_difference(id2)
Out[156]: Index(['a', 'c'], dtype='object')
```

上述的四类运算还可以用等价的符号表示代替如下：

```
In [157]: id1 & id2
Out[157]: Index(['b'], dtype='object')

In [158]: id1 | id2
Out[158]: Index(['a', 'b', 'c'], dtype='object')

In [159]: (id1 ^ id2) & id1
Out[159]: Index(['a'], dtype='object')

In [160]: id1 ^ id2 # ^ 符号即对称差
Out[160]: Index(['a', 'c'], dtype='object')
```

若两张表需要做集合运算的列并没有被设置索引，一种办法是先转成索引，运算后再恢复，另一种方法是利用 isin 函数，例如在重置索引的第一张表中选出 id 列交集的所在行：

```
In [161]: df_set_in_col_1 = df_set_1.reset_index()

In [162]: df_set_in_col_2 = df_set_2.reset_index()

In [163]: df_set_in_col_1
Out[163]:
  id1  0  1
0   a  0  1
1   b  1  2
2   a  3  4

In [164]: df_set_in_col_2
Out[164]:
  id2  0  1
0   b  4  5
1   b  2  6
2   c  7  1

In [165]: df_set_in_col_1[df_set_in_col_1.id1.isin(df_set_in_col_2.id2)]
Out[165]:
  id1  0  1
1   b  1  2
```

3.5 练习

3.5.1 Ex1：公司员工数据集

现有一份公司员工数据集：

```
In [166]: df = pd.read_csv('data/company.csv')

In [167]: df.head(3)
Out[167]:
   EmployeeID birthdate_key  age  city_name department      job_title gender
0        1318      1/3/1954   61  Vancouver  Executive            CEO      M
1        1319      1/3/1957   58  Vancouver  Executive      VP Stores      F
2        1320      1/2/1955   60  Vancouver  Executive  Legal Counsel      F
```

1. 分别只使用 query 和 loc 选出年龄不超过四十岁且工作部门为 Dairy 或 Bakery 的男性。
2. 选出员工 ID 号为奇数所在行的第 1、第 3 和倒数第 2 列。
3. 按照以下步骤进行索引操作：

- 把后三列设为索引后交换内外两层
- 恢复中间一层
- 修改外层索引名为 Gender
- 用下划线合并两层行索引
- 把行索引拆分为原状态
- 修改索引名为原表名称
- 恢复默认索引并将列保持为原表的相对位置

3.5.2 Ex2：巧克力数据集

现有一份关于巧克力评价的数据集，：

```
In [168]: df = pd.read_csv('data/chocolate.csv')

In [169]: df.head(3)
Out[169]:
    Company  Review\nDate Cocoa\nPercent Company\nLocation  Rating
0  A. Morin          2016            63%            France    3.75
1  A. Morin          2015            70%            France    2.75
2  A. Morin          2015            70%            France    3.00
```

1. 把列索引名中的 \n 替换为空格。
2. 巧克力 Rating 评分为 1 至 5，每 0.25 分一档，请选出 2.75 分及以下且可可含量 Cocoa Percent 高于中位数的样本。
3. 将 Review Date 和 Company Location 设为索引后，选出 Review Date 在 2012 年之后且 Company Location 不属于 France, Canada, Amsterdam, Belgium 的样本。

第 4 章　分组

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

In [2]: import pandas as pd
```

4.1　分组模式及其对象

4.1.1　分组的一般模式

分组操作在日常生活中使用极其广泛，例如：

- 依据 性别分组，统计全国人口 寿命的 平均值
- 依据 季节分组，对每一个季节的 温度进行 组内标准化
- 依据 班级分组，筛选出组内 数学分数的 平均值超过 80 分的班级

从上述的几个例子中不难看出，想要实现分组操作，必须明确三个要素：分组依据、数据来源、操作及其返回结果。同时从充分性的角度来说，如果明确了这三方面，就能确定一个分组操作，从而分组代码的一般模式即：

```
df.groupby(分组依据)[数据来源].使用操作
```

例如第一个例子中的代码就应该如下：

```
df.groupby('Gender')['Longevity'].mean()
```

现在返回到学生体测的数据集上，如果想要按照性别统计身高中位数，就可以如下写出：

```
In [3]: df = pd.read_csv('data/learn_pandas.csv')

In [4]: df.groupby('Gender')['Height'].median()
Out[4]:
```

(continues on next page)

(continued from previous page)

```
Gender
Female    159.6
Male      173.4
Name: Height, dtype: float64
```

4.1.2 分组依据的本质

前面提到的若干例子都是以单一维度进行分组的，比如根据性别，如果现在需要根据多个维度进行分组，该如何做？事实上，只需在 groupby 中传入相应列名构成的列表即可。例如，现想根据学校和性别进行分组，统计身高的均值就可以如下写出：

```
In [5]: df.groupby(['School', 'Gender'])['Height'].mean()
Out[5]:
School                          Gender
Fudan University                Female    158.776923
                                Male      174.212500
Peking University               Female    158.666667
                                Male      172.030000
Shanghai Jiao Tong University   Female    159.122500
                                Male      176.760000
Tsinghua University             Female    159.753333
                                Male      171.638889
Name: Height, dtype: float64
```

目前为止，groupby 的分组依据都是直接可以从列中按照名字获取的，那如果想要通过一定的复杂逻辑来分组，例如根据学生体重是否超过总体均值来分组，同样还是计算身高的均值。

首先应该先写出分组条件：

```
In [6]: condition = df.Weight > df.Weight.mean()
```

然后将其传入 groupby 中：

```
In [7]: df.groupby(condition)['Height'].mean()
Out[7]:
Weight
False    159.034646
True     172.705357
Name: Height, dtype: float64
```

练一练

请根据上下四分位数分割，将体重分为 high、normal、low 三组，统计身高的均值。

从索引可以看出，其实最后产生的结果就是按照条件列表中元素的值（此处是 True 和 False ）来分组，下面用随机传入字母序列来验证这一想法：

```
In [8]: item = np.random.choice(list('abc'), df.shape[0])

In [9]: df.groupby(item)['Height'].mean()
Out[9]:
a    163.924242
b    162.928814
c    162.708621
Name: Height, dtype: float64
```

此处的索引就是原先 item 中的元素，如果传入多个序列进入 groupby ，那么最后分组的依据就是这两个序列对应行的唯一组合：

```
In [10]: df.groupby([condition, item])['Height'].mean()
Out[10]:
Weight
False   a    160.193617
        b    158.921951
        c    157.756410
True    a    173.152632
        b    172.055556
        c    172.873684
Name: Height, dtype: float64
```

由此可以看出，之前传入列名只是一种简便的记号，事实上等价于传入的是一个或多个列，最后分组的依据来自于数据来源组合的 unique 值，通过 drop_duplicates 就能知道具体的组类别：

```
In [11]: df[['School', 'Gender']].drop_duplicates()
Out[11]:
                          School  Gender
0   Shanghai Jiao Tong University  Female
1              Peking University    Male
2   Shanghai Jiao Tong University    Male
3               Fudan University  Female
4               Fudan University    Male
5            Tsinghua University  Female
9              Peking University  Female
16           Tsinghua University    Male

In [12]: df.groupby([df['School'], df['Gender']])['Height'].mean()
```

(continues on next page)

(continued from previous page)

```
Out[12]:
School                          Gender
Fudan University                Female    158.776923
                                Male      174.212500
Peking University               Female    158.666667
                                Male      172.030000
Shanghai Jiao Tong University   Female    159.122500
                                Male      176.760000
Tsinghua University             Female    159.753333
                                Male      171.638889
Name: Height, dtype: float64
```

4.1.3 Groupby 对象

能够注意到，最终具体做分组操作时，所调用的方法都来自于 pandas 中的 groupby 对象，这个对象上定义了许多方法，也具有一些方便的属性。

```
In [13]: gb = df.groupby(['School', 'Grade'])

In [14]: gb
Out[14]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002266AC32D48>
```

通过 ngroups 属性，可以访问分为了多少组：

```
In [15]: gb.ngroups
Out[15]: 16
```

通过 groups 属性，可以返回从 组名映射到 组索引列表的字典：

```
In [16]: res = gb.groups

In [17]: res.keys() # 字典的值由于是索引，元素个数过多，此处只展示字典的键
Out[17]: dict_keys([('Fudan University', 'Freshman'), ('Fudan University', 'Junior'), ('Fudan␣
↪University', 'Senior'), ('Fudan University', 'Sophomore'), ('Peking University', 'Freshman'), (
↪'Peking University', 'Junior'), ('Peking University', 'Senior'), ('Peking University', 'Sophomore
↪'), ('Shanghai Jiao Tong University', 'Freshman'), ('Shanghai Jiao Tong University', 'Junior'), (
↪'Shanghai Jiao Tong University', 'Senior'), ('Shanghai Jiao Tong University', 'Sophomore'), (
↪'Tsinghua University', 'Freshman'), ('Tsinghua University', 'Junior'), ('Tsinghua University',
↪'Senior'), ('Tsinghua University', 'Sophomore')])
```

练一练

> 上一小节介绍了可以通过 drop_duplicates 得到具体的组类别，现请用 groups 属性完成类似的功能。

当 size 作为 DataFrame 的属性时，返回的是表长乘以表宽的大小，但在 groupby 对象上表示统计每个组的元素个数：

```
In [18]: gb.size()
Out[18]:
School                          Grade
Fudan University                Freshman      9
                                Junior       12
                                Senior       11
                                Sophomore     8
Peking University               Freshman     13
                                Junior        8
                                Senior        8
                                Sophomore     5
Shanghai Jiao Tong University   Freshman     13
                                Junior       17
                                Senior       22
                                Sophomore     5
Tsinghua University             Freshman     17
                                Junior       22
                                Senior       14
                                Sophomore    16
dtype: int64
```

通过 get_group 方法可以直接获取所在组对应的行，此时必须知道组的具体名字：

```
In [19]: gb.get_group(('Fudan University', 'Freshman')).iloc[:3, :3] # 展示一部分
Out[19]:
              School     Grade             Name
15  Fudan University  Freshman  Changqiang Yang
28  Fudan University  Freshman     Gaoqiang Qin
63  Fudan University  Freshman     Gaofeng Zhao
```

这里列出了 2 个属性和 2 个方法，而先前的 mean 、median 都是 groupby 对象上的方法，这些函数和许多其他函数的操作具有高度相似性，将在之后的小节进行专门介绍。

4.1.4 分组的三大操作

熟悉了一些分组的基本知识后，重新回到开头举的三个例子，可能会发现一些端倪，即这三种类型的分组返回数据的结果型态并不一样：

- 第一个例子中，每一个组返回一个标量值，可以是平均值、中位数、组容量 size 等
- 第二个例子中，做了原序列的标准化处理，也就是说每组返回的是一个 Series 类型
- 第三个例子中，既不是标量也不是序列，返回的整个组所在行的本身，即返回了 DataFrame 类型

由此，引申出分组的三大操作：聚合、变换和过滤，分别对应了三个例子的操作，下面就要分别介绍相应的 agg 、transform 和 filter 函数及其操作。

4.2 聚合函数

4.2.1 内置聚合函数

在介绍 agg 之前，首先要了解一些直接定义在 groupby 对象的聚合函数，因为它的速度基本都会经过内部的优化，使用功能时应当优先考虑。根据返回标量值的原则，包括如下函数：max/min/mean/median/count/all/any/idxmax/idxmin/mad/nunique/skew/quantile/sum/std/var/sem/size/prod 。

```
In [20]: gb = df.groupby('Gender')['Height']

In [21]: gb.idxmin()
Out[21]:
Gender
Female    143
Male      199
Name: Height, dtype: int64

In [22]: gb.quantile(0.95)
Out[22]:
Gender
Female    166.8
Male      185.9
Name: Height, dtype: float64
```

练一练

请查阅文档，明确 all/any/mad/skew/sem/prod 函数的含义。

这些聚合函数当传入的数据来源包含多个列时，将按照列进行迭代计算：

```
In [23]: gb = df.groupby('Gender')[['Height', 'Weight']]

In [24]: gb.max()
Out[24]:
        Height  Weight
Gender
Female   170.2    63.0
Male     193.9    89.0
```

4.2.2 agg 方法

虽然在 groupby 对象上定义了许多方便的函数，但仍然有以下缺陷：

- 无法同时使用多个函数
- 无法对特定的列使用特定的聚合函数
- 无法使用自定义的聚合函数
- 无法直接对结果的列名在聚合前进行自定义命名

下面说明如何通过 agg 函数解决这四个缺陷：

【a】使用多个函数

当使用多个聚合函数时，需要用列表的形式把内置聚合函数的对应的字符串传入，先前提到的所有字符串都是合法的。

```
In [25]: gb.agg(['sum', 'idxmax', 'skew'])
Out[25]:
         Height                   Weight
            sum idxmax      skew     sum idxmax      skew
Gender
Female  21014.0     28 -0.219253  6469.0     28 -0.268482
Male     8854.9    193  0.437535  3929.0      2 -0.332393
```

从结果看，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，分别逐一对列使用聚合，因此结果为 6 列。

【b】对特定的列使用特定的聚合函数

对于方法和列的特殊对应，可以通过构造字典传入 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。

```
In [26]: gb.agg({'Height':['mean','max'], 'Weight':'count'})
Out[26]:
           Height        Weight
             mean    max  count
Gender
Female  159.19697  170.2    135
Male    173.62549  193.9     54
```

练一练

请使用【b】中的传入字典的方法完成【a】中等价的聚合任务。

【c】使用自定义函数

在 agg 中可以使用具体的自定义函数，需要注意传入函数的参数是之前数据源中的列，逐列进行计算。下面分组计算身高和体重的极差：

```
In [27]: gb.agg(lambda x: x.mean()-x.min())
Out[27]:
          Height     Weight
Gender
Female  13.79697  13.918519
Male    17.92549  21.759259
```

练一练

在 groupby 对象中可以使用 describe 方法进行统计信息汇总，请同时使用多个聚合函数，完成与该方法相同的功能。

由于传入的是序列，因此序列上的方法和属性都是可以在函数中使用的，只需保证返回值是标量即可。下面的例子是指，如果组的指标均值，超过该指标的总体均值，返回 High，否则返回 Low。

```
In [28]: def my_func(s):
   ....:     res = 'High'
   ....:     if s.mean() <= df[s.name].mean():
   ....:         res = 'Low'
   ....:     return res
   ....:

In [29]: gb.agg(my_func)
Out[29]:
```

(continues on next page)

(continued from previous page)

```
       Height Weight
Gender
Female    Low    Low
Male     High   High
```

【d】聚合结果重命名

如果想要对结果进行重命名，只需要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，第二个位置为原来的函数，包括聚合字符串和自定义函数，现举若干例子说明：

```
In [30]: gb.agg([('range', lambda x: x.max()-x.min()), ('my_sum', 'sum')])
Out[30]:
       Height          Weight
        range   my_sum  range  my_sum
Gender
Female   24.8  21014.0   29.0  6469.0
Male     38.2   8854.9   38.0  3929.0
```

```
In [31]: gb.agg({'Height': [('my_func', my_func), 'sum'],
   ....:         'Weight': lambda x:x.max()})
   ....:
Out[31]:
        Height            Weight
       my_func      sum <lambda>
Gender
Female     Low  21014.0     63.0
Male      High   8854.9     89.0
```

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，重命名需要加方括号，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：

```
In [32]: gb.agg([('my_sum', 'sum')])
Out[32]:
         Height  Weight
         my_sum  my_sum
Gender
Female  21014.0  6469.0
Male     8854.9  3929.0
```

```
In [33]: gb.agg({'Height': [('my_func', my_func), 'sum'],
   ....:         'Weight': [('range', lambda x:x.max())]})
   ....:
Out[33]:
```

(continues on next page)

(continued from previous page)

```
        Height          Weight
       my_func      sum  range
Gender
Female     Low  21014.0   63.0
Male      High   8854.9   89.0
```

4.3 变换和过滤

4.3.1 变换函数与 transform 方法

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 ，它们的使用方式和聚合函数类似，只不过完成的是组内累计操作。此外在 groupby 对象上还定义了填充类和滑窗类的变换函数，这些函数的一般形式将会分别在第七章和第十章中讨论，此处略过。

```
In [34]: gb.cummax().head()
Out[34]:
   Height  Weight
0   158.9    46.0
1   166.5    70.0
2   188.9    89.0
3     NaN    46.0
4   188.9    89.0
```

练一练

> 在 groupby 对象中，rank 方法也是一个实用的变换函数，请查阅它的功能并给出一个使用的例子。

当用自定义变换时需要使用 transform 方法，被调用的自定义函数，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 。

现对身高和体重进行分组标准化，即减去组均值后除以组的标准差：

```
In [35]: gb.transform(lambda x: (x-x.mean())/x.std()).head()
Out[35]:
     Height    Weight
0 -0.058760 -0.354888
1 -1.010925 -0.355000
2  2.167063  2.089498
```

(continues on next page)

(continued from previous page)

```
3       NaN -1.279789
4  0.053133  0.159631
```

练一练

对于 transform 方法无法像 agg 一样，通过传入字典来对指定列使用特定的变换，如果需要在一次 transform 的调用中实现这种功能，请给出解决方案。

前面提到了 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。例如，构造两列新特征来分别表示样本所在性别组的身高均值和体重均值：

```
In [36]: gb.transform('mean').head() # 传入返回标量的函数也是可以的
Out[36]:
      Height     Weight
0  159.19697  47.918519
1  173.62549  72.759259
2  173.62549  72.759259
3  159.19697  47.918519
4  173.62549  72.759259
```

4.3.2 组索引与过滤

在上一章中介绍了索引的用法，那么索引和过滤有什么区别呢？

过滤在分组中是对于组的过滤，而索引是对于行的过滤，在第二章中的返回值，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，即如果筛选条件的则选入结果的表，否则不选入。

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，定义了 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，在之前例子中定义的 groupby 对象中，传入的就是 df[[‘Height’, ‘Weight’]]，因此所有表方法和属性都可以在自定义函数中相应地使用，同时只需保证自定义函数的返回为布尔值即可。

例如，在原表中通过过滤得到所有容量大于 100 的组：

```
In [37]: gb.filter(lambda x: x.shape[0] > 100).head()
Out[37]:
   Height  Weight
0   158.9    46.0
3     NaN    41.0
```

(continues on next page)

(continued from previous page)

```
5   158.0    51.0
6   162.5    52.0
7   161.9    50.0
```

练一练

从概念上说，索引功能是组过滤功能的子集，请使用 filter 函数完成 loc[.] 的功能，这里假设”.“是元素列表。

4.4 跨列分组

4.4.1 apply 的引入

之前几节介绍了三大分组操作，但事实上还有一种常见的分组场景，无法用前面介绍的任何一种方法处理，例如现在如下定义身体质量指数 BMI：

$$\text{BMI} = \frac{\text{Weight}}{\text{Height}^2}$$

其中体重和身高的单位分别为千克和米，需要分组计算组 BMI 的均值。

首先，这显然不是过滤操作，因此 filter 不符合要求；其次，返回的均值是标量而不是序列，因此 transform 不符合要求；最后，似乎使用 agg 函数能够处理，但是之前强调过聚合函数是逐列处理的，而不能够 多列数据同时处理。由此，引出了 apply 函数来解决这一问题。

4.4.2 apply 的使用

在设计上，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。现如下解决上述计算问题：

```
In [38]: def BMI(x):
   ....:     Height = x['Height']/100
   ....:     Weight = x['Weight']
   ....:     BMI_value = Weight/Height**2
   ....:     return BMI_value.mean()
   ....:

In [39]: gb.apply(BMI)
Out[39]:
Gender
```

(continues on next page)

(continued from previous page)

```
Female    18.860930
Male      24.318654
dtype: float64
```

除了返回标量之外，apply 方法还可以返回一维 Series 和二维 DataFrame ，但它们产生的数据框维数和多级索引的层数应当如何变化？下面举三组例子就非常容易明白结果是如何生成的：

【a】标量情况：结果得到的是 Series ，索引与 agg 的结果一致

```
In [40]: gb = df.groupby(['Gender','Test_Number'])[['Height','Weight']]

In [41]: gb.apply(lambda x: 0)
Out[41]:
Gender  Test_Number
Female  1              0
        2              0
        3              0
Male    1              0
        2              0
        3              0
dtype: int64

In [42]: gb.apply(lambda x: [0, 0]) # 虽然是列表，但是作为返回值仍然看作标量
Out[42]:
Gender  Test_Number
Female  1              [0, 0]
        2              [0, 0]
        3              [0, 0]
Male    1              [0, 0]
        2              [0, 0]
        3              [0, 0]
dtype: object
```

【b】Series 情况：得到的是 DataFrame ，行索引与标量情况一致，列索引为 Series 的索引

```
In [43]: gb.apply(lambda x: pd.Series([0,0],index=['a','b']))
Out[43]:
                    a  b
Gender Test_Number
Female 1            0  0
       2            0  0
       3            0  0
Male   1            0  0
       2            0  0
```

(continues on next page)

(continued from previous page)

```
      3           0  0
```

练一练

请尝试在 apply 传入的自定义函数中，根据组的某些特征返回相同长度但索引不同的 Series，会报错吗？

【c】DataFrame 情况：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。

```
In [44]: gb.apply(lambda x: pd.DataFrame(np.ones((2,2)),
   ....:                                 index = ['a','b'],
   ....:                                 columns=pd.Index([('w','x'),('y','z')])))
   ....:
Out[44]:
                        w    y
                        x    z
Gender Test_Number
Female 1           a  1.0  1.0
                   b  1.0  1.0
       2           a  1.0  1.0
                   b  1.0  1.0
       3           a  1.0  1.0
                   b  1.0  1.0
Male   1           a  1.0  1.0
                   b  1.0  1.0
       2           a  1.0  1.0
                   b  1.0  1.0
       3           a  1.0  1.0
                   b  1.0  1.0
```

练一练

请尝试在 apply 传入的自定义函数中，根据组的某些特征返回相同大小但列索引不同的 DataFrame，会报错吗？如果只是行索引不同，会报错吗？

最后需要强调的是，apply 函数的灵活性是以牺牲一定性能为代价换得的，除非需要使用跨列处理的分组处理，否则应当使用其他专门设计的 groupby 对象方法，否则在性能上会存在较大的差距。同时，在使用聚合函数和变换函数时，也应当优先使用内置函数，它们经过了高度的性能优化，一般而言在速度上都会快于用自定义函数来实现。

练一练

在 groupby 对象中还定义了 cov 和 corr 函数，从概念上说也属于跨列的分组处理。请利用之前定义的 gb 对象，使用 apply 函数实现与 gb.cov() 同样的功能并比较它们的性能。

4.5 练习

4.5.1 Ex1：汽车数据集

现有一份汽车数据集，其中 Brand, Disp., HP 分别代表汽车品牌、发动机蓄量、发动机输出。

```
In [45]: df = pd.read_csv('data/car.csv')

In [46]: df.head(3)
Out[46]:
             Brand  Price Country  Reliability  Mileage   Type  Weight  Disp.   HP
0   Eagle Summit 4   8895     USA          4.0       33  Small    2560     97  113
1  Ford Escort   4   7402     USA          2.0       33  Small    2345    114   90
2   Ford Festiva 4   6319   Korea          4.0       37  Small    1845     81   63
```

1. 先过滤出所属 Country 数超过 2 个的汽车，即若该汽车的 Country 在总体数据集中出现次数不超过 2 则剔除，再按 Country 分组计算价格均值、价格变异系数、该 Country 的汽车数量，其中变异系数的计算方法是标准差除以均值，并在结果中把变异系数重命名为 CoV 。
2. 按照表中位置的前三分之一、中间三分之一和后三分之一分组，统计 Price 的均值。
3. 对类型 Type 分组，对 Price 和 HP 分别计算最大值和最小值，结果会产生多级索引，请用下划线把多级列索引合并为单层索引。
4. 对类型 Type 分组，对 HP 进行组内的 min-max 归一化。
5. 对类型 Type 分组，计算 Disp. 与 HP 的相关系数。

4.5.2 Ex2：实现 transform 函数

- groupby 对象的构造方法是 my_groupby(df, group_cols)
- 支持单列分组与多列分组
- 支持带有标量广播的 my_groupby(df)[col].transform(my_func) 功能
- pandas 的 transform 不能跨列计算，请支持此功能，即仍返回 Series 但 col 参数为多列

- 无需考虑性能与异常处理，只需实现上述功能，在给出测试样例的同时与 pandas 中的 transform 对比结果是否一致

第 5 章　变形

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

In [2]: import pandas as pd
```

5.1　长宽表的变形

什么是长表？什么是宽表？这个概念是对于某一个特征而言的。例如：一个表中把性别存储在某一个列中，那么它就是关于性别的长表；如果把性别作为列名，列中的元素是某一其他的相关特征数值，那么这个表是关于性别的宽表。下面的两张表就分别是关于性别的长表和宽表：

```
In [3]: pd.DataFrame({'Gender':['F','F','M','M'],
   ...:               'Height':[163, 160, 175, 180]})
   ...:
Out[3]:
  Gender  Height
0      F     163
1      F     160
2      M     175
3      M     180

In [4]: pd.DataFrame({'Height: F':[163, 160],
   ...:               'Height: M':[175, 180]})
   ...:
Out[4]:
   Height: F  Height: M
0        163        175
1        160        180
```

显然这两张表从信息上是完全等价的，它们包含相同的身高统计数值，只是这些数值的呈现方式不同，而其呈现方式主要又与性别一列选择的布局模式有关，即到底是以 long 的状态存储还是以 wide 的状态存储。因此，pandas 针对此类长宽表的变形操作设计了一些有关的变形函数。

5.1.1 pivot

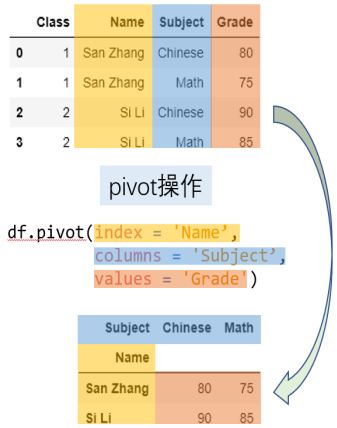
pivot 是一种典型的长表变宽表的函数，首先来看一个例子：下表存储了张三和李四的语文和数学分数，现在想要把语文和数学分数作为列来展示。

```
In [5]: df = pd.DataFrame({'Class':[1,1,2,2],
   ...:                    'Name':['San Zhang','San Zhang','Si Li','Si Li'],
   ...:                    'Subject':['Chinese','Math','Chinese','Math'],
   ...:                    'Grade':[80,75,90,85]})
   ...:

In [6]: df
Out[6]:
   Class       Name  Subject  Grade
0      1  San Zhang  Chinese     80
1      1  San Zhang     Math     75
2      2      Si Li  Chinese     90
3      2      Si Li     Math     85
```

对于一个基本的长变宽的操作而言，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。新生成表的列索引是 columns 对应列的 unique 值，而新表的行索引是 index 对应列的 unique 值，而 values 对应了想要展示的数值列。

```
In [7]: df.pivot(index='Name', columns='Subject', values='Grade')
Out[7]:
Subject    Chinese  Math
Name
San Zhang       80    75
Si Li           90    85
```

通过颜色的标记，更容易地能够理解其变形的过程：



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 对应两个列的行组合必须唯一。例如，现在把原表中第二行张三的数学改为语文就会报错，这是由于 Name 与 Subject 的组合中两次出现 ("San Zhang", "Chinese") ，从而最后不能够确定到底变形后应该是填写 80 分还是 75 分。

```
In [8]: df.loc[1, 'Subject'] = 'Chinese'

In [9]: try:
   ...:  df.pivot(index='Name', columns='Subject', values='Grade')
   ...: except Exception as e:
   ...:    Err_Msg = e
   ...:

