(continued from previous page)

```
In [122]: df3.sample(n=1, axis=1)
Out[122]:
   col1
0     1
1     2
2     3
```

Finally, one can also set a seed for `sample`'s random number generator using the `random_state` argument, which will accept either an integer (as a seed) or a NumPy RandomState object.

```
In [123]: df4 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

# With a given seed, the sample will always draw the same rows.
In [124]: df4.sample(n=2, random_state=2)
Out[124]:
   col1  col2
2     3     4
1     2     3

In [125]: df4.sample(n=2, random_state=2)
Out[125]:
   col1  col2
2     3     4
1     2     3
```

2.2.11 Setting with enlargement

The `.loc/[]` operations can perform enlargement when setting a non-existent key for that axis.

In the `Series` case this is effectively an appending operation.

```
In [126]: se = pd.Series([1, 2, 3])

In [127]: se
Out[127]:
0    1
1    2
2    3
dtype: int64

In [128]: se[5] = 5.

In [129]: se
Out[129]:
0    1.0
1    2.0
2    3.0
5    5.0
dtype: float64
```

A `DataFrame` can be enlarged on either axis via `.loc`.

```
In [130]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
   .....:                    columns=['A', 'B'])
```

(continues on next page)

(continued from previous page)

```
   .....:

In [131]: dfi
Out[131]:
   A  B
0  0  1
1  2  3
2  4  5

In [132]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [133]: dfi
Out[133]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an `append` operation on the `DataFrame`.

```
In [134]: dfi.loc[3] = 5

In [135]: dfi
Out[135]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```

2.2.12 Fast scalar value getting and setting

Since indexing with `[]` must handle a lot of cases (single-label access, slicing, boolean indexing, etc.), it has a bit of overhead in order to figure out what you're asking for. If you only want to access a scalar value, the fastest way is to use the `at` and `iat` methods, which are implemented on all of the data structures.

Similarly to `loc`, `at` provides **label** based scalar lookups, while, `iat` provides **integer** based lookups analogously to `iloc`

```
In [136]: s.iat[5]
Out[136]: 5

In [137]: df.at[dates[5], 'A']
Out[137]: -0.6736897080883706

In [138]: df.iat[3, 0]
Out[138]: 0.7215551622443669
```

You can also set using these same indexers.

```
In [139]: df.at[dates[5], 'E'] = 7

In [140]: df.iat[3, 0] = 7
```

`at` may enlarge the object in-place as above if the indexer is missing.

```
In [141]: df.at[dates[-1] + pd.Timedelta('1 day'), 0] = 7

In [142]: df
Out[142]:
                   A         B         C         D    E    0
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632  NaN  NaN
2000-01-02  1.212112 -0.173215  0.119209 -1.044236  NaN  NaN
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804  NaN  NaN
2000-01-04  7.000000 -0.706771 -1.039575  0.271860  NaN  NaN
2000-01-05 -0.424972  0.567020  0.276232 -1.087401  NaN  NaN
2000-01-06 -0.673690  0.113648 -1.478427  0.524988  7.0  NaN
2000-01-07  0.404705  0.577046 -1.715002 -1.039268  NaN  NaN
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885  NaN  NaN
2000-01-09       NaN       NaN       NaN       NaN  NaN  7.0
```

2.2.13 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: `|` for `or`, `&` for `and`, and `~` for `not`. These **must** be grouped by using parentheses, since by default Python will evaluate an expression such as `df['A'] > 2 & df['B'] < 3` as `df['A'] > (2 & df['B']) < 3`, while the desired evaluation order is `(df['A > 2) & (df['B'] < 3)`.

Using a boolean vector to index a Series works exactly as in a NumPy ndarray:

```
In [143]: s = pd.Series(range(-3, 4))

In [144]: s
Out[144]:
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int64

In [145]: s[s > 0]
Out[145]:
4    1
5    2
6    3
dtype: int64

In [146]: s[(s < -1) | (s > 0.5)]
Out[146]:
0   -3
1   -2
4    1
5    2
6    3
dtype: int64

In [147]: s[~(s < 0)]
Out[147]:
3    0
```

(continues on next page)

(continued from previous page)

```
4    1
5    2
6    3
dtype: int64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame's index (for example, something derived from one of the columns of the DataFrame):

