Deliverable 4

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

Table of Contents

Chosen Feature

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

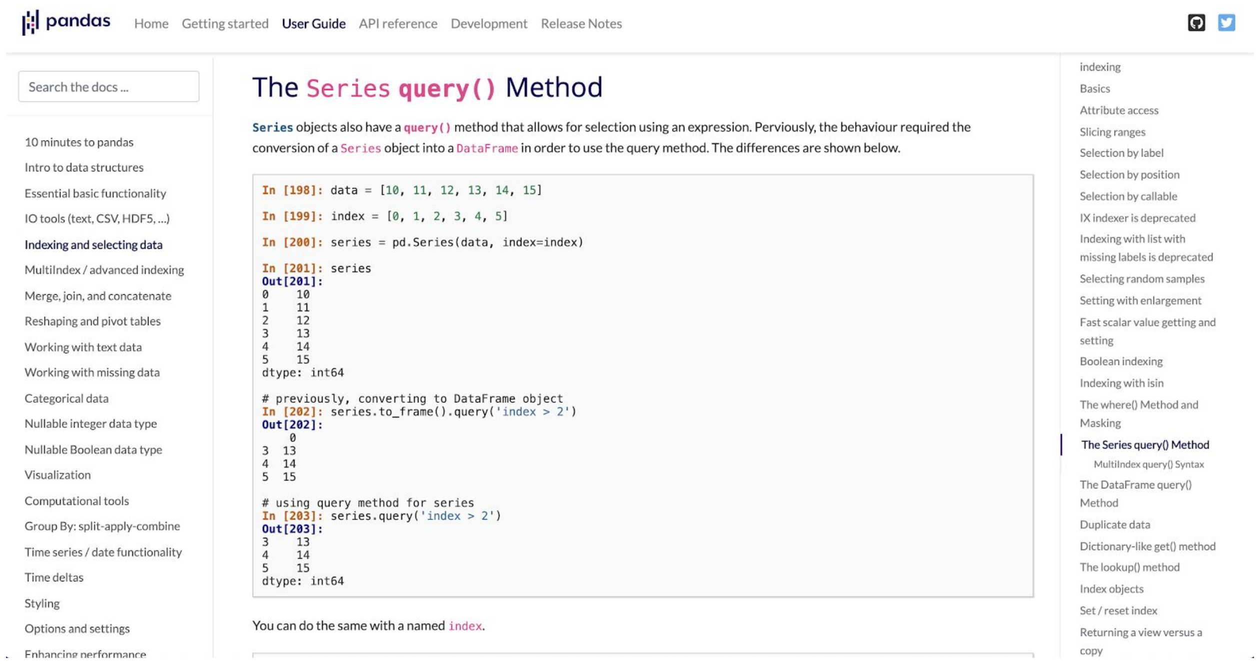
User Guide

The modified user guide is available to view, in raw, here.

pandas uses the user guide to outline how to use features by giving examples and linking the api reference. Documentation for the user guide is stored in the pandas/doc folder. Restructured text (.rst) is used to store the technical documentation and could be made into html, latex, or a clean format. We decided to add our user guide to these files so we could adhere to the pandas contribution guidelines. To make the files you should run the following command in the pandas/doc directory.

```
>>> python make.py html
```

To build the files you need to have the optional dependencies for pandas installed. The build takes a bit of time so we have provided screenshots below.


pandas
Home
Getting started
User Guide
API reference
Development
Release Notes
Search the docs ...
10 minutes to pandas
Intro to data structures
Essential basic functionality
IO tools (text, CSV, HDF5, ...)
Indexing and selecting data
MultiIndex / advanced indexing
Merge, join, and concatenate
Reshaping and pivot tables
Working with text data
Working with missing data
Categorical data
Nullable integer data type
Nullable Boolean data type
Visualization
Computational tools
Group By: split-apply-combine
Time series / date functionality
Time deltas
Styling
Options and settings
Enhancing performance
indexing
Basics
Attribute access
Slicing ranges
Selection by label
Selection by position
Selection by callable
IX indexer is deprecated
Indexing with list with missing labels is deprecated
Selecting random samples
Setting with enlargement
Fast scalar value getting and setting
Boolean indexing
Indexing with isin
The where() Method and Masking
The Series query() Method
MultiIndex query() Syntax
The DataFrame query() Method
Duplicate data
Dictionary-like get() method
The lookup() method
Index objects
Set / reset index
Returning a view versus a copy

The Series query() Method

Series objects also have a query() method that allows for selection using an expression. Perviously, the behaviour required the conversion of a Series object into a DataFrame in order to use the query method. The differences are shown below.

```
In [198]: data = [10, 11, 12, 13, 14, 15]

In [199]: index = [0, 1, 2, 3, 4, 5]

In [200]: series = pd.Series(data, index=index)

In [201]: series
Out[201]:
0    10
1    11
2    12
3    13
4    14
5    15
dtype: int64

# previously, converting to DataFrame object
In [202]: series.to_frame().query('index > 2')
Out[202]:
    0
3  13
4  14
5  15

# using query method for series
In [203]: series.query('index > 2')
Out[203]:
3    13
4    14
5    15
dtype: int64
```

You can do the same with a named index.

You can do the same with a named `index`.

```
In [204]: data = [10, 11, 12, 13, 14, 15]

In [205]: index = [0, 1, 2, 3, 4, 5]

In [206]: series = pd.Series(data, index=index)

In [207]: series.index.name = 'foo'

In [208]: series.query('foo > 2')
Out[208]:
foo
3    13
4    14
5    15
dtype: int64
```

The `query` method also has a `inplace` keyword that lets you modify the originail `Series`. It is set to `False` by default.

`MultiIndex` `query()` Syntax

You can use a `Series` with a **`MultiIndex`** with the query method.

```
In [209]: data = list(range(0, 10))

In [210]: foos = np.random.choice(['foo1', 'foo2'], size=10)

In [211]: bars = np.random.choice(['bar1', 'bar2'], size=10)

In [212]: data
Out[212]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [213]: foos
Out[213]:
array(['foo1', 'foo1', 'foo1', 'foo1', 'foo2', 'foo2', 'foo2', 'foo2',
       'foo1', 'foo2'], dtype='<U4')

In [214]: bars
Out[214]:
array(['bar2', 'bar2', 'bar2', 'bar1', 'bar1', 'bar2', 'bar2', 'bar2',
       'bar2', 'bar2'], dtype='<U4')

In [215]: index = pd.MultiIndex.from_arrays([foos, bars], names=['foos', 'bars'])

In [216]: index
Out[216]:
MultiIndex([('foo1', 'bar2'),
            ('foo1', 'bar2'),
            ('foo1', 'bar2'),
            ('foo1', 'bar1'),
            ('foo2', 'bar1'),
            ('foo2', 'bar2'),
            ('foo2', 'bar2'),
            ('foo2', 'bar2'),
            ('foo1', 'bar2'),
            ('foo2', 'bar2')],
           names=['foos', 'bars'])
```

```
In [217]: series = pd.Series(data, index=index)

In [218]: series
Out[218]:
foos  bars
foo1  bar2    0
      bar2    1
      bar2    2
      bar1    3
foo2  bar1    4
      bar2    5
      bar2    6
      bar2    7
foo1  bar2    8
foo2  bar2    9
dtype: int64

In [219]: series.query('foos == "foo1"')
Out[219]:
foos  bars
foo1  bar2    0
      bar2    1
      bar2    2
      bar1    3
      bar2    8
dtype: int64

In [220]: series.query('foos == "foo1" and bars == "bar2"')
Out[220]:
foos  bars
foo1  bar2    0
      bar2    1
      bar2    2
      bar2    8
dtype: int64
```

