Chapter 18

Window Functions

Contents

1 Window Functions in Pandas

- Pandas has two functions, `expanding` and `rolling` which do SQL style windows aggregations, using a syntax similar to `groupby`.
- An important difference between how Pandas and SQL implement window functions is how sorting is done. In SQL you *never* assume that rows have any order and always apply an ORDER BY clause to sort the data. In Pandas, the sort order is set by operation and you assume that it hasn't change when additional operators are applied. In other words, when we user window functions in SQL we set the row order via the window function but, when we use Pandas, we sort the data ahead of time and assume that the data retains that order.
- The difference between the `rolling` and `expanding` operators is the length of the window under consideration. The `expanding` operator has a window which increases to the start of the DataFrame while the `rolling` operator goes a fixed number of rows behind.
- The `rolling` method has one required parameter, which is the window length. This is similar to setting the `ROWS BETWEEN` operator in SQL.
- The `rolling` method has a fixed window length and, by default, sets all rows which have less data than the window length to NaN.[1]
- Let's consider a simple example to show how this works. We will start by building a simple DataFrame (`df`), which has two columns.

```
>>> d_1 = pd.DataFrame({'c1': [0, 1, 2, np.nan, 4], 'c2' : [0,1,2,3,4]})

>>> d_1
    c1  c2
0  0.0   0
1  1.0   1
2  2.0   2
3  NaN   3
4  4.0   4
```

- Just like `groupby` we use the `rolling` operator on the DataFrame. In this case we are going to choose a window length of two to create a rolling object:

```
>>> x_1 = d_1.rolling(2)

>>> type(x_1)
<class 'pandas.core.window.rolling.Rolling'>
```

- And, just like `groupby` we take this object and apply aggregations to it, using the syntax we have learned before.

  Apply function directly:

[1] This is different than SQL which fills in NULL values when the window length is less than the number of rows.

```
>>> x_1.mean()
    c1   c2
0  NaN  NaN
1  0.5  0.5
2  1.5  1.5
3  NaN  2.5
4  NaN  3.5
```

`agg` with list:

```
>>> x_1.agg('mean')
    c1   c2
0  NaN  NaN
1  0.5  0.5
2  1.5  1.5
3  NaN  2.5
4  NaN  3.5
```

`agg` with dict:

```
>>> x_1.agg( {'c1' : ['mean'], 'c2' : ['mean']})
    c1   c2
  mean mean
0  NaN  NaN
1  0.5  0.5
2  1.5  1.5
3  NaN  2.5
4  NaN  3.5
```

Looking at the above, in the first row, both columns have returned NaN. This is because we have set the window size to 2 and, by default, this means that any window of length less than two is set to NaN. We also see that there are two NaN's in the columns `c1`. This is because NaN added to any other number returns NaN.

- We can change the number of observations required to get a response using the `min_periods` argument:

```
>>> d_1.rolling(2, min_periods=1).mean()
    c1   c2
0  0.0  0.0
1  0.5  0.5
2  1.5  1.5
3  2.0  2.5
4  4.0  3.5
```

Note that this changes the result considerably. Since the first row now has a single observation it no longer returns NaN. Surprisingly, even the row with index 3 now has a value since there is one non-NaN value!

- To partition our data, we mix our `rolling` command with the `groupby` operator. In the following command we are going to only look at *inbound* traffic for the sake of simplicity.

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .groupby('plaza')
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
              vehiclescash  vehiclesez
plaza
1     103440           NaN         NaN
      103442           NaN         NaN
      103444        1855.0  558.666667
      103446        1976.0  580.333333
      103448        1806.0  479.000000
```

- Once again, remember that the sort order is set via code and should not be assumed.
- Pandas accumulates distinct values together – even if they are not connected within the original DataFrame. In the second example above, despite the plaza being after the hour column in the sort order this does not mean that multiple plazas are generated per hour. Since the `groupby` is on plaza this means that all similar values, independent of their row order are placed together.
- Take a look at what is returned in the example above and, specifically, what is being returned as the index. Since there was no index in the DataFrame before `rolling` was applied, the command keeps the original `RangeIndex` that was in the DataFrame! This is so that we could merge it back to the DataFrame before the `rolling` command.
- Alternatively, we could have moved our identifying columns into an index *before* specifying the `rolling` command so that we could merge it back onto the original DataFrame:

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .set_index(['plaza', 'mtadt', 'hr'])
            .groupby('plaza')
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                           vehiclescash  vehiclesez
plaza plaza mtadt      hr
1     1     2010-01-01 0            NaN         NaN
                       1            NaN         NaN
                       2         1855.0  558.666667
                       3         1976.0  580.333333
                       4         1806.0  479.000000
```

which would yield two `plaza` columns. However, just turning off `as_index` in the `groupby` won't change this issue:

