owo what's this???

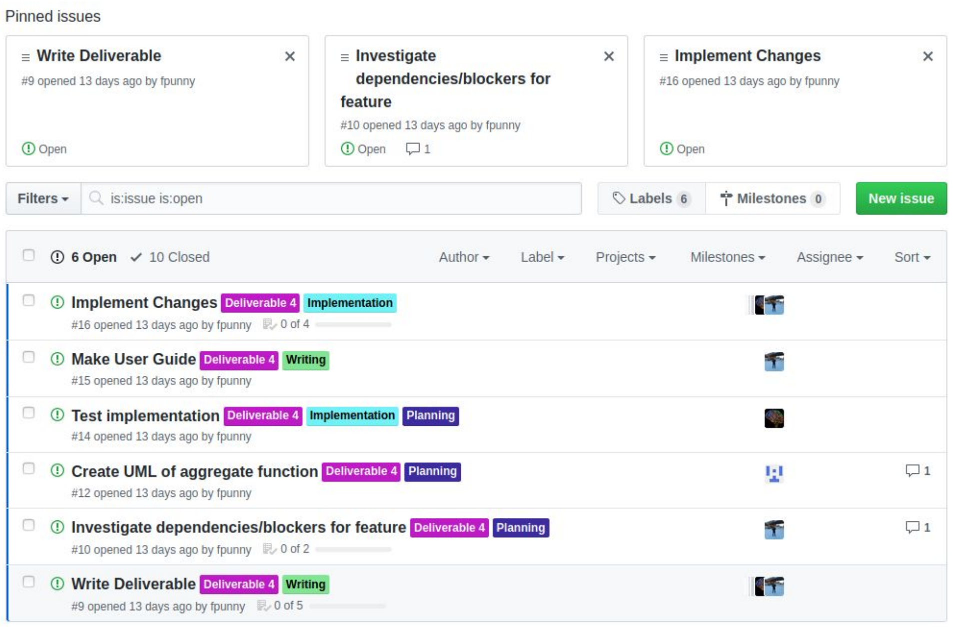
Team 24 - Deliverable 4

Table of Contents

The Feature

Overview

Deriving from what we proposed and argued from deliverable 3, the feature we tackled is an enhancement which allows for named aggregation with multiple columns - Issue 29268. Following our waterfall process, we broke the task into multiple processes and tasks as seen in our issue board and project on Github. We would now go over the main issues in chronological order to outline the requirements gathering step of our development process.


Pinned issues

Write Deliverable
#9 opened 13 days ago by fpunny
Open

Investigate dependencies/blockers for feature
#10 opened 13 days ago by fpunny
Open 1

Implement Changes
#16 opened 13 days ago by fpunny
Open

Filters is:issue is:open Labels 6 Milestones 0 New issue

6 Open ✓ 10 Closed Author Label Projects Milestones Assignee Sort

Implement Changes Deliverable 4 Implementation
#16 opened 13 days ago by fpunny 0 of 4

Make User Guide Deliverable 4 Writing
#15 opened 13 days ago by fpunny

Test implementation Deliverable 4 Implementation Planning
#14 opened 13 days ago by fpunny

Create UML of aggregate function Deliverable 4 Planning 1
#12 opened 13 days ago by fpunny

Investigate dependencies/blockers for feature Deliverable 4 Planning 1
#10 opened 13 days ago by fpunny 0 of 2

Write Deliverable Deliverable 4 Writing
#9 opened 13 days ago by fpunny 0 of 5

Investigate Functionality

Issue 11

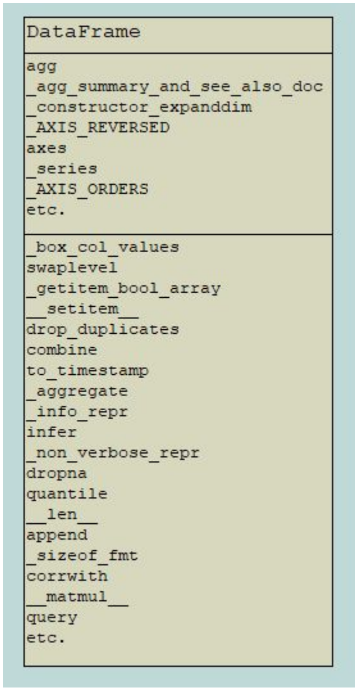
Before jumping to implementation strategies, we first investigated the functionality of aggregate. Being relatively new to the codebase, it was important we get an idea of the flow of the aggregation to ensure consistency in both functionality and integrity. From our investigations, we have deducted the following information.