You can use special names to refer to unnamed levels of `MultiIndex`. The example below uses `ilevel_1`, which is a common convention to refer to the index at the "1st" level.

```
In [221]: series.index.names = [None, None]

In [222]: series
Out[222]:
foo1  bar2    0
      bar2    1
      bar2    2
      bar1    3
foo2  bar1    4
      bar2    5
      bar2    6
      bar2    7
foo1  bar2    8
foo2  bar2    9
dtype: int64

In [223]: series.query('ilevel_1 == "bar1"')
Out[223]:
foo1  bar1    3
foo2  bar1    4
dtype: int64
```

The `Series.eval` method

A new function added alongside the `Series.query` method that allows you to evaluate an expression in the "context" of a **`Series`**.

```
In [22]: index = list(range(2010, 2015))

In [23]: data = np.random.randn(5)

In [24]: series = pd.Series(data, index=index, name='data')

In [25]: series.index.name = 'years'

In [26]: series
Out[26]:
years
2010    1.109084
2011   -1.355831
2012    1.086861
2013   -0.219075
2014   -1.415843
Name: data, dtype: float64

In [27]: series.eval('(years > 2010) & (data > 0)')
Out[27]:
years
2010    False
2011    False
2012     True
2013    False
2014    False
dtype: bool
```

There is an `inplace` keyword that currently defaults to `True`. This maybe change in future versions of pandas and it is recommended to use the `inplace` keyword.

If a `series index` does not have a name you can use "index" to reference the index.

```
In [28]: index = list(range(2010, 2015))

In [29]: data = np.random.randn(5)

In [30]: series = pd.Series(data, index=index)

In [31]: series
Out[31]:
2010   -0.210220
2011   -0.158656
2012   -0.976323
2013    0.060449
2014   -1.088247
dtype: float64

In [32]: series.eval('(index > 2010) & (index != 2013)')
Out[32]:
2010    False
2011     True
2012     True
2013    False
2014     True
dtype: bool
```

The function can also be used on a `series` with a **`MultiIndex`**.

```
In [33]: data = np.random.randn(10)

In [34]: foos = np.random.choice(['foo1', 'foo2'], size=10)

In [35]: years = list(range(2010, 2020))

In [36]: data
Out[36]:
array([ 0.864 , -0.116 , -1.2159,  1.6708, -2.6214, -1.2979, -0.8359,
        0.3152, -1.3373,  0.7724])

In [37]: foos
Out[37]:
array(['foo1', 'foo1', 'foo2', 'foo1', 'foo1', 'foo1', 'foo1', 'foo1',
       'foo2', 'foo2'], dtype='<U4')

In [38]: years
Out[38]: [2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019]

In [39]: index = pd.MultiIndex.from_arrays([foos, years], names=[None, 'years'])

In [40]: index
Out[40]:
MultiIndex([('foo1', 2010),
            ('foo1', 2011),
            ('foo2', 2012),
            ('foo1', 2013),
            ('foo1', 2014),
            ('foo1', 2015),
            ('foo1', 2016),
            ('foo1', 2017),
            ('foo2', 2018),
            ('foo2', 2019)],
           names=[None, 'years'])
```

```
In [41]: series = pd.Series(data, index=index)

In [42]: series
Out[42]:
      years
foo1  2010     0.863987
      2011    -0.115998
foo2  2012    -1.215949
foo1  2013     1.670837
      2014    -2.621419
      2015    -1.297879
      2016    -0.835893
      2017     0.315213
foo2  2018    -1.337334
      2019     0.772364
dtype: float64

In [43]: series.eval('ilevel_0 == "foo1" and years > 2013')
Out[43]:
      years
foo1  2010     False
      2011     False
foo2  2012     False
foo1  2013     False
      2014      True
      2015      True
      2016      True
      2017      True
foo2  2018     False
      2019     False
dtype: bool
```

`ilevel_0` can be used to reference a `MultiIndex` by the level. In this case the 0th index did not have a name.

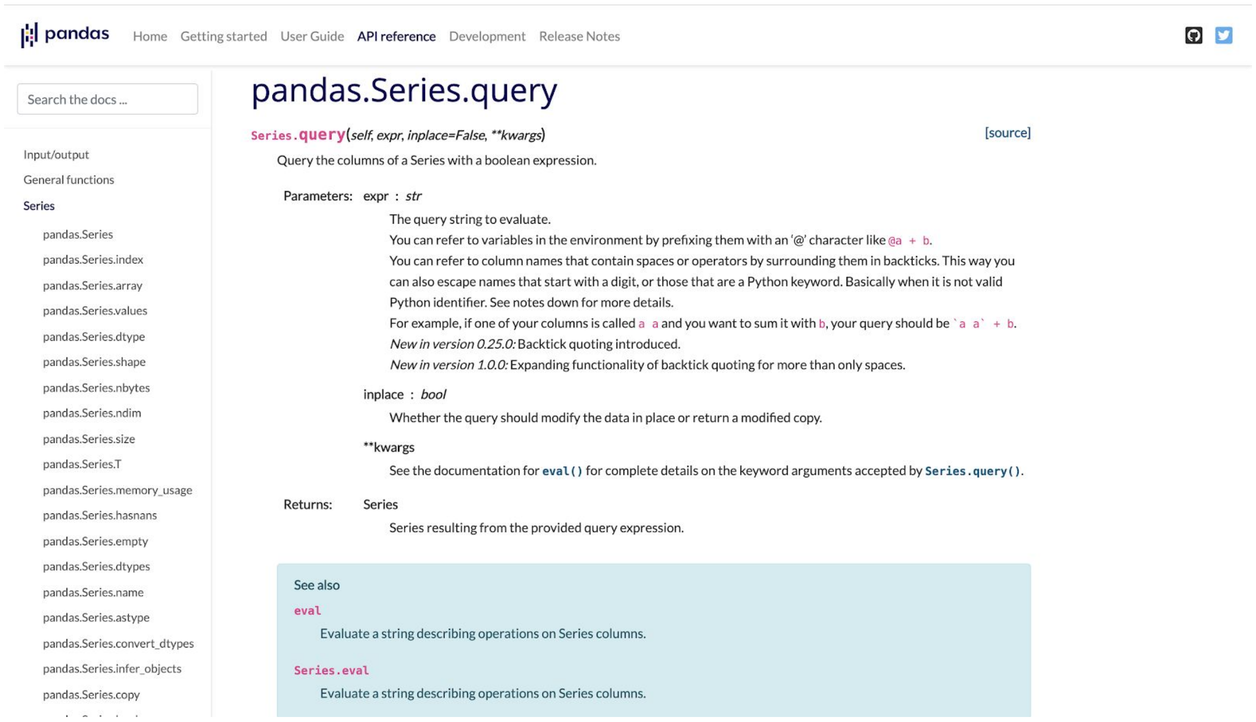
API Reference

The modified API reference is available to view, in raw, here.

pandas uses the API reference guide as an overview of all public pandas objects, functions and methods. Documentation for the API reference guide is pulled from method documentation in the pandas' code by referencing information in doc/source/reference. To make the files you should run the following command in the pandas/doc directory.

```
>>> python pandas/doc/make.py clean (for a clean build if needed)
>>> python pandas/doc/make.py html
```

To build the files you need to have the optional dependencies for pandas installed. The build takes a bit of time so we have provided screenshots below.



pandas.Series.query

`Series.query(self, expr, inplace=False, **kwargs)`

Query the columns of a Series with a boolean expression.

Parameters: expr : *str*

The query string to evaluate.

