Aggregating in Pandas

Justus Perlwitz

2017-12-26

Contents

1 1. Introduction

The other day I caught myself writing one too many `df.groupby()` in Pandas, and thought to myself: Isn't there a way I can simplify and reduce the amount of `df.groupby()` calls? What I group by hardly changes and only the operation that follows a group by varies from IPython cell to IPython cell. Aggregates shall come to the rescue.

Pandas DataFrames allow many useful calculations to be performed on them. One of them is the so called `.aggregate()` method. Using `.aggregate()`, a user can perform many calculations on a group by object or resampler at once. This can be quite handy in many situations and performs much faster than calculating all required aggregate values in separate steps. As a matter of fact, I was quite surprised while writing this article when I found out how much flexibility exactly a user can get while using `aggregate()`.

2 2. Setup

First things first and we do business as usual. We import `pandas` in order to create the DataFrames we use in our examples.

```
import pandas as pd
```

We want to define a DataFrame with a variety of illustrative data types.

The example that we come up with here is a list of fruits that our fictitious friends *Franz*, *Hans* and *Gerhard* have eaten in the last week. Furthermore, we note down whether our friends have actually liked the fruit or not.

So the columns are:

- `index`: Name of person
- `Fruit`: Fruit consumed
- `Satisfaction`: Satisfaction with consumed fruit
- `Weight`: Weight of the consumed fruit in grams

```
df = pd.DataFrame(
    [
        ['Apple', 'full', 100],
        ['Orange', 'none', 200],
        ['Pear', 'full', 300],
        ['Pear', 'partial', 100],
        ['Banana', 'full', 400],
        ['Banana', 'full', 300],
    ],
    columns=[
        'Fruit',
        'Satisfaction',
        'Weight'
    ],
    index=[
        'Franz',
        'Gerhard',
        'Gerhard',
        'Hans',
        'Hans',
        'Hans',
    ],
)
df.Satisfaction = df.Satisfaction.astype('category')
df
```

Output:

	Fruit	Satisfaction	Weight
Franz	Apple	full	100
Gerhard	Orange	none	200
Gerhard	Pear	full	300
Hans	Pear	partial	100
Hans	Banana	full	400
Hans	Banana	full	300

Are you as excited as I am to learn the first few magic incantations of `.aggregate()`? Let's move on and start working on the DataFrame.

3 3. Aggregate Heaven

3.1 3.1. `count`

Using the `count` aggregation, we can retrieve the amount of rows that are counted in a group by expression. This is not particularly exciting, but will make the following steps a little bit clearer.

First we turn towards our Pandas DataFrame and try to count the number of fruits that each person has consumed.

For this, we have to group the DataFrame by its index as the index contains the person names. In order to group by the DataFrame's index, we can simply group by using `.groupby(level=0)`. We use `level=0` to indicate that we want to group by the DataFrames index. We could use `.groupby(df.index)` instead, but this way we can

leave it implicit and save us a bit of typing. See the docs for some more information on how `groupby()` can be invoked.

```
df.groupby(level=0)
```

Output:

```
<pandas.core.groupby.DataFrameGroupBy object at 0x10b88ada0>
```

Calling `.groupby()` by itself does not do much. We need to perform an aggregation on it in order to get a meaningful result. Let's do the actual calculation now and see what we get. We call `.aggregate('count')` on the `DataFrameGroupBy` object.

```
df.groupby(level=0).aggregate('count')
```

Output:

	Fruit	Satisfaction	Weight
Franz	1	1	1
Gerhard	2	2	2
Hans	3	3	3

Our informed reader will surely let us know, that `count()` can also be invoked more directly and will give us the same result:

```
df.groupby(level=0).count()
```

Output:

	Fruit	Satisfaction	Weight
Franz	1	1	1
Gerhard	2	2	2
Hans	3	3	3

That is absolutely true, and in simple situations just using `.count()` on a group by object will be much easier to understand. On the other hand, `aggregate()` allows us to do one nifty thing that wouldn't be possible easily any other way. It lets us perform several calculations at once and neatly formats the results using nested columns. As a matter of fact, Pandas gives us a lot of freedom in how exactly we want the aggregates to look like. Let's get to a slightly more complicated example to illustrate the true flexibility right away.