```
In [148]: df[df['A'] > 0]
Out[148]: 
                   A         B         C         D   E   0
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632 NaN NaN
2000-01-02  1.212112 -0.173215  0.119209 -1.044236 NaN NaN
2000-01-04  7.000000 -0.706771 -1.039575  0.271860 NaN NaN
2000-01-07  0.404705  0.577046 -1.715002 -1.039268 NaN NaN
```

List comprehensions and the `map` method of Series can also be used to produce more complex criteria:

```
In [149]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'three', 'two', 'one', 'six
↪'],
   .....:                     'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                     'c': np.random.randn(7)})
   .....: 

# only want 'two' or 'three'
In [150]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [151]: df2[criterion]
Out[151]: 
       a  b         c
2    two  y  0.041290
3  three  x  0.361719
4    two  y -0.238075

# equivalent but slower
In [152]: df2[[x.startswith('t') for x in df2['a']]]
Out[152]: 
       a  b         c
2    two  y  0.041290
3  three  x  0.361719
4    two  y -0.238075

# Multiple criteria
In [153]: df2[criterion & (df2['b'] == 'x')]
Out[153]: 
       a  b         c
3  three  x  0.361719
```

With the choice methods *Selection by Label*, *Selection by Position*, and *Advanced Indexing* you may select along more than one axis using boolean vectors combined with other indexing expressions.

```
In [154]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
Out[154]: 
   b         c
3  x  0.361719
```

2.2.14 Indexing with isin

Consider the `isin()` method of `Series`, which returns a boolean vector that is true wherever the `Series` elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

```
In [155]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')

In [156]: s
Out[156]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [157]: s.isin([2, 4, 6])
Out[157]:
4    False
3    False
2     True
1    False
0     True
dtype: bool

In [158]: s[s.isin([2, 4, 6])]
Out[158]:
2    2
0    4
dtype: int64
```

The same method is available for `Index` objects and is useful for the cases when you don't know which of the sought labels are in fact present:

```
In [159]: s[s.index.isin([2, 4, 6])]
Out[159]:
4    0
2    2
dtype: int64

# compare it to the following
In [160]: s.reindex([2, 4, 6])
Out[160]:
2    2.0
4    0.0
6    NaN
dtype: float64
```

In addition to that, `MultiIndex` allows selecting a separate level to use in the membership check:

```
In [161]: s_mi = pd.Series(np.arange(6),
   .....:                  index=pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c
↪']]))
   .....:

In [162]: s_mi
Out[162]:
0  a    0
```

(continues on next page)

(continued from previous page)

```
   b    1
   c    2
1  a    3
   b    4
   c    5
dtype: int64

In [163]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
Out[163]:
0  c    2
1  a    3
dtype: int64

In [164]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
Out[164]:
0  a    0
   c    2
1  a    3
   c    5
dtype: int64
```

DataFrame also has an *isin()* method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

```
In [165]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
   .....:                    'ids2': ['a', 'n', 'c', 'n']})
   .....:

In [166]: values = ['a', 'b', 1, 3]

In [167]: df.isin(values)
Out[167]:
    vals    ids   ids2
0   True   True   True
1  False   True  False
2   True  False  False
3  False  False  False
```

Oftentimes you'll want to match certain values with certain columns. Just make values a `dict` where the key is the column, and the value is a list of items you want to check for.

```
In [168]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [169]: df.isin(values)
Out[169]:
    vals    ids   ids2
0   True   True  False
1  False   True  False
2   True  False  False
3  False  False  False
```

Combine DataFrame's `isin` with the `any()` and `all()` methods to quickly select subsets of your data that meet a given criteria. To select a row where each column meets its own criterion:

```
In [170]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}
```

(continues on next page)

(continued from previous page)

```
In [171]: row_mask = df.isin(values).all(1)

In [172]: df[row_mask]
Out[172]:
   vals ids ids2
0     1   a    a
```

2.2.15 The `where()` Method and Masking

Selecting values from a Series with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows:

```
In [173]: s[s > 0]
Out[173]:
3    1
2    2
1    3
0    4
dtype: int64
```

To return a Series of the same shape as the original:

```
In [174]: s.where(s > 0)
Out[174]:
4    NaN
3    1.0
2    2.0
1    3.0
0    4.0
dtype: float64
```

Selecting values from a DataFrame with a boolean criterion now also preserves input data shape. `where` is used under the hood as the implementation. The code below is equivalent to `df.where(df < 0)`.