You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

You can refer to column names that contain spaces or operators by surrounding them in backticks. This way you can also escape names that start with a digit, or those that are a Python keyword. Basically when it is not valid Python identifier. See notes down for more details.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be `` `a a` + b ``.

New in version 0.25.0: Backtick quoting introduced.

New in version 1.0.0: Expanding functionality of backtick quoting for more than only spaces.

inplace : *bool*

Whether the query should modify the data in place or return a modified copy.

**kwargs

See the documentation for `eval()` for complete details on the keyword arguments accepted by `Series.query()`.

Returns: Series

Series resulting from the provided query expression.

See also

`eval`

Evaluate a string describing operations on Series columns.

`Series.eval`

Evaluate a string describing operations on Series columns.

Notes

The result of the evaluation of this expression is first passed to `Series.loc` and if that fails because of a multidimensional key (e.g., a Series) then the result will be passed to `Series.__getitem__()`.

This method uses the top-level `eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `Series.index` attribute of the `Series` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, ``` `it's` > `that's` ``` will raise an error, as it forms a quoted string (``` 's > `that' ```) with a backtick inside.

Backtick quoted variables are parsed as literal Python code and are converted internally to a Python valid identifier. This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error. This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser. For example, `` `it's` > `that's` `` will raise an error, as it forms a quoted string (``'s > `that'``) with a backtick inside.

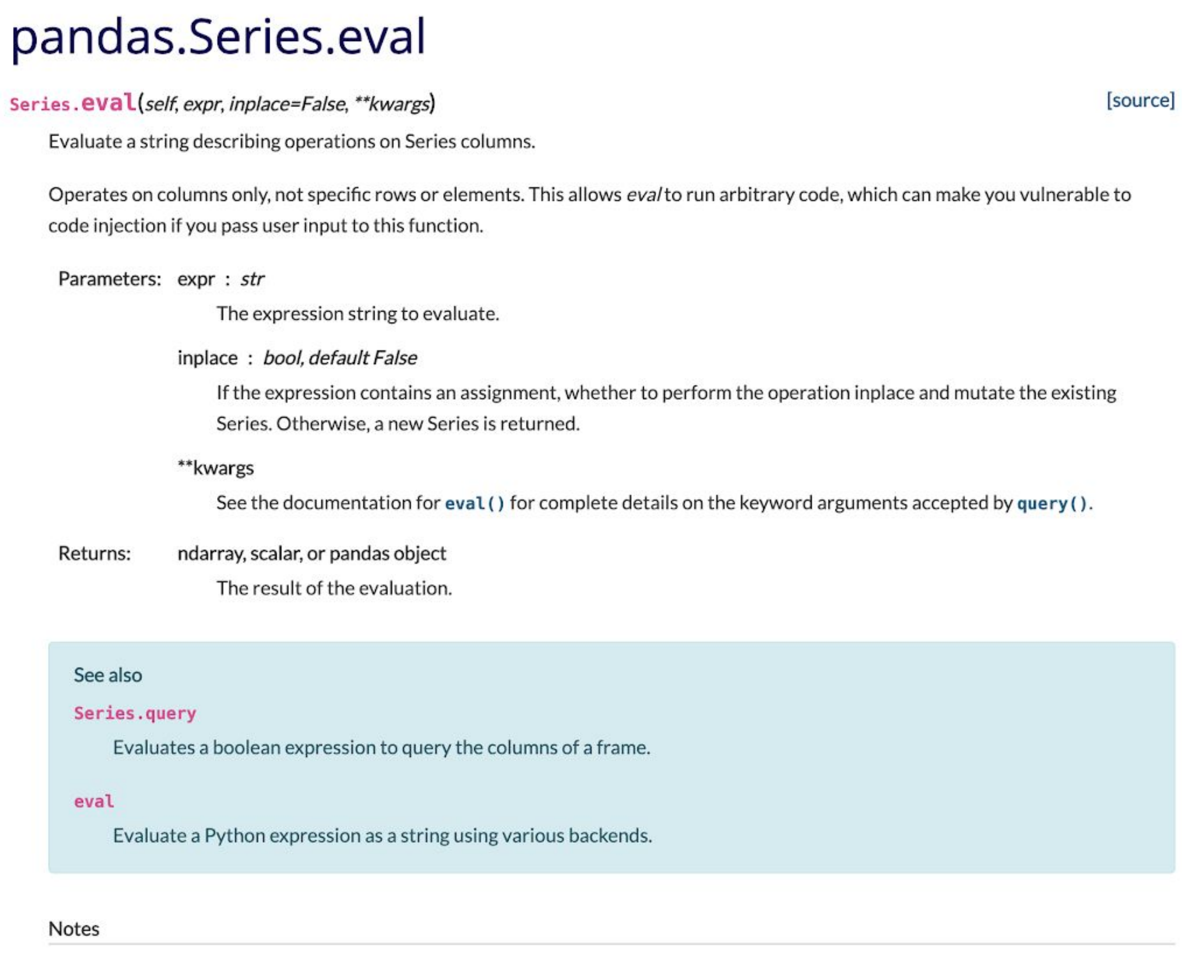
See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in **`pandas.core.computation.parsing`**.

Examples

```
>>> series = pd.Series(range(5))
>>> series
0    0
1    1
2    2
3    3
4    4
dtype: int64
>>> series.query('index > 2')
3    3
4    4
dtype: int64
```

<< pandas.Series.quantile

pandas.Series.radd >>



pandas.Series.eval

`Series.eval`(*self*, *expr*, *inplace=False*, ***kwargs*) [source]

Evaluate a string describing operations on Series columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters: expr : *str*

The expression string to evaluate.

inplace : *bool, default False*

If the expression contains an assignment, whether to perform the operation inplace and mutate the existing Series. Otherwise, a new Series is returned.

**kwargs

See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

Returns: ndarray, scalar, or pandas object

The result of the evaluation.

See also

`Series.query`

Evaluates a boolean expression to query the columns of a frame.

`eval`

Evaluate a Python expression as a string using various backends.

Notes

For more details see the API documentation for **`eval()`**. For detailed examples see enhancing performance with eval.

Examples

```
>>> series = pd.Series(1, index=range(5))
>>> series
0    1
1    1
2    1
3    1
4    1
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> op = f"1.2 {'+'} index"
>>> series.eval(op)
0    1.2
1    2.2
2    3.2
3    4.2
4    5.2
dtype: float64
```

Use `inplace=True` to modify the original DataFrame.

