Lecture 04.3 MergeGroupby updated

September 5, 2023

1 Merge and Groupby

Duncan Callaway

This notebook gives an introduction to using Pandas' `merge` and `groupby` methods.

[1]:

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

1.1 Row and column labels

The columns are identified with a list of values. Let's look at the fruit data set again:

[2]:

```
fruit_info_df = pd.read_csv('fruit_info.csv', index_col= False)
fruit_info_df
```

[2]:

```
       fruit   color  weight
0      apple     red     120
1     banana  yellow     150
2     orange  orange     250
3  raspberry    pink      15
```

1.1.1 Q: How do I print out just the columns?

[3]:

```
fruit_info_df.columns
```

[3]:

```
Index(['fruit', 'color', 'weight'], dtype='object')
```

1.1.2 Q: And the rows?

The rows are similarly labeled:

[4]:

```
fruit_info_df.index
```

[4]:

```
RangeIndex(start=0, stop=4, step=1)
```

1.2 Merging

Lets make another data frame and tack it on to the first

```
[5]: price_df = pd.DataFrame({'price':[0.5, 0.65, 1, 0.15],
                             'frut':['apple', 'banana', 'orange', 'rasberry']})
     price_df
```

```
[5]:    price      frut
     0   0.50     apple
     1   0.65    banana
     2   1.00    orange
     3   0.15  rasberry
```

Now let’s blindly merge:

```
[6]: pd.merge(price_df,fruit_info_df)
```

```
---------------------------------------------------------------------------
MergeError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 pd.merge(price_df,fruit_info_df)

File /opt/homebrew/lib/python3.11/site-packages/pandas/core/reshape/merge.py:
 ↪148, in merge(left, right, how, on, left_on, right_on, left_index,␣
 ↪right_index, sort, suffixes, copy, indicator, validate)
    131 @Substitution("\nleft : DataFrame or named Series")
    132 @Appender(_merge_doc, indents=0)
    133 def merge(
   (…)
    146     validate: str | None = None,
    147 ) -> DataFrame:
--> 148     op = _MergeOperation(
    149         left,
    150         right,
    151         how=how,
    152         on=on,
    153         left_on=left_on,
    154         right_on=right_on,
    155         left_index=left_index,
    156         right_index=right_index,
    157         sort=sort,
    158         suffixes=suffixes,
    159         indicator=indicator,
    160         validate=validate,
    161     )
    162     return op.get_result(copy=copy)

File /opt/homebrew/lib/python3.11/site-packages/pandas/core/reshape/merge.py:
 ↪719, in _MergeOperation.__init__(self, left, right, how, on, left_on,␣
 ↪right_on, axis, left_index, right_index, sort, suffixes, indicator, validate)
    712     msg = (
    713         "Not allowed to merge between different levels. "
    714         f"({_left.columns.nlevels} levels on the left, "
    715         f"{_right.columns.nlevels} on the right)"
    716     )
    717     raise MergeError(msg)
--> 719 self.left_on, self.right_on =␣
  ↪self._validate_left_right_on(left_on, right_on)
    721 cross_col = None
    722 if self.how == "cross":

File /opt/homebrew/lib/python3.11/site-packages/pandas/core/reshape/merge.py:
  ↪1500, in _MergeOperation._validate_left_right_on(self, left_on, right_on)
   1498 common_cols = left_cols.intersection(right_cols)
   1499 if len(common_cols) == 0:
-> 1500     raise MergeError(
   1501         "No common columns to perform merge on. "
   1502         f"Merge options: left_on={left_on}, "
   1503         f"right_on={right_on}, "
   1504         f"left_index={self.left_index}, "
   1505         f"right_index={self.right_index}"
   1506     )
   1507 if (
   1508     not left_cols.join(common_cols, how="inner").is_unique
   1509     or not right_cols.join(common_cols, how="inner").is_unique
   1510 ):
   1511     raise MergeError(f"Data columns not unique: {repr(common_cols)}")

MergeError: No common columns to perform merge on. Merge options: left_on=None,␣
  ↪right_on=None, left_index=False, right_index=False
```

What went wrong?

First, we didn't spell fruit correctly. Two ways to fix. First, specify the columns directly:

[7]:
```
pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on = 'fruit')
```

[7]:
```
   price    frut   fruit   color  weight
0   0.50   apple   apple     red     120
1   0.65  banana  banana  yellow     150
2   1.00  orange  orange  orange     250
```

Second, fix the spelling and *don't* tell pandas. In this case pandas works to figure out what's in common.

