pandas 数据规整

合并数据集

* pandas.merge 可根据一个或多个键将不同 DataFrame 中的行连接起来。
* pandas.concat 可以沿着一条轴将多个对象堆叠到一起
* 实例方法 combine\_first 可以用一个对象中的值填充另一个对象中对应位置的缺失值

使用键参数的 DataFrame 合并

pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=False, suffixes=('\_x', '\_y'), copy=True) 用于通过一个或多个键将两个数据集的**行**连接起来，类似于 SQL 中的 JOIN。该函数的典型应用场景是，针对同一个主键存在两张包含不同字段的表，现在我们想把他们整合到一张表里。在此典型情况下，结果集的行数并没有增加，列数则为两个元数据的列数和减去连接键的数量。

* on=None 用于显示指定列名（键名），如果该列在两个对象上的列名不同，则可以通过 left\_on=None, right\_on=None 来分别指定。或者想直接使用行索引作为连接键的话，就将 left\_index=False, right\_index=False 设为 True。
* how='inner' 参数指的是当左右两个对象中存在不重合的键时，取结果的方式：inner 代表交集；outer 代表并集；left 和 right 分别为取一边。
* suffixes=('\_x','\_y') 指的是当左右对象中存在除连接键外的同名列时，结果集中的区分方式，可以各加一个小尾巴。
* 对于多对多连接，结果采用的是行的笛卡尔积。

示例：

>>> df1 = DataFrame({'key':['a','a','b','b'],'data1':range(4)})

>>> df2 = DataFrame({'key':['b','b','c','c'],'data2':range(4)})

>>> pd.merge(df1,df2)

data1 key data2

0 2 b 0

1 2 b 1

2 3 b 0

3 3 b 1

[4 rows x 3 columns]

>>> pd.merge(df1,df2,how='left')

data1 key data2

0 0 a NaN

1 1 a NaN

2 2 b 0

3 2 b 1

4 3 b 0

5 3 b 1

[6 rows x 3 columns]

>>> pd.merge(df1,df2,left\_index=True,right\_index=True)

data1 key\_x data2 key\_y

0 0 a 0 b

1 1 a 1 b

2 2 b 2 c

3 3 b 3 c

[4 rows x 4 columns]

DataFrame 还有一个方法：.join(self, other, on=None, how='left', lsuffix='', rsuffix='', sort=False)，它能更方便地实现按索引合并。它还可以用于合并多个带有相同或相似索引的 DataFrame 对象，而不管他们之间有没有重叠的列。值得注意的是它的参数里 lsuffix='' , rsuffix='' 并没有给出默认值，所以当你的对象中有**列重叠**（columns overlap）时需要显示指定 suffix 参数，否则会报 ValueError：

>>> df1.join(df2,rsuffix='\_2')

data1 key data2 key\_2

0 0 a 0 b

1 1 a 1 b

2 2 b 2 c

3 3 b 3 c

[4 rows x 4 columns]

轴向连接

merge 算是一种整合的话，轴向连接 pd.concat() 就是单纯地把两个表拼在一起，这个过程也被称作连接（concatenation）、绑定（binding）或堆叠（stacking）。

因此可以想见，这个函数的关键参数应该是 axis，用于指定连接的轴向。在默认的 axis=0 情况下，pd.concat([obj1,obj2]) 函数的效果与 obj1.append(obj2) 是相同的；而在 axis=1 的情况下，pd.concat([df1,df2],axis=1) 的效果与 pd.merge(df1,df2,left\_index=True,right\_index=True,how='outer') 是相同的。可以理解为 concat 函数使用索引作为“连接键”。

本函数的全部参数为：pd.concat(objs, axis=0, join='outer', join\_axes=None, ignore\_index=False, keys=None, levels=None, names=None, verify\_integrity=False)。

* objs 就是需要连接的对象集合，一般是列表或字典；axis=0 是连接轴向
* join='outer' 参数作用于当另一条轴的 index 不重叠的时候，只有 'inner' 和 'outer' 可选（顺带展示 ignore\_index=True 的用法）：