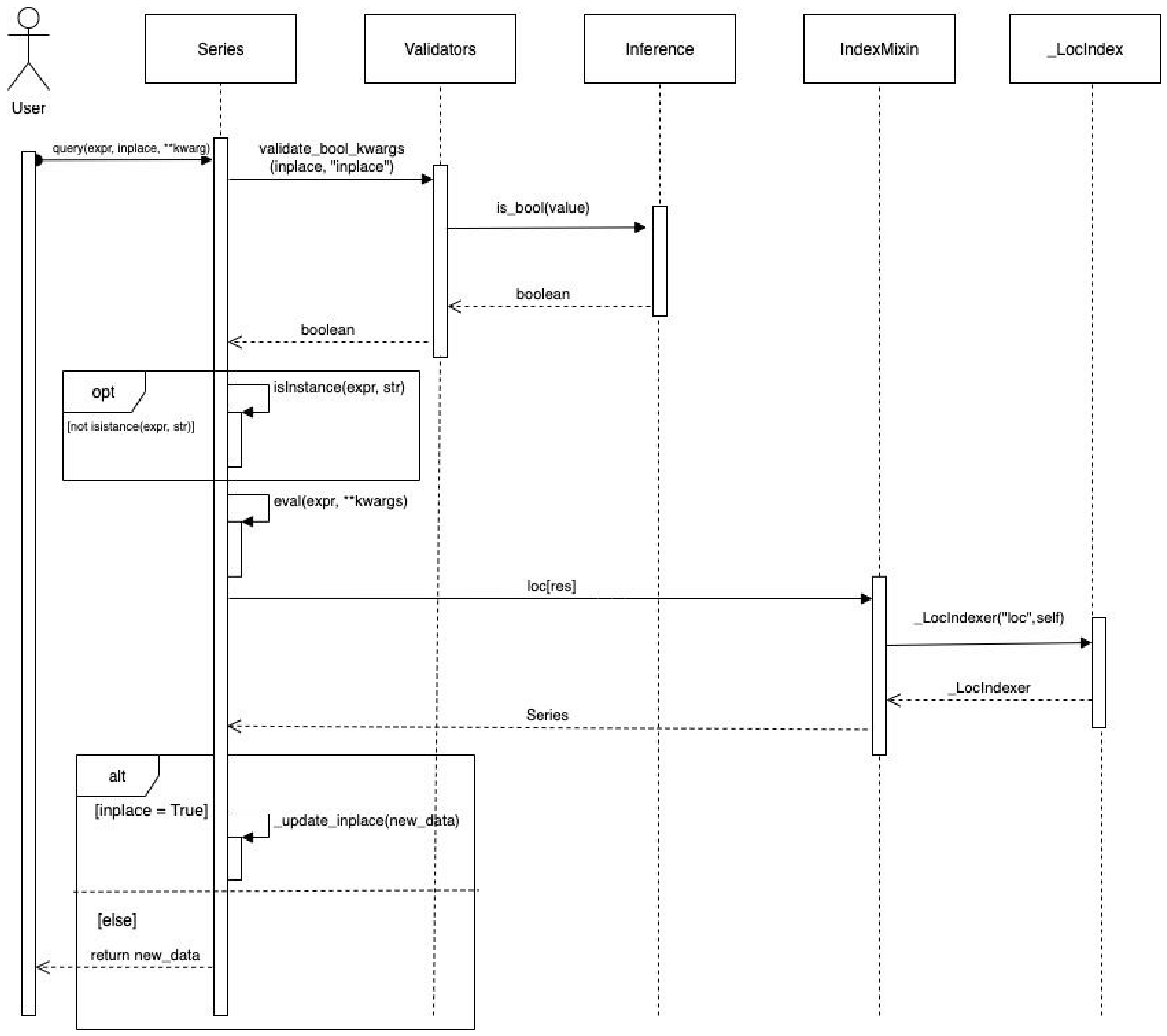
```
>>> dict1 = {"a": 1}
>>> dict2 = {"b": 2}
>>> series.eval("c = a + b", inplace=True, resolvers=[dict1, dict2])
>>> series
0    1
1    1
2    1
3    1
4    1
c    3
dtype: int64
```

<< pandas.Series.equals

pandas.Series.ewm >>

Design Document

Sequence Diagram for Series.query()



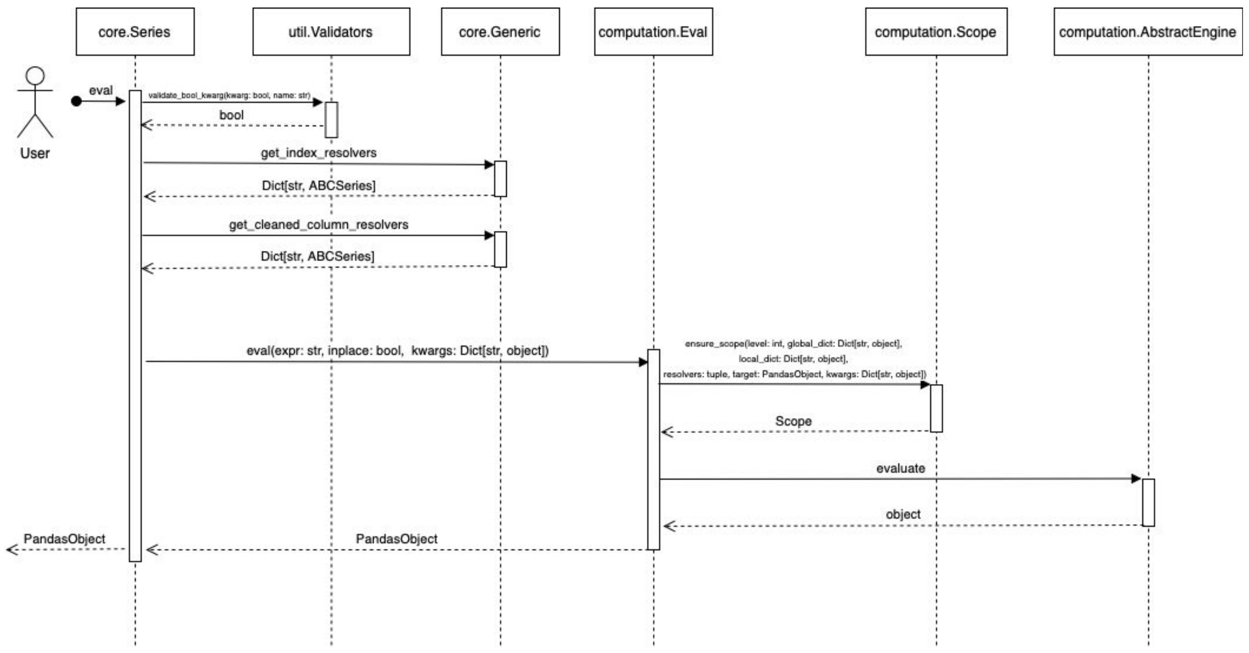
The team's implementation of Series.query() method is shown through the above sequence diagram. When a user calls query() under Series, the method will first validate whether the key work argument 'kwarg' is of Boolean type by calling validate_bool_kwargs() under util.Validators. Validators will check if the value passed in is of Boolean type through calling is_bool() under Inference and return the value back to Series if it is.

Then the method will call itself to check if the expression passed is of String type by calling isInstance(expr, str). eval(expr, **kwarg) is called afterwards to evaluate a string describing operations on Series columns.

Then loc[res] under IndexMixin object is called to access a group of rows and columns by label(s) or a boolean array. ".loc[]" is primarily label based, but may also be used with a boolean array. Labels are like “5” and “a” which can be used as keys to access the content of Series. IndexMixin then calls _LocIndex() so that a _LocIndexer object can be returned. After that, IndexMixin accesses the data by using _LocIndexer and returns new_data back to Series.

Finally, depending on whether the user sets “inplace” as True or False, there are two cases. If “inplace” is False, the new_date will be returned as it is. If it is True, the passed in value will be updated with new_data by calling the method _update_inplace(new_date) in itself. The updated value is returned.