In [10]: Err_Msg
Out[10]: ValueError('Index contains duplicate entries, cannot reshape')
```

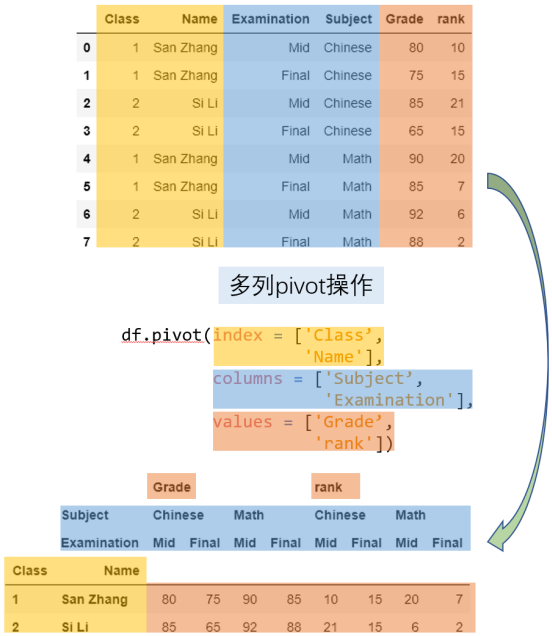
pandas 从 1.1.0 开始，pivot 相关的三个参数允许被设置为列表，这也意味着会返回多级索引。这里构造一个相应的例子来说明如何使用：下表中六列分别为班级、姓名、测试类型（期中考试和期末考试）、科目、成绩、排名。

```
In [11]: df = pd.DataFrame({'Class':[1, 1, 2, 2, 1, 1, 2, 2],
   ....:                    'Name':['San Zhang', 'San Zhang', 'Si Li', 'Si Li',
   ....:                            'San Zhang', 'San Zhang', 'Si Li', 'Si Li'],
   ....:                    'Examination': ['Mid', 'Final', 'Mid', 'Final',
   ....:                                    'Mid', 'Final', 'Mid', 'Final'],
   ....:                    'Subject':['Chinese', 'Chinese', 'Chinese', 'Chinese',
   ....:                               'Math', 'Math', 'Math', 'Math'],
   ....:                    'Grade':[80, 75, 85, 65, 90, 85, 92, 88],
   ....:                    'rank':[10, 15, 21, 15, 20, 7, 6, 2]})
   ....:

In [12]: df
Out[12]:
   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
```

现在想要把测试类型和科目联合组成的四个类别（期中语文、期末语文、期中数学、期末数学）转到列索引，并且同时统计成绩和排名：

```
In [13]: pivot_multi = df.pivot(index = ['Class', 'Name'],
   ....:                        columns = ['Subject','Examination'],
   ....:                        values = ['Grade','rank'])
   ....:

In [14]: pivot_multi
Out[14]:
                  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
```

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（与 index 类似）。从下面的示意图中能够比较容易地理解相应的操作：



5.1.2 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。

```
In [15]: df = pd.DataFrame({'Name':['San Zhang', 'San Zhang',
   ....:                            'San Zhang', 'San Zhang',
   ....:                            'Si Li', 'Si Li', 'Si Li', 'Si Li'],
   ....:                    'Subject':['Chinese', 'Chinese', 'Math', 'Math',
   ....:                               'Chinese', 'Chinese', 'Math', 'Math'],
   ....:                    'Grade':[80, 90, 100, 90, 70, 80, 85, 95]})
   ....:

In [16]: df
Out[16]:
        Name  Subject  Grade
0  San Zhang  Chinese     80
1  San Zhang  Chinese     90
2  San Zhang     Math    100
3  San Zhang     Math     90
4      Si Li  Chinese     70
5      Si Li  Chinese     80
6      Si Li     Math     85
7      Si Li     Math     95
```

pandas 中提供了 pivot_table 来实现，其中的 aggfunc 参数就是使用的聚合函数。上述场景可以如下写出：

```
In [17]: df.pivot_table(index = 'Name',
   ....:                columns = 'Subject',
   ....:                values = 'Grade',
   ....:                aggfunc = 'mean')
   ....:
Out[17]:
Subject    Chinese  Math
Name
San Zhang       85    95
Si Li           75    90
```

这里传入 aggfunc 包含了上一章中介绍的所有合法聚合字符串，此外还可以传入以序列为输入标量为输出的聚合函数来实现自定义操作，上述功能可以等价写出：

```
In [18]: df.pivot_table(index = 'Name',
   ....:                columns = 'Subject',
   ....:                values = 'Grade',
   ....:                aggfunc = lambda x:x.mean())
   ....:
Out[18]:
Subject    Chinese  Math
Name
```

(continues on next page)

(continued from previous page)

```
San Zhang       85    95
Si Li           75    90
```

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。下面就分别统计了语文均分和数学均分、张三均分和李四均分，以及总体所有分数的均分：

```
In [19]: df.pivot_table(index = 'Name',
   ....:                columns = 'Subject',
   ....:                values = 'Grade',
   ....:                aggfunc='mean',
   ....:                margins=True)
   ....:
Out[19]:
Subject    Chinese  Math    All
Name
San Zhang       85  95.0  90.00
Si Li           75  90.0  82.50
All             80  92.5  86.25
```

练一练

在上面的边际汇总例子中，行或列的汇总为新表中行元素或者列元素的平均值，而总体的汇总为新表中四个元素的平均值。这种关系一定成立吗？若不成立，请给出一个例子来说明。

5.1.3 melt

长宽表只是数据呈现方式的差异，但其包含的信息量是等价的，前面提到了利用 pivot 把长表转为宽表，那么就可以通过相应的逆操作把宽表转为长表，melt 函数就起到了这样的作用。在下面的例子中，Subject 以列索引的形式存储，现在想要将其压缩到一个列中。

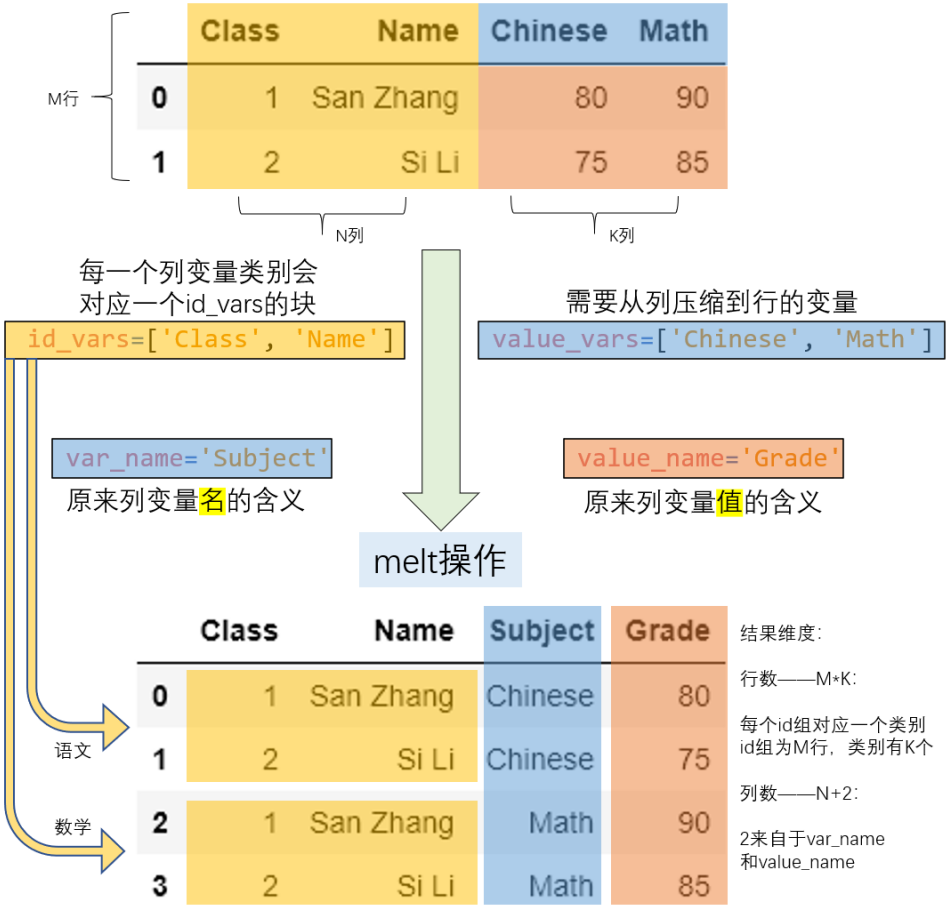
```
In [20]: df = pd.DataFrame({'Class':[1,2],
   ....:                    'Name':['San Zhang', 'Si Li'],
   ....:                    'Chinese':[80, 90],
   ....:                    'Math':[80, 75]})
   ....:

In [21]: df
Out[21]:
   Class       Name  Chinese  Math
```

(continues on next page)

(continued from previous page)

```
0      1  San Zhang        80    80
1      2       Si Li       90    75

In [22]: df_melted = df.melt(id_vars = ['Class', 'Name'],
   ....:                     value_vars = ['Chinese', 'Math'],
   ....:                     var_name = 'Subject',
   ....:                     value_name = 'Grade')
   ....:

In [23]: df_melted
Out[23]:
   Class       Name  Subject  Grade
0      1  San Zhang  Chinese     80
1      2       Si Li  Chinese     90
2      1  San Zhang     Math     80
3      2       Si Li     Math     75
```

melt 的主要参数和压缩的过程如下图所示：



前面提到了 melt 和 pivot 是一组互逆过程，那么就一定可以通过 pivot 操作把 df_melted 转回 df 的形式：

```
In [24]: df_unmelted = df_melted.pivot(index = ['Class', 'Name'],
   ....:                               columns='Subject',
   ....:                               values='Grade')
   ....:

In [25]: df_unmelted # 下面需要恢复索引，并且重命名列索引名称
Out[25]:
Subject          Chinese  Math
Class Name
1     San Zhang       80    80
2     Si Li           90    75

In [26]: df_unmelted = df_unmelted.reset_index().rename_axis(
   ....:                               columns={'Subject':''})
   ....:

In [27]: df_unmelted.equals(df)
Out[27]: True
```

5.1.4 wide_to_long

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，比如期中期末的类别和语文数学的类别，那么想要把 values_name 对应的 Grade 扩充为两列分别对应语文分数和数学分数，只把期中期末的信息压缩，这种需求下就要使用 wide_to_long 函数来完成。

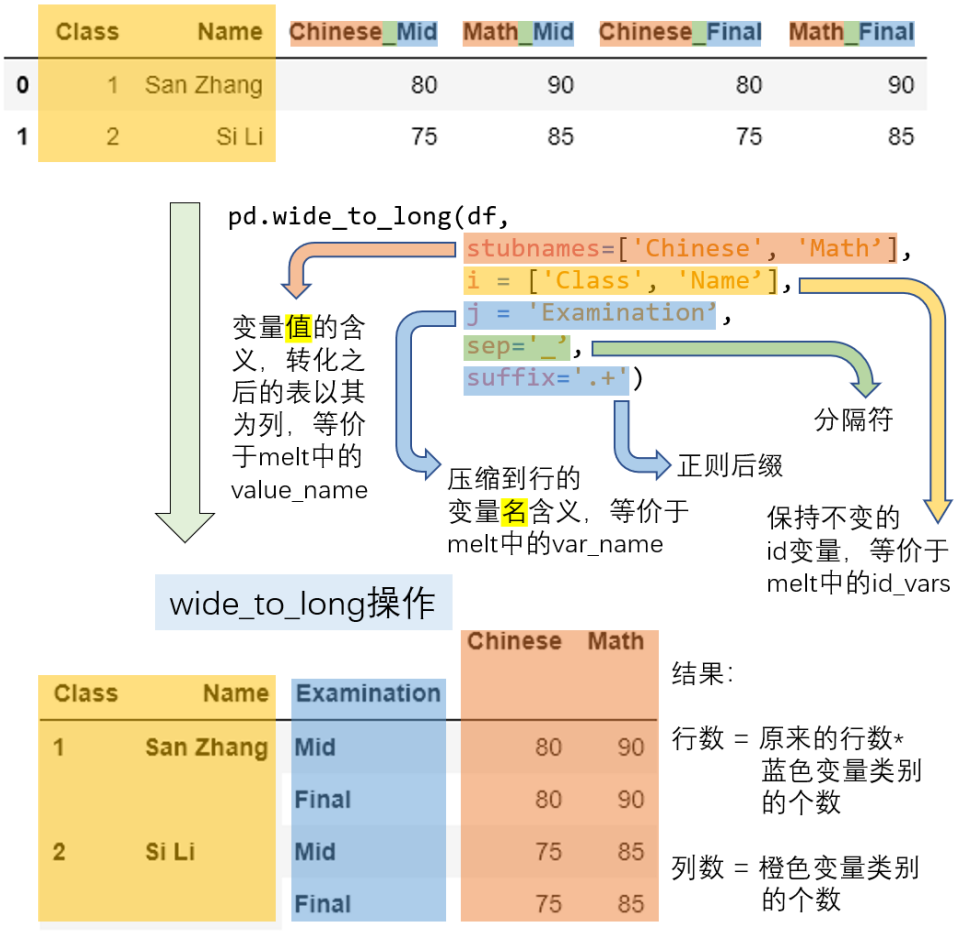
```
In [28]: df = pd.DataFrame({'Class':[1,2],'Name':['San Zhang', 'Si Li'],
   ....:                    'Chinese_Mid':[80, 75], 'Math_Mid':[90, 85],
   ....:                    'Chinese_Final':[80, 75], 'Math_Final':[90, 85]})
   ....:

In [29]: df
Out[29]:
   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

In [30]: pd.wide_to_long(df,
   ....:                 stubnames=['Chinese', 'Math'],
   ....:                 i = ['Class', 'Name'],
   ....:                 j='Examination',
```

(continues on next page)

(continued from previous page)

```
   ....:                 sep='_',
   ....:                 suffix='.+')
   ....:
Out[30]:
                           Chinese  Math
Class Name      Examination
1     San Zhang Mid             80    90
                Final           80    90
2     Si Li     Mid             75    85
                Final           75    85
```

具体的变换过程由下图进行展示，属相同概念的元素使用了一致的颜色标出：



下面给出一个比较复杂的案例，把之前在 pivot 一节中多列操作的结果（产生了多级索引），利用 wide_to_long 函数，将其转为原来的形态。其中，使用了第八章的 str.split 函数，目前暂时只需将其理解为对序列按照某个分隔符进行拆分即可。

```
In [31]: res = pivot_multi.copy()

In [32]: res.columns = res.columns.map(lambda x:'_'.join(x))

In [33]: res = res.reset_index()

In [34]: res = pd.wide_to_long(res, stubnames=['Grade', 'rank'],
   ....:                               i = ['Class', 'Name'],
   ....:                               j = 'Subject_Examination',
   ....:                               sep = '_',
   ....:                               suffix = '.+')
   ....:

In [35]: res = res.reset_index()

In [36]: res[['Subject', 'Examination']] = res[
   ....:                  'Subject_Examination'].str.split('_', expand=True)
   ....:

In [37]: res = res[['Class', 'Name', 'Examination',
   ....:             'Subject', 'Grade', 'rank']].sort_values('Subject')
   ....:

In [38]: res = res.reset_index(drop=True)

In [39]: res
Out[39]:
   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
```

5.2 索引的变形

5.2.1 stack 与 unstack

在第二章中提到了利用 swaplevel 或者 reorder_levels 进行索引内部的层交换，下面就要讨论 行列索引之间的交换，由于这种交换带来了 DataFrame 维度上的变化，因此属于变形操作。在第一节中提到的 4 种变形函数与其不同之处在于，它们都属于某一列或几列 元素和 列索引之间的转换，而不是索引之间的转换。

unstack 函数的作用是把行索引转为列索引，例如下面这个简单的例子：

```
In [40]: df = pd.DataFrame(np.ones((4,2)),
   ....:                   index = pd.Index([('A', 'cat', 'big'),
   ....:                                     ('A', 'dog', 'small'),
   ....:                                     ('B', 'cat', 'big'),
   ....:                                     ('B', 'dog', 'small')]),
   ....:                   columns=['col_1', 'col_2'])
   ....:

In [41]: df
Out[41]:
              col_1  col_2
A cat big       1.0    1.0
  dog small     1.0    1.0
B cat big       1.0    1.0
  dog small     1.0    1.0

In [42]: df.unstack()
Out[42]:
      col_1       col_2
        big small   big small
A cat   1.0   NaN   1.0   NaN
  dog   NaN   1.0   NaN   1.0
B cat   1.0   NaN   1.0   NaN
  dog   NaN   1.0   NaN   1.0
```

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：

```
In [43]: df.unstack(2)
Out[43]:
      col_1       col_2
        big small   big small
A cat   1.0   NaN   1.0   NaN
  dog   NaN   1.0   NaN   1.0
B cat   1.0   NaN   1.0   NaN
```

(continues on next page)

(continued from previous page)

```
  dog   NaN   1.0   NaN   1.0

In [44]: df.unstack([0,2])
Out[44]:
    col_1                 col_2
        A          B          A          B
      big small  big small   big small  big small
cat   1.0   NaN  1.0   NaN   1.0   NaN  1.0   NaN
dog   NaN   1.0  NaN   1.0   NaN   1.0  NaN   1.0
```

类似于 pivot 中的唯一性要求，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，例如把前两个列索引改成相同的破坏唯一性，那么就会报错：

```
In [45]: my_index = df.index.to_list()

In [46]: my_index[1] = my_index[0]

In [47]: df.index = pd.Index(my_index)

In [48]: df
Out[48]:
             col_1  col_2
A cat big      1.0    1.0
      big      1.0    1.0
B cat big      1.0    1.0
  dog small    1.0    1.0

In [49]: try:
   ....:     df.unstack()
   ....: except Exception as e:
   ....:     Err_Msg = e
   ....:

In [50]: Err_Msg
Out[50]: ValueError('Index contains duplicate entries, cannot reshape')
```

与 unstack 相反，stack 的作用就是把列索引的层压入行索引，其用法完全类似。

```
In [51]: df = pd.DataFrame(np.ones((4,2)),
   ....:                   index = pd.Index([('A', 'cat', 'big'),
   ....:                                     ('A', 'dog', 'small'),
   ....:                                     ('B', 'cat', 'big'),
   ....:                                     ('B', 'dog', 'small')]),
```

(continues on next page)

(continued from previous page)

```
   ....:                     columns=['index_1', 'index_2']).T
   ....:

In [52]: df
Out[52]:
           A          B
         cat   dog  cat   dog
         big small  big small
index_1  1.0   1.0  1.0   1.0
index_2  1.0   1.0  1.0   1.0

In [53]: df.stack()
Out[53]:
                 A         B
               cat  dog  cat  dog
index_1 big    1.0  NaN  1.0  NaN
        small  NaN  1.0  NaN  1.0
index_2 big    1.0  NaN  1.0  NaN
        small  NaN  1.0  NaN  1.0

In [54]: df.stack([1, 2])
Out[54]:
                     A    B
index_1 cat big    1.0  1.0
        dog small  1.0  1.0
index_2 cat big    1.0  1.0
        dog small  1.0  1.0
```

5.2.2 聚合与变形的关系

在上面介绍的所有函数中，除了带有聚合效果的 pivot_table 以外，所有的函数在变形前后并不会带来 values 个数的改变，只是这些值在呈现的形式上发生了变化。在上一章讨论的分组聚合操作，由于生成了新的行列索引，因此必然也属于某种特殊的变形操作，但由于聚合之后把原来的多个值变为了一个值，因此 values 的个数产生了变化，这也是分组聚合与变形函数的最大区别。

5.3 其他变形函数

5.3.1 crosstab

crosstab 并不是一个值得推荐使用的函数，因为它能实现的所有功能 pivot_table 都能完成，并且速度更快。在默认状态下，crosstab 可以统计元素组合出现的频数，即 count 操作。例如统计 learn_pandas 数据集中学校和转系情况对应的频数：

```
In [55]: df = pd.read_csv('data/learn_pandas.csv')

In [56]: pd.crosstab(index = df.School, columns = df.Transfer)
Out[56]:
Transfer                         N  Y
School
Fudan University                38  1
Peking University               28  2
Shanghai Jiao Tong University   53  0
Tsinghua University             62  4
```

这等价于如下 crosstab 的如下写法，这里的 aggfunc 即聚合参数：

```
In [57]: pd.crosstab(index = df.School, columns = df.Transfer,
   ....:             values = [0]*df.shape[0], aggfunc = 'count')
   ....:
Out[57]:
Transfer                           N    Y
School
Fudan University                38.0  1.0
Peking University               28.0  2.0
Shanghai Jiao Tong University   53.0  NaN
Tsinghua University             62.0  4.0
```

同样，可以利用 pivot_table 进行等价操作，由于这里统计的是组合的频数，因此 values 参数无论传入哪一个列都不会影响最后的结果：

```
In [58]: df.pivot_table(index = 'School',
   ....:                columns = 'Transfer',
   ....:                values = 'Name',
   ....:                aggfunc = 'count')
   ....:
Out[58]:
Transfer                           N    Y
School
```

(continues on next page)

(continued from previous page)

```
Fudan University                38.0  1.0
Peking University               28.0  2.0
Shanghai Jiao Tong University   53.0  NaN
Tsinghua University             62.0  4.0
```

从上面可以看出这两个函数的区别在于，crosstab 的对应位置传入的是具体的序列，而 pivot_table 传入的是被调用表对应的名字，若传入序列对应的值则会报错。

除了默认状态下的 count 统计，所有的聚合字符串和返回标量的自定义函数都是可用的，例如统计对应组合的身高均值：

```
In [59]: pd.crosstab(index = df.School, columns = df.Transfer,
   ....:             values = df.Height, aggfunc = 'mean')
   ....:
Out[59]:
Transfer                              N       Y
School
Fudan University                162.043750  177.20
Peking University               163.429630  162.40
Shanghai Jiao Tong University   163.953846     NaN
Tsinghua University             163.253571  164.55
```

练一练

前面提到了 crosstab 的性能劣于 pivot_table ，请选用多个聚合方法进行验证。

5.3.2 explode

explode 参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。

```
In [60]: df_ex = pd.DataFrame({'A': [[1, 2],
   ....:                             'my_str',
   ....:                             {1, 2},
   ....:                             pd.Series([3, 4])],
   ....:                          'B': 1})
   ....:

In [61]: df_ex.explode('A')
Out[61]:
        A  B
```

(continues on next page)

(continued from previous page)

```
0       1  1
0       2  1
1  my_str  1
2  {1, 2}  1
3       3  1
3       4  1
```

5.3.3 get_dummies

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。例如，对年级一列转为指示变量，属于某一个年级的对应列标记为 1，否则为 0：

```
In [62]: pd.get_dummies(df.Grade).head()
Out[62]:
   Freshman  Junior  Senior  Sophomore
0         1       0       0          0
1         1       0       0          0
2         0       0       1          0
3         0       0       0          1
4         0       0       0          1
```

5.4 练习

5.4.1 Ex1：美国非法药物数据集

现有一份关于美国非法药物的数据集，其中 SubstanceName, DrugReports 分别指药物名称和报告数量：

```
In [63]: df = pd.read_csv('data/drugs.csv').sort_values([
   ....:       'State','COUNTY','SubstanceName'],ignore_index=True)
   ....:

In [64]: df.head(3)
Out[64]:
   YYYY State COUNTY  SubstanceName  DrugReports
0  2011    KY  ADAIR  Buprenorphine            3
1  2012    KY  ADAIR  Buprenorphine            5
2  2013    KY  ADAIR  Buprenorphine            4
```

1. 将数据转为如下的形式：

	State	COUNTY	SubstanceName	2010	2011	2012	2013	2014	2015	2016	2017
0	KY	ADAIR	Buprenorphine	NaN	3.0	5.0	4.0	27.0	5.0	7.0	10.0
1	KY	ADAIR	Codeine	NaN	NaN	1.0	NaN	NaN	NaN	NaN	1.0
2	KY	ADAIR	Fentanyl	NaN	NaN	1.0	NaN	NaN	NaN	NaN	NaN
3	KY	ADAIR	Heroin	NaN	NaN	1.0	2.0	NaN	1.0	NaN	2.0
4	KY	ADAIR	Hydrocodone	6.0	9.0	10.0	10.0	9.0	7.0	11.0	3.0

2. 将第 1 问中的结果恢复为原表。

3. 按 State 分别统计每年的报告数量总和，其中 State, YYYY 分别为列索引和行索引，要求分别使用 pivot_table 函数与 groupby+unstack 两种不同的策略实现，并体会它们之间的联系。

5.4.2 Ex2：特殊的 wide_to_long 方法

从功能上看，melt 方法应当属于 wide_to_long 的一种特殊情况，即 stubnames 只有一类。请使用 wide_to_long 生成 melt 一节中的 df_melted 。（提示：对列名增加适当的前缀）

```
In [65]: df = pd.DataFrame({'Class':[1,2],
   ....:                    'Name':['San Zhang', 'Si Li'],
   ....:                    'Chinese':[80, 90],
   ....:                    'Math':[80, 75]})
   ....:

In [66]: df
Out[66]:
   Class       Name  Chinese  Math
0      1  San Zhang       80    80
1      2      Si Li       90    75
```

第 6 章　连接

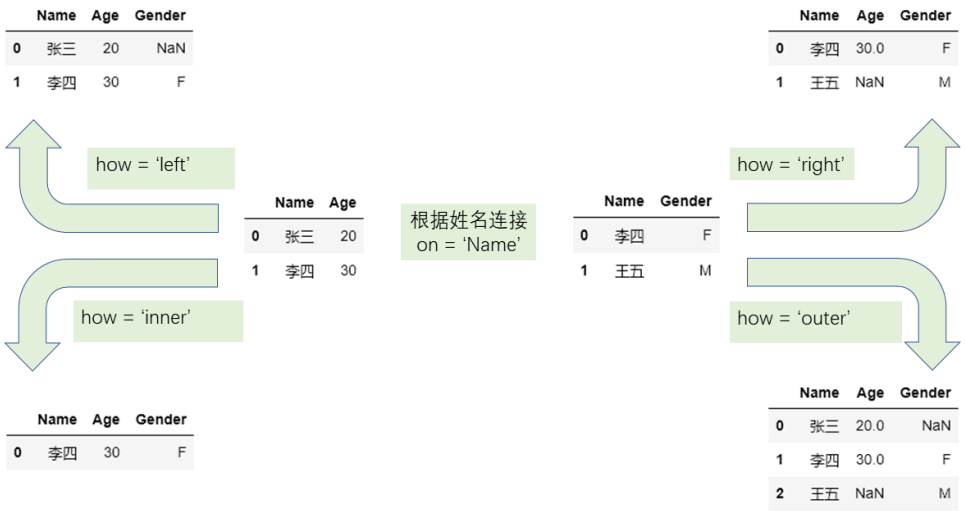
```
In [1]: import numpy as np