3.2 3.2. Aggregator Functions

If we would like to find out what the most frequent value in a column is, we need to use a custom aggregator. Pandas does not include a method for this out of the box, so we can either define a function or a lambda to give us the desired result.

In our case we would like to simply define a lambda to facilitate this calculation:

```
most_frequent = lambda s: s.value_counts().idxmax()
```

We can try calling `most_frequent` on the whole DataFrame and check the result. We use `.apply()` in this case to apply a function to every column in a DataFrame.

```
df.apply(most_frequent).to_frame()
```

Output:

	0
Fruit	Banana
Satisfaction	full
Weight	100

Let's put everything together and calculate the aggregate.

```
df.groupby(level=0).aggregate(lambda s: s.value_counts().idxmax())
```

Output:

	Fruit	Satisfaction	Weight
Franz	Apple	full	100
Gerhard	Pear	none	300
Hans	Banana	full	400

3.3 3.3. Combining Aggregates

Let us find out, what

- the first and last fruit that each person has eaten is,
- while also counting the total amount of fruits consumed, and
- what the most frequently given satisfaction rating for each person is.

In order to do this, we first have to define an aggregate dictionary. It contains instructions on which calculations to perform on which column. It even allows defining custom aggregators using Python functions or lambdas.

The aggregate dictionary contains two entries corresponding to the two columns in the DataFrame. For the fruit column, we add 3 desired aggregates in the form of a list:

```
['first', 'last', 'count']
```

And for the satisfaction column, we add a named aggregator function by specifying a list containing one tuple with the `most_frequent` lambda that we have defined before:

```
[('most_frequent', most_frequent)]
```

Attaching a name to the aggregator is useful in our case, since it will tell Pandas what to name the result column after calculating the aggregates. Otherwise, using a lambda will lead to the resulting column to also be called `lambda`.

The full dictionary is defined as follows:

```
aggregate = {
    'Fruit': [
        'first',
        'last',
        'count',
    ],
    'Satisfaction': [
        ('most_frequent', most_frequent),
    ]
}
```

Let's run the actual calculation then and see what the result looks like.

```
df.groupby(level=0).aggregate(aggregate)
```

Output:
Fruit
Satisfaction
first
last
count
most_frequent
Franz
Apple
Apple
1
full
Gerhard
Orange
Pear
2
none
Hans
Pear
Banana
3
full

We can see that the DataFrame's columns are formatted and calculated exactly as we have defined in the dictionary used for our `.aggregate()` call.

4 4. Side Note: Data Types

4.1 4.1. Pandas dtypes

Note that when running aggregates, the result type of an aggregated column can be different from the source column. To get back to our `count()` example, observe the following data types for the source DataFrame:

```
df.dtypes.to_frame()
```

Output:

	0
Fruit	object
Satisfaction	category
Weight	int64

- The `Fruit` column contains data of the type `object`, which is the way Pandas stores Python `str` (string) data in columns,
- `Satisfaction` contains category data, as indicated previously, and
- `Weight` contains `int64` data.

Now, observe one more time what happens when the count aggregate is retrieved on the same DataFrame.

```
df.groupby(level=0).aggregate('count')
```

Output:

	Fruit	Satisfaction	Weight
Franz	1	1	1
Gerhard	2	2	2
Hans	3	3	3

We look at the `.dtypes` attribute of our aggregate.

```
df.groupby(level=0).aggregate('count').dtypes.to_frame()
```

Output:

	0
Fruit	int64
Satisfaction	int64
Weight	int64

This reveals that `count` will be stored as `int64` unlike the original columns, which had the data types `object` and `category`. Pandas returns a different data type after performing aggregates, depending on what the result of a calculation is.

In the case of `count` data, integers such as `int64` (a 64 bit signed integer) are the sensible choice for storing them. NumPy `ndarray` is the internal storage format used for almost all data in Pandas, except for indices. More info on `ndarray` can be found here. The reason why Pandas uses `ndarray` objects is that it is a space and time efficient way of storing fixed length lists of numbers.