>>> df1 = DataFrame({'a':range(3),'b':range(3)})

>>> df2 = DataFrame({'a':range(4)})

>>> pd.concat([df1,df2])

a b

0 0 0

1 1 1

2 2 2

0 0 NaN

1 1 NaN

2 2 NaN

3 3 NaN

[7 rows x 2 columns]

>>> pd.concat([df1,df2],join='inner',ignore\_index=True)

a

0 0

1 1

2 2

3 0

4 1

5 2

6 3

[7 rows x 1 columns]

* join\_axes=None 参数用于详细制定其他轴上使用的索引，优先级可以覆盖 join 参数，join\_axes 的类型是一个列表，其中的元素为其他轴的 index 。比如上例两条命令等价于这样：pd.concat([df1,df2],join\_axes=[['a','b']]) 、 pd.concat([df1,df2],join\_axes=[['a']])
* keys=None 参数的作用是在结果集中对源数据进行区分。前例中可以看到，结果集中的项无法区分来源，因此使用一个列表型的 keys 参数可以在连接轴上创建一个层次化索引；另一个隐式使用 keys 参数的方法是传入 objs 参数时使用字典，字典的键就会被当做 keys。

>>> s1

a 0

b 1

dtype: int64

>>> s2

c 2

d 3

e 4

dtype: int64

>>> pd.concat([s1,s2],keys=['one','two'])

one a 0

b 1

two c 2

d 3

e 4

dtype: int64

>>> pd.concat({'one':s1,'two':s2})

one a 0

b 1

two c 2

d 3

e 4

dtype: int64

* levels=None 和 names=None 参数与 keys 参数有关，这里 pass；verify\_integrity=False 参数用于检查结果对象新连接轴上的索引是否有重复项，有的话引发 ValueError，可以看到这个参数的作用与 ignore\_index 是互斥的。

合并重叠数据

obj.combine\_first(other) 方法的作用是使用 other 中的数据去填补 obj 中的 NA 值，就像打补丁。而且可以自动对齐。

>>> s1 = Series(range(5))

>>> s2 = Series(range(1,5),index=range(1,5))

>>> s1[2] = np.nan#设置一个 NA

>>> s1

0 0

1 1

2 NaN

3 3

4 4

dtype: float64

>>> s2

1 1

2 2

3 3

4 4

dtype: int32

>>> s1.combine\_first(s2)

0 0

1 1

2 2

3 3

4 4

dtype: float64

重塑和轴向旋转

有许多用于重新排列表格数据的基础运算。这些函数称为重塑（reshape）或轴向旋转（pivot）运算。

重塑层次化索引

层次化索引为 DataFrame 数据的重排任务提供了一种具有良好一致性的方式。重塑层次化索引通过以下两个方法完成：

* .stack() 将列 “压缩” 为行的下级层次化索引
* .unstack() stack 的逆操作——将层次化的行索引 “展开” 为列

示例：

>>> hdf

opening closing

sh 600000 0 1

600001 2 3

sz 000001 4 5

000002 6 7

[4 rows x 2 columns]

>>> hdf.unstack()

opening closing

000001 000002 600000 600001 000001 000002 600000 600001

sh NaN NaN 0 2 NaN NaN 1 3

sz 4 6 NaN NaN 5 7 NaN NaN

[2 rows x 8 columns]

>>> hdf.stack()

sh 600000 opening 0

closing 1

600001 opening 2

closing 3

sz 000001 opening 4

closing 5

000002 opening 6

closing 7

dtype: int32

可见，如果是普通的多列 DataFrame ，调用一次 stack 后就会变成 Series 了。

默认情况下，unstack 操作的是最内层（stack 亦如此）。传入分层级别的编号或 name 即可对其他级别进行操作。

lang:python

>>> hdf.unstack(0)#展开外层

opening closing

sh sz sh sz

000001 NaN 4 NaN 5

000002 NaN 6 NaN 7

600000 0 NaN 1 NaN

600001 2 NaN 3 NaN

[4 rows x 4 columns]