- The files which aggregation uses are located in **core/groupby/generic.py**, **core/aggregation.py**, and **core/base.py**. The file **core/aggregation.py** is not very important and/or interesting as it's simply a utility for standard operations such as normalization and checking.

- Aggregations can be divided into two categories - 1D and 2D. These types are represented by SeriesGroupBy and DataFrameGroupBy. The types are created during the aggregation process using `self._gotItem()` from the base aggregation from core/base.py.
- When an aggregation is performed, all 2D operations would eventually be reduced to SeriesGroupBy since aggregations only support 1D operations (A blocker).
- The inputs for an aggregation are both unique and generic. The aggregation from core/base.py acts as a generic wrapper to orchestrate the correct aggregation and data types. As SeriesGroupBy and DataFrameGroupBy perform the input normalization and actual aggregation in Cython.

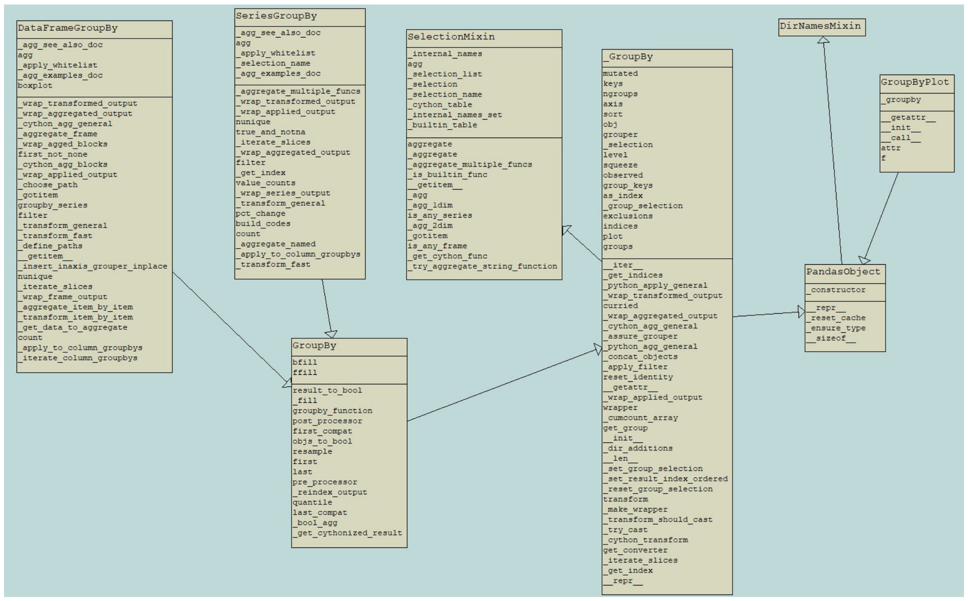
Generating UML

Issue 12

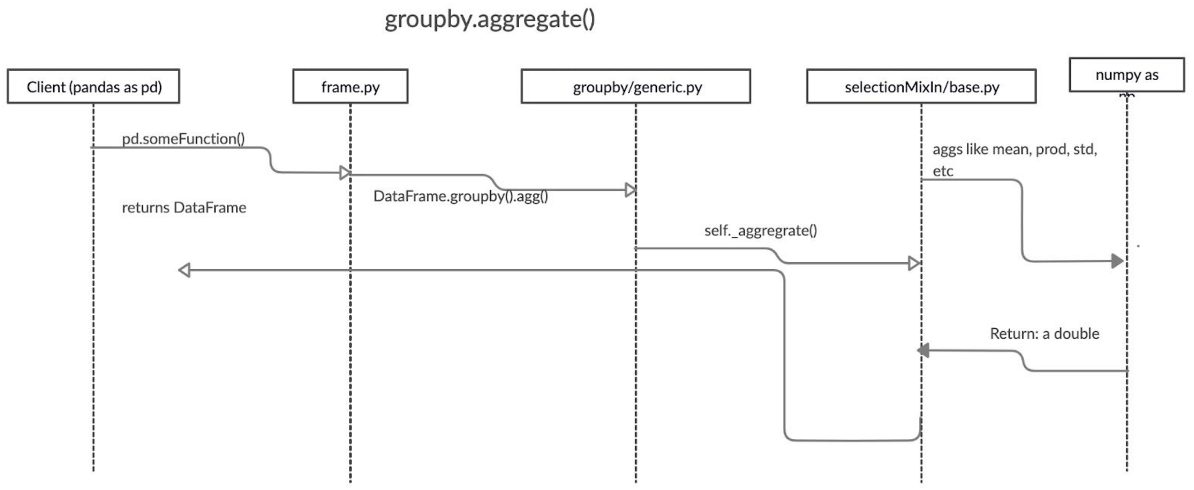


We constructed UMLs to model, and understand, the relevant components and objects are being manipulated. The data which aggregate is called upon comes from DataFrame, more specifically the array series from within.