We can easily take a look at the underlying data type of a Pandas column by accessing the `.values` attribute.

```
type(df.Fruit.values)
```

Output:

```
numpy.ndarray
```

And simply evaluating `df.Fruit.values` reveals that it has the data type `object`, as is indicated by the `dtype` value.

```
df.Fruit.values
```

Output:

```
array(['Apple', 'Orange', 'Pear', 'Pear', 'Banana', 'Banana'], dtype=object)
```

`object` is used to store `str` (string) data in NumPy and Pandas. Unlike fixed-width integers, such as `int64`, string data cannot be stored efficiently inside a continuous NumPy `ndarray`. The `ndarray` instance in this case only contains a collection of pointers to objects, quite similarly to how a regular Python `list` works. The space efficiency characteristics of a `ndarray` instance with data type set to `object` are slightly better than just using `list`. We will prove this in the next subsection.

4.2 4.2. Memory Profiling

Our claim that NumPy ndarrays store strings more efficiently than Python lists can be verified using a memory profiler. Luckily, a PyPi package called `pympler` allows us to quickly measured memory usage by Python objects. We furthermore import NumPy to directly create NumPy arrays without requiring Pandas.

```
from pympler import asizeof
import numpy as np
```

We decide to create an `ndarray` containing `10**6` strings "hello".

```
numpy_hello = np.zeros(10 ** 6, dtype=object)
numpy_hello.fill('hello')
numpy_hello
```

Output:

```
array(['hello', 'hello', 'hello', ..., 'hello', 'hello', 'hello'], dtype=object)
```

We create an equivalent Python `list` that contains "hello" `10**6` times.

```
python_hello = ["hello" for _ in range(10 ** 6)]
# Print the first 3 items
python_hello[:3]
```

Output:

```
['hello', 'hello', 'hello']
```

Now for the evaluation: We print the size of the NumPy `ndarray` in bytes using Pympler's `asizeof` method.

```
asizeof.asizeof(numpy_hello)
```

Output:

```
8000096
```

To compare, we print the size of the Python `list`.

```
asizeof.asizeof(python_hello)
```

Output:

```
8697520
```

```
print("NumPy array size in relation to Python list size: {:2.2%}".format(
    asizeof.asizeof(numpy_hello) /
    asizeof.asizeof(python_hello)
))
```

Output:

```
NumPy array size in relation to Python list size: 91.98%
```

The NumPy array only takes 91.98% of the space required by the Python list, even though the data that they store is the same.

About Pympler

Note that Pympler prints accurate sizes by traversing the object and summing up attribute sizes for all descendant attributes. This is unlike `sys.getsizeof`, which does not perform a deep traversal of an object and its attributes, especially for user defined classes.

We can easily see the difference:

```
from sys import getsizeof

getsizeof(python_hello)
```

Output:

```
8697464
```

Now here the difference is minimal, but as soon as we nest objects even further, the difference of `sys.getsizeof` and `asizeof` becomes quite obvious:

```
getsizeof([[[]]])
```

Output:

```
72
```

```
asizeof.asizeof([[[]]])
```

Output:

```
208
```

As a bonus, we compare the previous two objects `numpy_hello` and `python_hello` to using a `tuple()` instead.

```
tuple_hello = tuple("hello" for _ in range(10 ** 6))
# Print the first 3 items
tuple_hello[:3]
```

Output:

```
('hello', 'hello', 'hello')
```

`tuple_hello` is smaller than the list object `python_hello`, but still bigger than `numpy_hello`:

```
asizeof.asizeof(tuple_hello)
```

Output:

```
8000104
```

From this we can safely conclude that Numpy `ndarray` is the most efficient way of storing fixed-size array data. This concludes our short excursion on Python and NumPy memory usage.

5 5. More About Aggregates

5.1 5.1. `nunique`

In Pandas, `nunique` counts the number of unique values in a column. We can apply this to the whole DataFrame and get a count of the unique fruit and satisfaction values:

```
df.nunique().to_frame()
```

Output:

	0
Fruit	4
Satisfaction	3
Weight	4

Furthermore, the method can also be applied on a group by object to retrieve the unique number of values per group. We see below the number of unique fruits and satisfactions that have been assigned to each person.

```
df.groupby(level=0).nunique()
```

Output:

	Fruit	Satisfaction	Weight
Franz	1	1	1
Gerhard	2	2	2
Hans	2	2	3

Now, `nunique` is also available in aggregates. The reason why we would use `nunique` in aggregates is if we want to retrieve multiple results for one group by expression at the same time. This can not only save us some typing, but can potentially also save us some computational time, as a group only needs to be created once and each operation can then be applied to it one after another.

```
df.groupby(level=0).agg('nunique')
```

Output:

	Fruit	Satisfaction	Weight
Franz	1	1	1
Gerhard	2	2	2
Hans	2	2	3

5.2 5.2. Using Aggregate Results

Having run this, we now know that Franz has only consumed one type of fruit. Hans is the champion of trying out many types of fruits. We can then use this value to compare it to the total count of fruits consumed. This allows us to calculate a variety score for each person. We define a variety of 100 % as a fruit consumption pattern in which a new fruit is tried every time.

```
fruit_counts = df.groupby(level=0).Fruit.aggregate(['nunique', 'count'])
fruit_counts
```

Output:

	nunique	count
Franz	1	1
Gerhard	2	2
Hans	2	3

We decide to neatly display the variety counts with a quick `.apply` call in which we format the resulting floats using a Python format string.

```
(
    fruit_counts['nunique'] / fruit_counts['count']
).apply(
    lambda v: "{:.2%}".format(v)
).to_frame('Variety')
```

Output:

	Variety
Franz	100.00%
Gerhard	100.00%
Hans	66.67%

5.3 5.3. All Built-In Aggregates

What follows now is a list of all the built-in aggregates that I could find in Pandas.

These are the aggregates that work with **all** data types. To save some space we limit ourselves to the Fruit column.

```
df.groupby(level=0).Fruit.aggregate([
    'count',
    'min',
    'max',
    'first',
    'last',
    'nunique',
```

```
]).applymap(
    lambda v: v if isinstance(v, str) else "{:d}".format(v)
)
```

Output:

	count	min	max	first	last	nunique
Franz	1	Apple	Apple	Apple	Apple	1
Gerhard	2	Orange	Pear	Orange	Pear	2
Hans	3	Banana	Pear	Pear	Banana	2

Here are the data types that work with **numerical** data types.

```
df.groupby(level=0).aggregate([
    'mean',
    'std',
    'var',
    'median',
    'prod',
    'sum',
    'mad',
    'sem',
    'skew',
    'quantile',  # 50 % quantile
]).applymap(
    lambda v: v if isinstance(v, str) else "{:.2f}".format(v)
)
```

Output:

Weight
mean
std
var
median
prod
sum
mad
sem
skew
quantile
Franz
100.00
nan
nan
100.00
100.00
100.00
0.00
nan
nan
100.00
Gerhard
250.00
70.71
5000.00
250.00

60000.00
500.00
50.00
50.00
nan
250.00
Hans
266.67
152.75
23333.33
300.00
12000000.00
800.00
111.11
88.19
-0.94
300.00

6 6. Aggregate performance

Let's put the claim to the test that `.aggregate()` calls are faster than aggregating directly on a group by, and find out in which circumstances this statement holds. We would first like to find out which is faster: Multiple aggregates in one `.aggregate()` call, or separate aggregates applied to a new group by object each time? We choose to aggregate `min` and `max` on our DataFrame.

```
%%timeit
df.groupby(df.index).min()
df.groupby(df.index).max()
```

Output:

```
26.1 ms ± 819 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
df.groupby(df.index).aggregate(['min', 'max'])
```

Output:

```
9.02 ms ± 75.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We can clearly see that calling `.aggregate()` performs much faster when multiple aggregate values are needed compared to aggregating twice on a `.groupby()`. Furthermore, we can observe in the next two cells that there is hardly any difference when only one aggregate value is required. In this case, the shorter amount of code should win the contest, since it simply requires less typing.

```
%%timeit
df.groupby(df.index).min()
```

Output:

```
13.6 ms ± 945 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
df.groupby(df.index).aggregate('min')
```

Output:

```
16.5 ms ± 4.14 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

7 7. Conclusion

I hope I was able to demonstrate a few use cases for Pandas aggregates. It is certainly nice to be able to save some typing and have better performance when dealing with multiple aggregate calculations, especially in a group by setting.

I certainly could not do without `.aggregate()`, as it saves me a lot of time when typing out IPython notebooks.