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .set_index(['plaza', 'mtadt', 'hr'])
            .groupby('plaza', as_index=False)
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                          vehiclescash  vehiclesez
plaza plaza mtadt      hr
1     1     2010-01-01 0            NaN         NaN
                       1            NaN         NaN
                       2         1855.0  558.666667
                       3         1976.0  580.333333
                       4         1806.0  479.000000
```

Instead you need to have your index set to the returning variables you care about. Note that the `as_index` has no effect on what gets returned in this situation, as the `rolling` command will put `plaza` into the index no matter what.

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .set_index(['mtadt', 'hr'])
            .groupby('plaza', as_index=False)
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                     vehiclescash  vehiclesez
plaza mtadt      hr
1     2010-01-01 0            NaN         NaN
                 1            NaN         NaN
                 2         1855.0  558.666667
                 3         1976.0  580.333333
                 4         1806.0  479.000000
```

- So what have we learned:
  - The `rolling` command will take whatever is in the index and pass it through to the resultant DataFrame.
  - The `rolling` command will add whatever groupby column appears *as an index*, no matter what options you put in the `groupby` function.
  - Make sure that your DataFrame is *sorted* before applying the `rolling` operation.
- The other command used when doing window functions is the `expanding` operator. This operator calculates the aggregation back to the beginning of the frame in question, rather than based on a fixed window size.
- For example, if we want to return a running sum we could do the following and, we could verify that the changeover happens correctly:

```
>>> d_1 = (dfMTA
           .sort_values(['plaza', 'mtadt', 'hr'])
           .groupby('plaza')
           .expanding().agg({'vehiclesez' : 'sum'})
          )

>>> d_1.iloc[122975:122980]
                vehiclesez
plaza
1     1163399  147065302.0
2     206928          457.0
      206929          986.0
      206930         1526.0
      206931         2273.0
```

2 Some gotchas

Adding Back

- These types of functions are *very* easy to use in ways that cause problems.
- The biggest reason for this is that to run these commands the indexes have to be set *just right*.
- After running these commands we then want to put this data back into our original DataFrame, but this means then either changing the original DataFrame to conform with the result of our operation OR changing the result of our operation so that it conforms to our original DataFrame.
- In either case it is easy to end up in a place where functions do not return an error – but also aren't doing exactly what you want. The commands below are one way that we can take a DataFrame, do our aggregation functions and then add them back to our original DataFrame. Note the complexity required to make sure that the indexes align properly.

```
>>> d_1 = (dfMTA
            .set_index(['plaza', 'mtadt', 'hr', 'direction'])
        )

>>> d_2 = (d_1
        .reset_index(['plaza', 'direction'])
        .sort_values(['plaza', 'mtadt', 'hr'])
        .groupby(['plaza', 'direction'])
        .rolling(3)
        .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'})
        .reset_index()
        .set_index(['plaza', 'mtadt', 'hr', 'direction'])
        ).copy()

>>> d_1.loc[:, 'rcash'] = d_2.loc[:, 'vehiclescash' ]

>>> d_1.loc[:, 'rez'] = d_2.loc[:, 'vehiclesez' ]

>>> d_1.head()
                              vehiclesez  vehiclescash  rcash         rez
plaza mtadt      hr direction
1     2015-11-28 0  I                 477           205  817.0  653.333333
                    O                 486           252  998.0  694.333333
                 1  I                 350           171  646.0  499.666667
                    O                 307           182  797.0  509.333333
                 2  I                 280           133  509.0  369.000000
```

Offsetting

- There are no options within `rolling` or `expanding` to offset the data in some way.
- To do this we have to use the `shift` operator.

3 Reshaping Data: Transpose, Stack and Unstack

- In this section we look at the three commonly used commands for reshaping data between wide- and long-formats: `transpose`, `stack` and `unstack`.
- These operations strongly rely on indexes on both rows and columns. My common workflow with these operations is:

  1. Realize that I need to reshape the data.
  2. Figure out what index I need.
  3. Create index.
  4. Reshape data.
  5. Drop the index.

  I don't use indexes that much, preferring to leave the data "raw", rather than in named index columns. Because of this pattern, when I do need to reshape I have to define the appropriate indexes. This is a bit backward, but my preference is to avoid the complexity of indexes.