[8]:
```
price_df.columns[0]='fruit'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
Cell In[8], line 1
----> 1 price_df.columns[0]='fruit'

File /opt/homebrew/lib/python3.11/site-packages/pandas/core/indexes/base.py:
 ↪5157, in Index.__setitem__(self, key, value)
   5155 @final
   5156 def __setitem__(self, key, value):
-> 5157     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Bummer! Can't mutate index values. What to do?

[9]:
```
col_list = list(price_df.columns)
col_list
```

[9]:
```
['price', 'frut']
```

[10]:
```
col_list[1] = 'fruit'
```

[11]:
```
price_df.columns = col_list
price_df
```

[11]:
```
   price     fruit
0   0.50     apple
1   0.65    banana
2   1.00    orange
3   0.15  rasberry
```

[12]:
```
pd.merge(fruit_info_df,price_df)
```

[12]:
```
    fruit   color  weight  price
0   apple     red     120   0.50
1  banana  yellow     150   0.65
2  orange  orange     250   1.00
```

Note we can use different syntax:

[13]:
```
fruit_info_df.merge(price_df)
```

[13]:
```
    fruit   color  weight  price
0   apple     red     120   0.50
1  banana  yellow     150   0.65
2  orange  orange     250   1.00
```

1.2.1 Q: Now we're still missing raspberries – why?

Again, spelling error in the new frame. Let's fix:

[14]:

```
price_df.loc[3,'fruit'] = 'raspberry'
```

Note we could change individual entries in the data frame itself. They are mutable.

[15]:

```
fruit_info_df.merge(price_df)
```

[15]:

```
       fruit   color  weight  price
0      apple     red     120   0.50
1     banana  yellow     150   0.65
2     orange  orange     250   1.00
3  raspberry    pink      15   0.15
```

Another few things to takeaway from this 1. Merge can be brutal. That is, it'll drop data without telling you. BUT that's if we use the default 'inner' merge. In a few lecture we'll talk about alternative ways to merge that are a little less draconian. 2. It's important to review your results. How many rows do you expect? How many do you actually get? Did something important get chucked out? The ensuing solutions are the non-glamorous tasks of data cleaning.

Note, there are other commands – `join`, `concat`, and these do similar things to `merge`.

I've found merge seems to work well for most purposes.

FWIW, `pd.concat` seems to be a little more brute force – requires more careful syntax, but likely does unexpected things less often once you understand the syntax.

[16]:

```
merged_df = fruit_info_df.merge(price_df)
merged_df
```

[16]:

```
       fruit   color  weight  price
0      apple     red     120   0.50
1     banana  yellow     150   0.65
2     orange  orange     250   1.00
3  raspberry    pink      15   0.15
```

1.3 Merge Types: Inner, Outer, Left, Right

Let's load up the dataframe again and experiment with different merge types:

[17]:

```
fruit_info_df = pd.read_csv('fruit_info.csv', index_col= False)
price_df = pd.DataFrame({'price':[0.5, 0.65, 1, 0.15],
                         'frut':['apple', 'banana', 'orange', 'rasberry']})
```

[18]:

```
merged_df_inner = pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on =␣
  ↪'fruit', how = 'inner')
merged_df_inner
```

[18]:

```
   price    frut   fruit   color  weight
0   0.50   apple   apple     red     120
1   0.65  banana  banana  yellow     150
2   1.00  orange  orange  orange     250
```

That's what we got above. `pd.merge` gives an inner join by default.

[19]:
```
merged_df_outer = pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on =␣
  ↪'fruit', how = 'outer')
merged_df_outer
```

[19]:
```
   price      frut      fruit   color  weight
0   0.50     apple      apple     red   120.0
1   0.65    banana     banana  yellow   150.0
2   1.00    orange     orange  orange   250.0
3   0.15  rasberry        NaN     NaN     NaN
4    NaN       NaN  raspberry    pink    15.0
```

You can see we kept *every* row from both dataframes, and populated with NaNs where keys don't match.

Let's try left and right:

[20]:
```
merged_df_left = pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on =␣
  ↪'fruit', how = 'left')
merged_df_left
```

[20]:
```
   price      frut   fruit   color  weight
0   0.50     apple   apple     red   120.0
1   0.65    banana  banana  yellow   150.0
2   1.00    orange  orange  orange   250.0
3   0.15  rasberry     NaN     NaN     NaN
```

[21]:
```
merged_df_right = pd.merge(price_df,fruit_info_df, left_on = 'frut', right_on =␣
  ↪'fruit', how = 'right')
merged_df_right
```

[21]:
```
   price    frut      fruit   color  weight
0   0.50   apple      apple     red     120
1   0.65  banana     banana  yellow     150
2   1.00  orange     orange  orange     250
3    NaN     NaN  raspberry    pink      15
```

We can streamline by replacing the index number with the fruit column.

1.3.1 Q: in the following, what's the `inplace` command for?

[22]:
```
merged_df.set_index('fruit', inplace = True)
merged_df
```

[22]:
```
            color  weight  price
fruit
apple         red     120   0.50
banana     yellow     150   0.65
```

