Deliverable 3

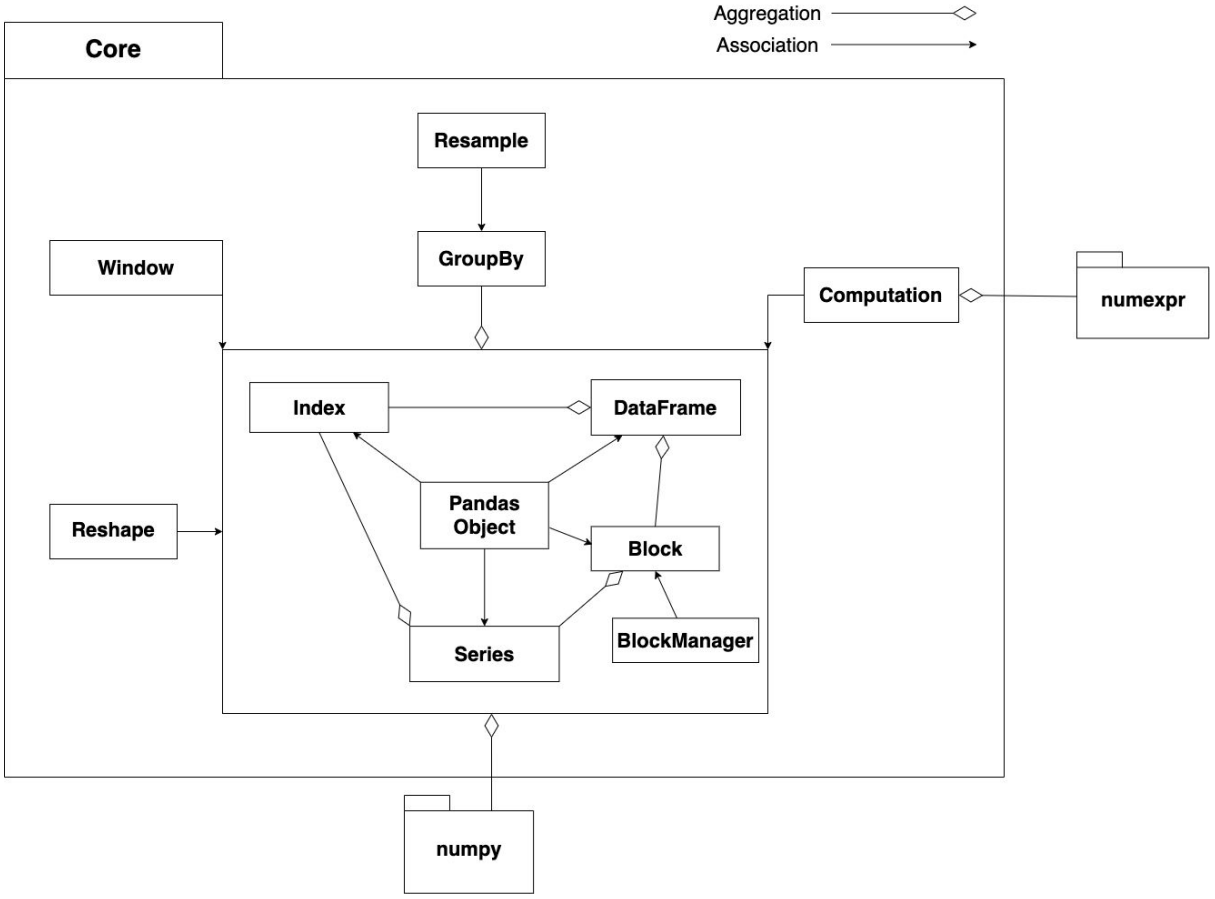
Team_01 (GitHub):
Abel Debalkew
Carl Alvares
Jaya Thirugnanasampanthan
Kemar Harris
Lan Yao
Seemin Syed