In [2]: import pandas as pd
```

6.1　关系型连接

6.1.1　连接的基本概念

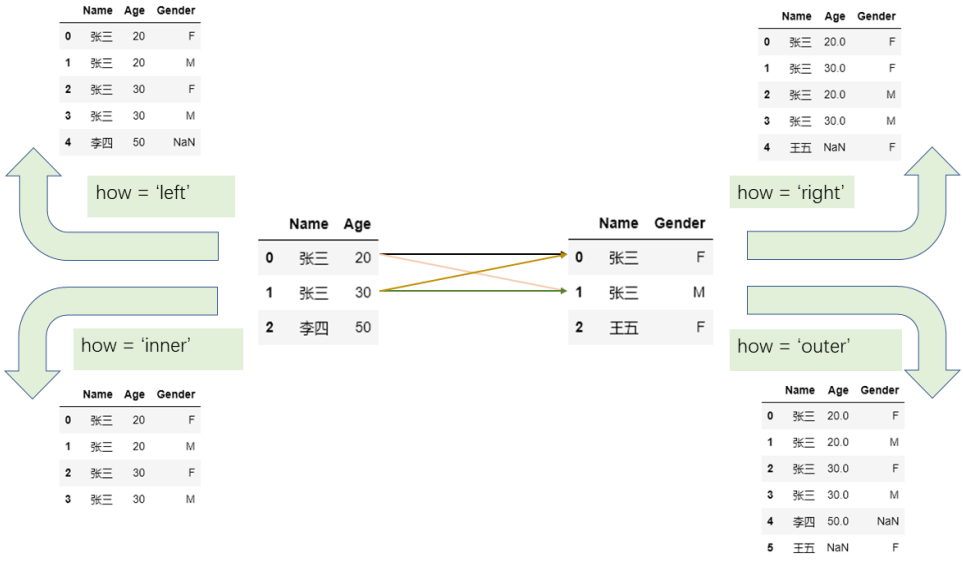
把两张相关的表按照某一个或某一组键连接起来是一种常见操作，例如学生期末考试各个科目的成绩表按照姓名和 班级连接成总的成绩表，又例如对企业员工的各类信息表按照 员工 ID 号进行连接汇总。由此可以看出，在关系型连接中，键是十分重要的，往往用 on 参数表示。

另一个重要的要素是连接的形式。在 pandas 中的关系型连接函数 merge 和 join 中提供了 how 参数来代表连接形式，分为左连接 left 、右连接 right 、内连接 inner 、外连接 outer ，它们的区别可以用如下示意图表示：



从图中可以看到，所谓左连接即以左边的键为准，如果右边表中的键于左边存在，那么就添加到左边，否则则处理为缺失值，右连接类似处理。内连接只负责合并两边同时出现的键，而外连接则会在内连接的基础上包含只在左边出现以及只在右边出现的值，因此外连接又叫全连接。

上面这个简单的例子中，同一个表中的键没有出现重复的情况，那么如果出现重复的键应该如何处理？只需把握一个原则，即只要两边同时出现的值，就以笛卡尔积的方式加入，如果单边出现则根据连接形式进行处理。其中，关于笛卡尔积可用如下例子说明：设左表中键 张三 出现两次，右表中的 张三 也出现两次，那么逐个进行匹配，最后产生的表必然包含 2*2 个姓名为 张三 的行。下面是一个对应例子的示意图：



显然在不同的场合应该使用不同的连接形式。其中左连接和右连接是等价的，由于它们的结果中的键是被一侧的表确定的，因此常常用于有方向性地添加到目标表。内外连接两侧的表，经常是地位类似的，想取出键的交集或者并集，具体的操作还需要业务的需求来判断。

6.1.2 值连接

在上面示意图中的例子中，两张表根据某一列的值来连接，事实上还可以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，例如第一张图的左连接：

```
In [3]: df1 = pd.DataFrame({'Name':['San Zhang','Si Li'],
   ...:                     'Age':[20,30]})
   ...:

In [4]: df2 = pd.DataFrame({'Name':['Si Li','Wu Wang'],
   ...:                     'Gender':['F','M']})
   ...:

In [5]: df1.merge(df2, on='Name', how='left')
```

(continues on next page)

(continued from previous page)

```
Out[5]:
        Name  Age Gender
0  San Zhang   20    NaN
1      Si Li   30      F
```

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：

```
In [6]: df1 = pd.DataFrame({'df1_name':['San Zhang','Si Li'],
   ...:                     'Age':[20,30]})
   ...:

In [7]: df2 = pd.DataFrame({'df2_name':['Si Li','Wu Wang'],
   ...:                     'Gender':['F','M']})
   ...:

In [8]: df1.merge(df2, left_on='df1_name', right_on='df2_name', how='left')
Out[8]:
    df1_name  Age df2_name Gender
0  San Zhang   20      NaN    NaN
1      Si Li   30    Si Li      F
```

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。例如合并考试成绩的时候，第一个表记录了语文成绩，第二个是数学成绩：

```
In [9]: df1 = pd.DataFrame({'Name':['San Zhang'],'Grade':[70]})

In [10]: df2 = pd.DataFrame({'Name':['San Zhang'],'Grade':[80]})

In [11]: df1.merge(df2, on='Name', how='left', suffixes=['_Chinese','_Math'])
Out[11]:
        Name  Grade_Chinese  Grade_Math
0  San Zhang             70          80
```

在某些时候出现重复元素是麻烦的，例如两位同学来自不同的班级，但是姓名相同，这种时候就要指定 on 参数为多个列使得正确连接：

```
In [12]: df1 = pd.DataFrame({'Name':['San Zhang', 'San Zhang'],
   ....:                     'Age':[20, 21],
   ....:                     'Class':['one', 'two']})
   ....:

In [13]: df2 = pd.DataFrame({'Name':['San Zhang', 'San Zhang'],
   ....:                     'Gender':['F', 'M'],
```

(continues on next page)

(continued from previous page)

```
   ....:                     'Class':['two', 'one']})
   ....:

In [14]: df1
Out[14]:
        Name  Age Class
0  San Zhang   20   one
1  San Zhang   21   two

In [15]: df2
Out[15]:
        Name Gender Class
0  San Zhang      F   two
1  San Zhang      M   one

In [16]: df1.merge(df2, on='Name', how='left') # 错误的结果
Out[16]:
        Name  Age Class_x Gender Class_y
0  San Zhang   20     one      F     two
1  San Zhang   20     one      M     one
2  San Zhang   21     two      F     two
3  San Zhang   21     two      M     one

In [17]: df1.merge(df2, on=['Name', 'Class'], how='left') # 正确的结果
Out[17]:
        Name  Age Class Gender
0  San Zhang   20   one      M
1  San Zhang   21   two      F
```

从上面的例子来看，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。举例中的行数很少，但如果实际数据中有几十万到上百万行的进行合并时，如果想要保证唯一性，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。

练一练

上面以多列为键的例子中，错误写法显然是一种多对多连接，而正确写法是一对一连接，请修改原表，使得以多列为键的正确写法能够通过 validate='1:m' 的检验，但不能通过 validate='m:1' 的检验。

6.1.3 索引连接

所谓索引连接，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。

```
In [18]: df1 = pd.DataFrame({'Age':[20,30]},
   ....:                    index=pd.Series(
   ....:                    ['San Zhang','Si Li'],name='Name'))
   ....:

In [19]: df2 = pd.DataFrame({'Gender':['F','M']},
   ....:                    index=pd.Series(
   ....:                    ['Si Li','Wu Wang'],name='Name'))
   ....:

In [20]: df1.join(df2, how='left')
Out[20]:
           Age Gender
Name
San Zhang   20    NaN
Si Li       30      F
```

仿照第 2 小节的例子，写出语文和数学分数合并的 join 版本：

```
In [21]: df1 = pd.DataFrame({'Grade':[70]},
   ....:                    index=pd.Series(['San Zhang'],
   ....:                    name='Name'))
   ....:

In [22]: df2 = pd.DataFrame({'Grade':[80]},
   ....:                    index=pd.Series(['San Zhang'],
   ....:                    name='Name'))
   ....:

In [23]: df1.join(df2, how='left', lsuffix='_Chinese', rsuffix='_Math')
Out[23]:
           Grade_Chinese  Grade_Math
Name
San Zhang             70          80
```

如果想要进行类似于 merge 中以多列为键的操作的时候，join 需要使用多级索引，例如在 merge 中的最后一个例子可以如下写出：

```
In [24]: df1 = pd.DataFrame({'Age':[20,21]},
   ....:                    index=pd.MultiIndex.from_arrays(
   ....:                    [['San Zhang', 'San Zhang'],['one', 'two']],
   ....:                    names=('Name','Class')))
   ....:

In [25]: df2 = pd.DataFrame({'Gender':['F', 'M']},
   ....:                    index=pd.MultiIndex.from_arrays(
   ....:                    [['San Zhang', 'San Zhang'],['two', 'one']],
   ....:                    names=('Name','Class')))
   ....:

In [26]: df1
Out[26]:
                 Age
Name      Class
San Zhang one     20
          two     21

In [27]: df2
Out[27]:
                Gender
Name      Class
San Zhang two        F
          one        M

In [28]: df1.join(df2)
Out[28]:
                 Age Gender
Name      Class
San Zhang one     20      M
          two     21      F
```

6.2 方向连接

6.2.1 concat

前面介绍了关系型连接，其中最重要的参数是 on 和 how ，但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，为这种需求，pandas 中提供了 concat 函数来实现。

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。这里需要特别注意，join 和 keys 与之前提到的 join 函数和键的概念没有任

何关系。

在默认状态下的 axis=0 ，表示纵向拼接多个表，常常用于多个样本的拼接；而 axis=1 表示横向拼接多个表，常用于多个字段或特征的拼接。

例如，纵向合并各表中人的信息：

```
In [29]: df1 = pd.DataFrame({'Name':['San Zhang','Si Li'],
   ....:                     'Age':[20,30]})
   ....:

In [30]: df2 = pd.DataFrame({'Name':['Wu Wang'], 'Age':[40]})

In [31]: pd.concat([df1, df2])
Out[31]:
        Name  Age
0  San Zhang   20
1      Si Li   30
0    Wu Wang   40
```

横向合并各表中的字段：

```
In [32]: df2 = pd.DataFrame({'Grade':[80, 90]})

In [33]: df3 = pd.DataFrame({'Gender':['M', 'F']})

In [34]: pd.concat([df1, df2, df3], 1)
Out[34]:
        Name  Age  Grade Gender
0  San Zhang   20     80      M
1      Si Li   30     90      F
```

虽然说 concat 不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。纵向拼接会根据列索引对其，默认状态下 join=outer ，表示保留所有的列，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列。横向拼接则根据行索引对齐，join 参数可以类似设置。

```
In [35]: df2 = pd.DataFrame({'Name':['Wu Wang'], 'Gender':['M']})

In [36]: pd.concat([df1, df2])
Out[36]:
        Name   Age Gender
0  San Zhang  20.0    NaN
1      Si Li  30.0    NaN
0    Wu Wang   NaN      M
```

(continues on next page)

(continued from previous page)

```
In [37]: df2 = pd.DataFrame({'Grade':[80, 90]}, index=[1, 2])

In [38]: pd.concat([df1, df2], 1)
Out[38]:
        Name   Age  Grade
0  San Zhang  20.0    NaN
1      Si Li  30.0   80.0
2        NaN   NaN   90.0

In [39]: pd.concat([df1, df2], axis=1, join='inner')
Out[39]:
    Name  Age  Grade
1  Si Li   30     80
```

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。

最后，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 参数产生多级索引进行标记。例如，第一个表中都是一班的同学，而第二个表中都是二班的同学，可以使用如下方式合并：

```
In [40]: df1 = pd.DataFrame({'Name':['San Zhang','Si Li'],
   ....:                     'Age':[20,21]})
   ....:

In [41]: df2 = pd.DataFrame({'Name':['Wu Wang'],'Age':[21]})

In [42]: pd.concat([df1, df2], keys=['one', 'two'])
Out[42]:
            Name  Age
one 0  San Zhang   20
    1      Si Li   21
two 0    Wu Wang   21
```

6.2.2 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 对新序列对应索引的自动标号，否则必须对 Series 指定 name 属性。

```
In [43]: s = pd.Series(['Wu Wang', 21], index = df1.columns)

In [44]: df1.append(s, ignore_index=True)
Out[44]:
        Name  Age
0  San Zhang   20
1      Si Li   21
2    Wu Wang   21
```

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：

```
In [45]: s = pd.Series([80, 90])

In [46]: df1.assign(Grade=s)
Out[46]:
        Name  Age  Grade
0  San Zhang   20     80
1      Si Li   21     90

In [47]: df1['Grade'] = s

In [48]: df1
Out[48]:
        Name  Age  Grade
0  San Zhang   20     80
1      Si Li   21     90
```

6.3 类连接操作

除了上述介绍的若干连接函数之外，pandas 中还设计了一些函数能够对两个表进行某些操作，这里把它们统称为类连接操作。

6.3.1 比较

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：

```
In [49]: df1 = pd.DataFrame({'Name':['San Zhang', 'Si Li', 'Wu Wang'],
   ....:                     'Age':[20, 21 ,21],
   ....:                     'Class':['one', 'two', 'three']})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [50]: df2 = pd.DataFrame({'Name':['San Zhang', 'Li Si', 'Wu Wang'],
   ....:                     'Age':[20, 21 ,21],
   ....:                     'Class':['one', 'two', 'Three']})
   ....:

In [51]: df1.compare(df2)
Out[51]:
    Name         Class
    self  other   self  other
1  Si Li  Li Si    NaN    NaN
2    NaN    NaN  three  Three
```

结果中返回了不同值所在的行列，如果相同则会被填充为缺失值 NaN ，其中 other 和 self 分别指代传入的参数表和被调用的表自身。

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 ：

```
In [52]: df1.compare(df2, keep_shape=True)
Out[52]:
    Name         Age        Class
    self  other self other   self  other
0    NaN    NaN  NaN   NaN    NaN    NaN
1  Si Li  Li Si  NaN   NaN    NaN    NaN
2    NaN    NaN  NaN   NaN  three  Three
```

6.3.2 组合

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，例如下面这个例子会依次传入 A,B,C,D 四组序列，每组为左右表的两个序列。同时，进行 A 列比较的时候，s1 指代的就是一个全空的序列，因为它在被调用的表中并不存在，并且来自第一个表的序列索引会被 reindex 成两个索引的并集。具体的过程可以通过在传入的函数中插入适当的 print 方法查看。

下面的例子表示选出对应索引位置较小的元素：

```
In [53]: 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
   ....:
```

(continues on next page)

(continued from previous page)

```
In [54]: df1 = pd.DataFrame({'A':[1,2], 'B':[3,4], 'C':[5,6]})

In [55]: df2 = pd.DataFrame({'B':[5,6], 'C':[7,8], 'D':[9,10]}, index=[1,2])

In [56]: df1.combine(df2, choose_min)
Out[56]:
    A    B    C   D
0 NaN  NaN  NaN NaN
1 NaN  4.0  6.0 NaN
2 NaN  NaN  NaN NaN
```

练一练

请在上述代码的基础上修改，保留 df2 中 4 个未被 df1 替换的相应位置原始值。

此外，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：

```
In [57]: df1.combine(df2, choose_min, overwrite=False)
Out[57]:
     A    B    C   D
0  1.0  NaN  NaN NaN
1  2.0  4.0  6.0 NaN
2  NaN  NaN  NaN NaN
```

练一练

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。下面给出一个例子，请用 combine 函数完成相同的功能。

```
In [58]: df1 = pd.DataFrame({'A':[1,2], 'B':[3,np.nan]})

In [59]: df2 = pd.DataFrame({'A':[5,6], 'B':[7,8]}, index=[1,2])

In [60]: df1.combine_first(df2)
Out[60]:
     A    B
0  1.0  3.0
```

(continues on next page)

(continued from previous page)

```
1  2.0  7.0
2  6.0  8.0
```

6.4 练习

6.4.1 Ex1：美国疫情数据集

现有美国 4 月 12 日至 11 月 16 日的疫情报表，请将 New York 的 Confirmed, Deaths, Recovered, Active 合并为一张表，索引为按如下方法生成的日期字符串序列：

```
In [61]: date = pd.date_range('20200412', '20201116').to_series()

In [62]: date = date.dt.month.astype('string').str.zfill(2
   ....:         ) +'-'+ date.dt.day.astype('string'
   ....:         ).str.zfill(2) +'-'+ '2020'
   ....:

In [63]: date = date.tolist()

In [64]: date[:5]
Out[64]: ['04-12-2020', '04-13-2020', '04-14-2020', '04-15-2020', '04-16-2020']
```

6.4.2 Ex2：实现 join 函数

请实现带有 how 参数的 join 函数

- 假设连接的两表无公共列
- 调用方式为 join(df1, df2, how="left")
- 给出测试样例

第 7 章　缺失数据

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

In [2]: import pandas as pd
```

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，通过和 sum 的组合可以计算出每列缺失值的比例：

```
In [3]: df = pd.read_csv('data/learn_pandas.csv',
   ...:                  usecols = ['Grade', 'Name', 'Gender', 'Height',
   ...:                             'Weight', 'Transfer'])
   ...:

In [4]: df.isna().head()
Out[4]:
   Grade   Name  Gender  Height  Weight  Transfer
0  False  False   False   False   False     False
1  False  False   False   False   False     False
2  False  False   False   False   False     False
3  False  False   False    True   False     False
4  False  False   False   False   False     False

In [5]: df.isna().sum()/df.shape[0] # 查看缺失的比例
Out[5]:
Grade       0.000
Name        0.000
Gender      0.000
Height      0.085
```

(continues on next page)

(continued from previous page)

```
Weight      0.055
Transfer    0.060
dtype: float64
```

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。例如，查看身高缺失的行：

```
In [6]: df[df.Height.isna()].head()
Out[6]:
        Grade          Name  Gender  Height  Weight Transfer
3   Sophomore  Xiaojuan Sun  Female     NaN    41.0        N
12     Senior      Peng You  Female     NaN    48.0      NaN
26     Junior     Yanli You  Female     NaN    48.0        N
36   Freshman  Xiaojuan Qin    Male     NaN    79.0        Y
60   Freshman    Yanpeng Lv    Male     NaN    65.0        N
```

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。例如，对身高、体重和转系情况这 3 列分别进行这三种情况的检索：

```
In [7]: sub_set = df[['Height', 'Weight', 'Transfer']]

In [8]: df[sub_set.isna().all(1)] # 全部缺失
Out[8]:
      Grade          Name Gender  Height  Weight Transfer
102  Junior  Chengli Zhao   Male     NaN     NaN      NaN

In [9]: df[sub_set.isna().any(1)].head() # 至少有一个缺失
Out[9]:
        Grade           Name  Gender  Height  Weight Transfer
3   Sophomore   Xiaojuan Sun  Female     NaN    41.0        N
9      Junior        Juan Xu  Female   164.8     NaN        N
12     Senior      Peng You  Female     NaN    48.0      NaN
21     Senior  Xiaopeng Shen    Male   166.0    62.0      NaN
26     Junior     Yanli You  Female     NaN    48.0        N

In [10]: df[sub_set.notna().all(1)].head() # 没有缺失
Out[10]:
       Grade            Name  Gender  Height  Weight Transfer
0   Freshman    Gaopeng Yang  Female   158.9    46.0        N
1   Freshman  Changqiang You    Male   166.5    70.0        N
2     Senior         Mei Sun    Male   188.9    89.0        N
4  Sophomore     Gaojuan You    Male   174.0    74.0        N
5   Freshman     Xiaoli Qian  Female   158.0    51.0        N
```

7.1.2 缺失信息的删除

数据处理中经常需要根据缺失值的大小、比例或其他特征来进行行样本或列特征的删除，pandas 中提供了 dropna 函数来进行操作。

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。

例如，删除身高体重至少有一个缺失的行：

```
In [11]: res = df.dropna(how = 'any', subset = ['Height', 'Weight'])

In [12]: res.shape
Out[12]: (174, 6)
```

例如，删除超过 15 个缺失值的列：

```
In [13]: res = df.dropna(1, thresh=df.shape[0]-15) # 身高被删除

In [14]: res.head()
Out[14]:
       Grade            Name  Gender  Weight Transfer
0   Freshman    Gaopeng Yang  Female    46.0        N
1   Freshman  Changqiang You    Male    70.0        N
2     Senior         Mei Sun    Male    89.0        N
3  Sophomore    Xiaojuan Sun  Female    41.0        N
4  Sophomore     Gaojuan You    Male    74.0        N
```

当然，不用 dropna 同样是可行的，例如上述的两个操作，也可以使用布尔索引来完成：

```
In [15]: res = df.loc[df[['Height', 'Weight']].notna().all(1)]

In [16]: res.shape
Out[16]: (174, 6)

In [17]: res = df.loc[:, ~(df.isna().sum()>15)]

In [18]: res.head()
Out[18]:
       Grade            Name  Gender  Weight Transfer
0   Freshman    Gaopeng Yang  Female    46.0        N
1   Freshman  Changqiang You    Male    70.0        N
2     Senior         Mei Sun    Male    89.0        N
3  Sophomore    Xiaojuan Sun  Female    41.0        N
```

(continues on next page)

(continued from previous page)

```
4  Sophomore     Gaojuan You    Male    74.0        N
```

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；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 参数表示连续缺失值的最大填充次数。

下面构造一个简单的 Series 来说明用法：

```
In [19]: s = pd.Series([np.nan, 1, np.nan, np.nan, 2, np.nan],
   ....:               list('aaabcd'))
   ....:

In [20]: s
Out[20]:
a    NaN
a    1.0
a    NaN
b    NaN
c    2.0
d    NaN
dtype: float64

In [21]: s.fillna(method='ffill') # 用前面的值向后填充
Out[21]:
a    NaN
a    1.0
a    1.0
b    1.0
c    2.0
d    2.0
dtype: float64

In [22]: s.fillna(method='ffill', limit=1) # 连续出现的缺失，最多填充一次
Out[22]:
a    NaN
a    1.0
a    1.0
```

(continues on next page)

(continued from previous page)

```
b    NaN
c    2.0
d    2.0
dtype: float64

In [23]: s.fillna(s.mean()) # value 为标量
Out[23]:
a    1.5
a    1.0
a    1.5
b    1.5
c    2.0
d    1.5
dtype: float64

In [24]: s.fillna({'a': 100, 'd': 200}) # 通过索引映射填充的值
Out[24]:
a    100.0
a      1.0
a    100.0
b      NaN
c      2.0
d    200.0
dtype: float64
```

有时为了更加合理地填充，需要先进行分组后再操作。例如，根据年级进行身高的均值填充：

```
In [25]: df.groupby('Grade')['Height'].transform(
   ....:                         lambda x: x.fillna(x.mean())).head()
   ....:
Out[25]:
0    158.900000
1    166.500000
2    188.900000
3    163.075862
4    174.000000
Name: Height, dtype: float64
```

练一练

对一个序列以如下规则填充缺失值：如果单独出现的缺失值，就用前后均值填充，如果连续出现的缺失值就不填充，即序列 [1, NaN, 3, NaN, NaN] 填充后为 [1, 2, 3, NaN, NaN]，请利用 fillna 函数实现。（提示：利用 limit 参数）

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。由于很多插值方法涉及到比较复杂的数学知识，因此这里只讨论比较常用且简单的三类情况，即线性插值、最近邻插值和索引插值。

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 。

```
In [26]: s = pd.Series([np.nan, np.nan, 1,
   ....:                np.nan, np.nan, np.nan,
   ....:                2, np.nan, np.nan])
   ....:

In [27]: s.values
Out[27]: array([nan, nan,  1., nan, nan, nan,  2., nan, nan])
```

例如，在默认线性插值法下分别进行 backward 和双向限制插值，同时限制最大连续条数为 1：

```
In [28]: res = s.interpolate(limit_direction='backward', limit=1)

In [29]: res.values
Out[29]: array([ nan, 1.  , 1.  ,  nan,  nan, 1.75, 2.  ,  nan,  nan])

In [30]: res = s.interpolate(limit_direction='both', limit=1)

In [31]: res.values
Out[31]: array([ nan, 1.  , 1.  , 1.25,  nan, 1.75, 2.  , 2.  ,  nan])
```

第二种常见的插值是最近邻插补，即缺失值的元素和离它最近的非缺失值元素一样：

```
In [32]: s.interpolate('nearest').values
Out[32]: array([nan, nan,  1.,  1.,  1.,  2.,  2., nan, nan])
```

最后来介绍索引插值，即根据索引大小进行线性插值。例如，构造不等间距的索引进行演示：

```
In [33]: s = pd.Series([0,np.nan,10],index=[0,1,10])

In [34]: s
Out[34]:
```

(continues on next page)

(continued from previous page)

```
0      0.0
1      NaN
10    10.0
dtype: float64

In [35]: s.interpolate() # 默认的线性插值，等价于计算中点的值
Out[35]:
0      0.0
1      5.0
10    10.0
dtype: float64

In [36]: 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
Out[36]:
0      0.0
1      1.0
10    10.0
dtype: float64
```

同时，这种方法对于时间戳索引也是可以使用的，有关时间序列的其他话题会在第十章进行讨论，这里举一个简单的例子：

```
In [37]: s = pd.Series([0,np.nan,10],
   ....:               index=pd.to_datetime(['20200101',
   ....:                                     '20200102',
   ....:                                     '20200111']))
   ....:

In [38]: s
Out[38]:
2020-01-01     0.0
2020-01-02     NaN
2020-01-11    10.0
dtype: float64

In [39]: s.interpolate()
Out[39]:
2020-01-01     0.0
2020-01-02     5.0
2020-01-11    10.0
dtype: float64

In [40]: s.interpolate(method='index')
```

(continues on next page)

(continued from previous page)

```
Out[40]:
2020-01-01     0.0
2020-01-02     1.0
2020-01-11    10.0
dtype: float64
```

关于 polynomial 和 spline 插值的注意事项

在 interpolate 中如果选用 polynomial 的插值方法，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，因此其实是样条插值而不是类似于 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 而不是普通的样条插值。这一部分的文档描述比较混乱，而且这种参数的设计也是不合理的，当使用这两类插值方法时，用户一定要小心谨慎地根据自己的实际需求选取恰当的插值方法。

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：

```
In [41]: None == None
Out[41]: True

In [42]: None == False
Out[42]: False

In [43]: None == []
Out[43]: False

In [44]: None == ''
Out[44]: False
```

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：

```
In [45]: np.nan == np.nan
Out[45]: False
```

(continues on next page)

(continued from previous page)

```
In [46]: np.nan == None
Out[46]: False

In [47]: np.nan == False
Out[47]: False
```

在 pandas 中无论是 None 还是 np.nan ，存入序列或表中都会以 np.nan 的形式存储：

```
In [48]: pd.Series([1, None])
Out[48]:
0    1.0
1    NaN
dtype: float64

In [49]: pd.Series([1, np.nan])
Out[49]:
0    1.0
1    NaN
dtype: float64
```

值得注意的是，虽然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 ：

```
In [50]: s1 = pd.Series([1, np.nan])

In [51]: s2 = pd.Series([1, 2])

In [52]: s3 = pd.Series([1, np.nan])

In [53]: s1 == 1
Out[53]:
0     True
1    False
dtype: bool

In [54]: s1.equals(s2)
Out[54]: False

In [55]: s1.equals(s3)
Out[55]: True
```

在时间序列的对象中，pandas 利用 pd.NaT 来指代缺失值，它的作用和 np.nan 是一致的（时间序列的对象和构造将在第十章讨论）：

```
In [56]: pd.to_timedelta(['30s', np.nan]) # Timedelta 中的 NaT
Out[56]: TimedeltaIndex(['0 days 00:00:30', NaT], dtype='timedelta64[ns]', freq=None)

In [57]: pd.to_datetime(['20200101', np.nan]) # Datetime 中的 NaT
Out[57]: DatetimeIndex(['2020-01-01', 'NaT'], dtype='datetime64[ns]', freq=None)
```

那么为什么要引入 pd.NaT 来表示时间对象中的缺失呢？仍然以 np.nan 的形式存放会有什么问题？在 pandas 中可以看到 object 类型的对象，而 object 是一种混杂对象类型，如果出现了多个类型的元素同时存储在 Series 中，它的类型就会变成 object 。例如，同时存放整数和字符串的列表：

```
In [58]: pd.Series([1, 'two'])
Out[58]:
0      1
1    two
dtype: object
```

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，这显然是不希望看到的。

```
In [59]: type(np.nan)
Out[59]: float
```

同时，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 ：

```
In [60]: pd.Series([1, np.nan]).dtype
Out[60]: dtype('float64')

In [61]: pd.Series([True, False, np.nan]).dtype
Out[61]: dtype('O')
```

因此，在进入 1.0.0 版本后，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 。

7.3.2 Nullable 类型的性质

从字面意义上看 Nullable 就是可空的，言下之意就是序列类型不受缺失值的影响。例如，在上述三个 Nullable 类型中存储缺失值，都会转为 pandas 内置的 pd.NA：

```
In [62]: pd.Series([np.nan, 1], dtype = 'Int64') # "i" 是大写的
Out[62]:
0    <NA>
1       1
```

(continues on next page)

(continued from previous page)

```
dtype: Int64

In [63]: pd.Series([np.nan, True], dtype = 'boolean')
Out[63]:
0    <NA>
1    True
dtype: boolean

In [64]: pd.Series([np.nan, 'my_str'], dtype = 'string')
Out[64]:
0      <NA>
1    my_str
dtype: string
```