```
orange      orange      250    1.00
raspberry     pink       15    0.15
```

1.3.2 A: It means the re-defined dataframe is assigned to the original name.

This is advantageous in memory constrained situations.

1.4 Multilevel indexing

We can also assign “multilevel” column or row names, like so:

[23]:
```
levels = [('categorical', 'color'),('quantitative',␣
 ↪'weight'),('quantitative','price')]
levels
```

[23]:
```
[('categorical', 'color'),
 ('quantitative', 'weight'),
 ('quantitative', 'price')]
```

Note the use of tuples (sets of values in parentheses) in setting up multiindex. This will come again later.

[24]:
```
merged_df.columns = pd.MultiIndex.from_tuples(levels)
merged_df
```

[24]:
```
          categorical quantitative
                color       weight price
fruit
apple             red          120  0.50
banana         yellow          150  0.65
orange         orange          250  1.00
raspberry        pink           15  0.15
```

Now we have categories and subcategories of columns:

[25]:
```
merged_df['quantitative']
```

[25]:
```
           weight  price
fruit
apple         120   0.50
banana        150   0.65
orange        250   1.00
raspberry      15   0.15
```

1.4.1 Q: How can we get data from an individual column?

Aim to get the `weight` column:

[26]:
```
merged_df[('quantitative','weight')]
```

```
[26]: fruit
      apple        120
      banana       150
      orange       250
      raspberry     15
      Name: (quantitative, weight), dtype: int64
```

1.5 Advanced multilevel (did not do in lecture)

Note, we can also drop and add things. With multilevel indexing things get a little tricky.

First, we can drop everything from the top level:

```
[27]: merged_test_df = merged_df.drop(columns=[('quantitative',)], axis = 1)
      merged_test_df
```

```
[27]:           categorical
                      color
      fruit
      apple             red
      banana         yellow
      orange         orange
      raspberry        pink
```

Note that I put the column identifier inside the parens, like a tuple, but it's not essential there.

However if we want to drop only a column from the second level, we get an error without the tuple syntax:

```
[28]: merged_test_df = merged_df.drop(columns=[('quantitative','price')], axis = 1)
      merged_test_df
```

```
[28]:           categorical quantitative
                      color       weight
      fruit
      apple             red          120
      banana         yellow          150
      orange         orange          250
      raspberry        pink           15
```

We can also drop rows:

```
[29]: merged_df.drop(index=[('apple')], axis = 0, inplace = True)
      merged_df
```

```
[29]:           categorical quantitative
                      color       weight price
      fruit
      banana         yellow          150  0.65
      orange         orange          250  1.00
```

```
raspberry        pink            15  0.15
```

Note indexing multilevels with `.loc` gets a little tricky. The thing to keep in mind is that you're working with tuples in each index location:

[30]:
```python
merged_df.loc['banana', ('quantitative', 'price')]
```

[30]:
```
0.65
```

If you leave an entry of the tuple empty you get all values.

[31]:
```python
merged_df.loc['banana', ('quantitative', )]
```

[31]:
```
weight    150.00
price       0.65
Name: banana, dtype: float64
```

You can also loop through the columns of the multilevel data frame like this:

[32]:
```python
for i, j in merged_df:
    print(merged_df.loc['banana', (i, j)])
```

```
yellow
150
0.65
```

Some added thoughts: 1. Multilevel indexing works for columns and index 2. It can be a powerful way to summarize your data and quickly reference subsets of it. 4. However it can also be a colossal pain in the rear – indexing with multilevel is often very hard to parse and debug.

1.6 Groupby

First, let's have another look at today's power point file. Now we'll learn about how groupby works.

Back to the notebook, let's make a toy DF (example taken from Wes McKinney's Python for Data Analysis:

[33]:
```python
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})
df
```

[33]:
```
  key1 key2     data1     data2
0    a  one -0.041104  0.041628
1    a  two  0.600520  0.112253
2    b  one -0.675769 -0.559583
3    b  two  1.516218 -0.021906
4    a  one -0.547718  1.414356
```

Let's group just the `data1` column by the `key1` column. A call to `groupby` does that.

Note, the syntax is to begin by invoking the portion of the dataframe we want to group (here, `df['data1']`), then we apply the groupby method with the portion of hte dataframe we want to group on (here `df['key1']`)

What is the object that results?

[34]:
```
grouped = df['data1'].groupby(df['key1'])
grouped
```

[34]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x11dc52d10>
```

As we see, it's not simply a new DataFrame. Instead, it's an object, in this case `SeriesGroupBy`. We'll see in a moment that if we group many columns of data we get a `DataFrameGroupBy` object.