>>> hdf.index.names=['Exchange','code']#分层命名

>>> hdf.unstack('Exchange')

opening closing

Exchange sh sz sh sz

code

000001 NaN 4 NaN 5

000002 NaN 6 NaN 7

600000 0 NaN 1 NaN

600001 2 NaN 3 NaN

[4 rows x 4 columns]

>>> hdf.unstack('code')

opening closing

code 000001 000002 600000 600001 000001 000002 600000 600001

Exchange

sh NaN NaN 0 2 NaN NaN 1 3

sz 4 6 NaN NaN 5 7 NaN NaN

[2 rows x 8 columns]

将 “长格式” 转换为 “宽格式”

时间序列数据通常都是以所谓的 “长格式”（long） 或 “堆叠格式”（stacked）存储在数据库或 CSV 中的：

>>> ldata

date item value

0 1959-03-31 realgdp 2710.349

1 1959-03-31 infl 0.000

2 1959-03-31 unemp 5.800

3 1959-06-30 realgdp 2778.801

4 1959-06-30 infl 2.340

5 1959-06-30 unemp 5.100

6 1959-09-30 realgdp 2775.488

7 1959-09-30 infl 2.740

8 1959-09-30 unemp 5.300

9 1959-12-31 realgdp 2785.204

10 1959-12-31 infl 0.270

11 1959-12-31 unemp 5.600

[12 rows x 3 columns]

这个 item 其实只包含三个字段——realgdp、infl 和 unemp，但每一个字段都单独存储为一行。这样做的好处是在数据库中维护了一个动态的 item 字段，以后如果 item 的项有增删的话，也不必改变表结构。但这种做法的冗余信息过多，而且操作起来很麻烦，需要额外输入很多命令，因此在处理数据前先将其 “展开” 为 “宽格式” 就显得很有必要。

这项任务其实在上一节中就已经给出了解决方法，不过本节要介绍的是一种 “快捷方式”——obj.pivot(index=None, columns=None, values=None) 方法。三个参数都应是来自 obj 的列名，或列对象。分别用于指定结果对象的 index、columns 和 values 属性。

>>> ldata.pivot('date','item','value')

item infl realgdp unemp

date

1959-03-31 0.00 2710.349 5.8

1959-06-30 2.34 2778.801 5.1

1959-09-30 2.74 2775.488 5.3

1959-12-31 0.27 2785.204 5.6

[4 rows x 3 columns]

数据转换

除了前面介绍的数据重排外，另一种重要操作是过滤、清理以及其他的转换工作 <br />

移除重复数据

移除重复数据操作有两个方法可用

* obj.duplicated() 本方法返回一个布尔型 Series，将重复的行标记为 True
* obj.drop\_duplicates() 本方法直接返回一个去除了重复行的新对象

这两个方法默认都会检查所有的列，如果想仅针对某一（些）列进行检查的话，可以传入 cols 参数，指定需要检查的列。

方法默认将第一个出现的值保留，还有一个 take\_last=False 参数，可将其改为 True 以保留最后的值。

利用函数或映射进行数据转换

Series 或 DataFrame 的列都可以调用一个 .map() 方法。该方法接受一个**函数或字典**作为参数，并将之应用于对象的每一个元素，最后返回一个包含所有结果的 Series。

>>> ser = Series(range(5))

>>> ser

0 0

1 1

2 2

3 3

4 4

dtype: int32

>>> ser.map(str).map(lambda x:x+'!')

0 0!

1 1!

2 2!

3 3!

4 4!

dtype: object

>>> ser.map(lambda x:str(x)+'!')

0 0!

1 1!

2 2!

3 3!