From the data that is being aggregated in DataFrame, we found out that aggregate goes through the following modules, outlined in the diagram below, located in **Generic.py**, **Aggregate.py** and **Base.py**:



When the aggregate function is called, Pandas drives through multiple different modules to access and compile data from **Dataframe** and the arrays that are contained within. This data then goes through aggregation methods - such as max, min, mean - and/or lambda operations through **Base.py** with assistance from numpy.



Our Implementation

Overview

When implementing the feature, we have to consider two audiences - the user and developer. The user represents the API of the functionality, which affects how effective the user is able to use and understand the new functionality. Some questions we might have to ask ourselves include,

- If the user tries existing syntax, how effective and relatable is it?
- If the user tries a different syntax, how learnable is it? Would the user understand it?
- And finally, does this collide with other params? How consistent is it to similar functionalities?

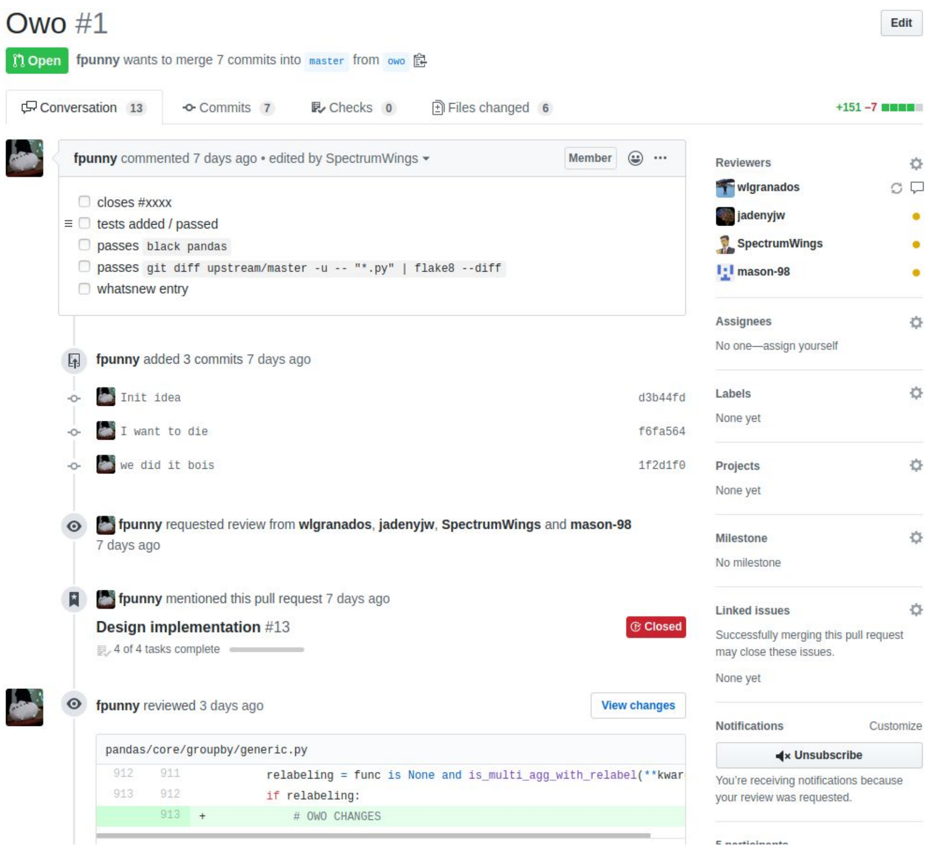
Implementation around only this issue is fairly simple. However considering this with the developer, creates the real challenge of this enhancement (explored further in the section). The developer represents performance and code consistency/reusability - which are main principles which the the contributors of Pandas strive for.