Table of Contents

Software Architecture

Overview

Figure 1

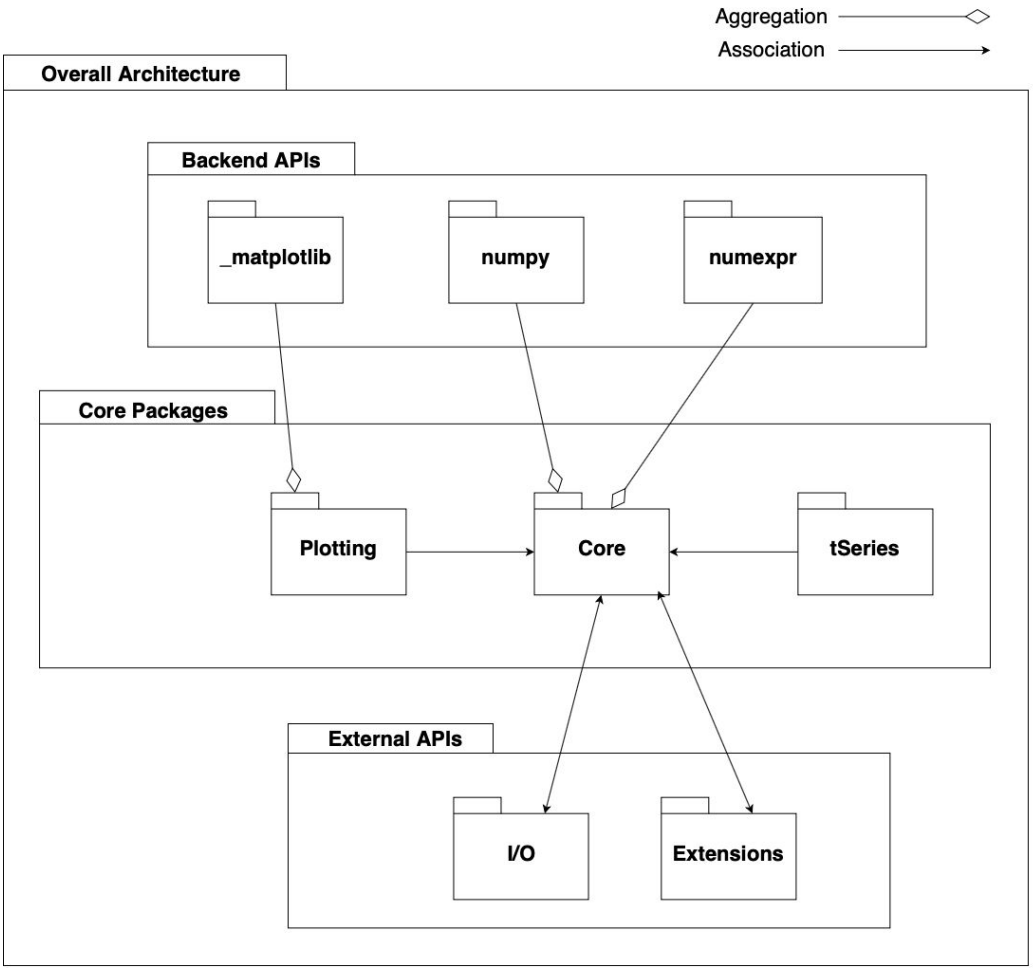


The core package is where the bulk of pandas' functionality is contained (Figure 1). At the base of this is Series, a pandas data structure which represents one-dimensional cross-sectional and time series data. Series extends the pandas Array object, which in turn is an extension of the Array object in the numpy library, a dependency of pandas. The DataFrame data structure is a dictionary of Series objects, and as such, is a two-dimensional container. Both Series and DataFrame make use of the different Index objects, depending on the type of data being analyzed, in order to characterize their data (for example, with labels). These basic objects are used throughout pandas.

Also within the core package are other classes which provide functionality for these basic objects. Reshape is a collection of data manipulation routines, such as pivot and merge, which can be used on the basic objects. GroupBy operations allow data within Series and DataFrame objects to be split, manipulated, and then grouped. Resampling contains functionality to repeatedly take samples from data contained in the basic objects or in GroupBy objects. Computations make use of the numexpr library (another dependency) to perform efficient calculations and evaluations of Python expressions within pandas, mainly

for use on columns in DataFrame objects. The Window class contains functions that allow for the calculation of statistics from multiple types of subsets of a larger dataset. These functions can be used on the basic objects or GroupBy objects.

Figure 2



Outside of the core, pandas is split into three layers (Figure 2). There is a backend layer, where libraries that pandas depends on are contained. In the core packages layer, there are the main APIs that users will interact with when using pandas. Finally, there are the external APIs, which allow pandas objects and functionalities to be imported, exported and extended to and from other sources.