Sequence Diagram for Series.eval()


core.Series
util.Validators
core.Generic
computation.Eval
computation.Scope
computation.AbstractEngine
User
eval
validate_bool_kwarg(kwarg: bool, name: str)
bool
get_index_resolvers
Dict[str, ABCSeries]
get_cleaned_column_resolvers
Dict[str, ABCSeries]
eval(expr: str, inplace: bool, kwargs: Dict[str, object])
ensure_scope(level: int, global_dict: Dict[str, object],
local_dict: Dict[str, object],
resolvers: tuple, target: PandasObject, kwargs: Dict[str, object])
Scope
evaluate
object
PandasObject
PandasObject

The team’s implementation of Series.eval() can be visualized in the above sequence diagram. The method begins by validating the inplace parameter provided by the user (which indicates whether or not to mutate the Series) using the validate_bool_kwarg() method in util.Validators. It then retrieves the index resolvers for the Series, which map each index level to the indices contained at that level. In order to do this, it uses the get_index_resolvers() method from core.Generic, which is a base class of Series. The method also retrieves the column resolvers from core.Generic using get_cleaned_column_resolvers(). These resolvers map each column of data (in this case, only one since the object is a Series) to a name or reference. Once both resolvers have been retrieved, Series.eval() calls the eval() method within the computation.Eval class with the expression given by the user, the inplace parameter that was validated, as well as kwargs (keyword arguments) provided by the user, in which the resolvers are added to. computation.Eval’s eval() method then uses these kwargs and the resolvers they contain to build a Scope (the environment containing all variables and references the expression has access to) for the expression using the computation.Scope ensure_scope() method. Then, computation.Eval’s eval() calls the evaluate() method of the computation.AbstractEngine class (the base class for the engines available to pandas, namely numexpr (a dependency of the pandas project) by default or python otherwise). The engine’s evaluate() method returns a result, which could be either a Series, a certain value, or any python object. computation.Eval’s eval() method takes this result and ensures it is a PandasObject before returning it to Series.eval(), which in turn returns the PandasObject to the user.

Differences Between Current Implementation and Intended Design

In the previous deliverable, the team intended to reuse Dataframe.query() to implement Series.query() (to be implemented in Series.py). In reusing Dataframe.query(), the team also had to reuse Dataframe.eval().

Initially, the team thought that the column resolvers which are present in DataFrame.eval() could be removed from the implementation of Series.eval() since they appeared to be unnecessary due to the fact that Series objects only have a single column. However, after implementation and testing, the team found that these resolvers are required for evaluation, since a single column still needs a name or reference attached to it in order to be queried or manipulated.

Interactions Between Old and New Code

pandas.Series.query() has been modelled off of pandas.DataFrame.query(), and so a lot of the functionality implemented by DataFrame was required by the Series object in the pandas.core folder.

The query() function added to the Series class uses an eval() function to evaluate the boolean expressions used to query the data. This eval function itself calls into core.computation.eval, which in turn extends various backends, mainly the numexpr package, although keyword arguments in both query() and eval() allow the python engine to be used instead. The implementation of eval() in Series followed that of DataFrame and did not require much modification, as core.computation.eval can work with and return a Series object. What needed to be ensured during said implementation was that the Series.query() method extended the eval() method appropriately, for more efficient expression evaluations.

During our preliminary investigation into query(), we found some files and functions that interacted with DataFrame's version of query() and might need to be modified for Series, specifically pandas.core.generic and core.computation.scope, which contain resolvers and cleaners for data contained in DataFrames, some of which is needed by Series. For example, the has_resolvers() method (pandas\core\computation\scope.py:151) is called on by both query() and eval() methods and checks to see if there is extra scope. This method, and other similar methods were not found to need extensive modification either.

A note made previously was that, after modification of the Series object to add querying, the team would need to ensure the functionality of generic getters and handlers. __getattr__() (in pandas\core\generic.py) triggered an error prior to implementation when the user tried to access the non existent Series.query() or Series.eval() methods, and one of the roles of the unit and acceptance tests has been to ensure the smooth functioning of this area of the code.

Acceptance Tests

The acceptance testing suite is available in acceptance.py, a separate python file alongside this document. As the acceptance tests are meant to represent a subset of all possible interactions to showcase the use of the features from a customer perspective, they are based on general use cases and examples provided by the author of the issue on the pandas repository. The are divided into five sections in the file, namely:

- Unnamed Index Query
    - Some ways to use query() on an unnamed series
- Unnamed Index Eval
    - Some ways to use eval() on an unnamed series
- Named MultiIndex Query
    - Some ways to use query() on a named multi indexed series
- Named MultiIndex Eval
    - Some ways to use eval() on a named multi indexed series
- Error Handling
    - How the code handles generic issues like unrecognized index names

Note that the results of the tests in the Error Handling section are not particularly user friendly, and this is due to the fact that the errors are handled generically and outside the scope of the feature's requirements. This default behaviour is true to how DataFrame handles said errors and as the behavior of Series.query() and Series.eval() was modelled after that of DataFrame.query() and DataFrame.eval(), the response of the acceptance tests is as expected.

Unit Tests

The team's unit tests for Series.query() and Series.eval() can be found here. There are a total of 77 tests.

To run these unit tests, pytest can be used in the following manner within the project directory:

```
>>> pytest pandas/tests/series/test_query_eval.py
```

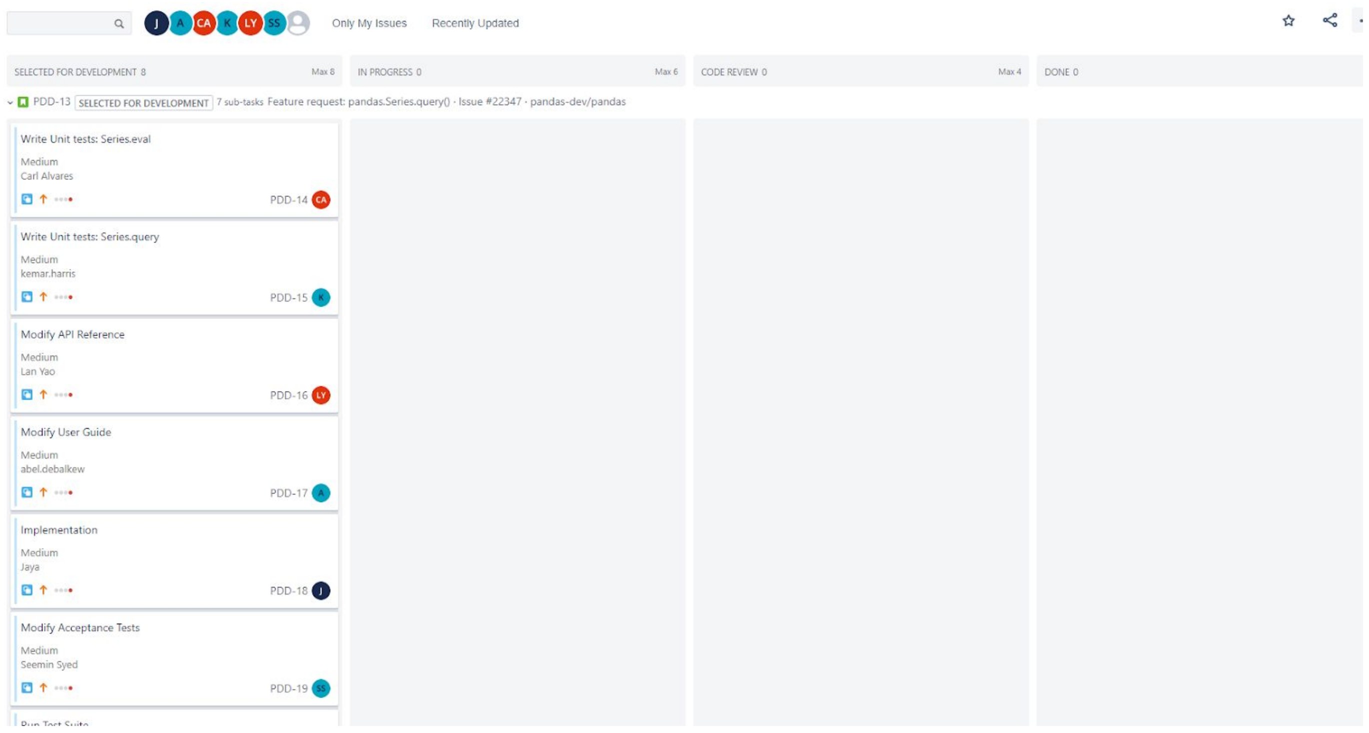
In addition, the entire pandas test suite (including the team's unit tests as well as the unit tests for the entire pandas project) can be ran with the following command, again in the project directory:

```
>>> pytest pandas
```

Be aware that due to the large number of tests contained in the entire suite, running them all will take around 20 minutes.