```
In [175]: df[df < 0]
Out[175]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525       NaN       NaN
2000-01-02 -0.352480       NaN -1.192319       NaN
2000-01-03 -0.864883       NaN -0.227870       NaN
2000-01-04       NaN -1.222082       NaN -1.233203
2000-01-05       NaN -0.605656 -1.169184       NaN
2000-01-06       NaN -0.948458       NaN -0.684718
2000-01-07 -2.670153 -0.114722       NaN -0.048048
2000-01-08       NaN       NaN -0.048788 -0.808838
```

In addition, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```
In [176]: df.where(df < 0, -df)
Out[176]:
                   A         B         C         D
```

(continues on next page)

(continued from previous page)

```
2000-01-01 -2.104139 -1.309525 -0.485855 -0.245166
2000-01-02 -0.352480 -0.390389 -1.192319 -1.655824
2000-01-03 -0.864883 -0.299674 -0.227870 -0.281059
2000-01-04 -0.846958 -1.222082 -0.600705 -1.233203
2000-01-05 -0.669692 -0.605656 -1.169184 -0.342416
2000-01-06 -0.868584 -0.948458 -2.297780 -0.684718
2000-01-07 -2.670153 -0.114722 -0.168904 -0.048048
2000-01-08 -0.801196 -1.392071 -0.048788 -0.808838
```

You may wish to set values based on some boolean criteria. This can be done intuitively like so:

```
In [177]: s2 = s.copy()

In [178]: s2[s2 < 0] = 0

In [179]: s2
Out[179]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [180]: df2 = df.copy()

In [181]: df2[df2 < 0] = 0

In [182]: df2
Out[182]:
                   A         B         C         D
2000-01-01  0.000000  0.000000  0.485855  0.245166
2000-01-02  0.000000  0.390389  0.000000  1.655824
2000-01-03  0.000000  0.299674  0.000000  0.281059
2000-01-04  0.846958  0.000000  0.600705  0.000000
2000-01-05  0.669692  0.000000  0.000000  0.342416
2000-01-06  0.868584  0.000000  2.297780  0.000000
2000-01-07  0.000000  0.000000  0.168904  0.000000
2000-01-08  0.801196  1.392071  0.000000  0.000000
```

By default, `where` returns a modified copy of the data. There is an optional parameter `inplace` so that the original data can be modified without creating a copy:

```
In [183]: df_orig = df.copy()

In [184]: df_orig.where(df > 0, -df, inplace=True)

In [185]: df_orig
Out[185]:
                   A         B         C         D
2000-01-01  2.104139  1.309525  0.485855  0.245166
2000-01-02  0.352480  0.390389  1.192319  1.655824
2000-01-03  0.864883  0.299674  0.227870  0.281059
2000-01-04  0.846958  1.222082  0.600705  1.233203
2000-01-05  0.669692  0.605656  1.169184  0.342416
2000-01-06  0.868584  0.948458  2.297780  0.684718
```

(continues on next page)

(continued from previous page)

```
2000-01-07  2.670153  0.114722  0.168904  0.048048
2000-01-08  0.801196  1.392071  0.048788  0.808838
```

Note: The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

```
In [186]: df.where(df < 0, -df) == np.where(df < 0, df, -df)
Out[186]:
               A     B     C     D
2000-01-01  True  True  True  True
2000-01-02  True  True  True  True
2000-01-03  True  True  True  True
2000-01-04  True  True  True  True
2000-01-05  True  True  True  True
2000-01-06  True  True  True  True
2000-01-07  True  True  True  True
2000-01-08  True  True  True  True
```

Alignment

Furthermore, `where` aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via `.loc` (but on the contents rather than the axis labels).

```
In [187]: df2 = df.copy()

In [188]: df2[df2[1:4] > 0] = 3

In [189]: df2
Out[189]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525  0.485855  0.245166
2000-01-02 -0.352480  3.000000 -1.192319  3.000000
2000-01-03 -0.864883  3.000000 -0.227870  3.000000
2000-01-04  3.000000 -1.222082  3.000000 -1.233203
2000-01-05  0.669692 -0.605656 -1.169184  0.342416
2000-01-06  0.868584 -0.948458  2.297780 -0.684718
2000-01-07 -2.670153 -0.114722  0.168904 -0.048048
2000-01-08  0.801196  1.392071 -0.048788 -0.808838
```