在 Int 的序列中，返回的结果会尽可能地成为 Nullable 的类型：

```
In [65]: pd.Series([np.nan, 0], dtype = 'Int64') + 1
Out[65]:
0    <NA>
1       1
dtype: Int64

In [66]: pd.Series([np.nan, 0], dtype = 'Int64') == 0
Out[66]:
0    <NA>
1    True
dtype: boolean

In [67]: pd.Series([np.nan, 0], dtype = 'Int64') * 0.5 # 只能是浮点
Out[67]:
0    NaN
1    0.0
dtype: float64
```

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：

第一点是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 ：

```
In [68]: s = pd.Series(['a', 'b'])

In [69]: s_bool = pd.Series([True, np.nan])

In [70]: s_boolean = pd.Series([True, np.nan]).astype('boolean')
```

(continues on next page)

(continued from previous page)

```
# s[s_bool] # 报错
In [71]: s[s_boolean]
Out[71]:
0    a
dtype: object
```

第二点是在进行逻辑运算时，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。那什么叫能否确定唯一结果呢？举个简单例子：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 。

```
In [72]: s_boolean & True
Out[72]:
0    True
1    <NA>
dtype: boolean

In [73]: s_boolean | True
Out[73]:
0    True
1    True
dtype: boolean

In [74]: ~s_boolean # 取反操作同样是无法唯一地判断缺失结果
Out[74]:
0    False
1     <NA>
dtype: boolean
```

关于 string 类型的具体性质将在下一章文本数据中进行讨论。

一般在实际数据处理时，可以在数据集读入后，先通过 convert_dtypes 转为 Nullable 类型：

```
In [75]: df = pd.read_csv('data/learn_pandas.csv')

In [76]: df = df.convert_dtypes()

In [77]: df.dtypes
Out[77]:
School          string
Grade           string
Name            string
Gender          string
Height         float64
```

(continues on next page)

(continued from previous page)

```
Weight            Int64
Transfer         string
Test_Number       Int64
Test_Date        string
Time_Record      string
dtype: object
```

7.3.3 缺失数据的计算和分组

当调用函数 sum, prob 使用加法和乘法的时候，缺失数据等价于被分别视作 0 和 1，即不改变原来的计算结果：

```
In [78]: s = pd.Series([2,3,np.nan,4,5])

In [79]: s.sum()
Out[79]: 14.0

In [80]: s.prod()
Out[80]: 120.0
```

当使用累计函数时，会自动跳过缺失值所处的位置：

```
In [81]: s.cumsum()
Out[81]:
0     2.0
1     5.0
2     NaN
3     9.0
4    14.0
dtype: float64
```

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 ：

```
In [82]: np.nan == 0
Out[82]: False

In [83]: pd.NA == 0
Out[83]: <NA>

In [84]: np.nan > 0
Out[84]: False
```

(continues on next page)

(continued from previous page)

```
In [85]: pd.NA > 0
Out[85]: <NA>

In [86]: np.nan + 1
Out[86]: nan

In [87]: np.log(np.nan)
Out[87]: nan

In [88]: np.add(np.nan, 1)
Out[88]: nan

In [89]: np.nan ** 0
Out[89]: 1.0

In [90]: pd.NA ** 0
Out[90]: 1

In [91]: 1 ** np.nan
Out[91]: 1.0

In [92]: 1 ** pd.NA
Out[92]: 1
```

另外需要注意的是，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被设为 0% 的变化率：

```
In [93]: s.diff()
Out[93]:
0    NaN
1    1.0
2    NaN
3    NaN
4    1.0
dtype: float64

In [94]: s.pct_change()
Out[94]:
0         NaN
1    0.500000
2    0.000000
3    0.333333
```

(continues on next page)

(continued from previous page)

```
4    0.250000
dtype: float64
```

对于一些函数而言，缺失可以作为一个类别处理，例如在 groupby, get_dummies 中可以设置相应的参数来进行增加缺失类别：

```
In [95]: df_nan = pd.DataFrame({'category':['a','a','b',np.nan,np.nan],
   ....:                        'value':[1,3,5,7,9]})
   ....:

In [96]: df_nan
Out[96]:
  category  value
0        a      1
1        a      3
2        b      5
3      NaN      7
4      NaN      9

In [97]: df_nan.groupby('category',
   ....:                 dropna=False)['value'].mean() # pandas 版本大于 1.1.0
   ....:
Out[97]:
category
a      2
b      5
NaN    8
Name: value, dtype: int64

In [98]: pd.get_dummies(df_nan.category, dummy_na=True)
Out[98]:
   a  b  NaN
0  1  0    0
1  1  0    0
2  0  1    0
3  0  0    1
4  0  0    1
```

7.4 练习

7.4.1 Ex1：缺失值与类别的相关性检验

在数据处理中，含有过多缺失值的列往往会被删除，除非缺失情况与标签强相关。下面有一份关于二分类问题的数据集，其中 X_1, X_2 为特征变量，y 为二分类标签。

```
In [99]: df = pd.read_csv('data/missing_chi.csv')

In [100]: df.head()
Out[100]:
    X_1  X_2  y
0   NaN  NaN  0
1   NaN  NaN  0
2   NaN  NaN  0
3  43.0  NaN  0
4   NaN  NaN  0

In [101]: df.isna().sum()/df.shape[0]
Out[101]:
X_1    0.855
X_2    0.894
y      0.000
dtype: float64

In [102]: df.y.value_counts(normalize=True)
Out[102]:
0    0.918
1    0.082
Name: y, dtype: float64
```

事实上，有时缺失值出现或者不出现本身就是一种特征，并且在一些场合下可能与标签的正负是相关的。关于缺失出现与否和标签的正负性，在统计学中可以利用卡方检验来断言它们是否存在相关性。按照特征缺失的正例、特征缺失的负例、特征不缺失的正例、特征不缺失的负例，可以分为四种情况，设它们分别对应的样例数为 $n_{11}, n_{10}, n_{01}, n_{00}$ 。假若它们是不相关的，那么特征缺失中正例的理论值，就应该接近于特征缺失总数 × 总体正例的比例，即：

$$E_{11} = n_{11} \approx (n_{11} + n_{10}) \times \frac{n_{11} + n_{01}}{n_{11} + n_{10} + n_{01} + n_{00}} = F_{11}$$

其他的三种情况同理。现将实际值和理论值分别记作 E_{ij}, F_{ij} ，那么希望下面的统计量越小越好，即代表实际值接近不相关情况的理论值：

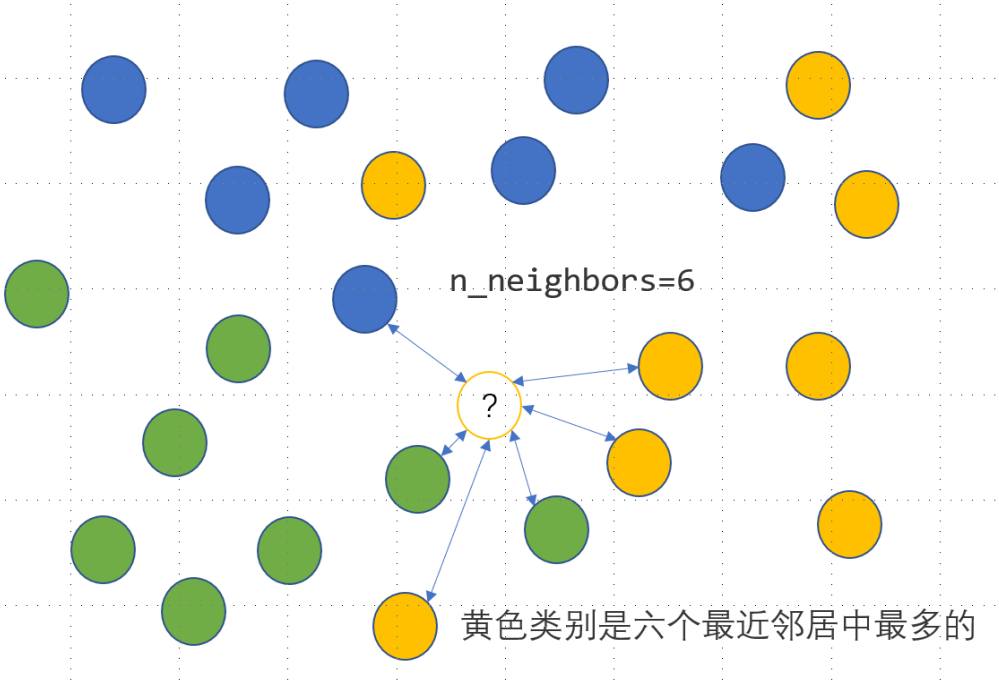
$$S = \sum_{i \in \{0,1\}} \sum_{j \in \{0,1\}} \frac{(E_{ij} - F_{ij})^2}{F_{ij}}$$

可以证明上面的统计量近似服从自由度为 1 的卡方分布，即 $S \sim \chi^2(1)$ 。因此，可通过计算 $P(\chi^2(1) > S)$ 的概率来进行相关性的判别，一般认为当此概率小于 0.05 时缺失情况与标签正负存在相关关系，即不相关条件下的理论值与实际值相差较大。

上面所说的概率即为统计学上关于 2×2 列联表检验问题的 p 值，它可以通过 scipy.stats.chi2(S, 1) 得到。请根据上面的材料，分别对 X_1, X_2 列进行检验。

7.4.2 Ex2：用回归模型解决分类问题

KNN 是一种监督式学习模型，既可以解决回归问题，又可以解决分类问题。对于分类变量，利用 KNN 分类模型可以实现其缺失值的插补，思路是度量缺失样本的特征与所有其他样本特征的距离，当给定了模型参数 n_neighbors=n 时，计算离该样本距离最近的 n 个样本点中最多的那个类别，并把这个类别作为该样本的缺失预测类别，具体如下图所示，未知的类别被预测为黄色：



上面有色点的特征数据提供如下：

```
In [103]: df = pd.read_excel('data/color.xlsx')

In [104]: df.head(3)
Out[104]:
    X1   X2 Color
0 -2.5  2.8  Blue
1 -1.5  1.8  Blue
2 -0.8  2.8  Blue
```

已知待预测的样本点为 $X_1 = 0.8, X_2 = -0.2$ ，那么预测类别可以如下写出：

```
In [105]: from sklearn.neighbors import KNeighborsClassifier

In [106]: clf = KNeighborsClassifier(n_neighbors=6)

In [107]: clf.fit(df.iloc[:,:2], df.Color)
Out[107]: KNeighborsClassifier(n_neighbors=6)

In [108]: clf.predict([[0.8, -0.2]])
Out[108]: array(['Yellow'], dtype=object)
```

1. 对于回归问题而言，需要得到的是一个具体的数值，因此预测值由最近的 n 个样本对应的平均值获得。请把上面的这个分类问题转化为回归问题，仅使用 KNeighborsRegressor 来完成上述的 KNeighborsClassifier 功能。

2. 请根据第 1 问中的方法，对 audit 数据集中的 Employment 变量进行缺失值插补。

```
In [109]: df = pd.read_csv('data/audit.csv')

In [110]: df.head(3)
Out[110]:
        ID  Age Employment    Marital     Income  Gender  Hours
0  1004641   38    Private  Unmarried   81838.00  Female     72
1  1010229   35    Private     Absent   72099.00    Male     30
2  1024587   32    Private   Divorced  154676.74    Male     40
```

第 8 章　文本数据

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

In [2]: import pandas as pd
```

8.1　str 对象

8.1.1　str 对象的设计意图

str 对象是定义在 Index 或 Series 上的属性，专门用于逐元素处理文本内容，其内部定义了大量方法，因此对一个序列进行文本处理，首先需要获取其 str 对象。在 Python 标准库中也有 str 模块，为了使用上的便利，有许多函数的用法 pandas 照搬了它的设计，例如字母转为大写的操作：

```
In [3]: var = 'abcd'

In [4]: str.upper(var) # Python 内置 str 模块
Out[4]: 'ABCD'

In [5]: s = pd.Series(['abcd', 'efg', 'hi'])

In [6]: s.str
Out[6]: <pandas.core.strings.StringMethods at 0x2266d50a4c8>

In [7]: s.str.upper() # pandas 中 str 对象上的 upper 方法
Out[7]:
0    ABCD
1     EFG
2      HI
dtype: object
```

根据文档 API 材料，在 pandas 的 50 个 str 对象方法中，有 31 个是和标准库中的 str 模块方法同名且功能一致，这为批量处理序列提供了有力的工具。

8.1.2 [] 索引器

对于 str 对象而言，可理解为其对字符串进行了序列化的操作，例如在一般的字符串中，通过 [] 可以取出某个位置的元素：

```
In [8]: var[0]
Out[8]: 'a'
```

同时也能通过切片得到子串：

```
In [9]: var[-1: 0: -2]
Out[9]: 'db'
```

通过对 str 对象使用 [] 索引器，可以完成完全一致的功能，并且如果超出范围则返回缺失值：

```
In [10]: s.str[0]
Out[10]:
0    a
1    e
2    h
dtype: object

In [11]: s.str[-1: 0: -2]
Out[11]:
0    db
1     g
2     i
dtype: object

In [12]: s.str[2]
Out[12]:
0      c
1      g
2    NaN
dtype: object
```

8.1.3 string 类型

在上一章提到，从 pandas 的 1.0.0 版本开始，引入了 string 类型，其引入的动机在于：原来所有的字符串类型都会以 object 类型的 Series 进行存储，但 object 类型只应当存储混合类型，例如同时存储浮点、字符串、字典、列表、自定义类型等，因此字符串有必要同数值型或 category 一样，具有自己的数据存放类型，从而引入了 string 类型。

总体上说，绝大多数对于 object 和 string 类型的序列使用 str 对象方法产生的结果是一致，但是在下面提到的两点上有较大差异：

首先，应当尽量保证每一个序列中的值都是字符串的情况下才使用 str 属性，但这并不是必须的，其必要条件是序列中至少有一个可迭代（Iterable）对象，包括但不限于字符串、字典、列表。对于一个可迭代对象，string 类型的 str 对象和 object 类型的 str 对象返回结果可能是不同的。

```
In [13]: s = pd.Series([{1: 'temp_1', 2: 'temp_2'}, ['a', 'b'], 0.5, 'my_string'])

In [14]: s.str[1]
Out[14]:
0    temp_1
1         b
2       NaN
3         y
dtype: object

In [15]: s.astype('string').str[1]
Out[15]:
0    1
1    '
2    .
3    y
dtype: string
```

除了最后一个字符串元素，前三个元素返回的值都不同，其原因在于当序列类型为 object 时，是对于每一个元素进行 [] 索引，因此对于字典而言，返回 temp_1 字符串，对于列表则返回第二个值，而第三个为不可迭代对象，返回缺失值，第四个是对字符串进行 [] 索引。而 string 类型的 str 对象先把整个元素转为字面意义的字符串，例如对于列表而言，第一个元素即“{ “，而对于最后一个字符串元素而言，恰好转化前后的表示方法一致，因此结果和 object 类型一致。

除了对于某些对象的 str 序列化方法不同之外，两者另外的一个差别在于，string 类型是 Nullable 类型，但 object 不是。这意味着 string 类型的序列，如果调用的 str 方法返回值为整数 Series 和布尔 Series 时，其分别对应的 dtype 是 Int 和 boolean 的 Nullable 类型，而 object 类型则会分别返回 int/float 和 bool/object，取决于缺失值的存在与否。同时，字符串的比较操作，也具有相似的特性，string 返回 Nullable 类型，但 object 不会。

```
In [16]: s = pd.Series(['a'])

In [17]: s.str.len()
Out[17]:
0    1
dtype: int64

In [18]: s.astype('string').str.len()
```

(continues on next page)

(continued from previous page)

```
Out[18]:
0    1
dtype: Int64

In [19]: s == 'a'
Out[19]:
0    True
dtype: bool

In [20]: s.astype('string') == 'a'
Out[20]:
0    True
dtype: boolean

In [21]: s = pd.Series(['a', np.nan]) # 带有缺失值

In [22]: s.str.len()
Out[22]:
0    1.0
1    NaN
dtype: float64

In [23]: s.astype('string').str.len()
Out[23]:
0       1
1    <NA>
dtype: Int64

In [24]: s == 'a'
Out[24]:
0     True
1    False
dtype: bool

In [25]: s.astype('string') == 'a'
Out[25]:
0    True
1    <NA>
dtype: boolean
```

最后需要注意的是，对于全体元素为数值类型的序列，即使其类型为 object 或者 category 也不允许直接使用 str 属性。如果需要把数字当成 string 类型处理，可以使用 astype 强制转换为 string 类型的 Series：

```
In [26]: s = pd.Series([12, 345, 6789])

In [27]: s.astype('string').str[1]
Out[27]:
0    2
1    4
2    7
dtype: string
```

8.2 正则表达式基础

这一节的两个表格来自于 learn-regex-zh 这个关于正则表达式项目，其使用 MIT 开源许可协议。这里只是介绍正则表达式的基本用法，需要系统学习的读者可参考 正则表达式必知必会 一书。

8.2.1 一般字符的匹配

正则表达式是一种按照某种正则模式，从左到右匹配字符串中内容的一种工具。对于一般的字符而言，它可以找到其所在的位置，这里为了演示便利，使用了 python 中 re 模块的 findall 函数来匹配所有出现过但不重叠的模式，第一个参数是正则表达式，第二个参数是待匹配的字符串。例如，在下面的字符串中找出 apple：

```
In [28]: import re

In [29]: re.findall('Apple', 'Apple! This Is an Apple!')
Out[29]: ['Apple', 'Apple']
```

8.2.2 元字符基础

元字符	描述
.	匹配除换行符以外的任意字符。
[ ]	字符类，匹配方括号中包含的任意字符。
[^]	否定字符类，匹配方括号中不包含的任意字符
*	匹配前面的子表达式零次或多次
+	匹配前面的子表达式一次或多次
?	匹配前面的子表达式零次或一次
{n,m}	花括号，匹配前面字符至少 n 次，但是不超过 m 次。
(xyz)	字符组，按照确切的顺序匹配字符 xyz。
|	分支结构，匹配符号之前的字符或后面的字符。
\	转义符，它可以还原元字符原来的含义
^	匹配行的开始
$	匹配行的结束

```
In [30]: re.findall('.', 'abc')
Out[30]: ['a', 'b', 'c']

In [31]: re.findall('[ac]', 'abc')
Out[31]: ['a', 'c']

In [32]: re.findall('[^ac]', 'abc')
Out[32]: ['b']

In [33]: re.findall('[ab]{2}', 'aaaabbbb') # {n} 指匹配 n 次
Out[33]: ['aa', 'aa', 'bb', 'bb']

In [34]: re.findall('aaa|bbb', 'aaaabbbb')
Out[34]: ['aaa', 'bbb']

In [35]: re.findall('a\\?|a\*', 'aa?a*a')
Out[35]: ['a?', 'a*']

In [36]: re.findall('a?.', 'abaacadaae')
Out[36]: ['ab', 'aa', 'c', 'ad', 'aa', 'e']
```

8.2.3 简写字符集

此外，正则表达式中还有一类简写字符集，其等价于一组字符的集合：

简写	描述
\w	匹配所有字母、数字、下划线: [a-zA-Z0-9_]
\W	匹配非字母和数字的字符: [^\w]
\d	匹配数字: [0-9]
\D	匹配非数字: [^\d]
\s	匹配空格符: [\t\n\f\r\p{Z}]
\S	匹配非空格符: [^\s]
\B	匹配一组非空字符开头或结尾的位置，不代表具体字符

```
In [37]: re.findall('.s', 'Apple! This Is an Apple!')
Out[37]: ['is', 'Is']

In [38]: re.findall('\w{2}', '09 8? 7w c_ 9q p@')
Out[38]: ['09', '7w', 'c_', '9q']

In [39]: re.findall('\w\W\B', '09 8? 7w c_ 9q p@')
Out[39]: ['8?', 'p@']

In [40]: re.findall('.\s.', 'Constant dropping wears the stone.')
Out[40]: ['t d', 'g w', 's t', 'e s']

In [41]: re.findall('上海市 (.{2,3} 区)(.{2,3} 路)(\d+ 号)',
   ....:            '上海市黄浦区方浜中路 249 号 上海市宝山区密山路 5 号')
   ....:
Out[41]: [('黄浦区', '方浜中路', '249 号'), ('宝山区', '密山路', '5 号')]
```

8.3 文本处理的五类操作

8.3.1 拆分

str.split 能够把字符串的列进行拆分，其中第一个参数为正则表达式，可选参数包括从左到右的最大拆分次数 n ，是否展开为多个列 expand 。

```
In [42]: s = pd.Series(['上海市黄浦区方浜中路 249 号',
   ....:                '上海市宝山区密山路 5 号'])
   ....:
```

(continues on next page)

(continued from previous page)

```
In [43]: s.str.split('[市区路]')
Out[43]:
0    [上海, 黄浦, 方浜中, 249 号]
1       [上海, 宝山, 密山, 5 号]
dtype: object

In [44]: s.str.split('[市区路]', n=2, expand=True)
Out[44]:
    0   1         2
0  上海  黄浦  方浜中路 249 号
1  上海  宝山     密山路 5 号
```

与其类似的函数是 str.rsplit ，其区别在于使用 n 参数的时候是从右到左限制最大拆分次数。但是当前版本下 rsplit 因为 bug 而无法使用正则表达式进行分割：

```
In [45]: s.str.rsplit('[市区路]', n=2, expand=True)
Out[45]:
                0
0  上海市黄浦区方浜中路 249 号
1     上海市宝山区密山路 5 号
```

8.3.2 合并

关于合并一共有两个函数，分别是 str.join 和 str.cat 。str.join 表示用某个连接符把 Series 中的字符串列表连接起来，如果列表中出现了字符串元素则返回缺失值：

```
In [46]: s = pd.Series([['a','b'], [1, 'a'], [['a', 'b'], 'c']])

In [47]: s.str.join('-')
Out[47]:
0    a-b
1    NaN
2    NaN
dtype: object
```

str.cat 用于合并两个序列，主要参数为连接符 sep 、连接形式 join` 以及缺失值替代符号 ``na_rep ，其中连接形式默认为以索引为键的左连接。

```
In [48]: s1 = pd.Series(['a','b'])

In [49]: s2 = pd.Series(['cat','dog'])
```

(continues on next page)

(continued from previous page)

```
In [50]: s1.str.cat(s2,sep='-')
Out[50]:
0    a-cat
1    b-dog
dtype: object

In [51]: s2.index = [1, 2]

In [52]: s1.str.cat(s2, sep='-', na_rep='?', join='outer')
Out[52]:
0      a-?
1    b-cat
2    ?-dog
dtype: object
```

8.3.3 匹配

str.contains 返回了每个字符串是否包含正则模式的布尔序列：

```
In [53]: s = pd.Series(['my cat', 'he is fat', 'railway station'])

In [54]: s.str.contains('\s\wat')
Out[54]:
0     True
1     True
2    False
dtype: bool
```

str.startswith 和 str.endswith 返回了每个字符串以给定模式为开始和结束的布尔序列，它们都不支持正则表达式：

```
In [55]: s.str.startswith('my')
Out[55]:
0     True
1    False
2    False
dtype: bool

In [56]: s.str.endswith('t')
Out[56]:
0     True
```

(continues on next page)

(continued from previous page)

```
1     True
2    False
dtype: bool
```

如果需要用正则表达式来检测开始或结束字符串的模式，可以使用 str.match ，其返回了每个字符串起始处是否符合给定正则模式的布尔序列：

```
In [57]: s.str.match('m|h')
Out[57]:
0     True
1     True
2    False
dtype: bool

In [58]: s.str[::-1].str.match('ta[f|g]|n') # 反转后匹配
Out[58]:
0    False
1     True
2     True
dtype: bool
```

当然，这些也能通过在 str.contains 的正则中使用 ^ 和 $ 来实现：

```
In [59]: s.str.contains('^[m|h]')
Out[59]:
0     True
1     True
2    False
dtype: bool

In [60]: s.str.contains('[f|g]at|n$')
Out[60]:
0    False
1     True
2     True
dtype: bool
```

除了上述返回值为布尔的匹配之外，还有一种返回索引的匹配函数，即 str.find 与 str.rfind ，其分别返回从左到右和从右到左第一次匹配的位置的索引，未找到则返回-1。需要注意的是这两个函数不支持正则匹配，只能用于字符子串的匹配：

```
In [61]: s = pd.Series(['This is an apple. That is not an apple.'])
```

(continues on next page)

(continued from previous page)

```
In [62]: s.str.find('apple')
Out[62]:
0    11
dtype: int64

In [63]: s.str.rfind('apple')
Out[63]:
0    33
dtype: int64
```

8.3.4 替换

str.replace 和 replace 并不是一个函数，在使用字符串替换时应当使用前者。

```
In [64]: s = pd.Series(['a_1_b','c_?'])