4 4!

dtype: object

一个例子写了两遍是为了展示 map 方法的嵌套用法。

替换值

fillna 方法填充缺失值可以看做**值替换**的一种特殊情况，map也可以用来修改对象的数据子集，而 .replace(to\_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None) 方法则提供了实现该功能的一种更简单、更灵活的方式。

to\_replace 参数可以是：str, regex, list, dict, Series, numeric, or None；value 参数可以是：scalar, dict, list, str, regex, default None。其他参数的特殊用法请使用 help 查看。

>>> ser.replace([1,2],'x')

0 0

1 x

2 x

3 3

4 4

dtype: object

重命名轴索引

前面应该提到过，pandas 对象的 index 参数是**不可变**（immutable）的，即不可以直接**对其元素**进行赋值操作。但你却可以对其使用 obj.index.map() 方法。

也可以直接对数组对象调用 obj.rename(index=None,columns=None) 方法。这里的 index 和 columns 参数并不是 index 对象，而是一个函数或字典：

>>> ldata[:3]

date item value

0 1959-03-31 realgdp 2710.349

1 1959-03-31 infl 0.000

2 1959-03-31 unemp 5.800

[3 rows x 3 columns]

>>> ldata[:3].rename(columns=str.title)

Date Item Value

0 1959-03-31 realgdp 2710.349

1 1959-03-31 infl 0.000

2 1959-03-31 unemp 5.800

[3 rows x 3 columns]

离散化和面元划分

为了便于分析，连续数据常常被离散化或拆分为 “面元”（bin）。这个过程要使用到 pandas 的 cut 函数：

cut(x, bins, right=True, labels=None, retbins=False, precision=3, include\_lowest=False)

核心参数为 x 和 bins，x 为被切对象，应当是个一维的类数组结构；bins 参数可以是序列、整数或标量。

* 序列：按序列的元素间隔划分 x，返回 x 各个元素的分组情况

>>> bins = [0,3,6,9]

>>> ser = Series(np.random.randint(1,10,6))

>>> ser

0 5

1 5

2 1

3 4

4 3

5 4

dtype: int32

>>> cats = pd.cut(ser,bins,labels=['small','middle','large'])

>>> cats

middle

middle

small

middle

small

middle

Levels (3): Index(['small', 'middle', 'large'], dtype=object)

* 整数：以 x 的上下界等长划分，可用 precision 参数调节精度。

>>> ser = Series([2,6,7,3,8])

>>> pd.cut(ser,3,precision=1)

(2, 4]

(4, 6]

(6, 8]

(2, 4]

(6, 8]

Levels (3): Index(['(2, 4]', '(4, 6]', '(6, 8]'], dtype=object)

right=True 参数用于控制序列型 bins 的边界，默认为右包含。labels 参数可以给 bins 添加代号。

最后我们来看一下 cut 函数返回的这个对象：

>>> type(cats)

<class 'pandas.core.categorical.Categorical'>

>>> cats.labels

array([1, 1, 0, 1, 0, 1], dtype=int64)

>>> cats.levels

Index(['small', 'middle', 'large'], dtype='object')

>>> pd.value\_counts(cats)

middle 4

small 2

dtype: int64

Categorical 对象是一个枚举型的序列对象，它的可选值都显示在 levels 属性里。

另一个 pd.qcut() 函数与 cut 类似，但它可以根据样本的分位数对数据进行面元划分：

>>> ser = np.random.randint(0,100,1000)

>>> cats = pd.qcut(ser,10)

>>> pd.value\_counts(cats)

(61, 70] 112

(41, 52] 104

[0, 9] 104

(20.8, 31] 103

(77, 88] 102

(31, 41] 100

(88, 99] 97

(9, 20.8] 96

(52, 61] 94

(70, 77] 88

dtype: int64

cut 与 qcut 的更多用法会在数据聚合与分组篇中提及。 <br />

检测和过滤异常值

异常值（outlier）的过滤或变换运算在很大程度上就是数组运算。如下一个 （1000,4）的标准正态分布数组

>>> data = DataFrame(np.random.randn(1000,4))

>>> data.describe()

0 1 2 3

count 1000.000000 1000.000000 1000.000000 1000.000000

mean -0.002069 -0.004543 -0.019383 0.015766

std 1.015236 1.007477 1.036879 0.989083

min -3.344487 -3.305229 -2.980726 -3.573460

25% -0.712828 -0.643239 -0.720927 -0.628151

50% 0.019140 0.019844 -0.048479 0.038351

75% 0.675520 0.669538 0.714605 0.691746

max 3.572161 3.178061 3.114121 3.946495

[8 rows x 4 columns]