Where can also accept `axis` and `level` parameters to align the input when performing the `where`.

```
In [190]: df2 = df.copy()

In [191]: df2.where(df2 > 0, df2['A'], axis='index')
Out[191]:
                   A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02 -0.352480  0.390389 -0.352480  1.655824
2000-01-03 -0.864883  0.299674 -0.864883  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196
```

This is equivalent to (but faster than) the following.

```
In [192]: df2 = df.copy()

In [193]: df.apply(lambda x, y: x.where(x > 0, y), y=df['A'])
Out[193]:
                   A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02 -0.352480  0.390389 -0.352480  1.655824
2000-01-03 -0.864883  0.299674 -0.864883  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196
```

`where` can accept a callable as condition and `other` arguments. The function must be with one argument (the calling Series or DataFrame) and that returns valid output as condition and `other` argument.

```
In [194]: df3 = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [4, 5, 6],
   .....:                     'C': [7, 8, 9]})
   .....:

In [195]: df3.where(lambda x: x > 4, lambda x: x + 10)
Out[195]:
    A   B  C
0  11  14  7
1  12   5  8
2  13   6  9
```

Mask

mask() is the inverse boolean operation of `where`.

```
In [196]: s.mask(s >= 0)
Out[196]:
4   NaN
3   NaN
2   NaN
1   NaN
0   NaN
dtype: float64

In [197]: df.mask(df >= 0)
Out[197]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525       NaN       NaN
2000-01-02 -0.352480       NaN -1.192319       NaN
2000-01-03 -0.864883       NaN -0.227870       NaN
2000-01-04       NaN -1.222082       NaN -1.233203
2000-01-05       NaN -0.605656 -1.169184       NaN
2000-01-06       NaN -0.948458       NaN -0.684718
2000-01-07 -2.670153 -0.114722       NaN -0.048048
2000-01-08       NaN       NaN -0.048788 -0.808838
```

2.2.16 The `query()` Method

`DataFrame` objects have a `query()` method that allows selection using an expression.

You can get the value of the frame where column `b` has values between the values of columns `a` and `c`. For example:

```
In [198]: n = 10

In [199]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [200]: df
Out[200]:
          a         b         c
0  0.438921  0.118680  0.863670
1  0.138138  0.577363  0.686602
2  0.595307  0.564592  0.520630
3  0.913052  0.926075  0.616184
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
6  0.792342  0.216974  0.564056
7  0.397890  0.454131  0.915716
8  0.074315  0.437913  0.019794
9  0.559209  0.502065  0.026437

# pure python
In [201]: df[(df['a'] < df['b']) & (df['b'] < df['c'])]
Out[201]:
          a         b         c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716

# query
In [202]: df.query('(a < b) & (b < c)')
Out[202]:
          a         b         c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716
```

Do the same thing but fall back on a named index if there is no column with the name `a`.

```
In [203]: df = pd.DataFrame(np.random.randint(n / 2, size=(n, 2)), columns=list('bc'))

In [204]: df.index.name = 'a'

In [205]: df
Out[205]:
   b  c
a
0  0  4
1  0  1
2  3  4
3  4  3
4  1  4
5  0  3
```

(continues on next page)

(continued from previous page)

```
6  0  1
7  3  4
8  2  3
9  1  1

In [206]: df.query('a < b and b < c')
Out[206]:
   b  c
a
2  3  4
```

If instead you don't want to or cannot name your index, you can use the name `index` in your query expression:

```
In [207]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)), columns=list('bc'))

In [208]: df
Out[208]:
   b  c
0  3  1
1  3  0
2  5  6
3  5  2
4  7  4
5  0  1
6  2  5
7  0  1
8  6  0
9  7  9

In [209]: df.query('index < b < c')
Out[209]:
   b  c
2  5  6
```

Note: If the name of your index overlaps with a column name, the column name is given precedence. For example,

```
In [210]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})

In [211]: df.index.name = 'a'

In [212]: df.query('a > 2')  # uses the column 'a', not the index
Out[212]:
   a
a
1  3
3  3
```

You can still use the index in a query expression by using the special identifier 'index':

```
In [213]: df.query('index > 2')
Out[213]:
   a
a
3  3
4  2
```

If for some reason you have a column named `index`, then you can refer to the index as `ilevel_0` as well, but at this point you should consider renaming your columns to something less ambiguous.