In [65]: s.str.replace('\d|\?', 'new')
Out[65]:
0    a_new_b
1      c_new
dtype: object
```

当需要对不同部分进行有差别的替换时，可以利用 子组 的方法，并且此时可以通过传入自定义的替换函数来分别进行处理，注意 group(k) 代表匹配到的第 k 个子组（圆括号之间的内容）：

```
In [66]: s = pd.Series(['上海市黄浦区方浜中路 249 号',
   ....:                '上海市宝山区密山路 5 号',
   ....:                '北京市昌平区北农路 2 号'])
   ....:

In [67]: pat = '(\w+ 市)(\w+ 区)(\w+ 路)(\d+ 号)'

In [68]: city = {'上海市': 'Shanghai', '北京市': 'Beijing'}

In [69]: district = {'昌平区': 'CP District',
   ....:             '黄浦区': 'HP District',
   ....:             '宝山区': 'BS District'}
   ....:

In [70]: road = {'方浜中路': 'Mid Fangbin Road',
   ....:         '密山路': 'Mishan Road',
   ....:         '北农路': 'Beinong Road'}
```

(continues on next page)

(continued from previous page)

```
   ....:

In [71]: def my_func(m):
   ....:     str_city = city[m.group(1)]
   ....:     str_district = district[m.group(2)]
   ....:     str_road = road[m.group(3)]
   ....:     str_no = 'No. ' + m.group(4)[:-1]
   ....:     return ' '.join([str_city,
   ....:                      str_district,
   ....:                      str_road,
   ....:                      str_no])
   ....:

In [72]: s.str.replace(pat, my_func)
Out[72]:
0    Shanghai HP District Mid Fangbin Road No. 249
1           Shanghai BS District Mishan Road No. 5
2           Beijing CP District Beinong Road No. 2
dtype: object
```

这里的数字标识并不直观，可以使用 命名子组 更加清晰地写出子组代表的含义：

```
In [73]: pat = '(?P< 市名>\w+ 市)(?P< 区名>\w+ 区)(?P< 路名>\w+ 路)(?P< 编号>\d+ 号)'

In [74]: def my_func(m):
   ....:     str_city = city[m.group('市名')]
   ....:     str_district = district[m.group('区名')]
   ....:     str_road = road[m.group('路名')]
   ....:     str_no = 'No. ' + m.group('编号')[:-1]
   ....:     return ' '.join([str_city,
   ....:                      str_district,
   ....:                      str_road,
   ....:                      str_no])
   ....:

In [75]: s.str.replace(pat, my_func)
Out[75]:
0    Shanghai HP District Mid Fangbin Road No. 249
1           Shanghai BS District Mishan Road No. 5
2           Beijing CP District Beinong Road No. 2
dtype: object
```

这里虽然看起来有些繁杂，但是实际数据处理中对应的替换，一般都会通过代码来获取数据从而构造字典映射，在具体写法上会简洁的多。

8.3.5 提取

提取既可以认为是一种返回具体元素（而不是布尔值或元素对应的索引位置）的匹配操作，也可以认为是一种特殊的拆分操作。前面提到的 str.split 例子中会把分隔符去除，这并不是用户想要的效果，这时候就可以用 str.extract 进行提取：

```
In [76]: pat = '(\w+ 市)(\w+ 区)(\w+ 路)(\d+ 号)'

In [77]: s.str.extract(pat)
Out[77]:
     0    1     2     3
0  上海市  黄浦区  方浜中路  249 号
1  上海市  宝山区   密山路    5 号
2  北京市  昌平区   北农路    2 号
```

通过子组的命名，可以直接对新生成 DataFrame 的列命名：

```
In [78]: pat = '(?P< 市名>\w+ 市)(?P< 区名>\w+ 区)(?P< 路名>\w+ 路)(?P< 编号>\d+ 号)'

In [79]: s.str.extract(pat)
Out[79]:
    市名   区名    路名    编号
0  上海市  黄浦区  方浜中路  249 号
1  上海市  宝山区   密山路    5 号
2  北京市  昌平区   北农路    2 号
```

str.extractall 不同于 str.extract 只匹配一次，它会把所有符合条件的模式全部匹配出来，如果存在多个结果，则以多级索引的方式存储：

```
In [80]: s = pd.Series(['A135T15,A26S5','B674S2,B25T6'], index = ['my_A','my_B'])

In [81]: pat = '[A|B](\d+)[T|S](\d+)'

In [82]: s.str.extractall(pat)
Out[82]:
              0   1
     match
my_A 0      135  15
     1       26   5
my_B 0      674   2
     1       25   6

In [83]: pat_with_name = '[A|B](?P<name1>\d+)[T|S](?P<name2>\d+)'
```

(continues on next page)

(continued from previous page)

```
In [84]: s.str.extractall(pat_with_name)
Out[84]:
           name1 name2
     match
my_A 0       135    15
     1        26     5
my_B 0       674     2
     1        25     6
```

str.findall 的功能类似于 str.extractall ，区别在于前者把结果存入列表中，而后者处理为多级索引，每个行只对应一组匹配，而不是把所有匹配组合构成列表。

```
In [85]: s.str.findall(pat)
Out[85]:
my_A    [(135, 15), (26, 5)]
my_B     [(674, 2), (25, 6)]
dtype: object
```

8.4 常用字符串函数

除了上述介绍的五类字符串操作有关的函数之外，str 对象上还定义了一些实用的其他方法，在此进行介绍：

8.4.1 字母型函数

upper, lower, title, capitalize, swapcase 这五个函数主要用于字母的大小写转化，从下面的例子中就容易领会其功能：

```
In [86]: s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])

In [87]: s.str.upper()
Out[87]:
0                 LOWER
1             CAPITALS
2    THIS IS A SENTENCE
3             SWAPCASE
dtype: object

In [88]: s.str.lower()
Out[88]:
0                 lower
```

(continues on next page)

(continued from previous page)

```
1              capitals
2    this is a sentence
3              swapcase
dtype: object

In [89]: s.str.title()
Out[89]:
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object

In [90]: s.str.capitalize()
Out[90]:
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object

In [91]: s.str.swapcase()
Out[91]:
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

8.4.2 数值型函数

这里着重需要介绍的是 pd.to_numeric 方法，它虽然不是 str 对象上的方法，但是能够对字符格式的数值进行快速转换和筛选。其主要参数包括 errors 和 downcast 分别代表了非数值的处理模式和转换类型。其中，对于不能转换为数值的有三种 errors 选项，raise, coerce, ignore 分别表示直接报错、设为缺失以及保持原来的字符串。

```
In [92]: s = pd.Series(['1', '2.2', '2e', '??', '-2.1', '0'])

In [93]: pd.to_numeric(s, errors='ignore')
Out[93]:
0       1
1     2.2
```

(continues on next page)

(continued from previous page)

```
2      2e
3      ??
4    -2.1
5       0
dtype: object

In [94]: pd.to_numeric(s, errors='coerce')
Out[94]:
0    1.0
1    2.2
2    NaN
3    NaN
4   -2.1
5    0.0
dtype: float64
```

在数据清洗时，可以利用 coerce 的设定，快速查看非数值型的行：

```
In [95]: s[pd.to_numeric(s, errors='coerce').isna()]
Out[95]:
2    2e
3    ??
dtype: object
```

8.4.3 统计型函数

count 和 len 的作用分别是返回出现正则模式的次数和字符串的长度：

```
In [96]: s = pd.Series(['cat rat fat at', 'get feed sheet heat'])

In [97]: s.str.count('[r|f]at|ee')
Out[97]:
0    2
1    2
dtype: int64

In [98]: s.str.len()
Out[98]:
0    14
1    19
dtype: int64
```

8.4.4 格式型函数

格式型函数主要分为两类，第一种是除空型，第二种时填充型。其中，第一类函数一共有三种，它们分别是 strip, rstrip, lstrip ，分别代表去除两侧空格、右侧空格和左侧空格。这些函数在数据清洗时是有用的，特别是列名含有非法空格的时候。

```
In [99]: my_index = pd.Index([' col1', 'col2 ', ' col3 '])

In [100]: my_index.str.strip().str.len()
Out[100]: Int64Index([4, 4, 4], dtype='int64')

In [101]: my_index.str.rstrip().str.len()
Out[101]: Int64Index([5, 4, 5], dtype='int64')

In [102]: my_index.str.lstrip().str.len()
Out[102]: Int64Index([4, 5, 5], dtype='int64')
```

对于填充型函数而言，pad 是最灵活的，它可以选定字符串长度、填充的方向和填充内容：

```
In [103]: s = pd.Series(['a','b','c'])

In [104]: s.str.pad(5,'left','*')
Out[104]:
0    ****a
1    ****b
2    ****c
dtype: object

In [105]: s.str.pad(5,'right','*')
Out[105]:
0    a****
1    b****
2    c****
dtype: object

In [106]: s.str.pad(5,'both','*')
Out[106]:
0    **a**
1    **b**
2    **c**
dtype: object
```

上述的三种情况可以分别用 rjust, ljust, center 来等效完成，需要注意 ljust 是指右侧填充而不是左侧填充：

```
In [107]: s.str.rjust(5, '*')
Out[107]:
0    ****a
1    ****b
2    ****c
dtype: object

In [108]: s.str.ljust(5, '*')
Out[108]:
0    a****
1    b****
2    c****
dtype: object

In [109]: s.str.center(5, '*')
Out[109]:
0    **a**
1    **b**
2    **c**
dtype: object
```

在读取 excel 文件时，经常会出现数字前补 0 的需求，例如证券代码读入的时候会把”000007”作为数值 7 来处理，pandas 中除了可以使用上面的左侧填充函数进行操作之外，还可用 zfill 来实现。

```
In [110]: s = pd.Series([7, 155, 303000]).astype('string')

In [111]: s.str.pad(6,'left','0')
Out[111]:
0    000007
1    000155
2    303000
dtype: string

In [112]: s.str.rjust(6,'0')
Out[112]:
0    000007
1    000155
2    303000
dtype: string

In [113]: s.str.zfill(6)
Out[113]:
0    000007
1    000155
```

(continues on next page)

(continued from previous page)

```
2    303000
dtype: string
```

8.5 练习

8.5.1 Ex1：房屋信息数据集

现有一份房屋信息数据集如下：

```
In [114]: df = pd.read_excel('data/house_info.xls', usecols=[
   .....:                    'floor','year','area','price'])
   .....:

In [115]: df.head(3)
Out[115]:
      floor    year    area price
0   高层（共 6 层）  1986 年建  58.23 ㎡  155 万
1  中层（共 20 层）  2020 年建     88 ㎡  155 万
2  低层（共 28 层）  2010 年建  89.33 ㎡  365 万
```

1. 将 year 列改为整数年份存储。
2. 将 floor 列替换为 Level, Highest 两列，其中的元素分别为 string 类型的层类别（高层、中层、低层）与整数类型的最高层数。
3. 计算房屋每平米的均价 avg_price，以 *** 元/平米 的格式存储到表中，其中 *** 为整数。

8.5.2 Ex2：《权力的游戏》剧本数据集

现有一份权力的游戏剧本数据集如下：

```
In [116]: df = pd.read_csv('data/script.csv')

In [117]: df.head(3)
Out[117]:
  Release Date    Season   Episode      Episode Title          Name                                ␣
↪           Sentence
0   2011-04-17  Season 1  Episode 1  Winter is Coming  waymar royce  What do you expect? They're␣
↪savages. One lot s...
1   2011-04-17  Season 1  Episode 1  Winter is Coming          will  I've never seen wildlings do␣
↪a thing like this...
```

(continues on next page)

(continued from previous page)

```
2   2011-04-17  Season 1  Episode 1  Winter is Coming  waymar royce                 ␣
↪How close did you get?
```

1. 计算每一个 Episode 的台词条数。
2. 以空格为单词的分割符号，请求出单句台词平均单词量最多的前五个人。
3. 若某人的台词中含有问号，那么下一个说台词的人即为回答者。若上一人台词中含有 n 个问号，则认为回答者回答了 n 个问题，请求出回答最多问题的前五个人。

第 9 章　分类数据

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

In [2]: import pandas as pd
```

9.1　cat 对象

9.1.1　cat 对象的属性

在 pandas 中提供了 category 类型，使用户能够处理分类类型的变量，将一个普通序列转换成分类变量可以使用 astype 方法。

```
In [3]: df = pd.read_csv('data/learn_pandas.csv',
   ...:                  usecols = ['Grade', 'Name', 'Gender', 'Height', 'Weight'])
   ...:

In [4]: s = df.Grade.astype('category')

In [5]: s.head()
Out[5]:
0     Freshman
1     Freshman
2       Senior
3    Sophomore
4    Sophomore
Name: Grade, dtype: category
Categories (4, object): ['Freshman', 'Junior', 'Senior', 'Sophomore']
```

在一个分类类型的 Series 中定义了 cat 对象，它和上一章中介绍的 str 对象类似，定义了一些属性和方法来进行分类类别的操作。

```
In [6]: s.cat
Out[6]: <pandas.core.arrays.categorical.CategoricalAccessor object at 0x0000022668D09A08>
```

对于一个具体的分类，有两个组成部分，其一为类别的本身，它以 Index 类型存储，其二为是否有序，它们都可以通过 cat 的属性被访问：

```
In [7]: s.cat.categories
Out[7]: Index(['Freshman', 'Junior', 'Senior', 'Sophomore'], dtype='object')

In [8]: s.cat.ordered
Out[8]: False
```

另外，每一个序列的类别会被赋予唯一的整数编号，它们的编号取决于 cat.categories 中的顺序，该属性可以通过 codes 访问：

```
In [9]: s.cat.codes.head()
Out[9]:
0    0
1    0
2    2
3    3
4    3
dtype: int8
```

9.1.2 类别的增加、删除和修改

通过 cat 对象的 categories 属性能够完成对类别的查询，那么应该如何进行“增改查删”的其他三个操作呢？

类别不得直接修改

在第三章中曾提到，索引 Index 类型是无法用 index_obj[0] = item 来修改的，而 categories 被存储在 Index 中，因此 pandas 在 cat 属性上定义了若干方法来达到相同的目的。

首先，对于类别的增加可以使用 add_categories ：

```
In [10]: s = s.cat.add_categories('Graduate') # 增加一个毕业生类别

In [11]: s.cat.categories
Out[11]: Index(['Freshman', 'Junior', 'Senior', 'Sophomore', 'Graduate'], dtype='object')
```

若要删除某一个类别可以使用 remove_categories ，同时所有原来序列中的该类会被设置为缺失。例如，删除大一的类别：

```
In [12]: s = s.cat.remove_categories('Freshman')

In [13]: s.cat.categories
Out[13]: Index(['Junior', 'Senior', 'Sophomore', 'Graduate'], dtype='object')

In [14]: s.head()
Out[14]:
0          NaN
1          NaN
2       Senior
3    Sophomore
4    Sophomore
Name: Grade, dtype: category
Categories (4, object): ['Junior', 'Senior', 'Sophomore', 'Graduate']
```

此外可以使用 set_categories 直接设置序列的新类别，原来的类别中如果存在元素不属于新类别，那么会被设置为缺失。

```
In [15]: s = s.cat.set_categories(['Sophomore','PhD']) # 新类别为大二学生和博士

In [16]: s.cat.categories
Out[16]: Index(['Sophomore', 'PhD'], dtype='object')

In [17]: s.head()
Out[17]:
0          NaN
1          NaN
2          NaN
3    Sophomore
4    Sophomore
Name: Grade, dtype: category
Categories (2, object): ['Sophomore', 'PhD']
```

如果想要删除未出现在序列中的类别，可以使用 remove_unused_categories 来实现：

```
In [18]: s = s.cat.remove_unused_categories() # 移除了未出现的博士生类别

In [19]: s.cat.categories
Out[19]: Index(['Sophomore'], dtype='object')
```

最后，“增改查删”中还剩下修改的操作，这可以通过 rename_categories 方法完成，同时需要注意的是，这个方法会对原序列的对应值也进行相应修改。例如，现在把 Sophomore 改成中文的 本科二年级学生 ：

```
In [20]: s = s.cat.rename_categories({'Sophomore':'本科二年级学生'})

In [21]: s.head()
Out[21]:
0        NaN
1        NaN
2        NaN
3    本科二年级学生
4    本科二年级学生
Name: Grade, dtype: category
Categories (1, object): ['本科二年级学生']
```

9.2 有序分类

9.2.1 序的建立

有序类别和无序类别可以通过 as_unordered 和 reorder_categories 互相转化，需要注意的是后者传入的参数必须是由当前序列的无需类别构成的列表，不能够增加新的类别，也不能缺少原来的类别，并且必须指定参数 ordered=True ，否则方法无效。例如，对年级高低进行相对大小的类别划分，然后再恢复无序状态：

```
In [22]: s = df.Grade.astype('category')

In [23]: s = s.cat.reorder_categories(['Freshman', 'Sophomore',
   ....:                               'Junior', 'Senior'],ordered=True)
   ....:

In [24]: s.head()
Out[24]:
0     Freshman
1     Freshman
2       Senior
3    Sophomore
4    Sophomore
Name: Grade, dtype: category
Categories (4, object): ['Freshman' < 'Sophomore' < 'Junior' < 'Senior']

In [25]: s.cat.as_unordered().head()
Out[25]:
0     Freshman
1     Freshman
2       Senior
3    Sophomore
```

(continues on next page)

(continued from previous page)

```
4    Sophomore
Name: Grade, dtype: category
Categories (4, object): ['Freshman', 'Sophomore', 'Junior', 'Senior']
```

类别不得直接修改

如果不想指定 ordered=True 参数，那么可以先用 s.cat.as_ordered() 转化为有序类别，再利用 reorder_categories 进行具体的相对大小调整。

9.2.2 排序和比较

在第二章中，曾提到了字符串和数值类型序列的排序，此时就要说明分类变量的排序：只需把列的类型修改为 category 后，再赋予相应的大小关系，就能正常地使用 sort_index 和 sort_values 。例如，对年级进行排序：

```
In [26]: df.Grade = df.Grade.astype('category')

In [27]: df.Grade = df.Grade.cat.reorder_categories(['Freshman',
   ....:                                             'Sophomore',
   ....:                                             'Junior',
   ....:                                             'Senior'],ordered=True)
   ....:

In [28]: df.sort_values('Grade').head() # 值排序
Out[28]:
        Grade            Name  Gender  Height  Weight
0    Freshman   Gaopeng Yang  Female   158.9    46.0
105  Freshman      Qiang Shi  Female   164.5    52.0
96   Freshman  Changmei Feng  Female   163.8    56.0
88   Freshman   Xiaopeng Han  Female   164.1    53.0
81   Freshman    Yanli Zhang  Female   165.1    52.0

In [29]: df.set_index('Grade').sort_index().head() # 索引排序
Out[29]:
                   Name  Gender  Height  Weight
Grade
Freshman   Gaopeng Yang  Female   158.9    46.0
Freshman      Qiang Shi  Female   164.5    52.0
Freshman  Changmei Feng  Female   163.8    56.0
Freshman   Xiaopeng Han  Female   164.1    53.0
Freshman    Yanli Zhang  Female   165.1    52.0
```

由于序的建立，因此就可以进行比较操作。分类变量的比较操作分为两类，第一种是 == 或 != 关系的比较，比较的对象可以是标量或者同长度的 Series （或 list ），第二种是 >,>=,<,<= 四类大小关系的比较，比较的对象和第一种类似，但是所有参与比较的元素必须属于原序列的 categories ，同时要和原序列具有相同的索引。

```
In [30]: res1 = df.Grade == 'Sophomore'

In [31]: res1.head()
Out[31]:
0    False
1    False
2    False
3     True
4     True
Name: Grade, dtype: bool

In [32]: res2 = df.Grade == ['PhD']*df.shape[0]

In [33]: res2.head()
Out[33]:
0    False
1    False
2    False
3    False
4    False
Name: Grade, dtype: bool

In [34]: res3 = df.Grade <= 'Sophomore'

In [35]: res3.head()
Out[35]:
0     True
1     True
2    False
3     True
4     True
Name: Grade, dtype: bool

In [36]: res4 = df.Grade <= df.Grade.sample(
   ....:                               frac=1).reset_index(
   ....:                                         drop=True) # 打乱后比较
   ....:

In [37]: res4.head()
```

(continues on next page)

(continued from previous page)

```
Out[37]:
0     True
1     True
2    False
3     True
4     True
Name: Grade, dtype: bool
```

9.3 区间类别

9.3.1 利用 cut 和 qcut 进行区间构造

区间是一种特殊的类别，在实际数据分析中，区间序列往往是通过 cut 和 qcut 方法进行构造的，这两个函数能够把原序列的数值特征进行装箱，即用区间位置来代替原来的具体数值。

首先介绍 cut 的常见用法：

其中，最重要的参数是 bin ，如果传入整数 n ，则代表把整个传入数组的按照最大和最小值等间距地分为 n 段。由于区间默认是左开右闭，需要进行调整把最小值包含进去，在 pandas 中的解决方案是在值最小的区间左端点再减去 0.001*(max-min) ，因此如果对序列 [1,2] 划分为 2 个箱子时，第一个箱子的范围 (0.999,1.5] ，第二个箱子的范围是 (1.5,2] 。如果需要指定左闭右开时，需要把 right 参数设置为 False ，相应的区间调整方法是在值最大的区间右端点再加上 0.001*(max-min) 。

```
In [38]: s = pd.Series([1,2])

In [39]: pd.cut(s, bins=2)
Out[39]:
0    (0.999, 1.5]
1      (1.5, 2.0]
dtype: category
Categories (2, interval[float64]): [(0.999, 1.5] < (1.5, 2.0]]

In [40]: pd.cut(s, bins=2, right=False)
Out[40]:
0      [1.0, 1.5)
1    [1.5, 2.001)
dtype: category
Categories (2, interval[float64]): [[1.0, 1.5) < [1.5, 2.001)]
```

bins 的另一个常见用法是指定区间分割点的列表（使用 np.infty 可以表示无穷大）：

```
In [41]: pd.cut(s, bins=[-np.infty, 1.2, 1.8, 2.2, np.infty])
Out[41]:
0    (-inf, 1.2]
1     (1.8, 2.2]
dtype: category
Categories (4, interval[float64]): [(-inf, 1.2] < (1.2, 1.8] < (1.8, 2.2] < (2.2, inf]]
```

另外两个常用参数为 labels 和 retbins ，分别代表了区间的名字和是否返回分割点（默认不返回）：

```
In [42]: s = pd.Series([1,2])

In [43]: res = pd.cut(s, bins=2, labels=['small', 'big'], retbins=True)

In [44]: res[0]
Out[44]:
0    small
1      big
dtype: category
Categories (2, object): ['small' < 'big']

In [45]: res[1] # 该元素为返回的分割点
Out[45]: array([0.999, 1.5  , 2.   ])
```

从用法上来说，qcut 和 cut 几乎没有差别，只是把 bins 参数变成的 q 参数，qcut 中的 q 是指 quantile 。这里的 q 为整数 n 时，指按照 n 等分位数把数据分箱，还可以传入浮点列表指代相应的分位数分割点。

```
In [46]: s = df.Weight

In [47]: pd.qcut(s, q=3).head()
Out[47]:
0    (33.999, 48.0]
1      (55.0, 89.0]
2      (55.0, 89.0]
3    (33.999, 48.0]
4      (55.0, 89.0]
Name: Weight, dtype: category
Categories (3, interval[float64]): [(33.999, 48.0] < (48.0, 55.0] < (55.0, 89.0]]

In [48]: pd.qcut(s, q=[0,0.2,0.8,1]).head()
Out[48]:
0      (44.0, 69.4]
1      (69.4, 89.0]
2      (69.4, 89.0]
3    (33.999, 44.0]
```

(continues on next page)

(continued from previous page)

```
4      (69.4, 89.0]
Name: Weight, dtype: category
Categories (3, interval[float64]): [(33.999, 44.0] < (44.0, 69.4] < (69.4, 89.0]]
```

9.3.2 一般区间的构造

对于某一个具体的区间而言，其具备三个要素，即左端点、右端点和端点的开闭状态，其中开闭状态可以指定 right, left, both, neither 中的一类：

```
In [49]: my_interval = pd.Interval(0, 1, 'right')

In [50]: my_interval
Out[50]: Interval(0, 1, closed='right')
```

其属性包含了 mid, length, right, left, closed, ，分别表示中点、长度、右端点、左端点和开闭状态。

使用 in 可以判断元素是否属于区间：

```
In [51]: 0.5 in my_interval
Out[51]: True
```

使用 overlaps 可以判断两个区间是否有交集：

```
In [52]: my_interval_2 = pd.Interval(0.5, 1.5, 'left')

In [53]: my_interval.overlaps(my_interval_2)
Out[53]: True
```

一般而言，pd.IntervalIndex 对象有四类方法生成，分别是 from_breaks, from_arrays, from_tuples, interval_range ，它们分别应用于不同的情况：

from_breaks 的功能类似于 cut 或 qcut 函数，只不过后两个是通过计算得到的风格点，而前者是直接传入自定义的分割点：

```
In [54]: pd.IntervalIndex.from_breaks([1,3,6,10], closed='both')
Out[54]:
IntervalIndex([[1, 3], [3, 6], [6, 10]],
              closed='both',
              dtype='interval[int64]')
```

from_arrays 是分别传入左端点和右端点的列表，适用于有交集并且知道起点和终点的情况：

```
In [55]: pd.IntervalIndex.from_arrays(left = [1,3,6,10],
   ....:                              right = [5,4,9,11],
   ....:                              closed = 'neither')
   ....:
Out[55]:
IntervalIndex([(1, 5), (3, 4), (6, 9), (10, 11)],
              closed='neither',
              dtype='interval[int64]')
```

from_tuples 传入的是起点和终点元组构成的列表：

```
In [56]: pd.IntervalIndex.from_tuples([(1,5),(3,4),(6,9),(10,11)],
   ....:                               closed='neither')
   ....:
Out[56]:
IntervalIndex([(1, 5), (3, 4), (6, 9), (10, 11)],
              closed='neither',
              dtype='interval[int64]')
```

一个等差的区间序列由起点、终点、区间个数和区间长度决定，其中三个量确定的情况下，剩下一个量就确定了，interval_range 中的 start, end, periods, freq 参数就对应了这四个量，从而就能构造出相应的区间：

```
In [57]: pd.interval_range(start=1,end=5,periods=8)
Out[57]:
IntervalIndex([(1.0, 1.5], (1.5, 2.0], (2.0, 2.5], (2.5, 3.0], (3.0, 3.5], (3.5, 4.0], (4.0, 4.5],␣
↪(4.5, 5.0]],
              closed='right',
              dtype='interval[float64]')

In [58]: pd.interval_range(end=5,periods=8,freq=0.5)
Out[58]:
IntervalIndex([(1.0, 1.5], (1.5, 2.0], (2.0, 2.5], (2.5, 3.0], (3.0, 3.5], (3.5, 4.0], (4.0, 4.5],␣
↪(4.5, 5.0]],
              closed='right',
              dtype='interval[float64]')
```

练一练

无论是 interval_range 还是下一章时间序列中的 date_range 都是给定了等差序列中四要素中的三个，从而确定整个序列。请回顾等差数列中的首项、末项、项数和公差的联系，写出 interval_range 中四个参数之间的恒等关系。

除此之外，如果直接使用 pd.IntervalIndex([...], closed=...) ，把 Interval 类型的列表组成传入其中转为区间

索引，那么所有的区间会被强制转为指定的 closed 类型，因为 pd.IntervalIndex 只允许存放同一种开闭区间的 Interval 对象。

```
In [59]: pd.IntervalIndex([my_interval, my_interval_2], closed='left')
Out[59]:
IntervalIndex([[0.0, 1.0), [0.5, 1.5)],
              closed='left',
              dtype='interval[float64]')
```

9.3.3 区间的属性与方法

IntervalIndex 上也定义了一些有用的属性和方法。同时，如果想要具体利用 cut 或者 qcut 的结果进行分析，那么需要先将其转为该种索引类型：

```
In [60]: id_interval = pd.IntervalIndex(pd.cut(s, 3))
```

与单个 Interval 类型相似，IntervalIndex 有若干常用属性：left, right, mid, length ，分别表示左右端点、两端点均值和区间长度。

```
In [61]: id_demo = id_interval[:5] # 选出前 5 个展示

In [62]: id_demo
Out[62]:
IntervalIndex([(33.945, 52.333], (52.333, 70.667], (70.667, 89.0], (33.945, 52.333], (70.667, 89.
↪0]],
              closed='right',
              name='Weight',
              dtype='interval[float64]')

In [63]: id_demo.left
Out[63]: Float64Index([33.945, 52.333, 70.667, 33.945, 70.667], dtype='float64')

In [64]: id_demo.right
Out[64]: Float64Index([52.333, 70.667, 89.0, 52.333, 89.0], dtype='float64')

In [65]: id_demo.mid
Out[65]: Float64Index([43.138999999999996, 61.5, 79.8335, 43.138999999999996, 79.8335], dtype=
↪'float64')

In [66]: id_demo.length
Out[66]:
Float64Index([18.387999999999998, 18.334000000000003, 18.333,
              18.387999999999998, 18.333],
             dtype='float64')
```

IntervalIndex 还有两个常用方法，包括 contains 和 overlaps ，分别指逐个判断每个区间是否包含某元素，以及是否和一个 pd.Interval 对象有交集。

```
In [67]: id_demo.contains(4)
Out[67]: array([False, False, False, False, False])

In [68]: id_demo.overlaps(pd.Interval(40,60))
Out[68]: array([ True,  True, False,  True, False])
```

9.4 练习

9.4.1 Ex1：统计未出现的类别

在第五章中介绍了 crosstab 函数，在默认参数下它能够对两个列的组合出现的频数进行统计汇总：

```
In [69]: df = pd.DataFrame({'A':['a','b','c','a'],
   ....:                    'B':['cat','cat','dog','cat']})
   ....:

In [70]: pd.crosstab(df.A, df.B)
Out[70]:
B  cat  dog
A
a    2    0
b    1    0
c    0    1
```

但事实上有些列存储的是分类变量，列中并不一定包含所有的类别，此时如果想要对这些未出现的类别在 crosstab 结果中也进行汇总，则可以指定 dropna 参数为 False ：

```
In [71]: df.B = df.B.astype('category').cat.add_categories('sheep')

In [72]: pd.crosstab(df.A, df.B, dropna=False)
Out[72]:
B  cat  dog  sheep
A
a    2    0      0
b    1    0      0
c    0    1      0
```

请实现一个带有 dropna 参数的 my_crosstab 函数来完成上面的功能。

9.4.2 Ex2：钻石数据集

现有一份关于钻石的数据集，其中 carat, cut, clarity, price 分别表示克拉重量、切割质量、纯净度和价格，样例如下：

```
In [73]: df = pd.read_csv('data/diamonds.csv')