In the core layer, there are the plotting, core, and tseries packages. The core package is described above. The plotting folder provides the API for using Matplotlib functionality in order to plot data contained in Series and DataFrame objects. It also provides an abstraction layer between Matplotlib and pandas, creating a separation between Matplotlib on the backend and this API, which is in the core layer of pandas. Matplotlib is meant to be a soft dependency. The tseries package separates calculation of time related data entirely from other types of data. It provides classes and implementations of time related concepts, such as weeks, months, business days, etc.

The backend layer contains the _matplotlib, numpy, and numexpr packages. Matplotlib is a library used for plotting data. pandas uses it to provide viewers with a visualization of their data. Numpy is a package in python for scientific computing with Python. pandas bases its main structures, Series and DataFrame, on this library. This allows pandas to focus its core functionality on shaping, modifying, and viewing data, while leaving the core computing functionality to numpy. Numexpr is a numerical expression evaluator for numpy. Expressions that operate on arrays are accelerated when using numexpr. Python can be slow when evaluating complex expressions on data structures like DataFrames, and since pandas focuses on data manipulation, numexpr is used in conjunction with numpy to speed up calculation.

Lastly, the External API layer has input/output and extensions packages. The input/output module provides functionality to read and write data from pandas to and from various sources, such as CSV, HTML, Excel, and more. It also allows pandas objects to be converted to and from other formats. The input/output module interacts with the core module, as this is where objects it converts to and from are defined. The extensions package allows users to extend pandas objects in their own libraries and projects.

Modifications

One modification to the original UML the team made is the inclusion of pandas Object (Figure 1). pandas Object is the base class of three pandas objects: Index, Series and DataFrame. pandas Object can be thought of as NumPy structured arrays with labeled rows and columns rather than just integers. It has functionalities like '__repr__()' that other pandas objects have. pandas Object is associated with the three core components because they have some similar methods to the ones in pandas Object, but they are not entirely implementing all the methods within pandas Object.

Another modification the team made to the UML is the inclusion of the Block object (Figure 1). Blocks are a collection of 2-D Series of the same dtype. Certain operations (like reindexing) are more efficient when run on a single, combined Series instead of multiple separate ones. The Block object allows multiple Series to be consolidated in order to perform these operations in the most efficient manner. The Series objects collected in a Block are all the same dtype so that they are not reduced to their generic pandas Object form when consolidated, ensuring that specific functionality is not lost. Blocks are managed by the BlockManager, which handles the grouping of Series into Blocks and performs Series consolidation whenever an operation which benefits from this consolidation is performed. One or more Block objects are contained in each DataFrame. The UML has been updated to place Block in between Series and DataFrame to reflect this change.

Design Patterns

With a deeper understanding of the architecture, the team noticed numerous design patterns present in the pandas project. The construction of core objects like Series and DataFrame is done using the Builder design pattern. These objects have a single, simple constructor that creates different representations of the respective object based on context. The Factory pattern is used when creating Series objects. The constructor takes values and infers which

dtype best matches those values, deferring to a subclass constructor to then create a Series of that specific dtype. The Computation module makes use of the Command pattern by separating the execution of various calculations from the objects that the calculations will be performed on. This allows for greater modularization, creating a consistent format that can be used wherever calculations are needed. The Strategy pattern is employed in core/algorithms, allowing different implementations of the specified algorithms to be used depending on what the developers see fit. The Singleton design pattern is used for the pandas NA object, which is a new value meant to represent missing values of all types. There is only one instance of this object present at all times, and any usage of NA refers to this single instance.

Interesting Solutions

An interesting part of the pandas architecture that the team found was that the plotting module allows for the use of third-party backends instead of the default (Matplotlib), effectively creating a soft dependency. This is done through a separate API layer which defines the functionality that plotting requires from third-party libraries, allowing users to bring in their backends of choice as long as they meet these requirements. This was a recent change by pandas contributors to offer greater flexibility to users.

Another interesting fact that the team noticed was that the plotting module and the tseries module are separate modules, despite performing similar functions. This is due to the fact that there are numerous differences between time-series graphs and other forms of graphs, and as such, pandas developers decided that there was little overlap in execution between the two.

Chosen Features/Bugs

Team FOSSkateers investigated 2 features/significant bug issues reported in the pandas project (Issues #23998 and #22347). They have chosen to implement Issue #22347.

Issue #23998 is concerned with displaying the number of rows added to a table when a DataFrame is converted to SQL. Issue #22347 is concerned with introducing a query method for Series.