MultiIndex query() Syntax

You can also use the levels of a `DataFrame` with a *MultiIndex* as if they were columns in the frame:

```
In [214]: n = 10

In [215]: colors = np.random.choice(['red', 'green'], size=n)

In [216]: foods = np.random.choice(['eggs', 'ham'], size=n)

In [217]: colors
Out[217]:
array(['red', 'red', 'red', 'green', 'green', 'green', 'green', 'green',
       'green', 'green'], dtype='<U5')

In [218]: foods
Out[218]:
array(['ham', 'ham', 'eggs', 'eggs', 'eggs', 'ham', 'ham', 'eggs', 'eggs',
       'eggs'], dtype='<U4')

In [219]: index = pd.MultiIndex.from_arrays([colors, foods], names=['color', 'food'])

In [220]: df = pd.DataFrame(np.random.randn(n, 2), index=index)

In [221]: df
Out[221]:
                   0         1
color food
red   ham   0.194889 -0.381994
      ham   0.318587  2.089075
      eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
      eggs -2.029766  0.792652
      ham   0.461007 -0.542749
      ham  -0.305384 -0.479195
      eggs  0.095031 -0.270099
      eggs -0.707140 -0.773882
      eggs  0.229453  0.304418

In [222]: df.query('color == "red"')
Out[222]:
                   0         1
color food
red   ham   0.194889 -0.381994
      ham   0.318587  2.089075
      eggs -0.728293 -0.090255
```

If the levels of the `MultiIndex` are unnamed, you can refer to them using special names:

```
In [223]: df.index.names = [None, None]

In [224]: df
Out[224]:
```

(continues on next page)

(continued from previous page)

```
                    0         1
red   ham    0.194889 -0.381994
      ham    0.318587  2.089075
      eggs  -0.728293 -0.090255
green eggs  -0.748199  1.318931
      eggs  -2.029766  0.792652
      ham    0.461007 -0.542749
      ham   -0.305384 -0.479195
      eggs   0.095031 -0.270099
      eggs  -0.707140 -0.773882
      eggs   0.229453  0.304418

In [225]: df.query('ilevel_0 == "red"')
Out[225]:
                  0         1
red ham    0.194889 -0.381994
    ham    0.318587  2.089075
    eggs  -0.728293 -0.090255
```

The convention is `ilevel_0`, which means "index level 0" for the 0th level of the `index`.

`query()` Use Cases

A use case for *`query()`* is when you have a collection of *`DataFrame`* objects that have a subset of column names (or index levels/names) in common. You can pass the same query to both frames *without* having to specify which frame you're interested in querying

```
In [226]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [227]: df
Out[227]:
          a         b         c
0  0.224283  0.736107  0.139168
1  0.302827  0.657803  0.713897
2  0.611185  0.136624  0.984960
3  0.195246  0.123436  0.627712
4  0.618673  0.371660  0.047902
5  0.480088  0.062993  0.185760
6  0.568018  0.483467  0.445289
7  0.309040  0.274580  0.587101
8  0.258993  0.477769  0.370255
9  0.550459  0.840870  0.304611

In [228]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)

In [229]: df2
Out[229]:
           a         b         c
0   0.357579  0.229800  0.596001
1   0.309059  0.957923  0.965663
2   0.123102  0.336914  0.318616
3   0.526506  0.323321  0.860813
4   0.518736  0.486514  0.384724
5   0.190804  0.505723  0.614533
6   0.891939  0.623977  0.676639
```

(continues on next page)