Hence in this section, we would go over the external and internal implementation. The external implementation would compare our API implementation ideas, exploring the pros and cons, and our rationale for the final implementation. As the internal implementation would explore our implementation within Pandas, the functionality itself, and explore how we follow the principles of Pandas - which is not perfect due to the nature of our functionality.

The Process

Our implementation of this feature is broken up in two parts - design implementation and implementation. This two step process is advantageous to us due to our lack of familiarity with the codebase and lack of skill as students. By having a pre-implementation and internal review, we are able to better transition and validate our plan with the actual implementation. As a result, we are able to produce a higher quality product for the official review to expedite the review process with the contributors.

To execute this, we created a PR to merge into our fork version of Pandas - which would only be visible by the team. The PR would allow for us to outline the exact code changes and a place for discussion on specific parts of the code. After this process, we created an official branch, issue-29268, with our changes to be merged into Pandas for review.


Owo #1

Edit

Open fpunny wants to merge 7 commits into `master` from `owo`

Conversation 13 Commits 7 Checks 0 Files changed 6 +151 −7

fpunny commented 7 days ago • edited by SpectrumWings Member

- [ ] closes #xxxx
- [ ] tests added / passed
- [ ] passes `black pandas`
- [ ] passes `git diff upstream/master -u -- "*.py" | flake8 --diff`
- [ ] whatsnew entry

fpunny added 3 commits 7 days ago

```
Init idea        d3b44fd
I want to die    f6fa564
we did it bois   1f2d1f0
```

fpunny requested review from wlgranados, jadenyjw, SpectrumWings and mason-98 7 days ago

fpunny mentioned this pull request 7 days ago

Design implementation #13 Closed

4 of 4 tasks complete

fpunny reviewed 3 days ago View changes

```
pandas/core/groupby/generic.py
912  911            relabeling = func is None and is_multi_agg_with_relabel(**kwar
913  912            if relabeling:
     913  +             # OWO CHANGES
```

Reviewers: wlgranados, jadenyjw, SpectrumWings, mason-98

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

Linked issues: Successfully merging this pull request may close these issues. None yet

Notifications Customize

Unsubscribe

You're receiving notifications because your review was requested.

External Implementation

From our investigation, we came up with 2 different ways to solve this problem - which we would call "the pipeline approach" and "native approach". We would go over both implementations and talk about the pros and cons of each. In conclusion, there is no perfect implementation and due to the nature of our feature - will result in some inconsistency.

The Pipeline Approach

The pipeline approach changes the way Pandas normally perform a named aggregation, which accepts a tuple of (string, string | lambda). Instead, we would create a new input, which is a list represented as [...args, lambda], where args is (string, string | lambda). The benefit of this approach is that it improves the reusability of aggregate, and code consistency of using normalize and consolidate. The side-effect of this approach is that we affect the user experience by introducing a brand new type of input. Another drawback to this approach is that it adds a fair amount of complexity in the checking and consolidation process - which is already fairly complex in big O and logic.

```
df.group("test").agg(
   example=[('a', "min"), ('b': "min"), lambda x: x['a'] - x['b']]
)
```

The Native Approach

The native approach, the approach we have taken, follows a similar format, like the example provided by the feature requester. The only difference however is that the first element of the tuple can now be a list of columns. The benefit of this approach is that the normalization and consolidation process is more or less intact and untouch - which improves the complexity slightly. The user experience is also good as the syntax is as similar as possible. The drawback to this approach is that this breaks the code consistency of aggregation - which we would go over more in depth in the internal implementation.

```
df.group("test").agg(
   example=(['a', 'b'], lambda x: x['a'].min() - x['b'].min())
)
```

Internal Implementation

How aggregations normalize data is fairly simple, arguments are passed in and normalize_keyword_aggregation would transform it into a dictionary and "metadata". The dictionary uses the original columns as keys and contains a list of aggregation operations to perform - which is passed into _aggregation to be aggregated using Cython in batches. The metadata would then help remap the result, in the form of DataFrames for some reason, back into a single DataFrame.