Issue #23998

Link:
[to_sql return inserted row count or list · Issue #23998 · pandas-dev/pandas]

Labels: API Design, IO SQL

Version:
Production: 1.0.1
Development: 1.1.0.dev0+873.ge72e2dd17

Description:
Sometimes, it would be useful to know how many rows have been inserted into a table when a Dataframe gets converted to SQL. At the moment the to_sql() function from pandas returns None. This feature requests for the function to_sql() to return the number of rows added to the sql database (which is the dataframe) after the function has completed.

Usage Example:

```
>>> import pandas as pd
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> resultProxy = df.to_sql('users', con=engine)
>>> resultProxy.rowcount
```

Expected Output:

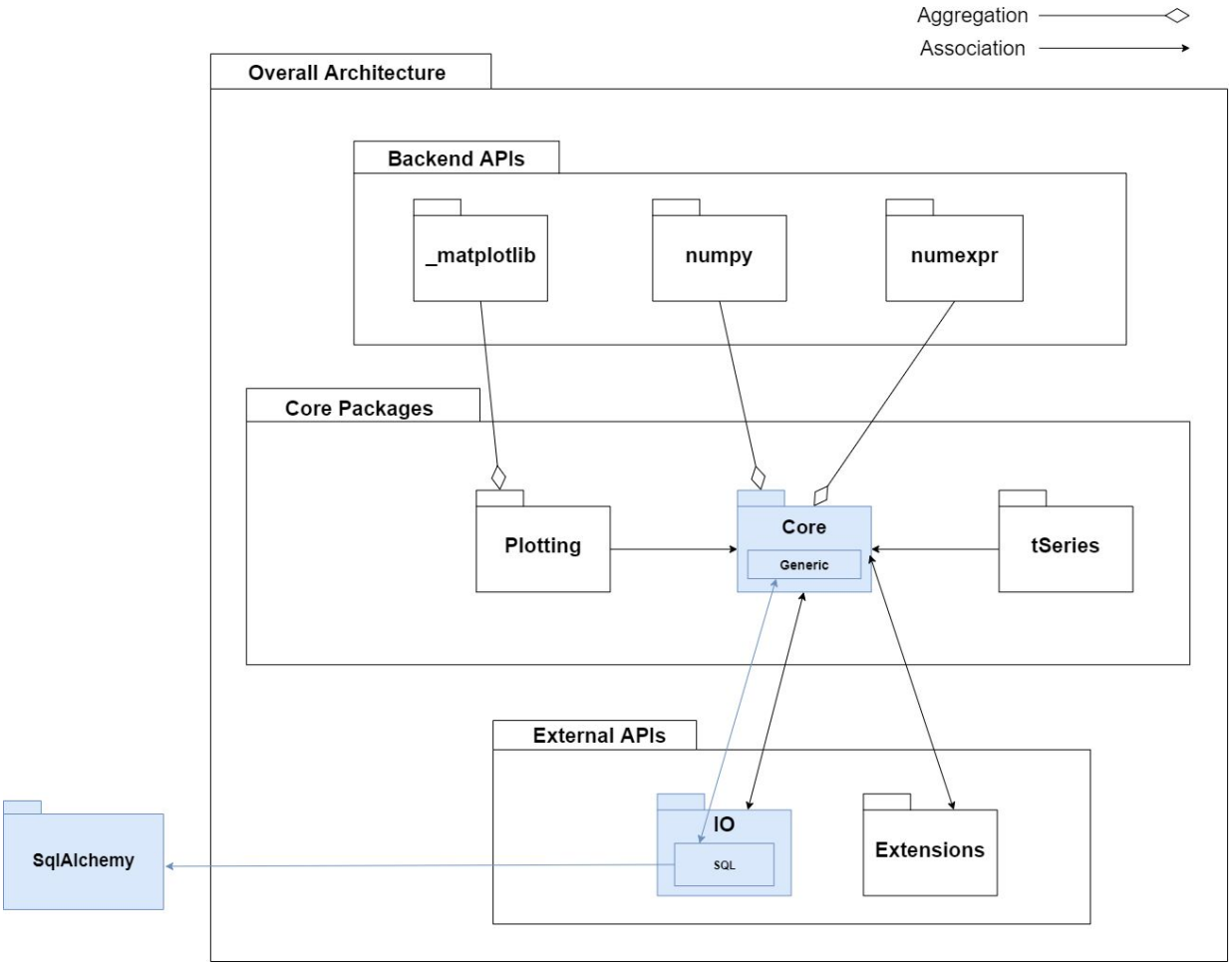
```
3
```

Code Trace:
>>> df.to_sql('users', con=engine)
pandas\core\generic.py, line 2631, in to_sql
sql.to_sql(self, name, con, schema=schema, if_exists=if_exists, index=index, index_label=index_label, chunksize=chunksize, dtype=dtype, method=method)
pandas\io\sql.py, line 510, in to_sql
pandas_sql.to_sql(self, name, con, schema=schema, if_exists=if_exists, index=index, index_label=index_label, chunksize=chunksize, dtype=dtype, method=method)
pandas\io\sql.py, line 1326, in to_sql
table.insert(chunksize, method=method)