In [74]: df.head(3)
Out[74]:
   carat      cut clarity  price
0   0.23    Ideal     SI2    326
1   0.21  Premium     SI1    326
2   0.23     Good     VS1    327
```

1. 分别对 df.cut 在 object 类型和 category 类型下使用 nunique 函数，并比较它们的性能。
2. 钻石的切割质量可以分为五个等级，由次到好分别是 Fair, Good, Very Good, Premium, Ideal ，纯净度有八个等级，由次到好分别是 I1, SI2, SI1, VS2, VS1, VVS2, VVS1, IF ，请对切割质量按照 **由好到次**的顺序排序，相同切割质量的钻石，按照纯净度进行 **由次到好**的排序。
3. 分别采用两种不同的方法，把 cut, clarity 这两列按照 **由好到次**的顺序，映射到从 0 到 n-1 的整数，其中 n 表示类别的个数。
4. 对每克拉的价格按照分别按照分位数（q=[0.2, 0.4, 0.6, 0.8]）与 [1000, 3500, 5500, 18000] 割点进行分箱得到五个类别 Very Low, Low, Mid, High, Very High ，并把按这两种分箱方法得到的 category 序列依次添加到原表中。
5. 第 4 问中按照整数分箱得到的序列中，是否出现了所有的类别？如果存在没有出现的类别请把该类别删除。
6. 对第 4 问中按照分位数分箱得到的序列，求每个样本对应所在区间的左右端点值和长度。

第 10 章　时序数据

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

In [2]: import pandas as pd
```

10.1　时序中的基本对象

时间序列的概念在日常生活中十分常见，但对于一个具体的时序事件而言，可以从多个时间对象的角度来描述。例如 2020 年 9 月 7 日周一早上 8 点整需要到教室上课，这个课会在当天早上 10 点结束，其中包含了哪些时间概念？

- 第一，会出现时间戳（Date times）的概念，即’2020-9-7 08:00:00’和’2020-9-7 10:00:00’这两个时间点分别代表了上课和下课的时刻，在 pandas 中称为 Timestamp 。同时，一系列的时间戳可以组成 DatetimeIndex ，而将它放到 Series 中后，Series 的类型就变为了 datetime64[ns] ，如果有涉及时区则为 datetime64[ns, tz]，其中 tz 是 timezone 的简写。
- 第二，会出现时间差（Time deltas）的概念，即上课需要的时间，两个 Timestamp 做差就得到了时间差，pandas 中利用 Timedelta 来表示。类似的，一系列的时间差就组成了 TimedeltaIndex，而将它放到 Series 中后，Series 的类型就变为了 timedelta64[ns] 。
- 第三，会出现时间段（Time spans）的概念，即在 8 点到 10 点这个区间都会持续地在上课，在 pandas 利用 Period 来表示。类似的，一系列的时间段就组成了 PeriodIndex，而将它放到 Series 中后，Series 的类型就变为了 Period 。
- 第四，会出现日期偏置（Date offsets）的概念，假设你只知道 9 月的第一个周一早上 8 点要去上课，但不知道具体的日期，那么就需要一个类型来处理此类需求。再例如，想要知道 2020 年 9 月 7 日后的第 30 个工作日是哪一天，那么时间差就解决不了你的问题，从而 pandas 中的 DateOffset 就出现了。同时，pandas 中没有为一列时间偏置专门设计存储类型，理由也很简单，因为需求比较奇怪，一般来说我们只需要对一批时间特征做一个统一的特殊日期偏置。

通过这个简单的例子，就能够容易地总结出官方文档中的这个 表格 ：

概念	单元素类型	数组类型	pandas 数据类型
Date times	Timestamp	DatetimeIndex	datetime64[ns]
Time deltas	Timedelta	TimedeltaIndex	timedelta64[ns]
Time spans	Period	PeriodIndex	period[freq]
Date offsets	DateOffset	None	None

由于时间段对象 Period/PeriodIndex 的使用频率并不高，因此将不进行讲解，而只涉及时间戳序列、时间差序列和日期偏置的相关内容。

10.2 时间戳

10.2.1 Timestamp 的构造与属性

单个时间戳的生成利用 pd.Timestamp 实现，一般而言的常见日期格式都能被成功地转换：

```
In [3]: ts = pd.Timestamp('2020/1/1')

In [4]: ts
Out[4]: Timestamp('2020-01-01 00:00:00')

In [5]: ts = pd.Timestamp('2020-1-1 08:10:30')

In [6]: ts
Out[6]: Timestamp('2020-01-01 08:10:30')
```

通过 year, month, day, hour, min, second 可以获取具体的数值：

```
In [7]: ts.year
Out[7]: 2020

In [8]: ts.month
Out[8]: 1

In [9]: ts.day
Out[9]: 1

In [10]: ts.hour
Out[10]: 8

In [11]: ts.minute
Out[11]: 10
```

(continues on next page)

(continued from previous page)

```
In [12]: ts.second
Out[12]: 30
```

在 pandas 中，时间戳的最小精度为纳秒 ns ，由于使用了 64 位存储，可以表示的时间范围大约可以如下计算：

$$\text{Time Range} = \frac{2^{64}}{10^9 \times 60 \times 60 \times 24 \times 365} \approx 585(\text{Years})$$

通过 pd.Timestamp.max 和 pd.Timestamp.min 可以获取时间戳表示的范围，可以看到确实表示的区间年数大小正如上述计算结果：

```
In [13]: pd.Timestamp.max
Out[13]: Timestamp('2262-04-11 23:47:16.854775807')

In [14]: pd.Timestamp.min
Out[14]: Timestamp('1677-09-21 00:12:43.145225')

In [15]: pd.Timestamp.max.year - pd.Timestamp.min.year
Out[15]: 585
```

10.2.2 Datetime 序列的生成

一组时间戳可以组成时间序列，可以用 to_datetime 和 date_range 来生成。其中，to_datetime 能够把一列时间戳格式的对象转换成为 datetime64[ns] 类型的时间序列：

```
In [16]: pd.to_datetime(['2020-1-1', '2020-1-3', '2020-1-6'])
Out[16]: DatetimeIndex(['2020-01-01', '2020-01-03', '2020-01-06'], dtype='datetime64[ns]',␣
↪freq=None)

In [17]: df = pd.read_csv('data/learn_pandas.csv')

In [18]: s = pd.to_datetime(df.Test_Date)

In [19]: s.head()
Out[19]:
0   2019-10-05
1   2019-09-04
2   2019-09-12
3   2020-01-03
4   2019-11-06
Name: Test_Date, dtype: datetime64[ns]
```

在极少数情况，时间戳的格式不满足转换时，可以强制使用 format 进行匹配：

```
In [20]: temp = pd.to_datetime(['2020\\1\\1','2020\\1\\3'],format='%Y\\%m\\%d')

In [21]: temp
Out[21]: DatetimeIndex(['2020-01-01', '2020-01-03'], dtype='datetime64[ns]', freq=None)
```

注意上面由于传入的是列表，而非 pandas 内部的 Series ，因此返回的是 DatetimeIndex ，如果想要转为 datetime64[ns] 的序列，需要显式用 Series 转化：

```
In [22]: pd.Series(temp).head()
Out[22]:
0   2020-01-01
1   2020-01-03
dtype: datetime64[ns]
```

另外，还存在一种把表的多列时间属性拼接转为时间序列的 to_datetime 操作，此时的列名必须和以下给定的时间关键词列名一致：

```
In [23]: df_date_cols = pd.DataFrame({'year': [2020, 2020],
   ....:                              'month': [1, 1],
   ....:                              'day': [1, 2],
   ....:                              'hour': [10, 20],
   ....:                              'minute': [30, 50],
   ....:                              'second': [20, 40]})
   ....:

In [24]: pd.to_datetime(df_date_cols)
Out[24]:
0   2020-01-01 10:30:20
1   2020-01-02 20:50:40
dtype: datetime64[ns]
```

date_range 是一种生成连续间隔时间的一种方法，其重要的参数为 start, end, freq, periods ，它们分别表示开始时间，结束时间，时间间隔，时间戳个数。其中，四个中的三个参数决定了，那么剩下的一个就随之确定了。这里要注意，开始或结束日期如果作为端点则它会被包含：

```
In [25]: pd.date_range('2020-1-1','2020-1-21', freq='10D') # 包含
Out[25]: DatetimeIndex(['2020-01-01', '2020-01-11', '2020-01-21'], dtype='datetime64[ns]', freq=
↪'10D')

In [26]: pd.date_range('2020-1-1','2020-2-28', freq='10D')
Out[26]:
DatetimeIndex(['2020-01-01', '2020-01-11', '2020-01-21', '2020-01-31',
               '2020-02-10', '2020-02-20'],
```

(continues on next page)

(continued from previous page)

```
               dtype='datetime64[ns]', freq='10D')

In [27]: pd.date_range('2020-1-1',
   ....:               '2020-2-28', periods=6) # 由于结束日期无法取到，freq 不为 10 天
   ....:
Out[27]:
DatetimeIndex(['2020-01-01 00:00:00', '2020-01-12 14:24:00',
               '2020-01-24 04:48:00', '2020-02-04 19:12:00',
               '2020-02-16 09:36:00', '2020-02-28 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

这里的 freq 参数与 DateOffset 对象紧密相关，将在第四节介绍其具体的用法。

练一练

Timestamp 上定义了一个 value 属性，其返回的整数值代表了从 1970 年 1 月 1 日零点到给定时间戳相差的纳秒数，请利用这个属性构造一个随机生成给定日期区间内日期序列的函数。

最后，要介绍一种改变序列采样频率的方法 asfreq ，它能够根据给定的 freq 对序列进行类似于 reindex 的操作：

```
In [28]: s = pd.Series(np.random.rand(5),
   ....:               index=pd.to_datetime([
   ....:                   '2020-1-%d'%i for i in range(1,10,2)]))
   ....:

In [29]: s.head()
Out[29]:
2020-01-01    0.836578
2020-01-03    0.678419
2020-01-05    0.711897
2020-01-07    0.487429
2020-01-09    0.604705
dtype: float64

In [30]: s.asfreq('D').head()
Out[30]:
2020-01-01    0.836578
2020-01-02         NaN
2020-01-03    0.678419
2020-01-04         NaN
2020-01-05    0.711897
```

(continues on next page)

(continued from previous page)

```
Freq: D, dtype: float64

In [31]: s.asfreq('12H').head()
Out[31]:
2020-01-01 00:00:00    0.836578
2020-01-01 12:00:00         NaN
2020-01-02 00:00:00         NaN
2020-01-02 12:00:00         NaN
2020-01-03 00:00:00    0.678419
Freq: 12H, dtype: float64
```

datetime64[ns] 序列的最值与均值

前面提到了 datetime64[ns] 本质上可以理解为一个大整数，对于一个该类型的序列，可以使用 max, min, mean ，来取得最大时间戳、最小时间戳和“平均”时间戳。

10.2.3 dt 对象

如同 category, string 的序列上定义了 cat, str 来完成分类数据和文本数据的操作，在时序类型的序列上定义了 dt 对象来完成许多时间序列的相关操作。这里对于 datetime64[ns] 类型而言，可以大致分为三类操作：取出时间相关的属性、判断时间戳是否满足条件、取整操作。

第一类操作的常用属性包括：date, time, year, month, day, hour, minute, second, microsecond, nanosecond, dayofweek, dayofyear, weekofyear, daysinmonth, quarter ，其中 daysinmonth, quarter 分别表示月中的第几天和季度。

```
In [32]: s = pd.Series(pd.date_range('2020-1-1','2020-1-3', freq='D'))

In [33]: s.dt.date
Out[33]:
0    2020-01-01
1    2020-01-02
2    2020-01-03
dtype: object

In [34]: s.dt.time
Out[34]:
0    00:00:00
1    00:00:00
2    00:00:00
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [35]: s.dt.day
Out[35]:
0    1
1    2
2    3
dtype: int64

In [36]: s.dt.daysinmonth
Out[36]:
0    31
1    31
2    31
dtype: int64
```

在这些属性中，经常使用的是 dayofweek ，它返回了周中的星期情况，周一为 0、周二为 1，以此类推：

```
In [37]: s.dt.dayofweek
Out[37]:
0    2
1    3
2    4
dtype: int64
```

此外，可以通过 month_name, day_name 返回英文的月名和星期名，注意它们是方法而不是属性：

```
In [38]: s.dt.month_name()
Out[38]:
0    January
1    January
2    January
dtype: object

In [39]: s.dt.day_name()
Out[39]:
0    Wednesday
1     Thursday
2       Friday
dtype: object
```

第二类判断操作主要用于测试是否为月/季/年的第一天或者最后一天：

```
In [40]: s.dt.is_year_start # 还可选 is_quarter/month_start
Out[40]:
```

(continues on next page)

(continued from previous page)

```
0     True
1    False
2    False
dtype: bool

In [41]: s.dt.is_year_end # 还可选 is_quarter/month_end
Out[41]:
0    False
1    False
2    False
dtype: bool
```

第三类的取整操作包含 round, ceil, floor ，它们的公共参数为 freq ，常用的包括 H, min, S （小时、分钟、秒），所有可选的 freq 可参考 此处 。

```
In [42]: s = pd.Series(pd.date_range('2020-1-1 20:35:00',
   ....:                             '2020-1-1 22:35:00',
   ....:                             freq='45min'))
   ....:

In [43]: s
Out[43]:
0   2020-01-01 20:35:00
1   2020-01-01 21:20:00
2   2020-01-01 22:05:00
dtype: datetime64[ns]

In [44]: s.dt.round('1H')
Out[44]:
0   2020-01-01 21:00:00
1   2020-01-01 21:00:00
2   2020-01-01 22:00:00
dtype: datetime64[ns]

In [45]: s.dt.ceil('1H')
Out[45]:
0   2020-01-01 21:00:00
1   2020-01-01 22:00:00
2   2020-01-01 23:00:00
dtype: datetime64[ns]

In [46]: s.dt.floor('1H')
Out[46]:
```

(continues on next page)

(continued from previous page)

```
0   2020-01-01 20:00:00
1   2020-01-01 21:00:00
2   2020-01-01 22:00:00
dtype: datetime64[ns]
```

10.2.4 时间戳的切片与索引

一般而言，时间戳序列作为索引使用。如果想要选出某个子时间戳序列，第一类方法是利用 dt 对象和布尔条件联合使用，另一种方式是利用切片，后者常用于连续时间戳。下面，举一些例子说明：

```
In [47]: s = pd.Series(np.random.randint(2,size=366),
   ....:                index=pd.date_range(
   ....:                        '2020-01-01','2020-12-31'))
   ....:

In [48]: idx = pd.Series(s.index).dt

In [49]: s.head()
Out[49]:
2020-01-01    1
2020-01-02    1
2020-01-03    0
2020-01-04    1
2020-01-05    0
Freq: D, dtype: int32
```

Example1：每月的第一天或者最后一天

```
In [50]: s[(idx.is_month_start|idx.is_month_end).values].head()
Out[50]:
2020-01-01    1
2020-01-31    0
2020-02-01    1
2020-02-29    1
2020-03-01    0
dtype: int32
```

Example2：双休日

```
In [51]: s[idx.dayofweek.isin([5,6]).values].head()
Out[51]:
2020-01-04    1
```

(continues on next page)

(continued from previous page)

```
2020-01-05    0
2020-01-11    0
2020-01-12    1
2020-01-18    1
dtype: int32
```

Example3：取出单日值

```
In [52]: s['2020-01-01']
Out[52]: 1

In [53]: s['20200101'] # 自动转换标准格式
Out[53]: 1
```

Example4：取出七月

```
In [54]: s['2020-07'].head()
Out[54]:
2020-07-01    0
2020-07-02    1
2020-07-03    0
2020-07-04    0
2020-07-05    0
Freq: D, dtype: int32
```

Example5：取出 5 月初至 7 月 15 日

```
In [55]: s['2020-05':'2020-7-15'].head()
Out[55]:
2020-05-01    0
2020-05-02    1
2020-05-03    0
2020-05-04    1
2020-05-05    1
Freq: D, dtype: int32

In [56]: s['2020-05':'2020-7-15'].tail()
Out[56]:
2020-07-11    0
2020-07-12    0
2020-07-13    1
2020-07-14    0
2020-07-15    1
Freq: D, dtype: int32
```

10.3 时间差

10.3.1 Timedelta 的生成

正如在第一节中所说，时间差可以理解为两个时间戳的差，这里也可以通过 pd.Timedelta 来构造：

```
In [57]: pd.Timestamp('20200102 08:00:00')-pd.Timestamp('20200101 07:35:00')
Out[57]: Timedelta('1 days 00:25:00')

In [58]: pd.Timedelta(days=1, minutes=25) # 需要注意加 s
Out[58]: Timedelta('1 days 00:25:00')

In [59]: pd.Timedelta('1 days 25 minutes') # 字符串生成
Out[59]: Timedelta('1 days 00:25:00')
```

生成时间差序列的主要方式是 pd.to_timedelta ，其类型为 timedelta64[ns] ：

```
In [60]: s = pd.to_timedelta(df.Time_Record)

In [61]: s.head()
Out[61]:
0   0 days 00:04:34
1   0 days 00:04:20
2   0 days 00:05:22
3   0 days 00:04:08
4   0 days 00:05:22
Name: Time_Record, dtype: timedelta64[ns]
```

与 date_range 一样，时间差序列也可以用 timedelta_range 来生成，它们两者具有一致的参数：

```
In [62]: pd.timedelta_range('0s', '1000s', freq='6min')
Out[62]: TimedeltaIndex(['0 days 00:00:00', '0 days 00:06:00', '0 days 00:12:00'], dtype=
↪'timedelta64[ns]', freq='6T')

In [63]: pd.timedelta_range('0s', '1000s', periods=3)
Out[63]: TimedeltaIndex(['0 days 00:00:00', '0 days 00:08:20', '0 days 00:16:40'], dtype=
↪'timedelta64[ns]', freq=None)
```

对于 Timedelta 序列，同样也定义了 dt 对象，上面主要定义了的属性包括 days, seconds, mircroseconds, nanoseconds ，它们分别返回了对应的时间差特征。需要注意的是，这里的 seconds 不是指单纯的秒，而是对天数取余后剩余的秒数：

```
In [64]: s.dt.seconds
Out[64]:
```

(continues on next page)

(continued from previous page)

```
0      274
1      260
2      322
3      248
4      322
      ...
195    271
196    243
197    288
198    298
199    305
Name: Time_Record, Length: 200, dtype: int64
```

如果不想对天数取余而直接对应秒数，可以使用 total_seconds

```
In [65]: s.dt.total_seconds()
Out[65]:
0      274.0
1      260.0
2      322.0
3      248.0
4      322.0
       ...
195    271.0
196    243.0
197    288.0
198    298.0
199    305.0
Name: Time_Record, Length: 200, dtype: float64
```

与时间戳序列类似，取整函数也是可以在 dt 对象上使用的：

```
In [66]: pd.to_timedelta(df.Time_Record).dt.round('min')
Out[66]:
0     0 days 00:05:00
1     0 days 00:04:00
2     0 days 00:05:00
3     0 days 00:04:00
4     0 days 00:05:00
            ...
195   0 days 00:05:00
196   0 days 00:04:00
197   0 days 00:05:00
```

(continues on next page)

(continued from previous page)

```
198   0 days 00:05:00
199   0 days 00:05:00
Name: Time_Record, Length: 200, dtype: timedelta64[ns]
```

10.3.2 Timedelta 的运算

时间差支持的常用运算有三类：与标量的乘法运算、与时间戳的加减法运算、与时间差的加减法与除法运算：

```
In [67]: td1 = pd.Timedelta(days=1)

In [68]: td2 = pd.Timedelta(days=3)

In [69]: ts = pd.Timestamp('20200101')

In [70]: td1 * 2
Out[70]: Timedelta('2 days 00:00:00')

In [71]: td2 - td1
Out[71]: Timedelta('2 days 00:00:00')

In [72]: ts + td1
Out[72]: Timestamp('2020-01-02 00:00:00')

In [73]: ts - td1
Out[73]: Timestamp('2019-12-31 00:00:00')
```

这些运算都可以移植到时间差的序列上：

```
In [74]: td1 = pd.timedelta_range(start='1 days', periods=5)

In [75]: td2 = pd.timedelta_range(start='12 hours',
   ....:                          freq='2H',
   ....:                          periods=5)
   ....:

In [76]: ts = pd.date_range('20200101', '20200105')

In [77]: td1 * 5
Out[77]: TimedeltaIndex(['5 days', '10 days', '15 days', '20 days', '25 days'], dtype=
↪'timedelta64[ns]', freq='5D')

In [78]: td1 * pd.Series(list(range(5))) # 逐个相乘
```

(continues on next page)

(continued from previous page)

```
Out[78]:
0    0 days
1    2 days
2    6 days
3   12 days
4   20 days
dtype: timedelta64[ns]

In [79]: td1 - td2
Out[79]:
TimedeltaIndex(['0 days 12:00:00', '1 days 10:00:00', '2 days 08:00:00',
                '3 days 06:00:00', '4 days 04:00:00'],
               dtype='timedelta64[ns]', freq=None)

In [80]: td1 + pd.Timestamp('20200101')
Out[80]:
DatetimeIndex(['2020-01-02', '2020-01-03', '2020-01-04', '2020-01-05',
               '2020-01-06'],
              dtype='datetime64[ns]', freq='D')

In [81]: td1 + ts # 逐个相加
Out[81]:
DatetimeIndex(['2020-01-02', '2020-01-04', '2020-01-06', '2020-01-08',
               '2020-01-10'],
              dtype='datetime64[ns]', freq=None)
```

10.4 日期偏置

10.4.1 Offset 对象

日期偏置是一种和日历相关的特殊时间差，例如回到第一节中的两个问题：如何求 2020 年 9 月第一个周一的日期，以及如何求 2020 年 9 月 7 日后的第 30 个工作日是哪一天。

```
In [82]: pd.Timestamp('20200831') + pd.offsets.WeekOfMonth(week=0,weekday=0)
Out[82]: Timestamp('2020-09-07 00:00:00')

In [83]: pd.Timestamp('20200907') + pd.offsets.BDay(30)
Out[83]: Timestamp('2020-10-19 00:00:00')
```

从上面的例子中可以看到，Offset 对象在 pd.offsets 中被定义。当使用 + 时获取离其最近的下一个日期，当使用 - 时获取离其最近的上一个日期：

```
In [84]: pd.Timestamp('20200831') - pd.offsets.WeekOfMonth(week=0,weekday=0)
Out[84]: Timestamp('2020-08-03 00:00:00')

In [85]: pd.Timestamp('20200907') - pd.offsets.BDay(30)
Out[85]: Timestamp('2020-07-27 00:00:00')

In [86]: pd.Timestamp('20200907') + pd.offsets.MonthEnd()
Out[86]: Timestamp('2020-09-30 00:00:00')
```

常用的日期偏置如下可以查阅这里的 文档 描述。在文档罗列的 Offset 中，需要介绍一个特殊的 Offset 对象 CDay ，其中的 holidays, weekmask 参数能够分别对自定义的日期和星期进行过滤，前者传入了需要过滤的日期列表，后者传入的是三个字母的星期缩写构成的星期字符串，其作用是只保留字符串中出现的星期：

```
In [87]: my_filter = pd.offsets.CDay(n=1,weekmask='Wed Fri',holidays=['20200109'])

In [88]: dr = pd.date_range('20200108', '20200111')

In [89]: dr.to_series().dt.dayofweek
Out[89]:
2020-01-08    2
2020-01-09    3
2020-01-10    4
2020-01-11    5
Freq: D, dtype: int64

In [90]: [i + my_filter for i in dr]
Out[90]:
[Timestamp('2020-01-10 00:00:00'),
 Timestamp('2020-01-10 00:00:00'),
 Timestamp('2020-01-15 00:00:00'),
 Timestamp('2020-01-15 00:00:00')]
```

上面的例子中，n 表示增加一天 CDay ，dr 中的第一天为 20200108 ，但由于下一天 20200109 被排除了，并且 20200110 是合法的周五，因此转为 20200110 ，其他后面的日期处理类似。

不要使用部分 `Offset`

在当前版本下由于一些 bug ，不要使用 Day 级别以下的 Offset 对象，比如 Hour, Second 等，请使用对应的 Timedelta 对象来代替。

10.4.2 偏置字符串

前面提到了关于 date_range 的 freq 取值可用 Offset 对象，同时在 pandas 中几乎每一个 Offset 对象绑定了日期偏置字符串（frequencies strings/offset aliases ），可以指定 Offset 对应的字符串来替代使用。下面举一些常见的例子。

```
In [91]: pd.date_range('20200101','20200331', freq='MS') # 月初
Out[91]: DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01'], dtype='datetime64[ns]', freq='MS
↪')

In [92]: pd.date_range('20200101','20200331', freq='M') # 月末
Out[92]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31'], dtype='datetime64[ns]', freq='M
↪')

In [93]: pd.date_range('20200101','20200110', freq='B') # 工作日
Out[93]:
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10'],
              dtype='datetime64[ns]', freq='B')

In [94]: pd.date_range('20200101','20200201', freq='W-MON') # 周一
Out[94]: DatetimeIndex(['2020-01-06', '2020-01-13', '2020-01-20', '2020-01-27'], dtype=
↪'datetime64[ns]', freq='W-MON')

In [95]: pd.date_range('20200101','20200201',
   ....:               freq='WOM-1MON') # 每月第一个周一
   ....:
Out[95]: DatetimeIndex(['2020-01-06'], dtype='datetime64[ns]', freq='WOM-1MON')
```

上面的这些字符串，等价于使用如下的 Offset 对象：

```
In [96]: pd.date_range('20200101','20200331',
   ....:               freq=pd.offsets.MonthBegin())
   ....:
Out[96]: DatetimeIndex(['2020-01-01', '2020-02-01', '2020-03-01'], dtype='datetime64[ns]', freq='MS
↪')

In [97]: pd.date_range('20200101','20200331',
   ....:               freq=pd.offsets.MonthEnd())
   ....:
Out[97]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31'], dtype='datetime64[ns]', freq='M
↪')

In [98]: pd.date_range('20200101','20200110', freq=pd.offsets.BDay())
```

(continues on next page)

(continued from previous page)

```
Out[98]:
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10'],
              dtype='datetime64[ns]', freq='B')

In [99]: pd.date_range('20200101','20200201',
   ....:               freq=pd.offsets.CDay(weekmask='Mon'))
   ....:
Out[99]: DatetimeIndex(['2020-01-06', '2020-01-13', '2020-01-20', '2020-01-27'], dtype=
↪'datetime64[ns]', freq='C')

In [100]: pd.date_range('20200101','20200201',
   .....:                freq=pd.offsets.WeekOfMonth(week=0,weekday=0))
   .....:
Out[100]: DatetimeIndex(['2020-01-06'], dtype='datetime64[ns]', freq='WOM-1MON')
```

关于时区问题的说明

各类时间对象的开发，除了使用 python 内置的 datetime 模块，pandas 还利用了 dateutil 模块，很大一部分是为了处理时区问题。总所周知，我国是没有夏令时调整时间一说的，但有些国家会有这种做法，导致了相对而言一天里可能会有 23/24/25 个小时，也就是 relativedelta ，这使得 Offset 对象和 Timedelta 对象有了对同一问题处理产生不同结果的现象，其中的规则也较为复杂，官方文档的写法存在部分描述错误，并且难以对描述做出统一修正，因为牵涉到了 Offset 相关的很多组件。因此，本教程完全不考虑时区处理，如果对时区处理的时间偏置有兴趣了解讨论，可以联系我或者参见 这里 的讨论。

10.5 时序中的滑窗与分组

10.5.1 滑动窗口

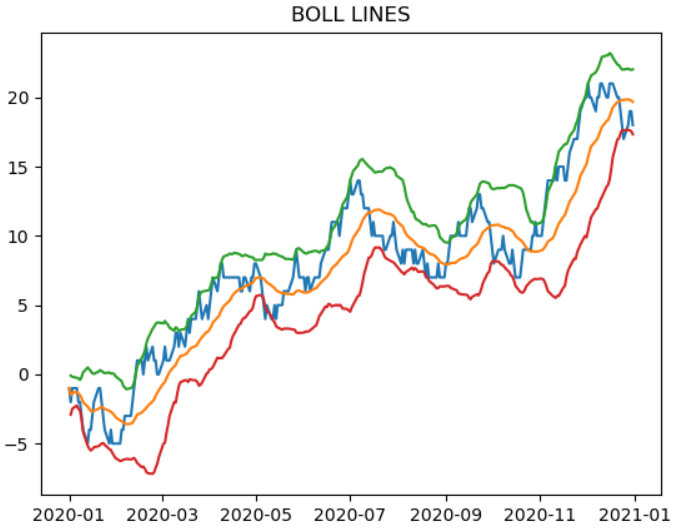
所谓时序的滑窗函数，即把滑动窗口用 freq 关键词代替，下面给出一个具体的应用案例：在股票市场中有一个指标为 BOLL 指标，它由中轨线、上轨线、下轨线这三根线构成，具体的计算方法分别是 N 日均值线、N 日均值加两倍 N 日标准差线、N 日均值减两倍 N 日标准差线。利用 rolling 对象计算 N=30 的 BOLL 指标可以如下写出：