Unfortunately, aggregations in Pandas are designed to be only 1D. Therefore, to transform something from 2D to 1D - would pose as the biggest obstacle to this. As a result, much of the normalization, aggregation, and checks don't support and/or can represent a list of columns. Hence, we would explore the two problems resulting from this - representing the list and transforming the result.

The representation of the list, in a way such that it can be extracted for aggregations but used as a key in a dictionary, was a simple issue. We simply had to serialize the list and deserialize it later to perform the aggregation. There are a fair amount of existing libraries in Pandas which allow for us to do this - json, numpy, and pickle. After some consideration, we have settled for `numpy.array().tobyte()` and `np.frombuffer()` to serialize/deserialize our list due to its performance and was already existing in all parts of the code.

The final issue was how to transform the data, which using the existing aggregation methods would never work and require implementing the logic in Cython. Hence, our solution was to create our custom aggregation function which unfortunately breaks the consistency of the code. This however was necessary as we lacked the skill to implement the changes in Cython. We hope that with some guidance from the Pandas team, to refine this implementation.

User Guide

As shown from the example below, the named aggregation performs similarly to the original functionality of a tuple of (string, string | lambda). Our enhancement includes an additional input of (string[], lambda), where string[] is a list of columns and lambda is an aggregation. Note that the second element does not take in native aggregation methods since those expect 1D inputs and the lambda function must return a 1D DataFrame as well.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(4,4), columns=list('abcd'))
df['group'] = [0, 0, 1, 1]
result = df.groupby('group').agg(
  cum_max_a_b=(['a', 'b'], lambda x: x['a'].max() + x['b'].max()),
  max_diff_a_b=(['a', 'b'], lambda x: x['a'].max() - x['b'].max()),
  mean_a_b=(['a', 'b'], lambda x: (x['a'].mean() + x['b'].mean())/2)
)

print(df)
print(result)

============================= output ================================
          a         b         c         d  group
0  0.873455  0.497850  0.905782  0.071701      0
1  0.931739  0.633920  0.366876  0.638039      0
2  0.037525  0.236030  0.814608  0.650021      1
3  0.429229  0.016485  0.438932  0.582659      1
       cum_max_a_b  max_diff_a_b  mean_a_b
group
0         1.565659      0.297819  0.734241
1         0.665259      0.193199  0.179817
```

Testing

Acceptance Testing

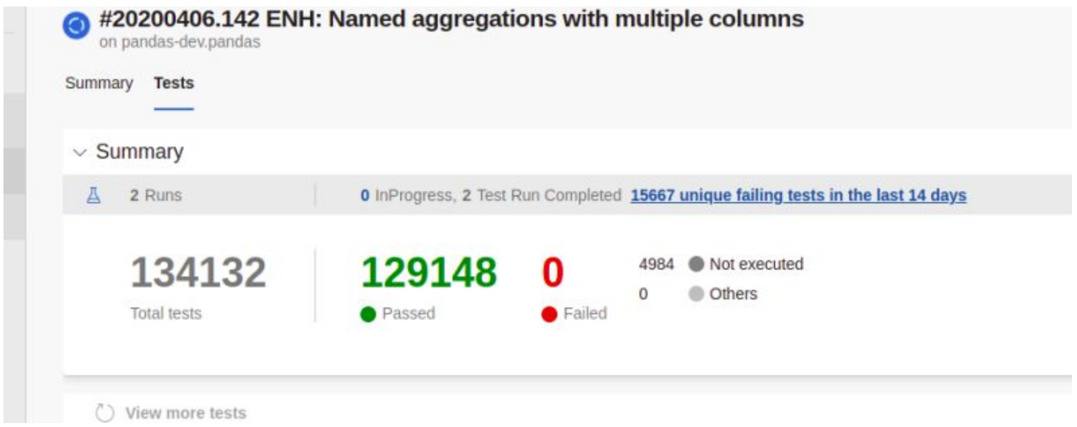
When performing an acceptance test, we used the following code sample below. The code sample would accept the changes if we are able to use the previous aggregations with the pandas library as well as with our own implementation. The goal is that we wouldn't run into any regression from the master branch compared to our current branch. This covers a wide range of the already existing functionalities such as batch aggregations, single aggregations, and different types of named aggregations.