Interactions with Code Base:
In the code, pandas executes its SQL queries using the sqlalchemy module. This module lets you create an SQL engine. These engines have a function called “execute()” that returns a ResultProxy. In the ticket, it is recommended to return the ResultProxy object, which has an attribute called “rowcount”, which shows the number of rows that were inserted into the database. To keep the layer of abstract between sqlalchemy and pandas, we could wrap the ResultProxy in an object called a PandasResultProxy. We could put the class definition in *pandas/io/sql.py.* In the case where frame.to_sql() would call execute() multiple times, an aggregation of the PandasResultProxy’s would be returned.

SQL query code is delegated from the “Core” module to the “I/O” module. The function call goes several levels in depth, and thus will modify the functions on the stack by changing their return value. _execute_insert(), _exec_insert_multi(), table.insert(), and pandas_sql.to_sql() will be modified in the “I/O” module, and the frame.to_sql() function will be modified in the “Core” module. All of these functions will be modified to return a PandasResultProxy.

UML:



Modifications will be made in the I/O and Core packages, with changes specifically occuring in the *pandas/io/sql.py* and *pandas/core/generic.py* files respectively. The sql module uses sqlalchemy, which is reflected in our updated UML diagram.

Estimate of work:
As we will be reusing the component from sqlalchemy, and the PandasResultProxy object would be a wrapper class, the real work would lie in the documentation of the new feature. Because we would only be adding return values to existing functions and because Python has loose typing for its return values, we would not be breaking any code. Therefore, implementing this feature would be extremely low risk. Creating the PandasResultProxy object would be simple as we could use the adapter design pattern and reuse the functionality implemented in ResultProxy. For the documentation however, we would have to create the documentation for PandasResultProxy, tests for this class, and tests for the feature that we implemented.

The work breakdown would be as follows:

- adding a unit test suite will take an hour
- possible modification of acceptance test suites, UML diagrams, and reanalysis of the code will take three hours
- adding documentation to the code will take ten minutes
- modifying the user guide in the pandas documentation will take three to four hours
- modifying the API references will take an hour
- actual implementation will take 5 minutes
- running unit tests and bug fixes will take around 30 minutes
- ensuring old test cases are functional will take around 30 minutes
- ensuring backwards compatibility will take an hour

Justification for Not Implementing:
We have decided to not implement this feature because we felt that it is not a priority for users in comparison to the other feature we investigated. While we are confident that we are able to design, implement, test, and document this feature within a short period of time, we also think that we are able to do more significant work and meet the same expectations (especially considering our team size).

Issue #22347

Link:
Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

Labels: API Design, Enhancement, expressions

Version:
Production: 1.1.0
Development: 1.1.0.dev0+689.gbdb4a0830

Description:
pandas.DataFrame.query is used to query the columns of a DataFrame object with a boolean expression and the issue requests that an equivalent function be implemented for Series objects.

While there are other ways to get equivalent behaviour to query for Series, for the sake of consistency with the attributes of other objects as well as for ease of acclimation for new pandas users, an explicit reference to the requested functionality is valuable.

Currently, users can get similar functionality by converting Series to a Dataframe using to_frame() and then querying said DataFrame and then converting this new DataFrame back into a Series, but this is neither explicit nor ideal; especially when numpy, upon which pandas is deeply dependent, observes no difference between 1D and 2D arrays in terms of operations possible on them.