(continued from previous page)

```
7   0.480559  0.378528  0.460858
8   0.420223  0.136404  0.141295
9   0.732206  0.419540  0.604675
10  0.604466  0.848974  0.896165
11  0.589168  0.920046  0.732716

In [230]: expr = '0.0 <= a <= c <= 0.5'

In [231]: map(lambda frame: frame.query(expr), [df, df2])
Out[231]: <map at 0x7f533c102cd0>
```

query() Python versus pandas Syntax Comparison

Full numpy-like syntax:

```
In [232]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=list('abc'))

In [233]: df
Out[233]:
   a  b  c
0  7  8  9
1  1  0  7
2  2  7  2
3  6  2  2
4  2  6  3
5  3  8  2
6  1  7  2
7  5  1  5
8  9  8  0
9  1  5  0

In [234]: df.query('(a < b) & (b < c)')
Out[234]:
   a  b  c
0  7  8  9

In [235]: df[(df['a'] < df['b']) & (df['b'] < df['c'])]
Out[235]:
   a  b  c
0  7  8  9
```

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than & and |).

```
In [236]: df.query('a < b & b < c')
Out[236]:
   a  b  c
0  7  8  9
```

Use English instead of symbols:

```
In [237]: df.query('a < b and b < c')
Out[237]:
   a  b  c
0  7  8  9
```

Pretty close to how you might write it on paper:

```
In [238]: df.query('a < b < c')
Out[238]:
   a  b  c
0  7  8  9
```

The `in` and `not in` operators

`query()` also supports special use of Python's `in` and `not in` comparison operators, providing a succinct syntax for calling the `isin` method of a `Series` or `DataFrame`.

```
# get all rows where columns "a" and "b" have overlapping values
In [239]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbbcccc'),
   .....:                    'c': np.random.randint(5, size=12),
   .....:                    'd': np.random.randint(9, size=12)})
   .....:

In [240]: df
Out[240]:
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
3   b  a  2  1
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

In [241]: df.query('a in b')
Out[241]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2

# How you'd do it in pure Python
In [242]: df[df['a'].isin(df['b'])]
Out[242]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2

In [243]: df.query('a not in b')
Out[243]:
    a  b  c  d
```

(continues on next page)

(continued from previous page)

```
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

# pure Python
In [244]: df[~df['a'].isin(df['b'])]
Out[244]:
    a  b  c  d
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2
```

You can combine this with other expressions for very succinct queries:

```
# rows where cols a and b have overlapping values
# and col c's values are less than col d's
In [245]: df.query('a in b and c < d')
Out[245]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
4  c  b  3  6
5  c  b  0  2

# pure Python
In [246]: df[df['b'].isin(df['a']) & (df['c'] < df['d'])]
Out[246]:
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
4   c  b  3  6
5   c  b  0  2
10  f  c  0  6
11  f  c  1  2
```

Note: Note that `in` and `not in` are evaluated in Python, since `numexpr` has no equivalent of this operation. However, **only the** `in`/`not in` **expression itself** is evaluated in vanilla Python. For example, in the expression

```
df.query('a in b + c + d')
```

`(b + c + d)` is evaluated by `numexpr` and *then* the `in` operation is evaluated in plain Python. In general, any operations that can be evaluated using `numexpr` will be.

Special use of the == operator with `list` objects

Comparing a `list` of values to a column using ==/!= works similarly to `in`/`not in`.

```
In [247]: df.query('b == ["a", "b", "c"]')
Out[247]:
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
3   b  a  2  1
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

# pure Python
In [248]: df[df['b'].isin(["a", "b", "c"])]
Out[248]:
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
3   b  a  2  1
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

In [249]: df.query('c == [1, 2]')
Out[249]:
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2

In [250]: df.query('c != [1, 2]')
Out[250]:
    a  b  c  d
1   a  a  4  7
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
8   e  c  4  3
10  f  c  0  6

# using in/not in
```

(continues on next page)

(continued from previous page)

```
In [251]: df.query('[1, 2] in c')
Out[251]:
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2

In [252]: df.query('[1, 2] not in c')
Out[252]:
    a  b  c  d
1   a  a  4  7
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
8   e  c  4  3
10  f  c  0  6

# pure Python
In [253]: df[df['c'].isin([1, 2])]
Out[253]:
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2
```

Boolean operators

You can negate boolean expressions with the word `not` or the ~ operator.

```
In [254]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [255]: df['bools'] = np.random.rand(len(df)) > 0.5

In [256]: df.query('~bools')
Out[256]:
          a         b         c  bools
2  0.697753  0.212799  0.329209  False
7  0.275396  0.691034  0.826619  False
8  0.190649  0.558748  0.262467  False

In [257]: df.query('not bools')
Out[257]:
          a         b         c  bools
2  0.697753  0.212799  0.329209  False
7  0.275396  0.691034  0.826619  False
8  0.190649  0.558748  0.262467  False

In [258]: df.query('not bools') == df[~df['bools']]
Out[258]:
```