- In the simplest case to reshape data we can simply "transpose" it using the operator `T`. Let's look at the following example:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & (dfMTA.loc[:, 'direction'] == 'I')
    & (dfMTA.loc[:, 'plaza']==1),
    ['hr', 'vehiclesez', 'vehiclescash']]
    .reset_index(drop=True))

>>> d_1.head()
   hr  vehiclesez  vehiclescash
0   0         669           315
1   1        1085           426
2   2         922           426
3   3         767           450
4   4         724           429
```

We have three columns of data and we wish to make it wide. There are two options for this data: one is that we have "hr" as a column index or we just have "hr" as a row. We can do either by choosing to set an index or not:

1. **Pure Transpose:** Swap everything.

```
>>> d_1.T
                0     1    2    3    4    5   ...    18    19   20   21   22   23
hr              0     1    2    3    4    5   ...    18    19   20   21   22   23
vehiclesez    669  1085  922  767  724  616   ...  1098  1107  971  844  783  626
vehiclescash  315   426  426  450  429  331   ...   489   482  378  369  344  283

[3 rows x 24 columns]
```

2. **Transpose with an Index:** Create a column index based on hour.

```
>>> d_1.set_index('hr').T
hr             0     1    2    3    4    5   ...    18    19   20   21   22   23
vehiclesez    669  1085  922  767  724  616   ...  1098  1107  971  844  783  626
vehiclescash  315   426  426  450  429  331   ...   489   482  378  369  344  283

[2 rows x 24 columns]
```

   Looking at the result there are only two rows this time since "hr" has been turned into a column index.

- To swap the data back to the original form use the `T` command again.
- Transpose works when you wish to reshape the *entire* DataFrame. Most of the time, however, that operation is too severe and you only wish to make some of the information change shape.
- The first command `stack` takes data which is "wide" and makes it long while `unstack` returns the data to its wide format. Let's look at an example, using the MTA data:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & (dfMTA.loc[:, 'direction'] == 'I')
    & ((dfMTA.loc[:, 'plaza']==1) | (dfMTA.loc[:, 'plaza'] == 2)),
    ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
    .reset_index(drop=True)
    .set_index(['plaza', 'hr'])
    .unstack('plaza')
    )

>>> d_1.head()
      vehiclesez      vehiclescash
plaza          1    2            1    2
hr
0            669  554          315  160
1           1085  799          426  259
2            922  670          426  320
3            767  518          450  187
4            724  423          429  180
```

- We created a dataset with four columns: plaza, hr, vehiclesez and vehiclescash. We then use unstack to take this "long" data and turn it "wide" along the plaza dimension. The resulting DataFrame will have 24 rows and four columns.
- We can undo this command by using `stack`:

```
>>> d_1.stack('plaza').head()
          vehiclesez  vehiclescash
hr plaza
0  1             669           315
   2             554           160
1  1            1085           426
   2             799           259
2  1             922           426
```

As you can see we have moved plaza from the column index back as a row index. The only difference between this and the original DataFrame is the order of the index, which we could remove with `reset_index`.

- This might seem like magic, but lets think through the operation a bit and see if we can make sense of it. First, when we stack a DataFrame all columns with the same values are treated the same in the resulting DataFrame. This makes the reshape that much easier to conceptualize: all examples of plazas with the same number are going to have the number when we stack.
- The `unstack` operation also only works if the index that is set is **unique for each row.** By doing this, there is no way to have a conflict on the reshape.
- If we make the data wide by `unstack`, there may not be values present in all varieties of each index value. The `stack` operation, on the other hand, does not create any new data, so missing values won't be created.
- To use these operations its important to consider the following:
  - What values do you want in the new rows and columns: Are they unique? If not, stop.

- Once you have identified which values are moving, determine what is a value and what should be in the index.
- Set the index
- Call `stack` or `unstack` with the appropriate variable, from the index, selected.

