Financial Data Analysis with Python

Instructor: Luping Yu

Mar 29, 2022

Lecture 05. Data Wrangling: Combine and Merge

In many applications, data may be spread across a number of files or databases or be arranged in a form that is not easy to analyze. This lecture focuses on tools to help combine and merge data.

First, I introduce the concept of hierarchical indexing in pandas, which is used extensively in some of these operations. I then dig into the particular data manipulations.

Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index levels on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form.

Let's start with a simple example; create a Series with a list of lists (or arrays) as the index:

In [219]:

```
import pandas as pd
import numpy as np
#np.random.randn: 生成随机数组

df = pd.Series(np.random.randn(9),
                 index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
                        [1,2,3,1,3,1,2,2,3]])

df
```

Out[219]:

```
a  1    0.569461
   2    1.102654
   3    0.580025
b  1    0.252372
   3    1.627715
c  1   -0.778286
   2   -0.678535
d  2   -0.847814
   3    0.845388
dtype: float64
```

What you're seeing is a prettified view of a Series with a **MultiIndex** as its index. The "gaps" in the index display mean "use the label directly above":

In [220]:

```
df.index
```

Out[220]:

```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select *subsets* of the data:

In [221]:

```
df['b']
```

Out[221]:

```
1    0.252372
3    1.627715
dtype: float64
```

In [222]:

```
df['b':'d']
```

Out[222]:

```
b  1    0.252372
   3    1.627715
c  1   -0.778286
   2   -0.678535
d  2   -0.847814
   3    0.845388
dtype: float64
```

In [223]:

```
df.loc[['b','d']]
```

Out[223]:

```
b  1    0.252372
   3    1.627715
d  2   -0.847814
   3    0.845388
dtype: float64
```

Selection is even possible from an **inner** level:

In [224]:

```
df.loc[:, 2]
```

Out[224]:

```
a    1.102654
c   -0.678535
d   -0.847814
dtype: float64
```

Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table. For example, you could rearrange the data into a DataFrame using its unstack method:

In [225]:

```
df.unstack()
```

Out[225]:

	1	2	3
a	0.569461	1.102654	0.580025
b	0.252372	NaN	1.627715
c	-0.778286	-0.678535	NaN
d	NaN	-0.847814	0.845388

The inverse operation of unstack is stack:

In [226]:

```
df.unstack().stack()
```

Out[226]:

```
a  1    0.569461
   2    1.102654
   3    0.580025
b  1    0.252372
   3    1.627715
c  1   -0.778286
   2   -0.678535
d  2   -0.847814
   3    0.845388
dtype: float64
```

Indexing with a DataFrame's columns

It's not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame's columns.

Here's an example DataFrame:

In [227]:

```
df = pd.DataFrame({'a': range(7),
                   'b': range(7, 0, -1),
                   'c': ['one', 'one', 'one', 'two', 'two','two', 'two'],
                   'd': [0,1,2,0,1,2,3]})

df
```

Out[227]:

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

DataFrame's set_index function will create a new DataFrame using one or more of its columns as the index:

In [228]:

```
df.set_index(['c', 'd'])
```

Out[228]:

		a	b
c	**d**		
one	**0**	0	7
	1	1	6
	2	2	5
two	**0**	3	4
	1	4	3
	2	5	2
	3	6	1

By default the columns are removed from the DataFrame, though you can leave them in:

In [229]:

```
df.set_index(['c', 'd'], drop=False)
```

Out[229]:

		a	b	c	d
c	**d**				
one	**0**	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	**0**	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

reset_index, on the other hand, does the opposite of set_index; the hierarchical index levels are moved into the columns:

In [230]:

```
df = df.set_index(['c', 'd'])

df
```

Out[230]:

		a	b
c	**d**		
one	**0**	0	7
	1	1	6
	2	2	5
two	**0**	3	4
	1	4	3
	2	5	2
	3	6	1

In [231]:

```
df.reset_index()
```

Out[231]:

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

Combining and Merging Datasets

Data contained in pandas objects can be combined together in a number of ways:

- **pandas.merge** connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database **join** operations.
- **pandas.concat** concatenates or “stacks” together objects along an axis.

Database-Style DataFrame Joins

Merge or **join** operations combine datasets by **linking rows using one or more keys**. These operations are central to relational databases (e.g., SQL-based). The merge function in pandas is the main entry point for using these algorithms on your data.

In [232]:

```
# Jupyter notebook display multiple pandas tables side by side (方便并排显示dataframe,
from IPython.display import display_html
from itertools import chain,cycle
def display_side_by_side(*args,titles=cycle([''])):
    html_str=''
    for df,title in zip(args, chain(titles,cycle(['</br>'])) ):
        html_str+='<th style="text-align:center"><td style="vertical-align:top">'
        html_str+=f'<h2>{title}</h2>'
        html_str+=df.to_html().replace('table','table style="display:inline"')
        html_str+='</td></th>'
    display_html(html_str,raw=True)
```

In [233]:

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                    'data1': range(7)})

df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
                    'data2': range(3)})

display_side_by_side(df1, df2)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

This is an example of a **many-to-one join**; the data in df1 has multiple rows labeled a and b, whereas df2 has only one row for each value in the key column. Calling merge with these objects we obtain:

In [234]:

```
df_merged = pd.merge(df1, df2)

display_side_by_side(df1, df2, df_merged)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

Note that I didn't specify which column to join on. If that information is not specified, merge uses the overlapping column names as the keys.

It's a good practice to specify explicitly, though:

In [235]:

```
df_merged = pd.merge(df1, df2, on='key')

display_side_by_side(df1, df2, df_merged)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

If the column names are different in each object, you can specify them separately:

In [236]:

```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                    'data1': range(7)})

df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
                    'data2': range(3)})
```

In [237]:

```
df_merged = pd.merge(df3, df4, left_on='lkey', right_on='rkey')

display_side_by_side(df3, df4, df_merged)
```

	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	rkey	data2
0	a	0
1	b	1
2	d	2

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

You may notice that the 'c' and 'd' values and associated data are missing from the result.

By default merge does an **inner join**: the keys in the result are the intersection, or the common set found in both tables. Other possible options are **left**, **right**, and **outer join**.

The **outer join** takes the union of the keys, combining the effect of applying both left and right joins:

In [238]:

```
df_merged_inner = pd.merge(df1, df2, how='inner')
df_merged_outer = pd.merge(df1, df2, how='outer')
```

In [239]:

```
display_side_by_side(df1, df2, df_merged_inner)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

In [240]:

```
display_side_by_side(df1, df2, df_merged_outer)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

	key	data2
0	a	0
1	b	1
2	d	2

	key	data1	data2
0	b	0.0	1.0
1	b	1.0	1.0
2	b	6.0	1.0
3	a	2.0	0.0
4	a	4.0	0.0
5	a	5.0	0.0
6	c	3.0	NaN
7	d	NaN	2.0

See following table for a summary of the options for *how*:

Option	Behavior
'inner'	Use only the key combinations observed in both tables
'left'	Use all key combinations found in the left table
'right'	Use all key combinations found in the right table
'output'	Use all key combinations observed in both tables together

Many-to-many merges have well-defined, though not necessarily intuitive, behavior. Here's an example:

In [241]:

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                    'data1': range(6)})

df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
                    'data2': range(5)})
```

In [242]:

```
df_merged = pd.merge(df1, df2, on='key', how='left')

display_side_by_side(df1, df2, df_merged)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

	key	data1	data2
0	b	0	1.0
1	b	0	3.0
2	b	1	1.0
3	b	1	3.0
4	a	2	0.0
5	a	2	2.0
6	c	3	NaN
7	a	4	0.0
8	a	4	2.0
9	b	5	1.0
10	b	5	3.0

Many-to-many joins form the **Cartesian product** of the rows. Since there were three 'b' rows in the left DataFrame and two in the right one, there are six 'b' rows in the result.

The join method only affects the distinct key values appearing in the result:

In [243]:

```
df_merged = pd.merge(df1, df2, how='inner')

display_side_by_side(df1, df2, df_merged)
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	b	5	1
5	b	5	3
6	a	2	0
7	a	2	2
8	a	4	0
9	a	4	2

To merge with multiple keys, pass a list of column names:

In [244]:

```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
                     'key2': ['one', 'two', 'one'],
                     'lval': [1, 2, 3]})

right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                      'key2': ['one', 'one', 'one', 'two'],
                      'rval': [4, 5, 6, 7]})
```

In [245]:

```
df_merged = pd.merge(left, right, on=['key1', 'key2'], how='outer')

display_side_by_side(left, right, df_merged)
```

	key1	key2	lval
0	foo	one	1
1	foo	two	2
2	bar	one	3

	key1	key2	rval
0	foo	one	4
1	foo	one	5
2	bar	one	6
3	bar	two	7

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key.

When you're joining columns-on-columns, the indexes on the passed DataFrame objects are **discarded**.

See the following table for an argument reference on merge:

Argument	Description
left	DataFrame to be merged on the left side.
right	DataFrame to be merged on the right side.
how	One of'inner','outer','left', or'right'; defaults to'inner'.
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names inleftandrightas the join keys.
left_on	Columns in left DataFrame to use as join keys.
right_on	Analogous to left_on for left DataFrame.
left_index	Use row index in left as its join key (or keys, if a MultiIndex).
right_index	Analogous to left_index.

Merging on Index

In some cases, the merge key(s) in a DataFrame will be found in its index. In this case, you can pass left_index=True or right_index=True (or both) to indicate that the index should be used as the merge key:

In [246]:

```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
                      'value': range(6)})

right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

In [247]:

```
df_merged = pd.merge(left1, right1, left_on='key', right_index=True)

display_side_by_side(left1, right1, df_merged)
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

	group_val
a	3.5
b	7.0

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

In [248]:

```
df_merged = pd.merge(left1, right1, left_on='key', right_index=True, how='outer')

display_side_by_side(left1, right1, df_merged)
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

	group_val
a	3.5
b	7.0

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Concatenating Along an Axis

Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking.

The concat function in pandas provides a consistent way to concat the datasets. Suppose we have three Series with no index overlap:

In [249]:

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

Calling concat with these objects in a list glues together the values and indexes:

In [250]:

```
pd.concat([s1, s2, s3])
```

Out[250]:

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

By default concat works along axis=0, producing another Series.

If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

In [251]:

```
pd.concat([s1, s2, s3], axis=1)
```

Out[251]:

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

A potential issue is that the concatenated pieces are not identifiable in the result.

Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

In [252]:

```
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

result
```

Out[252]:

```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

In the case of combining Series along axis=1, the keys become the DataFrame column headers:

In [253]:

```
pd.concat([s1, s1, s3], axis=1, keys=['one', 'two', 'three'])
```

Out[253]:

	one	two	three
a	0.0	0.0	NaN
b	1.0	1.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

The same logic extends to DataFrame objects:

In [254]:

```
df1 = pd.DataFrame([[0, 1],[2, 3],[4, 5]],
                   index=['a', 'b', 'c'],
                   columns=['one', 'two'])
df2 = pd.DataFrame([[5, 6],[7, 8]],
                   index=['a', 'c'],
                   columns=['two', 'three'])
```

In [255]:

```
df_concated = pd.concat([df1, df2])

display_side_by_side(df1, df2, df_concated)
```

	one	two
a	0	1
b	2	3
c	4	5

	two	three
a	5	6
c	7	8

	one	two	three
a	0.0	1	NaN
b	2.0	3	NaN
c	4.0	5	NaN
a	NaN	5	6.0
c	NaN	7	8.0

In [256]:

```
df_concated = pd.concat([df1, df2], axis=1)

display_side_by_side(df1, df2, df_concated)
```

	one	two
a	0	1
b	2	3
c	4	5

	two	three
a	5	6
c	7	8

	one	two	two	three
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0