Process

Kanban Board



SELECTED FOR DEVELOPMENT 8 Max 8

IN PROGRESS 0 Max 6

CODE REVIEW 0 Max 4

DONE 0

PDD-13 SELECTED FOR DEVELOPMENT 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

Write Unit tests: Series.eval
Medium
Carl Alvares
PDD-14 CA

Write Unit tests: Series.query
Medium
kemar.harris
PDD-15 K

Modify API Reference
Medium
Lan Yao
PDD-16 LY

Modify User Guide
Medium
abel.debalkew
PDD-17 A

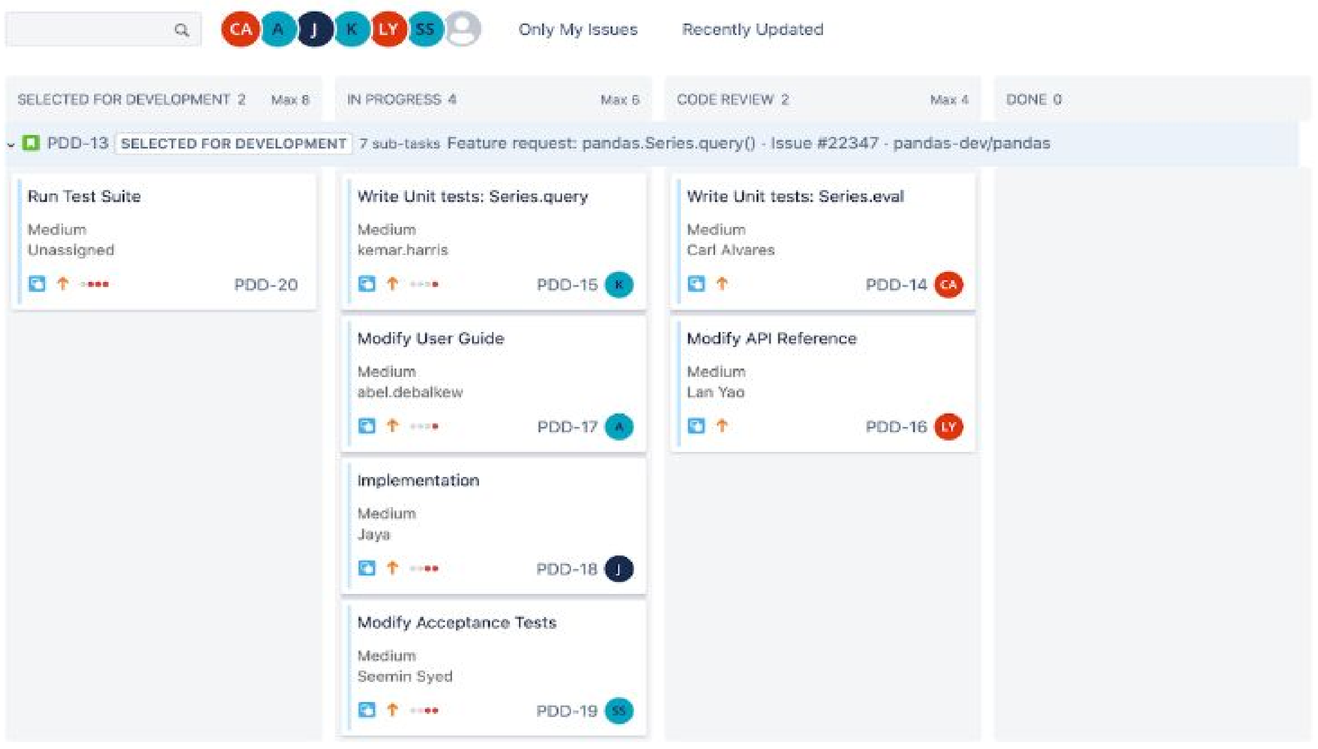
Implementation
Medium
Jaya
PDD-18 J

Modify Acceptance Tests
Medium
Seemin Syed
PDD-19 SS

Run Test Suite

As shown in the above screenshot, all team members have been delegated work. Note that at this point, the team has already begun/completed the following process activities in the previous deliverable: ‘Requirements Specification’, ‘Component Analysis’, ‘System Design with Reuse’.

As a result, at this point in the development process, the team intended to use the design document produced at the end of the ‘System Design with Reuse’ process activity to implement the feature. Also, only the task of formally writing up the User Guide was included as a task. To reiterate, the team had already gathered requirements during the ‘Requirements Specification’ process activity in the last deliverable.

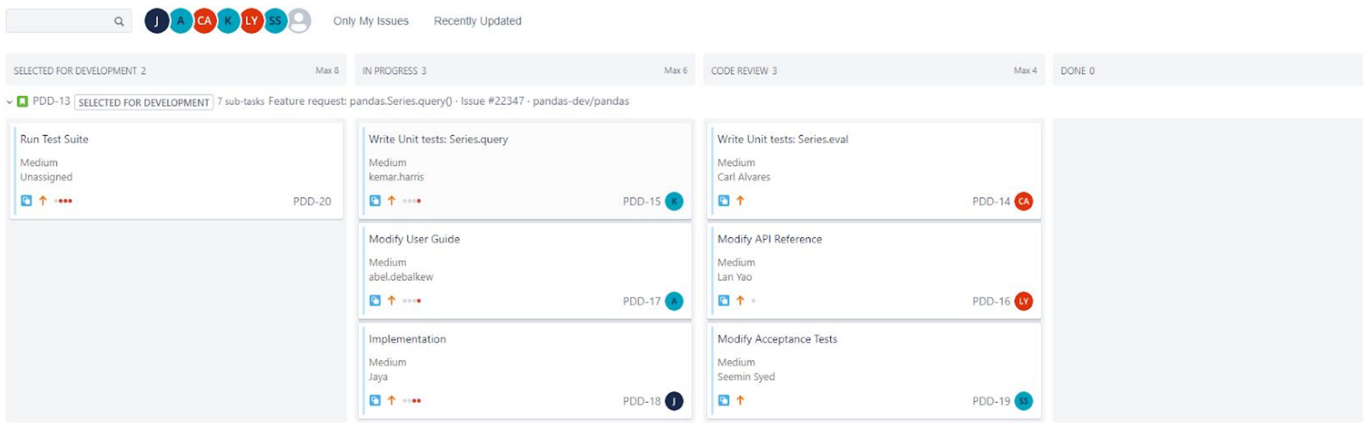

Only My Issues   Recently Updated

SELECTED FOR DEVELOPMENT 2 Max 8 | IN PROGRESS 4 Max 6 | CODE REVIEW 2 Max 4 | DONE 0

PDD-13 SELECTED FOR DEVELOPMENT 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

SELECTED FOR DEVELOPMENT 2	IN PROGRESS 4	CODE REVIEW 2	DONE 0
Run Test Suite Medium Unassigned PDD-20	Write Unit tests: Series.query Medium kemar.harris PDD-15	Write Unit tests: Series.eval Medium Carl Alvares PDD-14	
	Modify User Guide Medium abel.debalkew PDD-17	Modify API Reference Medium Lan Yao PDD-16	
	Implementation Medium Jaya PDD-18		
	Modify Acceptance Tests Medium Seemin Syed PDD-19		

As shown in the above screenshot, most tasks are under the ‘In-progress’ column, except for the task to run the test suite since this would be part of the ‘Validation’ process activity.