- Note that you can do multiple values in your reshaping by providing a list. Consider the following:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & ((dfMTA.loc[:, 'plaza']==1) | (dfMTA.loc[:, 'plaza'] == 2)),
    ['plaza', 'hr', 'vehiclesez', 'direction', 'vehiclescash']]
    .reset_index(drop=True)
    . set_index(['plaza', 'hr', 'direction'])
    .unstack(['plaza', 'direction'])
    )

>>> d_1.head()
          vehiclesez                  vehiclescash
plaza              1         2                 1         2
direction          I    O    I     O           I    O    I    O
hr
0                669  552  554   760         315  300  160  241
1               1085  896  799  1123         426  437  259  357
2                922  747  670   933         426  447  320  360
3                767  694  518   728         450  407  187  257
4                724  577  423   586         429  369  180  188
```

- Returning to the above, we can also do a "semi" stack:

```
>>> d_1.stack('plaza').head()
          vehiclescash      vehiclesez
direction            I    O          I     O
hr plaza
0  1               315  300        669   552
   2               160  241        554   760
1  1               426  437       1085   896
   2               259  357        799  1123
2  1               426  447        922   747
```

4 A Bunch of stuff to clean up

- You can see this in the below (no idea what we are talking about)
- When using `expanding` or `rolling` keep in mind that the DataFrame returned does *not* have a clean index system. Continuing with the above example:

```
>>> d_1.index.names
['plaza', 'mtadt', 'hr', 'direction']
```

Unexpected! There are two levels of the index: one generated from the plaza `groupby` and another with the name "None". Even if we decide to stop the index creation with the `groupby` we will end up with an unexpected result:

```
>>> d_2 = (dfMTA
    .sort_values(['plaza', 'mtadt', 'hr'])
    .groupby('plaza', as_index=False)
    .expanding().agg({'vehiclesez' : 'sum'})
    )

>>> d_2.index.names
['plaza', None]
```

Comparing the above, we see that both, dfMTAC and dfMTAC2 have an additional index column:

```
>>> d_1.head()
                                vehiclesez  vehiclescash  rcash         rez
plaza mtadt      hr direction
1     2015-11-28 0  I                 477           205  817.0  653.333333
                    O                 486           252  998.0  694.333333
                 1  I                 350           171  646.0  499.666667
                    O                 307           182  797.0  509.333333
                 2  I                 280           133  509.0  369.000000

>>> d_2.head()
              vehiclesez
plaza
1     103440       415.0
      103441       801.0
      103442      1503.0
      103443      2037.0
      103444      2596.0
```

- This additional index column has implications for how we combine this data with other DataFrames, as we will see below.

5 Combining with the original DataFrame

- In the previous examples we generated a new Series or DataFrame which contained the data that we were interested in. Frequently we wish to combine this new data with the DataFrame that generated it and, sadly, this can be difficult as we need to create the column and then somehow put it back on

the original dataset.[2]

- There are a few different possibilities when doing this:

  1. `rolling` or `expanding` without a `groupby`.
  2. `rolling` or `expanding` with a `groupby` by creating an index.
  3. `rolling` or `expanding` with a `groupby` by using an already present index.

  We will go over each in the section below.

Without a groupby

- When there is no `groupby` we simply compute the `expanding` or `rolling` values, reset the index and then select the column and join back on:

```
>>> d_1 = dfMTA.copy()

>>> d_1.loc[:, 'newcol'] = (d_1
                                .expanding()
                                .agg({'vehiclescash' : 'sum'})
                                .reset_index()
                                .loc[:, 'vehiclescash'])

>>> d_1.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2015-11-28   0         I         477           205   205.0
1      1 2015-11-28   0         O         486           252   457.0
2      1 2015-11-28   1         I         350           171   628.0
3      1 2015-11-28   1         O         307           182   810.0
4      1 2015-11-28   2         I         280           133   943.0
```

- In the case where we want to sort the data beforehand, it is import to `sort_values` as well as `reset_index` on the original DataFrame to make sure that everything stays aligned:

```
>>> d_1 = dfMTA.sort_values(['mtadt', 'hr']).reset_index(drop=True).copy()

>>> d_1.loc[:, 'newcol'] = (d_1
                .expanding()
                .agg({'vehiclescash' : 'sum'})
                .reset_index()
                .loc[:, 'vehiclescash'])

>>> d_1.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2010-01-01   0         I         415           474   474.0
1      1 2010-01-01   0         O         386           412   886.0
2      2 2010-01-01   0         I         457           290  1176.0
3      2 2010-01-01   0         O         529           321  1497.0
4      3 2010-01-01   0         I         701           406  1903.0
```

[2] I'm really open to being wrong on this, but after spending a significant amount of time on this, I haven't seen a consistent solution outside what is shown here.

Note that the only difference between the two previous code blocks is the `sort_values` and `reset_index` commands.

With a GroupBy and Creating an Index

- Let's say that we don't have an obvious set of index columns to use, but we still wish to use a `groupby` with a window function. In this case we need to create an index.
- Consider the following situation where we want to calculate the running sum of inbound cars over the entire DataFrame, but partitioned by plaza:

```
>>> d_1 = (dfMTA
           .loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
           .sort_values(['plaza', 'mtadt', 'hr'])
           .reset_index(drop=True)
          )

>>> d_1.index
RangeIndex(start=0, stop=613608, step=1)
```

At this stage we have set up our original dataset to be sorted correctly and created a new integer index. The reason for the `drop=True` line is to prevent the original index from being placed in the DataFrame.[3]

We now take this DataFrame and create our running sum, making sure to start from the sorted DataFrame:

```
>>> d_2 = (d_1
           .groupby('plaza', sort=False)
           .expanding()
           .agg({'vehiclescash' : 'sum'})
           )

>>> d_2.head()
          vehiclescash
plaza
1     0          474.0
      1         1191.0
      2         1855.0
      3         2450.0
      4         2997.0

>>> d_2.index.names
['plaza', None]
```

Looking at the above, we can see that the index is no longer a `RangeIndex` and has changed! Meaning that we probably can't merge it back onto the original DataFrame without some modification.

- Note also that we included the option "sort=False" in our `GroupBy`. We did this because we want to make sure that this method doesn't change the order of the data. Since we know that the order is going to be stable, we reset the index:

[3]The original index was also an RangeIndex, but since we dropped all of the outbound rows as well as sorted the DataFrame, the original index does not exist in the proper form.

```
>>> d_2.loc[ :, 'runningsum'] = d_2.reset_index().loc[:, 'vehiclescash']

>>> d_2.head()
          vehiclescash  runningsum
plaza
1     0          474.0         NaN
      1         1191.0         NaN
      2         1855.0         NaN
      3         2450.0         NaN
      4         2997.0         NaN
```

- Combining this all together into two lines:

```
>>> d_1 = (dfMTA
           .loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
           .sort_values(['plaza', 'mtadt', 'hr'])
           .reset_index(drop=True)
         )

>>> d_1['runningsum'] = (d_1
                         .groupby('plaza', sort=False)
                         .expanding()
                         .agg({'vehiclescash' : 'sum'})
                         .reset_index(drop=True)
                         .loc[:, 'vehiclescash']
                        )
```

With a GroupBy using an index

- Alternatively, we can rely on an unique set of index column if they are present in the DataFrame. Redoing the example above:

```
>>> d_1 = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
           .sort_values(['plaza', 'mtadt', 'hr'])
           .set_index(['plaza', 'mtadt', 'hr'])
         )

>>> d_1.loc[:, 'runningsum'] = (d_1
           .reset_index('plaza')
           .groupby('plaza', as_index=False, sort=False)
           .expanding()
           .agg({'vehiclescash' : 'sum'})
           .reset_index()
           .set_index(['plaza', 'mtadt', 'hr'])
           .loc[:, 'vehiclescash'])
```

- Looking at the above, we `set_index` on the original DataFrame and then set it again on the created dataset.

- IMPORTANT: A caveat to the above is that if the index columns are not unique then we can run into situations where the data is sorted differently in each and thus the merge may result in incorrect results. This method should *only* be used if there is a set of columns which uniquely define a row.

6 Moving the Window

- A limitation in how Pandas implement window functions is that they do not naturally have the ability to move the window – e.g. offset it by a number of rows.
- For example, lets say that I want to know the maximum value of a column up to, but not including the current row? This could occur because I want to know if the current row is higher than the previous maximum value. It's easy enough to calculate the maximum up to, and including the current row, but moving that window back one requires an additional operation.
- One way of doing this is to use the `shift` operator to move the data after the calculation occurs, such as in the example below which calculates the maximum vehicles which use an cash up to, but not including the current row (only in the inbound direction)

```
>>> d_1 = (dfMTA.loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
           .sort_values(['plaza', 'mtadt', 'hr'])
           .set_index(['plaza', 'mtadt', 'hr'])
          )

>>> d_1.loc[:, 'runningmax_no_current'] = (d_1
           .reset_index('plaza')
           .groupby('plaza', as_index=False, sort=False)
           .expanding()
           .agg({'vehiclescash' : 'max'})
           .reset_index()
           .set_index(['plaza', 'mtadt', 'hr'])
           .groupby('plaza', as_index=False)
           .shift(1)
           .loc[:, 'vehiclescash']
)

>>> d_1 = d_1.reset_index()
```

- The last line removes the index that we created.

7 Pivot / Melt

- While we won't cover it in this course, the `pivot` and `melt` commands are powerful way to reshape data.
- While they nearly map to `stack` and `unstack`, they do not require the use of an index.