```
In [101]: import matplotlib.pyplot as plt

In [102]: idx = pd.date_range('20200101', '20201231', freq='B')
```

(continues on next page)

(continued from previous page)

```
In [103]: np.random.seed(2020)

In [104]: data = np.random.randint(-1,2,len(idx)).cumsum() # 随机游动构造模拟序列

In [105]: s = pd.Series(data,index=idx)

In [106]: s.head()
Out[106]:
2020-01-01   -1
2020-01-02   -2
2020-01-03   -1
2020-01-06   -1
2020-01-07   -2
Freq: B, dtype: int32

In [107]: r = s.rolling('30D')

In [108]: plt.plot(s)
Out[108]: [<matplotlib.lines.Line2D at 0x22668e7bd08>]

In [109]: plt.title('BOLL LINES')
Out[109]: Text(0.5, 1.0, 'BOLL LINES')

In [110]: plt.plot(r.mean())
Out[110]: [<matplotlib.lines.Line2D at 0x22668e1a448>]

In [111]: plt.plot(r.mean()+r.std()*2)
Out[111]: [<matplotlib.lines.Line2D at 0x22668e1d148>]

In [112]: plt.plot(r.mean()-r.std()*2)
Out[112]: [<matplotlib.lines.Line2D at 0x22668e41608>]
```



对于 shift 函数而言，作用在 datetime64 为索引的序列上时，可以指定 freq 单位进行滑动：

```
In [113]: s.shift(freq='50D').head()
Out[113]:
2020-02-20   -1
2020-02-21   -2
2020-02-22   -1
2020-02-25   -1
2020-02-26   -2
dtype: int32
```

另外，datetime64[ns] 的序列进行 diff 后就能够得到 timedelta64[ns] 的序列，这能够使用户方便地观察有序时间序列的间隔：

```
In [114]: my_series = pd.Series(s.index)

In [115]: my_series.head()
Out[115]:
0   2020-01-01
1   2020-01-02
2   2020-01-03
3   2020-01-06
4   2020-01-07
dtype: datetime64[ns]

In [116]: my_series.diff(1).head()
Out[116]:
0      NaT
```

(continues on next page)

(continued from previous page)

```
1   1 days
2   1 days
3   3 days
4   1 days
dtype: timedelta64[ns]
```

10.5.2 重采样

重采样对象 resample 和第四章中分组对象 groupby 的用法类似，前者是针对时间序列的分组计算而设计的分组对象。

例如，对上面的序列计算每 10 天的均值：

```
In [117]: s.resample('10D').mean().head()
Out[117]:
2020-01-01   -2.000000
2020-01-11   -3.166667
2020-01-21   -3.625000
2020-01-31   -4.000000
2020-02-10   -0.375000
Freq: 10D, dtype: float64
```

同时，如果没有内置定义的处理函数，可以通过 apply 方法自定义：

```
In [118]: s.resample('10D').apply(lambda x:x.max()-x.min()).head() # 极差
Out[118]:
2020-01-01    3
2020-01-11    4
2020-01-21    4
2020-01-31    2
2020-02-10    4
Freq: 10D, dtype: int32
```

在 resample 中要特别注意组边界值的处理情况，默认情况下起始值的计算方法是从最小值时间戳对应日期的午夜 00:00:00 开始增加 freq ，直到不超过该最小时间戳的最大时间戳，由此对应的时间戳为起始值，然后每次累加 freq 参数作为分割结点进行分组，区间情况为左闭右开。下面构造一个不均匀的例子：

```
In [119]: idx = pd.date_range('20200101 8:26:35', '20200101 9:31:58', freq='77s')

In [120]: data = np.random.randint(-1,2,len(idx)).cumsum()

In [121]: s = pd.Series(data,index=idx)
```

(continues on next page)

(continued from previous page)

```
In [122]: s.head()
Out[122]:
2020-01-01 08:26:35   -1
2020-01-01 08:27:52   -1
2020-01-01 08:29:09   -2
2020-01-01 08:30:26   -3
2020-01-01 08:31:43   -4
Freq: 77S, dtype: int32
```

下面对应的第一个组起始值为 08:24:00 ，其是从当天 0 点增加 72 个 freq=7 min 得到的，如果再增加一个 freq 则超出了序列的最小时间戳 08:26:35 ：

```
In [123]: s.resample('7min').mean().head()
Out[123]:
2020-01-01 08:24:00   -1.750000
2020-01-01 08:31:00   -2.600000
2020-01-01 08:38:00   -2.166667
2020-01-01 08:45:00    0.200000
2020-01-01 08:52:00    2.833333
Freq: 7T, dtype: float64
```

有时候，用户希望从序列的最小时间戳开始依次增加 freq 进行分组，此时可以指定 origin 参数为 start ：

```
In [124]: s.resample('7min', origin='start').mean().head()
Out[124]:
2020-01-01 08:26:35   -2.333333
2020-01-01 08:33:35   -2.400000
2020-01-01 08:40:35   -1.333333
2020-01-01 08:47:35    1.200000
2020-01-01 08:54:35    3.166667
Freq: 7T, dtype: float64
```

在返回值中，要注意索引一般是取组的第一个时间戳，但 M, A, Q, BM, BA, BQ, W 这七个是取对应区间的最后一个时间戳：

```
In [125]: s = pd.Series(np.random.randint(2,size=366),
   .....:               index=pd.date_range('2020-01-01',
   .....:                                   '2020-12-31'))
   .....:

In [126]: s.resample('M').mean().head()
Out[126]:
```

(continues on next page)

(continued from previous page)

```
2020-01-31    0.451613
2020-02-29    0.448276
2020-03-31    0.516129
2020-04-30    0.566667
2020-05-31    0.451613
Freq: M, dtype: float64

In [127]: s.resample('MS').mean().head() # 结果一样，但索引不同
Out[127]:
2020-01-01    0.451613
2020-02-01    0.448276
2020-03-01    0.516129
2020-04-01    0.566667
2020-05-01    0.451613
Freq: MS, dtype: float64
```

10.6 练习

10.6.1 Ex1：太阳辐射数据集

现有一份关于太阳辐射的数据集：

```
In [128]: df = pd.read_csv('data/solar.csv', usecols=['Data','Time',
   .....:                  'Radiation','Temperature'])
   .....:

In [129]: df.head(3)
Out[129]:
                   Data      Time  Radiation  Temperature
0  9/29/2016 12:00:00 AM  23:55:26       1.21           48
1  9/29/2016 12:00:00 AM  23:50:23       1.21           48
2  9/29/2016 12:00:00 AM  23:45:26       1.23           48
```

1. 将 Datetime, Time 合并为一个时间列 Datetime ，同时把它作为索引后排序。
2. 每条记录时间的间隔显然并不一致，请解决如下问题：

(a) 找出间隔时间的前三个最大值所对应的三组时间戳。

(b) 是否存在一个大致的范围，使得绝大多数的间隔时间都落在这个区间中？如果存在，请对此范围内的样本间隔秒数画出柱状图，设置 bins=50 。

3. 求如下指标对应的 Series ：

(a) 温度与辐射量的 6 小时滑动相关系数

(b) 以三点、九点、十五点、二十一点为分割，该观测所在时间区间的温度均值序列

(c) 每个观测 6 小时前的辐射量（一般而言不会恰好取到，此时取最近时间戳对应的辐射量）

10.6.2 Ex2：水果销量数据集

现有一份 2019 年每日水果销量记录表：

```
In [130]: df = pd.read_csv('data/fruit.csv')

In [131]: df.head(3)
Out[131]:
         Date  Fruit  Sale
0  2019-04-18  Peach    15
1  2019-12-29  Peach    15
2  2019-06-05  Peach    19
```

1. 统计如下指标：

(a) 每月上半月（15 号及之前）与下半月葡萄销量的比值

(b) 每月最后一天的生梨销量总和

(c) 每月最后一天工作日的生梨销量总和

(d) 每月最后五天的苹果销量均值

2. 按月计算周一至周日各品种水果的平均记录条数，行索引外层为水果名称，内层为月份，列索引为星期。

3. 按天计算向前 10 个工作日窗口的苹果销量均值序列，非工作日的值用上一个工作日的结果填充。

第11章　参考答案

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

In [2]: import pandas as pd

In [3]: import matplotlib.pyplot as plt
```

11.1　预备知识

11.1.1　Ex1：利用列表推导式写矩阵乘法

```
In [4]: M1 = np.random.rand(2,3)

In [5]: M2 = np.random.rand(3,4)

In [6]: res = [[sum([M1[i][k] * M2[k][j] for k in range(M1.shape[1])]) for j in range(M2.
↪shape[1])] for i in range(M1.shape[0])]

In [7]: ((M1@M2 - res) < 1e-15).all()
Out[7]: True
```

11.1.2　Ex2：更新矩阵

```
In [8]: A = np.arange(1,10).reshape(3,-1)

In [9]: B = A*(1/A).sum(1).reshape(-1,1)

In [10]: B
Out[10]:
array([[1.83333333, 3.66666667, 5.5       ],
```

(continues on next page)

(continued from previous page)

```
       [2.46666667, 3.08333333, 3.7       ],
       [2.65277778, 3.03174603, 3.41071429]])
```

11.1.3 Ex3：卡方统计量

```
In [11]: np.random.seed(0)

In [12]: A = np.random.randint(10, 20, (8, 5))

In [13]: B = A.sum(0)*A.sum(1).reshape(-1, 1)/A.sum()

In [14]: res = ((A-B)**2/B).sum()

In [15]: res
Out[15]: 11.842696601945802
```

11.1.4 Ex4：改进矩阵计算的性能

原方法：

```
In [16]: np.random.seed(0)

In [17]: m, n, p = 100, 80, 50

In [18]: B = np.random.randint(0, 2, (m, p))

In [19]: U = np.random.randint(0, 2, (p, n))

In [20]: Z = np.random.randint(0, 2, (m, n))
```

```
In [21]: def solution(B=B, U=U, Z=Z):
   ....:     L_res = []
   ....:     for i in range(m):
   ....:         for j in range(n):
   ....:             norm_value = ((B[i]-U[:,j])**2).sum()
   ....:             L_res.append(norm_value*Z[i][j])
   ....:     return sum(L_res)
   ....:

In [22]: solution(B, U, Z)
Out[22]: 100566
```

改进方法：

令 $Y_{ij} = \|B_i - U_j\|_2^2$ ，则 $R = \sum_{i=1}^{m}\sum_{j=1}^{n} Y_{ij}Z_{ij}$ ，这在 Numpy 中可以用逐元素的乘法后求和实现，因此问题转化为了如何构造 Y 矩阵。

$$\begin{aligned} Y_{ij} &= \|B_i - U_j\|_2^2 \\ &= \sum_{k=1}^{p}(B_{ik} - U_{kj})^2 \\ &= \sum_{k=1}^{p} B_{ik}^2 + \sum_{k=1}^{p} U_{kj}^2 - 2\sum_{k=1}^{p} B_{ik}U_{kj} \end{aligned}$$

从上式可以看出，第一第二项分别为 B 的行平方和与 U 的列平方和，第三项是两倍的内积。因此，Y 矩阵可以写为三个部分，第一个部分是 $m \times n$ 的全 1 矩阵每行乘以 B 对应行的行平方和，第二个部分是相同大小的全 1 矩阵每列乘以 U 对应列的列平方和，第三个部分恰为 B 矩阵与 U 矩阵乘积的两倍。从而结果如下：

```
In [23]: ((np.ones((m,n))*(B**2).sum(1).reshape(-1,1) +
   ....:   np.ones((m,n))*(U**2).sum(0) - 2*B@U)*Z).sum()
   ....:
Out[23]: 100566.0
```

对比它们的性能：

```
In [24]: %timeit -n 30 solution(B, U, Z)
37.7 ms +- 719 us per loop (mean +- std. dev. of 7 runs, 30 loops each)
```

```
In [25]: %timeit -n 30 ((np.ones((m,n))*(B**2).sum(1).reshape(-1,1) +\
   ....:                    np.ones((m,n))*(U**2).sum(0) - 2*B@U)*Z).sum()
   ....:
671 us +- 7.86 us per loop (mean +- std. dev. of 7 runs, 30 loops each)
```

11.1.5 Ex5：连续整数的最大长度

```
In [26]: f = lambda x:np.diff(np.nonzero(np.r_[1,np.diff(x)!=1,1])).max()

In [27]: f([1,2,5,6,7])
Out[27]: 3

In [28]: f([3,2,1,2,3,4,6])
Out[28]: 4
```

11.2 pandas 基础

11.2.1 Ex1：口袋妖怪数据集

1.

```
In [29]: df = pd.read_csv('data/pokemon.csv')

In [30]: (df[['HP', 'Attack', 'Defense', 'Sp. Atk', 'Sp. Def', 'Speed'
   ....:    ]].sum(1)!=df['Total']).mean()
   ....:
Out[30]: 0.0
```

2.

(a)

```
In [31]: dp_dup = df.drop_duplicates('#', keep='first')

In [32]: dp_dup['Type 1'].nunique()
Out[32]: 18

In [33]: dp_dup['Type 1'].value_counts().index[:3]
Out[33]: Index(['Water', 'Normal', 'Grass'], dtype='object')
```

(b)

```
In [34]: attr_dup = dp_dup.drop_duplicates(['Type 1', 'Type 2'])

In [35]: attr_dup.shape[0]
Out[35]: 143
```

(c)

```
In [36]: L_full = [' '.join([i, j]) if i!=j else i for j in dp_dup['Type 1'
   ....:          ].unique() for i in dp_dup['Type 1'].unique()]
   ....:

In [37]: L_part = [' '.join([i, j]) if type(j)!=float else i for i, j in zip(
   ....:          attr_dup['Type 1'], attr_dup['Type 2'])]
   ....:

In [38]: res = set(L_full).difference(set(L_part))
```

(continues on next page)

(continued from previous page)

```
In [39]: len(res) # 太多，不打印了
Out[39]: 181
```

3.

(a)

```
In [40]: df['Attack'].mask(df['Attack']>120, 'high'
   ....:                   ).mask(df['Attack']<50, 'low').mask((50<=df['Attack']
   ....:                   )&(df['Attack']<=120), 'mid').head()
   ....:
Out[40]:
0    low
1    mid
2    mid
3    mid
4    mid
Name: Attack, dtype: object
```

(b)

```
In [41]: df['Type 1'].replace({i:str.upper(i) for i in df['Type 1'].unique()})
Out[41]:
0        GRASS
1        GRASS
2        GRASS
3        GRASS
4         FIRE
        ...
795       ROCK
796       ROCK
797    PSYCHIC
798    PSYCHIC
799       FIRE
Name: Type 1, Length: 800, dtype: object

In [42]: df['Type 1'].apply(lambda x:str.upper(x)).head()
Out[42]:
0    GRASS
1    GRASS
2    GRASS
3    GRASS
4     FIRE
Name: Type 1, dtype: object
```

(c)

```
In [43]: df['Deviation'] = df[['HP', 'Attack', 'Defense', 'Sp. Atk',
   ....:                       'Sp. Def', 'Speed']].apply(lambda x:np.max(
   ....:                       (x-x.median()).abs()), 1)
   ....:

In [44]: df.sort_values('Deviation', ascending=False).head()
Out[44]:
       #                 Name  Type 1  Type 2  Total   HP  Attack  Defense  Sp. Atk  Sp. Def␣
↪Speed  Deviation
230  213              Shuckle     Bug    Rock    505   20      10      230       10      230     ␣
↪5      215.0
121  113              Chansey  Normal     NaN    450  250       5        5       35      105     ␣
↪50      207.5
261  242              Blissey  Normal     NaN    540  255      10       10       75      135     ␣
↪55      190.0
333  306    AggronMega Aggron   Steel     NaN    630   70     140      230       60       80     ␣
↪50      155.0
224  208  SteelixMega Steelix   Steel  Ground    610   75     125      230       55       95     ␣
↪30      145.0
```

11.2.2 Ex2：指数加权窗口

1.

```
In [45]: np.random.seed(0)

In [46]: s = pd.Series(np.random.randint(-1,2,30).cumsum())

In [47]: s.ewm(alpha=0.2).mean().head()
Out[47]:
0   -1.000000
1   -1.000000
2   -1.409836
3   -1.609756
4   -1.725845
dtype: float64

In [48]: def ewm_func(x, alpha=0.2):
   ....:     win = (1-alpha)**np.arange(x.shape[0])[::-1]
   ....:     res = (win*x).sum()/win.sum()
   ....:     return res
   ....:
```

(continues on next page)

(continued from previous page)

```
In [49]: s.expanding().apply(ewm_func).head()
Out[49]:
0   -1.000000
1   -1.000000
2   -1.409836
3   -1.609756
4   -1.725845
dtype: float64
```

2.

新的权重为 $w_i = (1-\alpha)^i, i \in \{0, 1, ..., n-1\}$ ， y_t 更新如下：

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

```
In [50]: s.rolling(window=4).apply(ewm_func).head() # 无需对原函数改动
Out[50]:
0         NaN
1         NaN
2         NaN
3   -1.609756
4   -1.826558
dtype: float64
```

11.3 索引

11.3.1 Ex1：公司员工数据集

1.

```
In [51]: df = pd.read_csv('data/company.csv')

In [52]: dpt = ['Dairy', 'Bakery']

In [53]: df.query("(age <= 40)&(department == @dpt)&(gender=='M')").head(3)
Out[53]:
      EmployeeID birthdate_key  age city_name department     job_title gender
3611        5791     1/14/1975   40   Kelowna      Dairy  Dairy Person      M
```

(continues on next page)

(continued from previous page)

```
3613        5793     1/22/1975   40  Richmond     Bakery         Baker      M
3615        5795     1/30/1975   40   Nanaimo      Dairy  Dairy Person      M

In [54]: df.loc[(df.age<=40)&df.department.isin(dpt)&(df.gender=='M')].head(3)
Out[54]:
      EmployeeID birthdate_key  age city_name department     job_title gender
3611        5791     1/14/1975   40   Kelowna      Dairy  Dairy Person      M
3613        5793     1/22/1975   40  Richmond     Bakery         Baker      M
3615        5795     1/30/1975   40   Nanaimo      Dairy  Dairy Person      M
```

2.

```
In [55]: df.iloc[(df.EmployeeID%2==1).values,[0,2,-2]].head()
Out[55]:
   EmployeeID  age                     job_title
1        1319   58                     VP Stores
3        1321   56             VP Human Resources
5        1323   53      Exec Assistant, VP Stores
6        1325   51  Exec Assistant, Legal Counsel
8        1329   48                 Store Manager
```

3.

```
In [56]: df_op = df.copy()

In [57]: df_op = df_op.set_index(df_op.columns[-3:].tolist()).swaplevel(0,2,axis=0)

In [58]: df_op = df_op.reset_index(level=1)

In [59]: df_op = df_op.rename_axis(index={'gender':'Gender'})

In [60]: df_op.index = df_op.index.map(lambda x:'_'.join(x))

In [61]: df_op.index = df_op.index.map(lambda x:tuple(x.split('_')))

In [62]: df_op = df_op.rename_axis(index=['gender', 'department'])

In [63]: df_op = df_op.reset_index().reindex(df.columns, axis=1)

In [64]: df_op.equals(df)
Out[64]: True
```

11.3.2 Ex2：巧克力数据集

1.

```
In [65]: df = pd.read_csv('data/chocolate.csv')

In [66]: df.columns = [' '.join(i.split('\n')) for i in df.columns]

In [67]: df.head(3)
Out[67]:
    Company  Review Date Cocoa Percent Company Location  Rating
0  A. Morin         2016           63%           France    3.75
1  A. Morin         2015           70%           France    2.75
2  A. Morin         2015           70%           France    3.00
```

2.

```
In [68]: df['Cocoa Percent'] = df['Cocoa Percent'].apply(lambda x:float(x[:-1])/100)

In [69]: df.query('(Rating<3)&(`Cocoa Percent`>`Cocoa Percent`.median())').head(3)
Out[69]:
               Company  Review Date  Cocoa Percent Company Location  Rating
33  Akesson's (Pralus)         2010           0.75      Switzerland    2.75
34  Akesson's (Pralus)         2010           0.75      Switzerland    2.75
36       Alain Ducasse         2014           0.75           France    2.75
```

3.

```
In [70]: idx = pd.IndexSlice

In [71]: exclude = ['France', 'Canada', 'Amsterdam', 'Belgium']

In [72]: res = df.set_index(['Review Date', 'Company Location']).sort_index(level=0)

In [73]: res.loc[idx[2012:,~res.index.get_level_values(1).isin(exclude)],:].head(3)
Out[73]:
                                  Company  Cocoa Percent  Rating
Review Date Company Location
2012        Australia         Bahen & Co.           0.7     3.0
            Australia         Bahen & Co.           0.7     2.5
            Australia         Bahen & Co.           0.7     2.5
```

11.4 分组

11.4.1 Ex1：汽车数据集

现有一份关于汽车的数据集，其中 Brand, Disp., HP 分别代表汽车品牌、发动机蓄量、发动机输出。

```
In [74]: df = pd.read_csv('data/car.csv')

In [75]: df.head(3)
Out[75]:
             Brand  Price Country  Reliability  Mileage   Type  Weight  Disp.   HP
0   Eagle Summit 4   8895     USA          4.0       33  Small    2560     97  113
1  Ford Escort   4   7402     USA          2.0       33  Small    2345    114   90
2   Ford Festiva 4   6319   Korea          4.0       37  Small    1845     81   63
```

1.

```
In [76]: df.groupby('Country').filter(lambda x:x.shape[0]>2).groupby(
   ....:             'Country')['Price'].agg([(
   ....:             'CoV', lambda x: x.std()/x.mean()), 'mean', 'count'])
   ....:
Out[76]:
                CoV          mean  count
Country
Japan      0.387429  13938.052632     19
Japan/USA  0.240040  10067.571429      7
Korea      0.243435   7857.333333      3
USA        0.203344  12543.269231     26
```

2.

```
In [77]: df.shape[0]
Out[77]: 60

In [78]: condition = ['Head']*20+['Mid']*20+['Tail']*20

In [79]: df.groupby(condition)['Price'].mean()
Out[79]:
Head     9069.95
Mid     13356.40
Tail    15420.65
Name: Price, dtype: float64
```

3.

```
In [80]: res = df.groupby('Type').agg({'Price': ['max'], 'HP': ['min']})

In [81]: res.columns = res.columns.map(lambda x:'_'.join(x))

In [82]: res
Out[82]:
         Price_max  HP_min
Type
Compact      18900      95
Large        17257     150
Medium       24760     110
Small         9995      63
Sporty       13945      92
Van          15395     106
```

4.

```
In [83]: def normalize(s):
   ....:     s_min, s_max = s.min(), s.max()
   ....:     res = (s - s_min)/(s_max - s_min)
   ....:     return res
   ....:

In [84]: df.groupby('Type')['HP'].transform(normalize).head()
Out[84]:
0    1.00
1    0.54
2    0.00
3    0.58
4    0.80
Name: HP, dtype: float64
```

5.

```
In [85]: df.groupby('Type')[['HP', 'Disp.']].apply(
   ....:    lambda x:np.corrcoef(x['HP'].values, x['Disp.'].values)[0,1])
   ....:
Out[85]:
Type
Compact    0.586087
Large     -0.242765
Medium     0.370491
Small      0.603916
Sporty     0.871426
```

(continues on next page)

(continued from previous page)

```
Van        0.819881
dtype: float64
```

11.4.2 Ex2：实现 transform 函数

```
In [86]: class my_groupby:
   ....:     def __init__(self, my_df, group_cols):
   ....:         self.my_df = my_df.copy()
   ....:         self.groups = my_df[group_cols].drop_duplicates()
   ....:         if isinstance(self.groups, pd.Series):
   ....:             self.groups = self.groups.to_frame()
   ....:         self.group_cols = self.groups.columns.tolist()
   ....:         self.groups = {i: self.groups[i].values.tolist(
   ....:                         ) for i in self.groups.columns}
   ....:         self.transform_col = None
   ....:     def __getitem__(self, col):
   ....:         self.pr_col = [col] if isinstance(col, str) else list(col)
   ....:         return self
   ....:     def transform(self, my_func):
   ....:         self.num = len(self.groups[self.group_cols[0]])
   ....:         L_order, L_value = np.array([]), np.array([])
   ....:         for i in range(self.num):
   ....:             group_df = self.my_df.reset_index().copy()
   ....:             for col in self.group_cols:
   ....:                 group_df = group_df[group_df[col]==self.groups[col][i]]
   ....:             group_df = group_df[self.pr_col]
   ....:             if group_df.shape[1] == 1:
   ....:                 group_df = group_df.iloc[:, 0]
   ....:             group_res = my_func(group_df)
   ....:             if not isinstance(group_res, pd.Series):
   ....:                 group_res = pd.Series(group_res,
   ....:                                       index=group_df.index,
   ....:                                       name=group_df.name)
   ....:             L_order = np.r_[L_order, group_res.index]
   ....:             L_value = np.r_[L_value, group_res.values]
   ....:         self.res = pd.Series(pd.Series(L_value, index=L_order).sort_index(
   ....:                    ).values,index=self.my_df.reset_index(
   ....:                    ).index, name=my_func.__name__)
   ....:         return self.res
   ....:

In [87]: my_groupby(df, 'Type')
```

(continues on next page)

(continued from previous page)

```
Out[87]: <__main__.my_groupby at 0x2266e2cf648>
```

单列分组：

```
In [88]: def f(s):
   ....:     res = (s-s.min())/(s.max()-s.min())
   ....:     return res
   ....:

In [89]: my_groupby(df, 'Type')['Price'].transform(f).head()
Out[89]:
0    0.733592
1    0.372003
2    0.109712
3    0.186244
4    0.177525
Name: f, dtype: float64

In [90]: df.groupby('Type')['Price'].transform(f).head()
Out[90]:
0    0.733592
1    0.372003
2    0.109712
3    0.186244
4    0.177525
Name: Price, dtype: float64
```

多列分组：

```
In [91]: my_groupby(df, ['Type','Country'])['Price'].transform(f).head()
Out[91]:
0    1.000000
1    0.000000
2    0.000000
3    0.000000
4    0.196357
Name: f, dtype: float64

In [92]: df.groupby(['Type','Country'])['Price'].transform(f).head()
Out[92]:
0    1.000000
1    0.000000
2    0.000000
```

(continues on next page)

(continued from previous page)

```
3    0.000000
4    0.196357
Name: Price, dtype: float64
```

标量广播：

```
In [93]: my_groupby(df, 'Type')['Price'].transform(lambda x:x.mean()).head()
Out[93]:
0    7682.384615
1    7682.384615
2    7682.384615
3    7682.384615
4    7682.384615
Name: <lambda>, dtype: float64

In [94]: df.groupby('Type')['Price'].transform(lambda x:x.mean()).head()
Out[94]:
0    7682.384615
1    7682.384615
2    7682.384615
3    7682.384615
4    7682.384615
Name: Price, dtype: float64
```

跨列计算：

```
In [95]: my_groupby(df, 'Type')['Disp.', 'HP'].transform(
   ....:                 lambda x: x['Disp.']/x.HP).head()
   ....:
Out[95]:
0    0.858407
1    1.266667
2    1.285714
3    0.989130
4    1.097087
Name: <lambda>, dtype: float64
```

11.5 变形

11.5.1 Ex1：美国非法药物数据集

1.

```
In [96]: df = pd.read_csv('data/drugs.csv').sort_values([
   ....:       'State','COUNTY','SubstanceName'],ignore_index=True)
   ....:

In [97]: res = df.pivot(index=['State','COUNTY','SubstanceName'
   ....:                ], columns='YYYY', values='DrugReports'
   ....:                ).reset_index().rename_axis(columns={'YYYY':''})
   ....:

In [98]: res.head(5)
Out[98]:
  State COUNTY  SubstanceName  2010  2011  2012  2013  2014  2015  2016  2017
0    KY  ADAIR  Buprenorphine   NaN   3.0   5.0   4.0  27.0   5.0   7.0  10.0
1    KY  ADAIR        Codeine   NaN   NaN   1.0   NaN   NaN   NaN   NaN   1.0
2    KY  ADAIR       Fentanyl   NaN   NaN   1.0   NaN   NaN   NaN   NaN   NaN
3    KY  ADAIR         Heroin   NaN   NaN   1.0   2.0   NaN   1.0   NaN   2.0
4    KY  ADAIR    Hydrocodone   6.0   9.0  10.0  10.0   9.0   7.0  11.0   3.0
```

2.

```
In [99]: res_melted = res.melt(id_vars = ['State','COUNTY','SubstanceName'],
   ....:                       value_vars = res.columns[-8:],
   ....:                       var_name = 'YYYY',
   ....:                       value_name = 'DrugReports').dropna(
   ....:                       subset=['DrugReports'])
   ....:

In [100]: res_melted = res_melted[df.columns].sort_values([
   .....:                'State','COUNTY','SubstanceName'],ignore_index=True
   .....:                ).astype({'YYYY':'int64', 'DrugReports':'int64'})
   .....: res_melted.equals(df)
   .....:
Out[100]: True
```

3.