(continues on next page)

(continued from previous page)

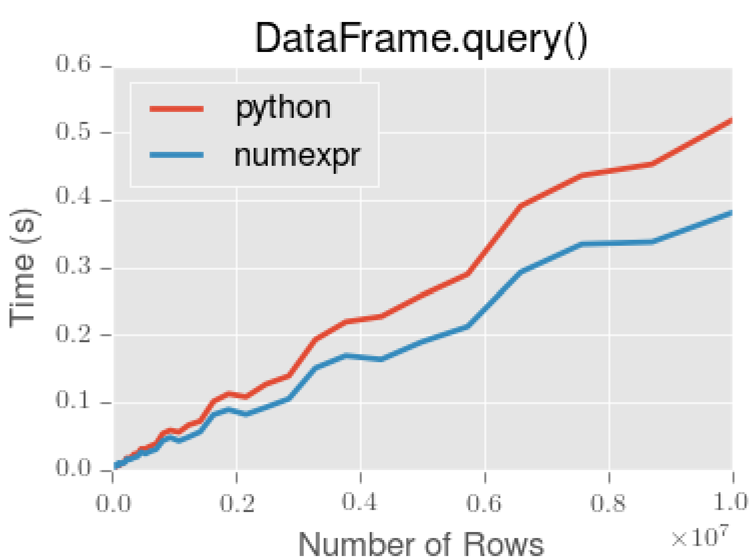
```
      a     b     c  bools
2  True  True  True   True
7  True  True  True   True
8  True  True  True   True
```

Of course, expressions can be arbitrarily complex too:

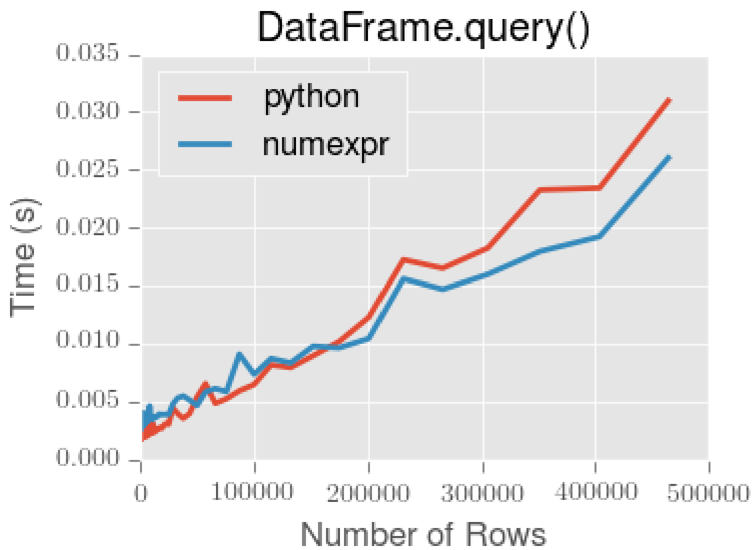
```
# short query syntax
In [259]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [260]: longer = df[(df['a'] < df['b'])
   .....:              & (df['b'] < df['c'])
   .....:              & (~df['bools'])
   .....:              | (df['bools'] > 2)]
   .....:

In [261]: shorter
Out[261]:
          a         b         c  bools
7  0.275396  0.691034  0.826619  False

In [262]: longer
Out[262]:
          a         b         c  bools
7  0.275396  0.691034  0.826619  False

In [263]: shorter == longer
Out[263]:
      a     b     c  bools
7  True  True  True   True
```

Performance of `query()`

`DataFrame.query()` using `numexpr` is slightly faster than Python for large frames.



Note: You will only see the performance benefits of using the `numexpr` engine with `DataFrame.query()` if your frame has more than approximately 200,000 rows.



This plot was created using a `DataFrame` with 3 columns each containing floating point values generated using `numpy.random.randn()`.

2.2.17 Duplicate data

If you want to identify and remove duplicate rows in a DataFrame, there are two methods that will help: `duplicated` and `drop_duplicates`. Each takes as an argument the columns to use to identify duplicated rows.