To look inside we need to use different syntax. The specific thing we're looking for are the groups of the object…but let's tab in to the grouped object to see what's there.

[ ]:
```
grouped.groups
```

That gave us the groups (a and b) and the indices of elements in the groups, but nothing else.

You can see this structure looks like a dict. a and b are the keys, and the data are lists associated with each key – the values.

But the `grouped` object is capable of making computations across all groups – this is where it gets powerful.

We can try things like `sum`, `min` and `max`.

Notice if you don't put the parens after the method, pandas returns information about what the method does, but not it's actual output.

[ ]:
```
grouped.sum()
```

You can also pass `numpy` functions into the aggregate command.

But it can be informative to look at what's inside. We can iterate over a `groupby` object, as we iterate we get pairs of `(name, group)`, where the `group` is either a `Series` or a `DataFrame`, depending on whether the `groupby` object is a `SeriesGroupBy` (as above) or a `DataFrameGroupBy` (see below).

Something quirky to note about the interaction between the grouped object and the for loop structure below: we're going to define variables `name` and `group` as being things in `grouped`. But there are no `name` or `group` attributes associated with the `grouped` object.

[ ]:
```
for name, group in grouped:
    print('Name:', name)
    display(group)
```

We can group on multiple keys, and the result is grouping by tuples:

[ ]:
```
g2 = df['data1'].groupby([df['key1'], df['key2']])
g2
```

[ ]:

```
g2.groups
```

Now we have a groupby object that has tuples as the keys.

[ ]:

```
g2.mean()
```

1.6.1 Aside (did not do in lecture)

We can also group the entire dataframe – not just one column of it – on a single key. This results in a `DataFrameGroupBy` object as the result:

[ ]:

```
k1g = df.groupby('key1')
k1g
```

[ ]:

```
k1g.groups
```

That output actually looks a lot like the output when we were only grouping one of the columns of the dataframe. But there is actually more information in the group itself.

[ ]:

```
for name, group in k1g:
    print('Name:', name)
    display(group)
```

The contents of each group is another dataframe.

[ ]:

```
k1g.mean()
```

Where did column `key2` go in the mean above? It's a *nuisance column*, which gets automatically eliminated from an operation where it doesn't make sense (such as a numerical mean).

1.6.2 Aside (did not do in lecture): Grouping over a different dimension

Above, we've been grouping data along the rows, using column keys as our selectors.

But we can also group along the *columns*,

What's even more cool? We can group by *data type*.

Here we'll group along columns, by data type:

[ ]:

```
df.dtypes
```

[ ]:

```
grouped = df.groupby(df.dtypes, axis=1)
for dtype, group in grouped:
    print(dtype)
    display(group)
```

1.7 Using groupby to re-ask our question

(did not do this in lecture, instead did CAISO forecasting error example) Which hour had the lowest average wind production?

[ ]:
```
cds = pd.read_csv('CAISO_2017to2018_stack.csv', index_col= 0)
```

[ ]:
```
cds.head()
```

It will help to have a column of hour of day values:

[ ]:
```
cds_time = pd.to_datetime(cds.index)
type(cds_time)
```

Let's add that list of values into the data frame.

[ ]:
```
cds['hour'] = cds_time.hour
```

[ ]:
```
cds.head(10)
```

1.7.1 Q: What groupby syntax would you use to arrange the data...

...so that you can examine production by hour and source?

See if you can do it yourself: we want to group MWh values by source AND hour.

[ ]:
```
cds_grouped = cds['MWh'].groupby([cds['Source'],cds['hour']])
```

1.7.2 Q: How to get *all* the means for all sources and hours?

Didn't need to do any fancy logical indexing or looping!

[ ]:
```
cds_grouped.mean()
```

Now it would be nice to see that information in a dataframe, wouldn't it?

[ ]:
```
averages = pd.DataFrame(cds_grouped.mean())
```

[ ]:
```
averages
```

And lo and behold, we have a multilevel index for the rows!

[ ]:
```
averages.loc[('WIND TOTAL',),:]
```

But now we can look at other sources

[ ]:
```
averages.loc[('SMALL HYDRO',),:]
```

Let's plot:

[ ]:
```
import matplotlib.pyplot as plt
```

[ ]:
```
plt.plot(averages.loc[('SMALL HYDRO',),:]);
```

[ ]:
```
plt.plot(averages.loc[('GEOTHERMAL',),:]);
```

[ ]:
```
plt.plot(averages.loc[('SOLAR PV',),:]);
```