```
import pandas as pd
import numpy as np

np.random.seed(0)
df = pd.DataFrame(np.random.rand(4,4), columns=list('abcd'))
df['group'] = [0, 0, 1, 1]

dfgb = df.groupby('group')
aggregation = dfgb.agg(
    a_sum=('a', 'sum'),
    a_mean=('a', 'mean'),
    b_mean=('b', 'mean'),
    c_sum=('c', 'sum'),
    a_max=('a', 'max'),
    d_range=('d', lambda x: x.max() - x.min()),
    diff_a_b=(['a', 'b'], lambda x: x['a'].max() - x['b'].max())
)
print(aggregation)
```

To perform an acceptance test on our new functionality with different parts of the code, we checked it with the 129,148 test cases in the Panda's test suite. This would allow us to ensure at the very least, no other features are broken from our changes.


#20200406.142 ENH: Named aggregations with multiple columns
on pandas-dev.pandas

Summary Tests

Summary

2 Runs | 0 InProgress, 2 Test Run Completed 15667 unique failing tests in the last 14 days

134132 Total tests | 129148 ● Passed | 0 ● Failed | 4984 ● Not executed | 0 ● Others

View more tests

Unit Testing

We added 3 new unit tests to the test suite. These check for correctness of calculation, the mapping of our serialization/deserialization, and ensure that the batch operations of our functionality is consistent with the existing code - which would affect the consolidation process of the aggregation.

The first test, as shown below, checks the batch aggregation functionality. If done incorrectly, we should expect an error after running _aggregation from /panda/core/base.py.

```
def test_agg_multiple_columns(self):
    df = pd.DataFrame({"A": [0, 0, 1, 1], "B": [1, 2, 3, 4], "C": [3,
4, 5, 6]})
    result = df.groupby("A").agg(
        add=(["B", "C"], lambda x: x["B"].max() + x["C"].min()),
        minus=(["C", "B"], lambda x: x["B"].max() - x["C"].min()),
    )
    expected = pd.DataFrame(
        {"add": [5, 9], "minus": [-1, -1]}, index=pd.Index([0, 1],
name="A")
    )
    tm.assert_frame_equal(result, expected)
```

The second test, checks the normalization process of the functionality. If done correctly, we should expect the custom error to arise. If any other errors, or if the operation has passed, we would find that column validations are lacking.

```
def test_agg_multi_missing_column_raises(self):
    df = pd.DataFrame({"A": [0, 0, 1, 1], "B": [1, 2, 3, 4], "C": [3,
4, 5, 6]})
    with pytest.raises(KeyError, match="Column 'D' does not exist"):
        df.groupby("A").agg(
            minus=(["D", "C"], lambda x: x["D"].max() -
x["C"].min()),
        )
```

The final test checks for our selection of B and C. Since we are only selecting two columns, only two columns should exist in the operations and nothing more.

```
def test_agg_multi_missing_key_raises(self):
    df = pd.DataFrame({"A": [0, 0, 1, 1], "B": [1, 2, 3, 4], "C": [3,
4, 5, 6], "D": [0, 0, 1, 1]})
    with pytest.raises(KeyError, match="D"):
        df.groupby("A").agg(
            minus=(["B", "C"], lambda x: x["D"].max() -
x["D"].min()),
        )
```

Pull Request

We have submitted a formal pull request with the implementation of this issue here. As it passes all of the integration and regression tests provided by the Pandas community, we are actively tracking its progress and monitoring any further requests from the community so that our implementation can go through and be included into Pandas itself.

With the completion of the required tasks, we closed the issues we had on our issues board in our repository - as well as moved our tickets in the project to done.