假设要找出某一列中绝对值大小超过 3 的项：

>>> col = data[3]

>>> col[np.abs(col)>3]

385 -3.573460

692 3.034318

763 3.946495

Name: 3, dtype: float64

要选出全部含有 “绝对值超过 3 的值” 的行，可以利用布尔型索引和 any 方法：

>>> data[(np.abs(data)>3).any(1)]

0 1 2 3

122 0.989242 -0.458811 3.114121 1.562819

215 3.572161 0.187996 -0.687865 1.378730

216 3.265406 -0.263109 0.682896 -0.637152

381 -3.344487 -0.622073 1.107529 -0.196075

385 -2.111132 -0.863913 -1.103775 -3.573460

426 0.210532 -3.208607 1.092182 -0.255276

452 3.203703 -0.992268 -1.396385 -2.701209

457 -1.361164 3.178061 -0.115614 0.709487

692 0.578040 1.480447 -1.927734 3.034318

763 -1.481627 1.136522 0.283987 3.946495

920 1.901519 -3.305229 -0.220002 -0.333692

[11 rows x 4 columns]

以下命令会将 data 的值全部限制在 [-3,3] 之间，通过将异常值替换为 -3 和 3 的方式。

>>> data[np.abs(data)>3] = np.sign(data)\*3

>>> data.describe()

0 1 2 3

count 1000.000000 1000.000000 1000.000000 1000.000000

mean -0.002766 -0.004207 -0.019497 0.015359

std 1.010851 1.005330 1.036540 0.983736

min -3.000000 -3.000000 -2.980726 -3.000000 #!

25% -0.712828 -0.643239 -0.720927 -0.628151

50% 0.019140 0.019844 -0.048479 0.038351

75% 0.675520 0.669538 0.714605 0.691746

max 3.000000 3.000000 3.000000 3.000000 #!

[8 rows x 4 columns]

np.sign() 函数可以返回一个由 -1 和 1 组成的数组，表示原始值的符号。 <br />

随机采样

随机采样的基本思路是：先利用 np.random 模块随机生成一个需要的索引，然后利用这个索引去源数据里过滤取值。随机采样的两个常用函数为

* np.random.randint(start, end, size) 这个函数一般用于实现 “可重取” 的随机采样，因为返回的数组中的元素可重复，而且 size 可变

>>> bag = np.array([5,7,-1,6,4])

>>> sampler = np.random.randint(0,len(bag),size=10)

>>> bag.take(sampler)

array([5, 5, 7, 5, 7, 5, 6, 7, 7, 4])

>>> bag[sampler]

array([5, 5, 7, 5, 7, 5, 6, 7, 7, 4])

* np.random.permutation(x) 函数用于随机排列一个序列类型。x 参数接受**整数或类序列类型**，实际处理过程中都是按序列来处理的——整型 x 会当做 range(x) 来处理。本函数会随机重排（shuffle）接收到的序列参数并返回一个新结果，显然这是一个 “不可重取” 的抽样，且 size 最大即为 len(x)。

>>> df = DataFrame(np.arange(20).reshape(5,4))

>>> sampler = np.random.permutation(5)

>>> sampler

array([2, 1, 0, 4, 3])

>>> df

0 1 2 3

0 0 1 2 3

1 4 5 6 7

2 8 9 10 11

3 12 13 14 15

4 16 17 18 19

[5 rows x 4 columns]

>>> df.reindex(sampler)

0 1 2 3

2 8 9 10 11

1 4 5 6 7

0 0 1 2 3

4 16 17 18 19

3 12 13 14 15

[5 rows x 4 columns]

>>> df.take(sampler)

0 1 2 3

2 8 9 10 11

1 4 5 6 7

0 0 1 2 3

4 16 17 18 19

3 12 13 14 15

[5 rows x 4 columns]

>>> df.ix[sampler]

0 1 2 3

2 8 9 10 11

1 4 5 6 7

0 0 1 2 3

4 16 17 18 19

3 12 13 14 15

[5 rows x 4 columns]