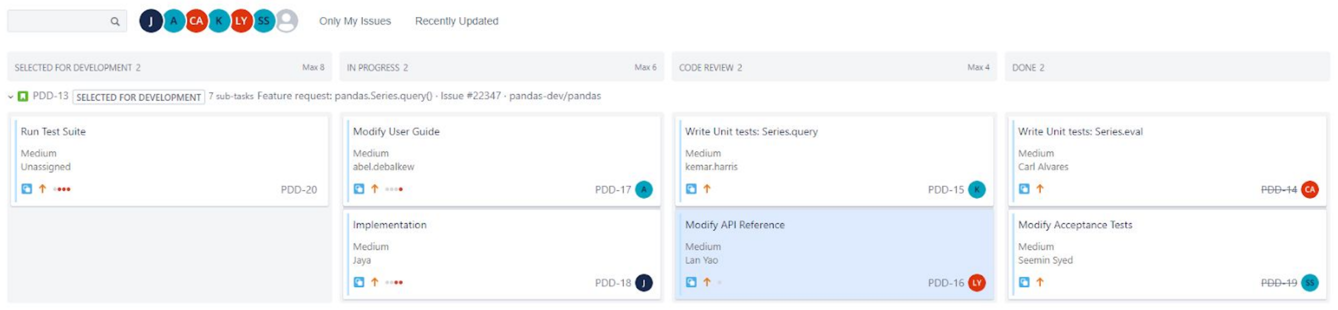

Only My Issues   Recently Updated

SELECTED FOR DEVELOPMENT 2 Max 8 | IN PROGRESS 3 Max 6 | CODE REVIEW 3 Max 4 | DONE 0

PDD-13 SELECTED FOR DEVELOPMENT 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

SELECTED FOR DEVELOPMENT 2	IN PROGRESS 3	CODE REVIEW 3	DONE 0
Run Test Suite Medium Unassigned PDD-20	Write Unit tests: Series.query Medium kemar.harris PDD-15	Write Unit tests: Series.eval Medium Carl Alvares PDD-14	
	Modify User Guide Medium abel.debalkew PDD-17	Modify API Reference Medium Lan Yao PDD-16	
	Implementation Medium Jaya PDD-18	Modify Acceptance Tests Medium Seemin Syed PDD-19	

As shown in the above screenshot, three tasks have been placed in the ‘Code Review’ column. Note the WIP limit for the ‘Code Review’ column.



SELECTED FOR DEVELOPMENT 2 Max 8 | IN PROGRESS 2 Max 6 | CODE REVIEW 2 Max 4 | DONE 2

PDD-13 SELECTED FOR DEVELOPMENT 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

Run Test Suite
Medium
Unassigned
PDD-20

Modify User Guide
Medium
abel.debalkew
PDD-17 A

Implementation
Medium
Jaya
PDD-18 J

Write Unit tests: Series.query
Medium
kemar.harris
PDD-15 K

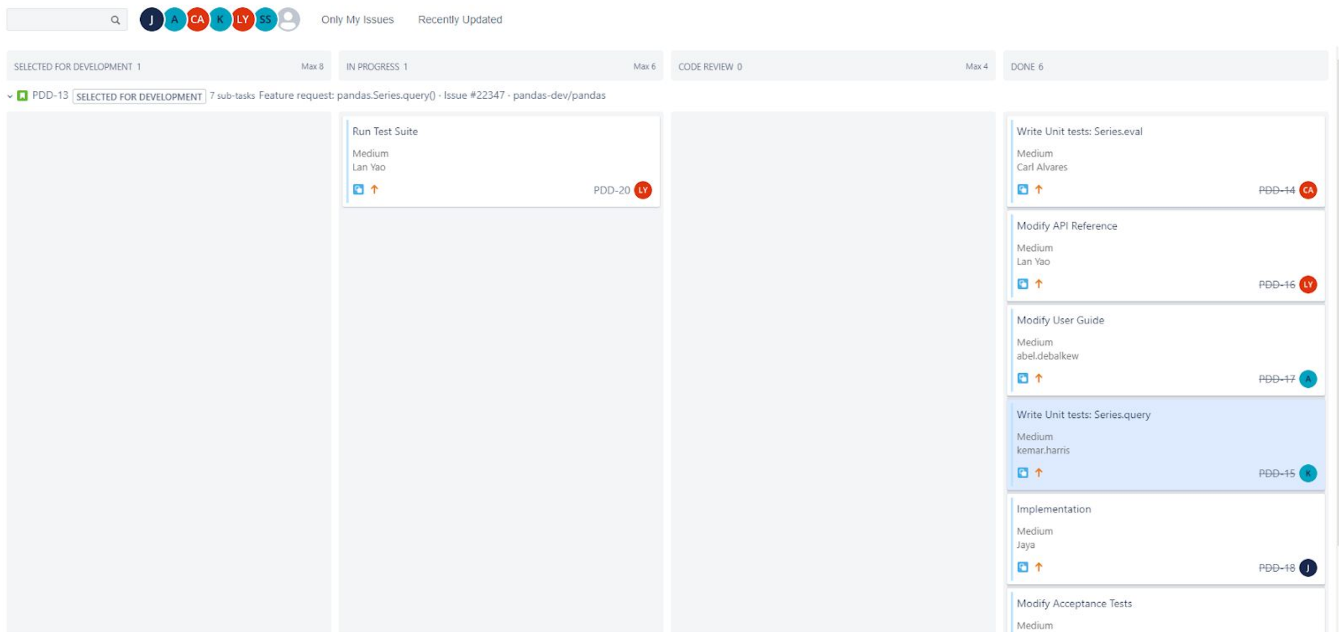
Modify API Reference
Medium
Lan Yao
PDD-16 LY

Write Unit tests: Series.eval
Medium
Carl Alvares
~~PDD-14~~ CA

Modify Acceptance Tests
Medium
Seemin Syed
~~PDD-19~~ SS

Although the number of cards in the ‘Code Review’ has not exceeded the WIP limit (which is four for the ‘Code Review’ column), some team members reviewed others’ work and moved these task cards to the ‘Done’ column before adding their cards to the ‘Code Review’ column. Click here to see an example PR made for a task under the ‘Code Review’ column. Here, the WIP limits helped the team ensure there is no accumulation of unreviewed work.

At this point, during the actual implementation of the feature, the team had to revisit the design plans as certain code from our reusable components (which we originally thought could be safe to remove) could not be safely removed without breaking the unit tests. This resulted in the team modifying the original design plan, which has been included in this report.