Usage Example:

Setup:

```
>>> import pandas as pd
>>> years = range(2002, 2005)
>>> fields = range(1, 5)
>>> index = pd.MultiIndex.from_product([years, fields], names=['year',
'field'])
>>> series = pd.Series(index=index)
      year  field
      2002  1       NaN
            2       NaN
            3       NaN
            4       NaN
      2003  1       NaN
                  ...
      2004  1       NaN
            2       NaN
            3       NaN
            4       NaN
      dtype: float64
```

Expected Implementation: Applying a boolean expression to a Series object directly using the query function, which returns the appropriate queried Series object.

```
>>> train_data = series.query('year != 2003')
      year  field
```

```
      2002  1       NaN
            2       NaN
            3       NaN
            4       NaN
      2004  1       NaN
            2       NaN
            3       NaN
            4       NaN
      Name: 0, dtype: float64
```

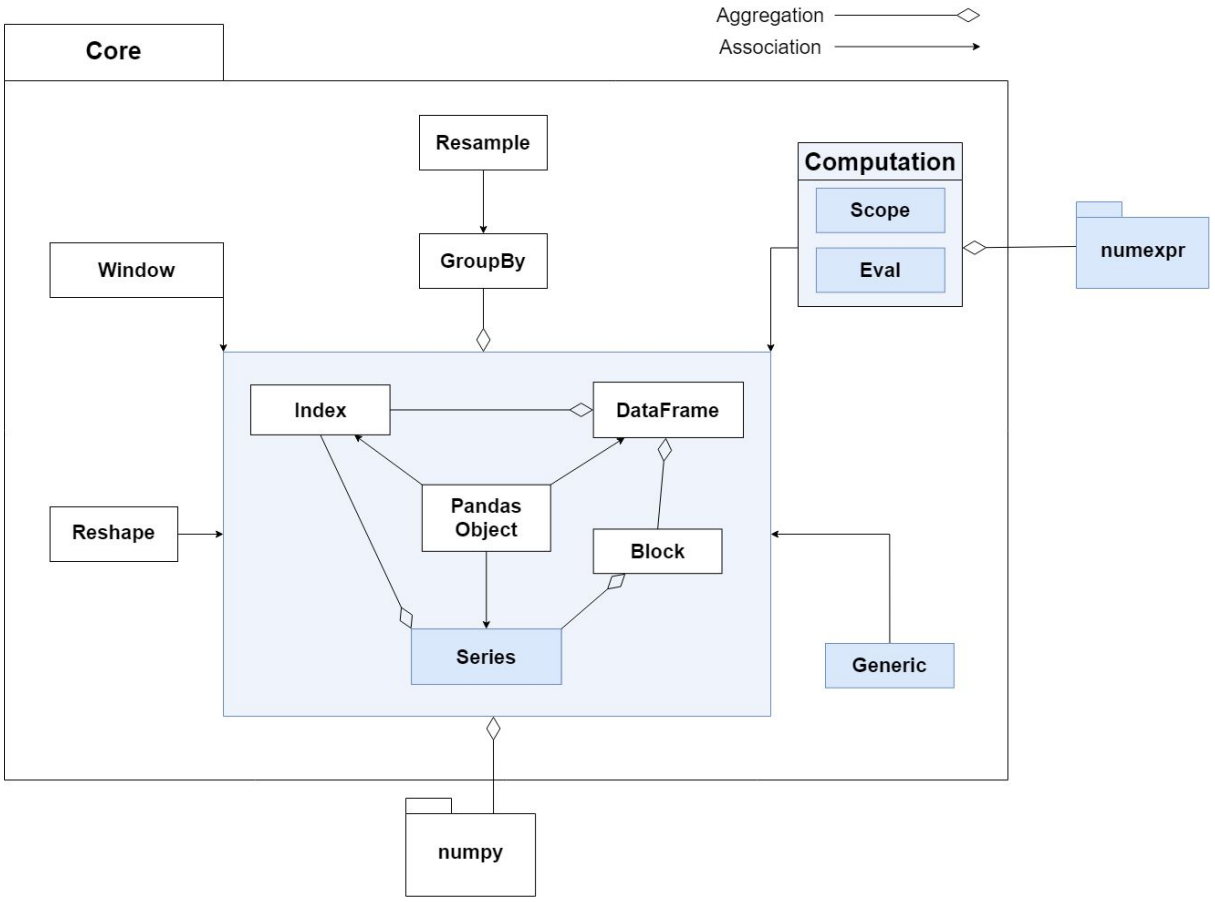
Current Alternatives: Converting Series to DataFrame, then running a query on it which returns a DataFrame and must be returned as a Series by accessing the 0th index.

```
>>> train_data = series.to_frame().query('year != 2003')[0]
      year  field
      2002  1       NaN
            2       NaN
            3       NaN
            4       NaN
      2004  1       NaN
            2       NaN
            3       NaN
            4       NaN
      Name: 0, dtype: float64
```

Code Trace and Interactions with Code Base:
As pandas.Series.query() will be modelled off of pandas.DataFrame.query(), a lot of the functionality implemented by DataFrame will also be required by Series, mainly within the pandas.core folder.