因为 sampler 是一个数组类型，所以用它在源数据中取值的方式有很多 ↑，如果不想全部取样的话，给 sampler 加个切片就可以了。 <br />

转换指标/哑变量

pd.get\_dummies(data, prefix=None, prefix\_sep='\_', dummy\_na=False) 函数可用来将**分类变量**（Categorical variable）转换为 “哑变量矩阵”（dummy matrix）或称 “指标矩阵”（indicator matrix）。更加便捷的是，data 参数并不限于 categorical 类型，而是可以直接使用一个类 Series 对象，比如 DataFrame 的列。本函数返回的是一个以 data 元素为列名的 1、0 矩阵。

>>> ser = Series(['b','b','a','c','a','b'],name='key')

>>> ser

0 b

1 b

2 a

3 c

4 a

5 b

Name: key, dtype: object

>>> pd.get\_dummies(ser)

a b c

0 0 1 0

1 0 1 0

2 1 0 0

3 0 0 1

4 1 0 0

5 0 1 0

[6 rows x 3 columns]

将本函数直接应用于 DataFrame 的列上，再与原数据剩余部分连接：

>>> df = DataFrame({'key':['b','b','a','c','a','b'],'value':range(6)})

>>> df

key value

0 b 0

1 b 1

2 a 2

3 c 3

4 a 4

5 b 5

[6 rows x 2 columns]

>>> pd.get\_dummies(df['key']).join(df['value'])

a b c value

0 0 1 0 0

1 0 1 0 1

2 1 0 0 2

3 0 0 1 3

4 1 0 0 4

5 0 1 0 5

[6 rows x 4 columns]

字符串操作

在对字符串元素进行规整化操作时，使用 .map() 方法的一个弊端是需要小心绕过 NA 值。为了解决这个问题，Series 直接提供了一些能够跳过 NA 值的字符串操作方法，全部通过 ser.str.xxx() 来访问。这些方法一般也都支持正则表达式。

>>> data = Series({'Dave':'dav@google.com','Steve':'steve@gmail.com',

'Rov':'rob@gmail.com','Wes':np.nan})

>>> data

Dave dav@google.com

Rov rob@gmail.com

Steve steve@gmail.com

Wes NaN

dtype: object

>>> data.str.contains('gmail')

Dave False

Rov True

Steve True

Wes NaN

dtype: object

>>> pattern = '([A-Z0-9.\_%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

>>> import re

>>> data.str.findall(pattern,flags=re.IGNORECASE)

Dave [(dav, google, com)]

Rov [(rob, gmail, com)]

Steve [(steve, gmail, com)]

Wes NaN

dtype: object

有两个办法可以实现矢量化的元素获取操作：要么使用 str.get，要么在 str 属性上使用索引。

>>> matches = data.str.match(pattern,flags=re.IGNORECASE)

>>> matches.str.get(1)

Dave google

Rov gmail

Steve gmail

Wes NaN

dtype: object

>>> matches.str[0]

Dave dav

Rov rob

Steve steve

Wes NaN

dtype: object

其他一些矢量化的字符串方法有：

|  |  |
| --- | --- |
| **函数** | **解释** |
| cat | 元素级的字符串连接操作，可指定分隔符 |
| contains | 返回表示各字符串是否包含指定模式的布尔型数组 |
| count | 模式的出现次数 |
| endswith, startswith | 元素级执行 x.endswith(pattern) |
| findall | 返回各字符串的模式列表 |
| get | 获取各元素的第 i 个字符 |
| join | 根据指定的分隔符将 Series 中的元素字符串连接起来 |
| len | 计算各字符串的长度 |
| lower, upper | 元素级转换大小写 |
| match | 根据指定的表达式对各元素执行 re.match |
| pad | 在字符串的左边、右边或两边添加空白符 |
| center | 相当于 pad(side='both') |
| repeat | 重复值，元素级执行 x\*n |
| replace | 用指定字符串替换找到的模式 |
| slice | 对 Series 各个字符串进行子串截取 |
| split | 根据分隔符或 re 对字符串进行拆分 |
| strip, rstrip, lstrip | 去除空白符，包括换行符。 |