SELECTED FOR DEVELOPMENT 1 Max 8 | IN PROGRESS 1 Max 6 | CODE REVIEW 0 Max 4 | DONE 6

PDD-13 SELECTED FOR DEVELOPMENT 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

Run Test Suite
Medium
Lan Yao
PDD-20 LY

Write Unit tests: Series.eval
Medium
Carl Alvares
~~PDD-14~~ CA

Modify API Reference
Medium
Lan Yao
~~PDD-16~~ LY

Modify User Guide
Medium
abel.debalkew
~~PDD-17~~ A

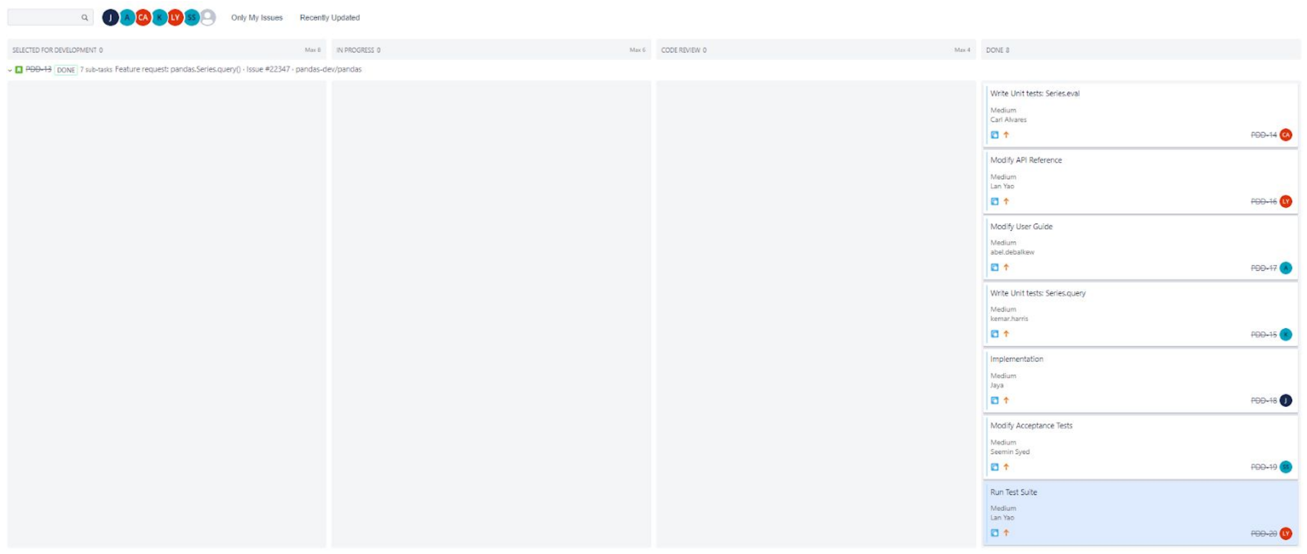
Write Unit tests: Series.query
Medium
kemar.harris
~~PDD-15~~ K

Implementation
Medium
Jaya
~~PDD-18~~ J

Modify Acceptance Tests
Medium

As shown in the screenshot above, most tasks have been placed under ‘Done’. As said in past deliverables, a task is considered to be ‘done’ if it has been reviewed and locally tested (if applicable). In addition, for the actual implementation of our feature, all unit and acceptance tests must pass in order for the task to be considered ‘done’. All code associated with tasks under the ‘Done’ column have also been merged to the series-query-feature branch.

The only remaining task is to run the test suite (which includes the pandas test suite, unit tests, and acceptance tests) on the series-query-feature branch and make any necessary changes to ensure these tests all pass; all these would be considered part of our ‘Validation’ process activity. These changes are intended to be recorded as new tasks on the Kanban board. However, in the end, no new tasks were created since all new tests passed. Although we did notice that some tests from the original suite failed, these same tests also failed before our changes, so our regression testing found no issues.



SELECTED FOR DEVELOPMENT 0 Max 8 | IN PROGRESS 0 Max 6 | CODE REVIEW 0 Max 4 | DONE 8

~~PDD-13~~ DONE 7 sub-tasks Feature request: pandas.Series.query() · Issue #22347 · pandas-dev/pandas

Write Unit tests: Series.eval
Medium
Carl Alvares
~~PDD-14~~ CA

Modify API Reference
Medium
Lan Yao
~~PDD-16~~ LY

Modify User Guide
Medium
abel.debalkew
~~PDD-17~~ A

Write Unit tests: Series.query
Medium
kemar.harris
~~PDD-15~~ K

Implementation
Medium
Jaya
~~PDD-18~~ J

Modify Acceptance Tests
Medium
Seemin Syed
~~PDD-19~~ SS

Run Test Suite
Medium
Lan Yao
~~PDD-20~~ LY

Kanban Daily Meetings: Summary

Note: All meetings were done via Google hangouts. The Kanban board guided the discussions.

<u>March 30, 2020 (12:00 - 12:10 pm)</u>

Abel: Reviewed the pandas user guide to see how the developers did their documentation for the functions.

Carl: Reviewed existing pandas unit tests to understand the format and capabilities of pytest and the utilities available for testing the pandas project.

Jaya: Have started the implementation of Series.query() by reusing components as specified in the design document. Simple tests pass, but want to play around with interfaces.

Kemar: Looked into DataFrame query methods to understand the range of possibilities for Series. Spent extra time investigating how the pandas team wanted the feature implemented, as there was some confusion in terms of syntax.

Lan: Read through pandas doc documentation to figure out how writing API reference works

Seemin: Design acceptance tests based on Deliverable 3 acceptance tests and feedback.

<u>March 31, 2020 (6: 00 - 6:05 pm)</u>
Abel: Moved my card into ‘In Progress’ and began working on the user guides for the Series.query() and Series.eval() functions.

Carl: Created initial unit tests using the DataFrame.eval() unit tests as a reference.

Jaya: No significant issues. Attempted to create a separate query interface (for Series and Dataframe to implement), but existing tests for Dataframe.query() and Dataframe.eval() fail….and new plan doesn’t really make much sense either. Continue with using design from the original design document.

Kemar: Wrote some base functionality tests for querying on an index and a multiindex for query. Merged with Carl’s existing tests.

Lan: Started writing API references but ran into some issues with pandas not recognizing the imported Validate module under numpydoc, so will spend time on fixing the issue.

Seemin: Check use case examples for DataFrame.Query() and DataFrame.Series() in pandas online documentation (User Guide and API Reference).

<u>April 5, 2020 (8:00 - 8:10 pm)</u>
Abel: I needed to test the examples from the user guide and compile it. So I made a separate branch locally that merged Jaya’s in order to do this. I found a few bugs in my examples and fixed them.

Carl: Opened a PR for my unit tests. Reviewed Lan’s API Reference PR since I noticed that we were nearing the WIP limit. Created a sequence diagram for Series.eval().

Jaya: Have spoken with Carl and Kemar re. not removing column resolvers since tests fail. Team will need to revisit the design document.

Kemar: Added more tests for an unnamed multiindex and backticks. Ran the existing tests to make sure they all passed, and fixed tests that were passing because of test syntax.

Lan: Solved the issue with module not recognized and pull requested the branch series_query_api, and will work on the sequence diagram

Seemin: Decide on acceptance tests types/subdivision based on use case and understandability.

April 6, 2020 (4:00 - 4:10 pm)
Abel: Reviewed Carl's unit tests and made sure that they passed on my machine. Moved my card into 'Code Review' and submitted a PR for the card on github. Made changes given by Carl.

Carl: Incorporated feedback from my PR. Added more coverage so that my unit tests were as comprehensive as Kemar's unit tests for Series.query(). Fixed some errors that caused some of my tests to fail. Reviewed Abel's PR for the User Guide and Kemar's PR for the Series.query() unit tests.

Jaya: Plan to put my card under the 'Code Review' column, but will review Seemin's work first since unreviewed work has started to pile up.

Kemar: Made my PR to the series-query-feature branch. Looked at Jaya's pull request for query documentation. Went over document diagrams.

Lan: Will review Abel's, Carl and Kemar's pull requests and finish up with sequence diagrams Also make changes to my pull request if any comments are given.

Seemin: Added acceptance test cases for subheading, from unnamed Series to named MultiIndexed Series, testing query, eval and error cases. Work needs to be reviewed. Will start putting together deliverable report.

April 7, 2020 (2:00 - 2:10 pm)
All code has been merged to the series-query-feature branch. We will need to run the test suite (pandas test suite, unit tests, and acceptance tests).