The query() function in the DataFrame class uses an eval() function to evaluate the boolean expressions used to query the data. This eval function calls into core.computation.eval, which in turn extends various backends, including one of the project's dependencies, the numexpr package.
Other areas that query() will interact with and that may require modification is in pandas.core.generic and core.computation.scope, which contain resolvers and cleaners for data contained in DataFrames, some of which may be needed by Series.

UML:



Modifications will be made in Series based on DataFrame, both of which are built on numpy, to call functions in Generic, Eval and Scope, the latter two of which are in computation and extend numexpr and some other backends. numexpr and numpy will not be modified explicitly, but the other folders mentioned above and in the image will be modified to some degree, depending on the requirements uncovered during implementation.

Estimate of work:
pandas.Series.query() will be modelled off of pandas.DataFrame.query() and as such, the latter will provide the components and expected behaviors for the former.

There may, however, arise issues with integrating said components for this new object, and as such, modifications made by developers to address edge cases and broken functionality will increase the difficulty of the project. Beyond that, the team will also have to implement test cases for the new functionality and ensure present test cases are not affected by the change, as it should be self contained.

Due to the complexity of the feature, the work required can be broken down into sections (with all developers working concurrently):

- adding the unit test suite will take approximately 3 hours
- possible modification of acceptance test suites, UML diagrams, and reanalysis of the code will take 3-5 hours
- adding documentation to the code will take 6-8 hours

- modifying the user guide in the pandas documentation will take 4 or 5 hours
- modifying the API references will take 3 or 4 hours
- actual implementation will take approximately 5 hours
- running unit tests and bug fixes will take around 1 or 2 hours
- ensuring old test cases are functional will take around 5 hours
- ensuring backwards compatibility will take an hour

Technical Commentary

Team FOSSkateers have chosen to implement the second issue listed above (Issue #22347), and have provided technical commentary as it relates to said issue.

Reasons:

- Reusability: Issue #22347 was selected because we are able to reuse components. As mentioned previously, the DataFrame.query method already exists, and so it is now a question of attempting to reuse existing components to implement the Series.query method. Being able to reuse components decreases development time, which in turn allows the team to deliver value to users sooner. Furthermore, as these reusable components are reliable (i.e., they have already undergone extensive review and testing), we will have fewer development risks. This also increases the likelihood of our solution being accepted to the pandas code base. All these reasons align with the team's goals of delivering real-world value to users in a short amount of time.

- Usability: We argue that this feature adds value to new users and users from the numpy project. Currently, users can get similar functionality by converting Series to a Dataframe using to_frame() and then querying said DataFrame and then converting this new DataFrame back into a Series, but this is neither explicit nor ideal; especially when numpy, upon which pandas is deeply dependent, observes no difference between 1D and 2D arrays in terms of operations possible on them. This was also pointed out by other contributors in the issue's discussion thread.

Anticipated Risk:

An obvious risk that will be difficult to mitigate is introducing new bugs. This is a likely possibility due to the nature of the feature (i.e., it interacts with many components, as shown in our diagrams). To mitigate this risk, the team should carefully analyze the components affected by our code changes before proceeding with implementation - this will be completed in the 'system design with reuse' process activity.

Implementation Plan:

The team has identified significant reusable components, what new code and documentation are needed, and what components may be affected by our code changes.

Code to Reuse

1. Reuse the DataFrame.query() method to implement the Series.query() method. See pandas\core\frame.py:2798.

It goes without saying that any references made to DataFrames in the DataFrame.query() method should be replaced with Series or removed entirely. As such, we must implement the Series.eval() method, which should evaluate an expression (which is passed as an argument) on a Series. In our case, we must do the following:

2. Reuse DataFrame.eval() method to implement Series.eval() method, for use by Series.query. See pandas\core\frame.py:2960.

When reusing the existing eval method, we will need to remove column_resolvers as it is not required for Series, though index_resolvers should be kept. Both Series.query() and Series.eval() should be added to series.py, which ensures high cohesion.

