GroupBy Overhaul Documentation

Release 0.1

Will Ayd

Jul 08, 2018

CONTENTS

CHAPTER

ONE

PROPOSAL

To make the GroupBy implementation more scalable, developer-/user friendly and predictable I propose the following:

- Removal of *apply* method
- Removal of *DataFrameGroupBy* and *SeriesGroupBy* classes
- Explicit default column naming
- Removal of *axis* argument

The pros/cons and potential design considerations of each are outlined in the below sections.

CHAPTER

TWO

REMOVAL OF APPLY

2.1 Reasoning

apply is a "workhorse" function that accepts an arbitrary UDF, applies it to a group and then theoretically pieces the results back together the way the end user wants. While that may happen in a variety of cases, the ones that do typically violate the "There should be one obvious way to do it" mantra and introduce performance discrepancies that may be non-obvious to the general user of pandas. To illustrate:

```
df.groupby('key')['value'].sum()
```

Will provide the exact same value as

```
df.groupby('key')['value'].apply(sum)
```

Though the former is order of magnitudes faster.

Counter-intuitively, running the above without an explicit selection of columns will yield two different results.

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame([['foo', 'baz']], columns=['a', 'b'])

In [3]: df
Out[3]:
     a    b
0  foo  baz

In [4]: df.groupby('b').sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:
       a
b
baz  foo

In [5]: df.groupby('b').apply(sum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]:
       a    b
b
baz  foo  baz
```

If we decided that the default groupby operations should vary from their builtin counterparts we would introduced another nuanced inconsistency for users. Specifically, if we decided to raise a `ValueError` when summing strings instead of concatenating them (we do this for *rank*), the above example would provide a vastly different result between the two methodologies.

Perhaps of more importance, it is arguably not feasible to infer the users' desired output from the application of an anonymous function. The current implementation is also buggy, as you can see in the difference of frames below:

```
In [6]: df = pd.DataFrame(np.zeros((3, 3)), columns=['foo', 'bar', 'baz'])

In [7]: df['key'] = list('abc')

# No MultiIndex here
In [8]: df.groupby('key').apply(lambda x: x)
Out[8]:
   foo  bar  baz key
0  0.0  0.0  0.0   a
1  0.0  0.0  0.0   b
2  0.0  0.0  0.0   c

# Now we get a MultiIndex?
In [9]: df.groupby('key').apply(lambda x: x.iloc[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
↪
     foo  bar  baz key
key
a    0.0  0.0  0.0   a
b    0.0  0.0  0.0   b
c    0.0  0.0  0.0   c
```

The inference required here adds a significant development burden and is not something that can reasonably be guaranteed for user functions. As an alternate, I believe that if we exposed an iterator over group objects to the users they can leave their UDFs untouched and they would not have to add safeguards against undesirable indexing inference that may occur.

2.2 Counter Arguments

In the past when issues have occurred within our code base *apply* has served as a fallback plan (see this comment). Some may view that "safety valve" as desirable.

There are no doubt many use cases where the return value of *apply just works*. This should happen if the UDF reduces to a scalar. I'd counter that by saying that *agg* would then be an equivalent use case and would again reinforce there being only "one way to do things," but this no doubt would require a behavioral change.

If we take away the implied reindexing behavior and replace with an iterator users would have to store those results in say an OrderedDict and subsequently reconstruct their DataFrame. The cost of not guessing what the user wants is that they have to code more, and it may be duplicative in a lot of instances.

We support apply on other major objects like *DataFrame* and *Window*. I'd argue that the latter may have as many if not more nuances than the *GroupBy* object does (albeit with less usage), though the former is certainly stable. From a non-developers' perspective it could be difficult to appreciate *why* supporting this is fraught with peril.

CHAPTER

THREE

REMOVAL OF *DATAFRAMEGROUPBY* AND *SERIESGROUPBY*

3.1 Reasoning

We currently subclass for the various pandas objects (Panel included but ignored for brevity) though in quite a few instances our method of applying functions is to iterate over the columns of the calling object and apply the desired function, storing the result in an `OrderedDict` before reconstructing the desired object. With that being the case, the dimensions of the calling object should not be a major consideration point since we end up taking one dimensional slices and piece them back together at the end. The main reason for this is the strongly typed Cythonized functions.

As a result, the *SeriesGroupBy* and *DataFrameGroupBy* subclasses' main use is arguably to control the formatting of the output. Note the difference between the below calls:

```
In [10]: df = pd.DataFrame([['a', 1], ['b', 1]], columns=['key', 'val'])

In [11]: df.groupby('key')['val'].sum()
Out[11]:
key
a    1
b    1
Name: val, dtype: int64

In [12]: df.groupby('key')[['val']].sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:
     val
key
a      1
b      1
```

Though potentially desirable, the two classes here do add a surprising amount of complexity and the behaviour may not be completely intuitive to a casual user. Instead I would counter that we do away with both classes and simply guarantee that the output of the operation be a *DataFrame*.

3.2 Counter Arguments

For extremely wide DataFrames iterating over the slices of the caller may have a performance penalty. I would argue that if that were the case we can model the iteration in Python now and search for a Cythonized solution in the future that can better handle data with an arbitrary shapes, but on the flip side I'm not sure of any currently measurable limitations or test cases with the proposed solution.

This also changes behavior today where passing a *Series* will typically return a *Series*. If the user had a workflow where they assign the result of a *GroupBy* operation to another *DataFrame* they would have to squeeze the result first.

CHAPTER

FOUR

EXPLICIT DEFAULT COLUMN NAMING

There were quite a few discussions about this at the pandas sprint. For a simple use case, let's assume that we are grouping a *Series* with the name `A`. If we were to take the sum of that, the name of the returned object would still be `A`. If we did something like `.agg({'A': 'sum'})` we would get a MultiIndex column whose first level was `A` and second level was `sum`.

I propose that in all cases we use `f"{func} of {colname}"` as the returned label, which mimics the behavior of a tool like Excel. This makes it so that *string*, *dict* and *list* arguments to `.agg` have a very consistent and identifiable return value (i.e. `Sum of A`). It removes the need of having to custom-mangle or flatten your columns after a *dict* invocation, and makes it so that you don't have ambiguous column names between your caller and your returned object.

CHAPTER

FIVE

REMOVAL OF AXIS ARGUMENT

We admittedly don't have a ton of reported issues with *axis=1*, but I wonder if that's because there is limited usage for this in the first place. This type of analysis is not supported in many other analytics tools (at least none that I have encountered) and I believe it just adds unnecessary cruft to our code base.

In instances where users do use this functionality I would propose that they can just transpose -> operate -> transpose to get the equivalent result in a fashion that is better supported. I can certainly seeing that as an inconvenience but again for the sake of cleaning up our *GroupBy* module and establishing a better API I think it is still worth that cost.