策略一：

```
In [101]: res = df.pivot_table(index='YYYY', columns='State',
   .....:                      values='DrugReports', aggfunc='sum')
   .....:

In [102]: res.head(3)
Out[102]:
State     KY     OH     PA    VA    WV
YYYY
2010   10453  19707  19814  8685  2890
2011   10289  20330  19987  6749  3271
2012   10722  23145  19959  7831  3376
```

策略二：

```
In [103]: res = df.groupby(['State', 'YYYY'])['DrugReports'].sum(
   .....:                 ).to_frame().unstack(0).droplevel(0,axis=1)
   .....:

In [104]: res.head(3)
Out[104]:
State     KY     OH     PA    VA    WV
YYYY
2010   10453  19707  19814  8685  2890
2011   10289  20330  19987  6749  3271
2012   10722  23145  19959  7831  3376
```

11.5.2 Ex2：特殊的 wide_to_long 方法

```
In [105]: df = pd.DataFrame({'Class':[1,2],
   .....:                    'Name':['San Zhang', 'Si Li'],
   .....:                    'Chinese':[80, 90],
   .....:                    'Math':[80, 75]})
   .....:

In [106]: df
Out[106]:
   Class       Name  Chinese  Math
0      1  San Zhang       80    80
1      2      Si Li       90    75
```

```
In [107]: df = df.rename(columns={'Chinese':'pre_Chinese', 'Math':'pre_Math'})
```

(continues on next page)

(continued from previous page)

```
In [108]: pd.wide_to_long(df,
   .....:                 stubnames=['pre'],
   .....:                 i = ['Class', 'Name'],
   .....:                 j='Subject',
   .....:                 sep='_',
   .....:                 suffix='.+').reset_index().rename(columns={'pre':'Grade'})
   .....:
Out[108]:
   Class       Name  Subject  Grade
0      1  San Zhang  Chinese     80
1      1  San Zhang     Math     80
2      2      Si Li  Chinese     90
3      2      Si Li     Math     75
```

11.6 连接

11.6.1 Ex1：美国疫情数据集

```
In [109]: date = pd.date_range('20200412', '20201116').to_series()

In [110]: date = date.dt.month.astype('string').str.zfill(2
   .....:        ) +'-'+ date.dt.day.astype('string'
   .....:        ).str.zfill(2) +'-'+ '2020'
   .....:

In [111]: date = date.tolist()
```

```
In [112]: L = []

In [113]: for d in date:
   .....:     df = pd.read_csv('data/us_report/' + d + '.csv', index_col='Province_State')
   .....:     data = df.loc['New York', ['Confirmed','Deaths',
   .....:                    'Recovered','Active']]
   .....:     L.append(data.to_frame().T)
   .....:

In [114]: res = pd.concat(L)

In [115]: res.index = date
```

(continues on next page)

(continued from previous page)

```
In [116]: res.head()
Out[116]:
           Confirmed Deaths Recovered  Active
04-12-2020    189033   9385     23887  179648
04-13-2020    195749  10058     23887  185691
04-14-2020    203020  10842     23887  192178
04-15-2020    214454  11617     23887  202837
04-16-2020    223691  14832     23887  208859
```

11.6.2 Ex2：实现 join 函数

```
In [117]: def join(df1, df2, how='left'):
   .....:     res_col = df1.columns.tolist() +  df2.columns.tolist()
   .....:     dup = df1.index.unique().intersection(df2.index.unique())
   .....:     res_df = pd.DataFrame(columns = res_col)
   .....:     for label in dup:
   .....:         cartesian = [list(i)+list(j) for i in df1.loc[label
   .....:                     ].values for j in df2.loc[label].values]
   .....:         dup_df = pd.DataFrame(cartesian, index = [label]*len(
   .....:                  cartesian), columns = res_col)
   .....:         res_df = pd.concat([res_df,dup_df])
   .....:     if how in ['left', 'outer']:
   .....:         for label in df1.index.unique().difference(dup):
   .....:             if isinstance(df1.loc[label], pd.DataFrame):
   .....:                 cat = [list(i)+[np.nan]*df2.shape[1
   .....:                       ] for i in df1.loc[label].values]
   .....:             else: cat = [list(i)+[np.nan]*df2.shape[1
   .....:                       ] for i in df1.loc[label].to_frame().values]
   .....:             dup_df = pd.DataFrame(cat, index = [label
   .....:                       ]*len(cat), columns = res_col)
   .....:             res_df = pd.concat([res_df,dup_df])
   .....:     if how in ['right', 'outer']:
   .....:         for label in df2.index.unique().difference(dup):
   .....:             if isinstance(df2.loc[label], pd.DataFrame):
   .....:                 cat = [[np.nan]+list(i)*df1.shape[1
   .....:                       ] for i in df2.loc[label].values]
   .....:             else: cat = [[np.nan]+list(i)*df1.shape[1
   .....:                       ] for i in df2.loc[label].to_frame().values]
   .....:             dup_df = pd.DataFrame(cat, index = [label
   .....:                       ]*len(cat), columns = res_col)
   .....:             res_df = pd.concat([res_df,dup_df])
   .....:     return res_df
```

(continues on next page)

(continued from previous page)

```
   .....:

In [118]: df1 = pd.DataFrame({'col1':list('01234')}, index=list('AABCD'))

In [119]: df1
Out[119]:
  col1
A    0
A    1
B    2
C    3
D    4

In [120]: df2 = pd.DataFrame({'col2':list('opqrst')}, index=list('ABBCEE'))

In [121]: df2
Out[121]:
  col2
A    o
B    p
B    q
C    r
E    s
E    t

In [122]: join(df1, df2, how='outer')
Out[122]:
  col1 col2
A    0    o
A    1    o
B    2    p
B    2    q
C    3    r
D    4  NaN
E  NaN    s
E  NaN    t
```

11.7 缺失数据

11.7.1 Ex1：缺失值与类别的相关性检验

```
In [123]: df = pd.read_csv('data/missing_chi.csv')

In [124]: cat_1 = df.X_1.fillna('NaN').mask(df.X_1.notna()).fillna("NotNaN")

In [125]: cat_2 = df.X_2.fillna('NaN').mask(df.X_2.notna()).fillna("NotNaN")

In [126]: df_1 = pd.crosstab(cat_1, df.y, margins=True)

In [127]: df_2 = pd.crosstab(cat_2, df.y, margins=True)

In [128]: def compute_S(my_df):
   .....:     S = []
   .....:     for i in range(2):
   .....:         for j in range(2):
   .....:             E = my_df.iat[i, j]
   .....:             F = my_df.iat[i, 2]*my_df.iat[2, j]/my_df.iat[2,2]
   .....:             S.append((E-F)**2/F)
   .....:     return sum(S)
   .....:

In [129]: res1 = compute_S(df_1)

In [130]: res2 = compute_S(df_2)

In [131]: from scipy.stats import chi2

In [132]: chi2.sf(res1, 1) # X_1 检验的 p 值 # 不能认为相关，剔除
Out[132]: 0.9712760884395901

In [133]: chi2.sf(res2, 1) # X_2 检验的 p 值 # 认为相关，保留
Out[133]: 7.459641265637543e-166
```

结果与 scipy.stats.chi2_contingency 在不使用 $Yates$ 修正的情况下完全一致：

```
In [134]: from scipy.stats import chi2_contingency

In [135]: chi2_contingency(pd.crosstab(cat_1, df.y), correction=False)[1]
Out[135]: 0.9712760884395901
```

(continues on next page)

(continued from previous page)

```
In [136]: chi2_contingency(pd.crosstab(cat_2, df.y), correction=False)[1]
Out[136]: 7.459641265637543e-166
```

11.7.2 Ex2：用回归模型解决分类问题

1.

```
In [137]: from sklearn.neighbors import KNeighborsRegressor

In [138]: df = pd.read_excel('data/color.xlsx')

In [139]: df_dummies = pd.get_dummies(df.Color)

In [140]: stack_list = []

In [141]: for col in df_dummies.columns:
   .....:     clf = KNeighborsRegressor(n_neighbors=6)
   .....:     clf.fit(df.iloc[:,:2], df_dummies[col])
   .....:     res = clf.predict([[0.8, -0.2]]).reshape(-1,1)
   .....:     stack_list.append(res)
   .....:

In [142]: code_res = pd.Series(np.hstack(stack_list).argmax(1))

In [143]: df_dummies.columns[code_res[0]]
Out[143]: 'Yellow'
```

2.

```
In [144]: from sklearn.neighbors import KNeighborsRegressor

In [145]: df = pd.read_csv('data/audit.csv')

In [146]: res_df = df.copy()

In [147]: df = pd.concat([pd.get_dummies(df[['Marital', 'Gender']]),
   .....:     df[['Age','Income','Hours']].apply(
   .....:         lambda x:(x-x.min())/(x.max()-x.min())), df.Employment],1)
   .....:

In [148]: X_train = df.query('Employment.notna()')
```

(continues on next page)

(continued from previous page)

```
In [149]: X_test = df.query('Employment.isna()')

In [150]: df_dummies = pd.get_dummies(X_train.Employment)

In [151]: stack_list = []

In [152]: for col in df_dummies.columns:
   .....:     clf = KNeighborsRegressor(n_neighbors=6)
   .....:     clf.fit(X_train.iloc[:,:-1], df_dummies[col])
   .....:     res = clf.predict(X_test.iloc[:,:-1]).reshape(-1,1)
   .....:     stack_list.append(res)
   .....:

In [153]: code_res = pd.Series(np.hstack(stack_list).argmax(1))

In [154]: cat_res = code_res.replace(dict(zip(list(
   .....:             range(df_dummies.shape[0])),df_dummies.columns)))
   .....:

In [155]: res_df.loc[res_df.Employment.isna(), 'Employment'] = cat_res.values

In [156]: res_df.isna().sum()
Out[156]:
ID            0
Age           0
Employment    0
Marital       0
Income        0
Gender        0
Hours         0
dtype: int64
```

11.8 文本数据

11.8.1 Ex1：房屋信息数据集

1.

```
In [157]: df = pd.read_excel('data/house_info.xls', usecols=[
   .....:                 'floor','year','area','price'])
   .....:
```

(continues on next page)

(continued from previous page)

```
In [158]: df.year = pd.to_numeric(df.year.str[:-2]).astype('Int64')

In [159]: df.head(3)
Out[159]:
      floor  year    area price
0   高层（共 6 层）  1986  58.23 ㎡  155 万
1  中层（共 20 层）  2020     88 ㎡  155 万
2  低层（共 28 层）  2010  89.33 ㎡  365 万
```

2.

```
In [160]: pat = '(\w 层)（共 (\d+) 层）'

In [161]: new_cols = df.floor.str.extract(pat).rename(
   .....:                     columns={0:'Level', 1:'Highest'})
   .....:

In [162]: df = pd.concat([df.drop(columns=['floor']), new_cols], 1)

In [163]: df.head(3)
Out[163]:
   year    area price Level Highest
0  1986  58.23 ㎡  155 万    高层       6
1  2020     88 ㎡  155 万    中层      20
2  2010  89.33 ㎡  365 万    低层      28
```

3.

```
In [164]: s_area = pd.to_numeric(df.area.str[:-1])

In [165]: s_price = pd.to_numeric(df.price.str[:-1])

In [166]: df['avg_price'] = ((s_price/s_area)*10000).astype(
   .....:                     'int').astype('string') + '元/平米'
   .....:

In [167]: df.head(3)
Out[167]:
   year    area price Level Highest  avg_price
0  1986  58.23 ㎡  155 万    高层       6  26618 元/平米
1  2020     88 ㎡  155 万    中层      20  17613 元/平米
2  2010  89.33 ㎡  365 万    低层      28  40859 元/平米
```

11.8.2 Ex2：《权力的游戏》剧本数据集

1.

```
In [168]: df = pd.read_csv('data/script.csv')

In [169]: df.columns = df.columns.str.strip()

In [170]: df.groupby(['Season', 'Episode'])['Sentence'].count().head()
Out[170]:
Season    Episode
Season 1  Episode 1     327
          Episode 10    266
          Episode 2     283
          Episode 3     353
          Episode 4     404
Name: Sentence, dtype: int64
```

2.

```
In [171]: df.set_index('Name').Sentence.str.split().str.len(
   .....:  ).groupby('Name').mean().sort_values(ascending=False).head()
   .....:
Out[171]:
Name
male singer          109.000000
slave owner           77.000000
manderly              62.000000
lollys stokeworth     62.000000
dothraki matron       56.666667
Name: Sentence, dtype: float64
```

3.

```
In [172]: s = pd.Series(df.Sentence.values, index=df.Name.shift(-1))

In [173]: s.str.count('\?').groupby('Name').sum().sort_values(ascending=False).head()
Out[173]:
Name
tyrion lannister    527
jon snow            374
jaime lannister     283
arya stark          265
cersei lannister    246
dtype: int64
```

11.9 分类数据

11.9.1 Ex1：统计未出现的类别

```
In [174]: def my_crosstab(s1, s2, dropna=True):
   .....:     idx1 = (s1.cat.categories if s1.dtype.name == 'category' and
   .....:                               not dropna else s1.unique())
   .....:     idx2 = (s2.cat.categories if s2.dtype.name == 'category' and
   .....:                               not dropna else s2.unique())
   .....:     res = pd.DataFrame(np.zeros((idx1.shape[0], idx2.shape[0])),
   .....:                        index=idx1, columns=idx2)
   .....:     for i, j in zip(s1, s2):
   .....:         res.at[i, j] += 1
   .....:     res = res.rename_axis(index=s1.name, columns=s2.name).astype('int')
   .....:     return res
   .....:

In [175]: df = pd.DataFrame({'A':['a','b','c','a'],
   .....:                    'B':['cat','cat','dog','cat']})
   .....:

In [176]: df.B = df.B.astype('category').cat.add_categories('sheep')

In [177]: my_crosstab(df.A, df.B)
Out[177]:
B  cat  dog
A
a    2    0
b    1    0
c    0    1

In [178]: my_crosstab(df.A, df.B, dropna=False)
Out[178]:
B  cat  dog  sheep
A
a    2    0      0
b    1    0      0
c    0    1      0
```

11.9.2 Ex2：钻石数据集

1.

```
In [179]: df = pd.read_csv('data/diamonds.csv')

In [180]: s_obj, s_cat = df.cut, df.cut.astype('category')
```

```
In [181]: %timeit -n 30 s_obj.nunique()
2.15 ms +- 21.9 us per loop (mean +- std. dev. of 7 runs, 30 loops each)
```

```
In [182]: %timeit -n 30 s_cat.nunique()
539 us +- 2.48 us per loop (mean +- std. dev. of 7 runs, 30 loops each)
```

2.

```
In [183]: df.cut = df.cut.astype('category').cat.reorder_categories([
   .....:         'Fair', 'Good', 'Very Good', 'Premium', 'Ideal'],ordered=True)
   .....:

In [184]: df.clarity = df.clarity.astype('category').cat.reorder_categories([
   .....:         'I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS2', 'VVS1', 'IF'],ordered=True)
   .....:

In [185]: res = df.sort_values(['cut', 'clarity'], ascending=[False, True])

In [186]: res.head(3)
Out[186]:
     carat    cut clarity  price
315   0.96  Ideal      I1   2801
535   0.96  Ideal      I1   2826
551   0.97  Ideal      I1   2830

In [187]: res.tail(3)
Out[187]:
       carat   cut clarity  price
47407   0.52  Fair      IF   1849
49683   0.52  Fair      IF   2144
50126   0.47  Fair      IF   2211
```

3.

```
In [188]: df.cut = df.cut.cat.reorder_categories(
   .....:         df.cut.cat.categories[::-1])
```

(continues on next page)

(continued from previous page)

```
   .....:

In [189]: df.clarity = df.clarity.cat.reorder_categories(
   .....:             df.clarity.cat.categories[::-1])
   .....:

In [190]: df.cut = df.cut.cat.codes # 方法一：利用 cat.codes

In [191]: clarity_cat = df.clarity.cat.categories

In [192]: df.clarity = df.clarity.replace(dict(zip(
   .....:             clarity_cat, np.arange(
   .....:                 len(clarity_cat))))) # 方法二：使用 replace 映射
   .....:

In [193]: df.head(3)
Out[193]:
   carat  cut  clarity  price
0   0.23    0        6    326
1   0.21    1        5    326
2   0.23    3        3    327
```

4.

```
In [194]: q = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [195]: point = [-np.infty, 1000, 3500, 5500, 18000, np.infty]

In [196]: avg = df.price / df.carat

In [197]: df['avg_cut'] = pd.cut(avg, bins=point, labels=[
   .....:                 'Very Low', 'Low', 'Mid', 'High', 'Very High'])
   .....:

In [198]: df['avg_qcut'] = pd.qcut(avg, q=q, labels=[
   .....:                 'Very Low', 'Low', 'Mid', 'High', 'Very High'])
   .....:

In [199]: df.head()
Out[199]:
   carat  cut  clarity  price avg_cut  avg_qcut
0   0.23    0        6    326     Low  Very Low
1   0.21    1        5    326     Low  Very Low
```

(continues on next page)

(continued from previous page)

```
2   0.23    3        3    327     Low  Very Low
3   0.29    1        4    334     Low  Very Low
4   0.31    3        6    335     Low  Very Low
```

5.

```
In [200]: df.avg_cut.unique()
Out[200]:
['Low', 'Mid', 'High']
Categories (3, object): ['Low' < 'Mid' < 'High']

In [201]: df.avg_cut.cat.categories
Out[201]: Index(['Very Low', 'Low', 'Mid', 'High', 'Very High'], dtype='object')

In [202]: df.avg_cut = df.avg_cut.cat.remove_categories([
   .....:              'Very Low', 'Very High'])
   .....:

In [203]: df.avg_cut.head(3)
Out[203]:
0    Low
1    Low
2    Low
Name: avg_cut, dtype: category
Categories (3, object): ['Low' < 'Mid' < 'High']
```

6.

```
In [204]: interval_avg = pd.IntervalIndex(pd.qcut(avg, q=q))

In [205]: interval_avg.right.to_series().reset_index(drop=True).head(3)
Out[205]:
0    2295.0
1    2295.0
2    2295.0
dtype: float64

In [206]: interval_avg.left.to_series().reset_index(drop=True).head(3)
Out[206]:
0    1051.162
1    1051.162
2    1051.162
dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [207]: interval_avg.length.to_series().reset_index(drop=True).head(3)
Out[207]:
0    1243.838
1    1243.838
2    1243.838
dtype: float64
```

11.10 时序数据

11.10.1 Ex1：太阳辐射数据集

1.

```
In [208]: df = pd.read_csv('data/solar.csv', usecols=['Data','Time',
   .....:                  'Radiation','Temperature'])
   .....:

In [209]: solar_date = df.Data.str.extract('([/|\w]+\s).+')[0]

In [210]: df['Data'] = pd.to_datetime(solar_date + df.Time)

In [211]: df = df.drop(columns='Time').rename(columns={'Data':'Datetime'}
   .....:              ).set_index('Datetime').sort_index()
   .....:

In [212]: df.head(3)
Out[212]:
                     Radiation  Temperature
Datetime
2016-09-01 00:00:08       2.58           51
2016-09-01 00:05:10       2.83           51
2016-09-01 00:20:06       2.16           51
```

2.

(a)

```
In [213]: s = df.index.to_series().reset_index(drop=True).diff().dt.total_seconds()

In [214]: max_3 = s.nlargest(3).index
```

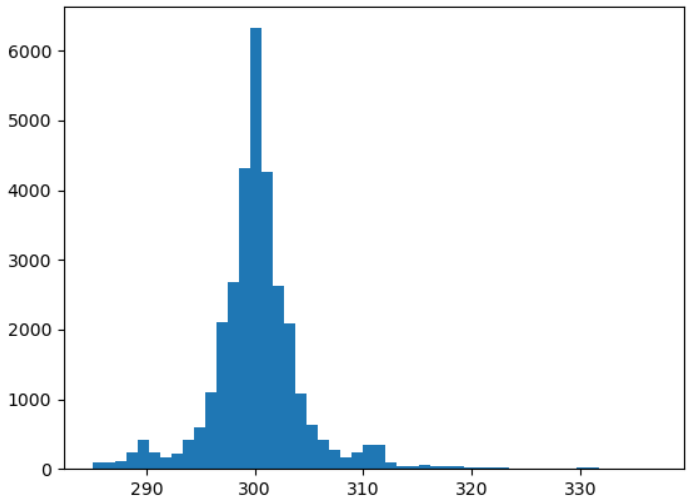
(continues on next page)

(continued from previous page)

```
In [215]: df.index[max_3.union(max_3-1)]
Out[215]:
DatetimeIndex(['2016-09-29 23:55:26', '2016-10-01 00:00:19',
               '2016-11-29 19:05:02', '2016-12-01 00:00:02',
               '2016-12-05 20:45:53', '2016-12-08 11:10:42'],
              dtype='datetime64[ns]', name='Datetime', freq=None)
```

(b)

```
In [216]: res = s.mask((s>s.quantile(0.99))|(s<s.quantile(0.01)))

In [217]: _ = plt.hist(res, bins=50)
```



3.

(a)

```
In [218]: res = df.Radiation.rolling('6H').corr(df.Temperature)

In [219]: res.tail(3)
Out[219]:
Datetime
2016-12-31 23:45:04    0.328574
2016-12-31 23:50:03    0.261883
2016-12-31 23:55:01    0.262406
dtype: float64
```

(b)

```
In [220]: res = df.Temperature.resample('6H', origin='03:00:00').mean()

In [221]: res.head(3)
Out[221]:
Datetime
2016-08-31 21:00:00    51.218750
2016-09-01 03:00:00    50.033333
2016-09-01 09:00:00    59.379310
Freq: 6H, Name: Temperature, dtype: float64
```

(c)

```
In [222]: my_dt = df.index.shift(freq='-6H')

In [223]: int_loc = [df.index.get_loc(i, method='nearest') for i in my_dt]

In [224]: res = df.Radiation.iloc[int_loc]

In [225]: res.tail(3)
Out[225]:
Datetime
2016-12-31 17:45:02    9.33
2016-12-31 17:50:01    8.49
2016-12-31 17:55:02    5.84
Name: Radiation, dtype: float64
```

11.10.2 Ex2：水果销量数据集

1.

(a)

```
In [226]: df = pd.read_csv('data/fruit.csv')

In [227]: df.Date = pd.to_datetime(df.Date)

In [228]: df_grape = df.query("Fruit == 'Grape'")

In [229]: res = df_grape.groupby([np.where(df_grape.Date.dt.day<=15,
   .....:                          'First', 'Second'),df_grape.Date.dt.month]
   .....:                          )['Sale'].mean().to_frame().unstack(0
   .....:                          ).droplevel(0,axis=1)
```

(continues on next page)

(continued from previous page)

```
   .....:

In [230]: res = (res.First/res.Second).rename_axis('Month')

In [231]: res.head()
Out[231]:
Month
1    1.174998
2    0.968890
3    0.951351
4    1.020797
5    0.931061
dtype: float64
```

(b)

```
In [232]: df[df.Date.dt.is_month_end].query("Fruit == 'Pear'"
   .....:                             ).groupby('Date').Sale.sum().head()
   .....:
Out[232]:
Date
2019-01-31    847
2019-02-28    774
2019-03-31    761
2019-04-30    648
2019-05-31    616
Name: Sale, dtype: int64
```

(c)

```
In [233]: df[df.Date.isin(pd.date_range('20190101', '20191231',
   .....:                  freq='BM'))].query("Fruit == 'Pear'"
   .....:                  ).groupby('Date').Sale.mean().head()
   .....:
Out[233]:
Date
2019-01-31    60.500000
2019-02-28    59.538462
2019-03-29    56.666667
2019-04-30    64.800000
2019-05-31    61.600000
Name: Sale, dtype: float64
```

(d)

```
In [234]: target_dt = df.drop_duplicates().groupby(df.Date.drop_duplicates(
   .....:             ).dt.month)['Date'].nlargest(5).reset_index(drop=True)
   .....:

In [235]: res = df.set_index('Date').loc[target_dt].reset_index(
   .....:             ).query("Fruit == 'Apple'")
   .....:

In [236]: res = res.groupby(res.Date.dt.month)['Sale'].mean(
   .....:             ).rename_axis('Month')
   .....:

In [237]: res.head()
Out[237]:
Month
1    65.313725
2    54.061538
3    59.325581
4    65.795455
5    57.465116
Name: Sale, dtype: float64
```

2.

```
In [238]: month_order = ['January','February','March','April',
   .....:                'May','June','July','August','September',
   .....:                'October','November','December']
   .....:

In [239]: week_order = ['Mon','Tue','Wed','Thu','Fri','Sat','Sum']

In [240]: group1 = df.Date.dt.month_name().astype('category').cat.reorder_categories(
   .....:         month_order, ordered=True)
   .....:

In [241]: group2 = df.Fruit

In [242]: group3 = df.Date.dt.dayofweek.replace(dict(zip(range(7),week_order))
   .....:          ).astype('category').cat.reorder_categories(
   .....:          week_order, ordered=True)
   .....:

In [243]: res = df.groupby([group1, group2,group3])['Sale'].count().to_frame(
   .....:          ).unstack(0).droplevel(0,axis=1)
```

(continues on next page)

(continued from previous page)

```
   .....:

In [244]: res.head()
Out[244]:
Date        January  February  March  April  May  June  July  August  September  October  November␣
↪ December
Fruit Date                                                                                  ␣
↪
Apple Mon        46        43     43     47   43    40    41      38         59       42        39␣
↪       45
      Tue        50        40     44     52   46    39    50      42         40       57        47␣
↪       47
      Wed        50        47     37     43   39    39    58      43         35       46        47␣
↪       38
      Thu        45        35     31     47   58    33    52      44         36       63        37␣
↪       40
      Fri        32        33     52     31   46    38    37      48         34       37        46␣
↪       41
```

3.

```
In [245]: df_apple = df[(df.Fruit=='Apple')&(
   .....:                ~df.Date.dt.dayofweek.isin([5,6]))]
   .....:

In [246]: s = pd.Series(df_apple.Sale.values,
   .....:                index=df_apple.Date).groupby('Date').sum()
   .....:

In [247]: res = s.rolling('10D').mean().reindex(
   .....:                pd.date_range('20190101','20191231')).fillna(method='ffill')
   .....:

In [248]: res.head()
Out[248]:
2019-01-01    189.000000
2019-01-02    335.500000
2019-01-03    520.333333
2019-01-04    527.750000
2019-01-05    527.750000
Freq: D, dtype: float64
```