New Code, Test Suites, and Documentation

As we are implementing a new method, a significant amount of time will be spent on documentation (i.e., writing docstrings). In addition, the team will need to add Series-specific tests to pandas\tests\computation\test_eval.py. Considering Series.query() will be based on DataFrame.query(), which already has its own testing suite (in pandas\tests\frame\test_query_eval.py), the team will use the already established criteria to inform the development of tests tailored to Series.query().

Interactions Between New Code and Existing Code

The eval() method (in pandas\core\computation\eval.py:269) is imported in the DataFrame.query() method. This eval() method “evaluates a Python expression as a string using various backends”. When implementing the Series.query() method, the team will need to ensure that the method uses the eval() method. The eval() method may be used without modification as it can return a Series object (specified in line 245).

Also, the has_resolvers method (pandas\core\computation\scope.py:151) is called on by both DataFrame.query() and DataFrame.eval() methods. The has_resolvers() method checks to see if there is extra scope. When implementing the Series.query() and Series.eval() methods, the team will need to ensure that these methods call on the has_resolvers() method appropriately. Based on our current understanding though, the has_resolvers() method may be used without modification.

After modification of the Series object to add querying, the team will need to ensure the functionality of generic getters and handlers. __getattr__() (in pandas\core\generic.py:5051) currently triggers an error when the user tries to access the non existent Series.query() method, and as part of developing said method, the team will need to check that the function finds our methods without issue.

Process

The software development process we are using for the final deliverable is Kanban along with a modified version of the reuse-oriented model. As mentioned in previous deliverables, the team's goal is to deliver value to users sooner. As such, this software development process will help us accomplish this.

Process Activities[1]:

- Requirements Specification: The team has begun functional requirements gathering and will continue to gather both functional and non-functional requirements by communicating with pandas members. In this activity, the team will begin writing the User Guide.

- Component Analysis: The team plans to reuse existing code used for DataFrame.query() method. Consequently, the team will need to allocate more time to analyze the code for DataFrame.query() and identify reusable components.

- System Design with Reuse: The team will need to reevaluate the design plans for Series.query() so that the reusable components identified in the previous activity are taken into consideration. We will also take into account any new code that needs to be written.

- Development and Integration: The team will break the feature into smaller tasks. Tasks will be prioritized so that they are worked on/pulled first. All members will adhere to the WIP limits so that the team is able to focus on current tasks and unfinished tasks do not pile up in any column. Unit tests must be written and successfully pass after task implementation. Furthermore, all code development must be done in branches and pass the style test. Finally, all code should be properly documented.

- Validation: Pull-requests will be made. Code and documentation must be reviewed by other members before it is merged to the master branch. All tests (including unit tests and pandas' own test suite) must be run after the merge to ensure that the feature meets expectations. Acceptance testing will also be conducted.

All process activities interleave which means that we can accommodate new changes easily without having to perform a significant amount of rework.

Acceptance Tests Summary:

To verify that the solution meets customers' needs (i.e., implement a query method for Series), the following instructions should be carried out.

1. In a terminal, open Python and import the pandas library.

```
>>> import pandas as pd
```

2. Initialize a pandas Series with data and indexes.

[1] In the modified version of the reuse-oriented model.

```
>>> data = [1, 2, 3]
>>> index  = [4, 5, 6]
>>> series = pd.Series(data, index=index)
>>> series
4    1
5    2
6    3
dtype: int64
```

3. Query the Series with a boolean expression (e.g., 'index > 4'). This returns a Series resulting from the given query expression.

```
>>> series.query('index > 4')
5  2
6  3
dtype: int64
```

Note that the query used above does not mutate the data in the original Series (i.e., the original Series remains unchanged); we only return a modified copy of the Series. This is because the second argument passed to Series.query() is ‘inplace’, which is set to ‘False’ by default. ‘inplace’ (which is a boolean) indicates whether the query should mutate the original Series or leave the original Series unchanged, producing a modified copy instead[2].

4. To mutate the Series, ‘inplace’ must be set to ‘True’. This returns the Series (not a copy though) resulting from the given query expression.

```
>>> series.query('index > 4', True) #inplace=True so Series is mutated
>>> series
5    2
6    3
dtype: int64
```

Note that the query could also take another parameter, **kwargs (keyword arguments). The result should be a Series.

5. Pass **kwargs to query().

```
>>> series.query('index > 4', False, engine='python')
5    2
6    3
dtype: int64
```

[2] The query method for Series should have the same arguments as the query method for DataFrame: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html.