- `duplicated` returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.
- `drop_duplicates` removes duplicate rows.

By default, the first observed row of a duplicate set is considered unique, but each method has a `keep` parameter to specify targets to be kept.

- `keep='first'` (default): mark / drop duplicates except for the first occurrence.
- `keep='last'`: mark / drop duplicates except for the last occurrence.
- `keep=False`: mark / drop all duplicates.

```
In [264]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two', 'three', 'four
↪'],
   .....:                     'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
   .....:                     'c': np.random.randn(7)})
   .....:

In [265]: df2
Out[265]:
       a  b         c
0    one  x -1.067137
```

(continues on next page)

(continued from previous page)

```
1    one  y  0.309500
2    two  x -0.211056
3    two  y -1.842023
4    two  x -0.390820
5  three  x -1.964475
6   four  x  1.298329

In [266]: df2.duplicated('a')
Out[266]:
0    False
1     True
2    False
3     True
4     True
5    False
6    False
dtype: bool

In [267]: df2.duplicated('a', keep='last')
Out[267]:
0     True
1    False
2     True
3     True
4    False
5    False
6    False
dtype: bool

In [268]: df2.duplicated('a', keep=False)
Out[268]:
0     True
1     True
2     True
3     True
4     True
5    False
6    False
dtype: bool

In [269]: df2.drop_duplicates('a')
Out[269]:
       a  b         c
0    one  x -1.067137
2    two  x -0.211056
5  three  x -1.964475
6   four  x  1.298329

In [270]: df2.drop_duplicates('a', keep='last')
Out[270]:
       a  b         c
1    one  y  0.309500
4    two  x -0.390820
5  three  x -1.964475
6   four  x  1.298329

In [271]: df2.drop_duplicates('a', keep=False)
```

(continues on next page)

(continued from previous page)

```
Out[271]:
       a  b         c
5  three  x -1.964475
6   four  x  1.298329
```

Also, you can pass a list of columns to identify duplications.

```
In [272]: df2.duplicated(['a', 'b'])
Out[272]:
0    False
1    False
2    False
3    False
4     True
5    False
6    False
dtype: bool

In [273]: df2.drop_duplicates(['a', 'b'])
Out[273]:
       a  b         c
0    one  x -1.067137
1    one  y  0.309500
2    two  x -0.211056
3    two  y -1.842023
5  three  x -1.964475
6   four  x  1.298329
```

To drop duplicates by index value, use `Index.duplicated` then perform slicing. The same set of options are available for the `keep` parameter.

```
In [274]: df3 = pd.DataFrame({'a': np.arange(6),
   .....:                     'b': np.random.randn(6)},
   .....:                    index=['a', 'a', 'b', 'c', 'b', 'a'])
   .....:

In [275]: df3
Out[275]:
   a         b
a  0  1.440455
a  1  2.456086
b  2  1.038402
c  3 -0.894409
b  4  0.683536
a  5  3.082764

In [276]: df3.index.duplicated()
Out[276]: array([False,  True, False, False,  True,  True])

In [277]: df3[~df3.index.duplicated()]
Out[277]:
   a         b
a  0  1.440455
b  2  1.038402
c  3 -0.894409
```

(continues on next page)

(continued from previous page)

```
In [278]: df3[~df3.index.duplicated(keep='last')]
Out[278]:
   a         b
c  3 -0.894409
b  4  0.683536
a  5  3.082764

In [279]: df3[~df3.index.duplicated(keep=False)]
Out[279]:
   a         b
c  3 -0.894409
```

2.2.18 Dictionary-like `get()` method

Each of Series or DataFrame have a `get` method which can return a default value.

```
In [280]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [281]: s.get('a')   # equivalent to s['a']
Out[281]: 1

In [282]: s.get('x', default=-1)
Out[282]: -1
```

2.2.19 The `lookup()` method

Sometimes you want to extract a set of values given a sequence of row labels and column labels, and the `lookup` method allows for this and returns a NumPy array. For instance:

```
In [283]: dflookup = pd.DataFrame(np.random.rand(20, 4), columns = ['A', 'B', 'C', 'D
↪'])

In [284]: dflookup.lookup(list(range(0, 10, 2)), ['B', 'C', 'A', 'B', 'D'])
Out[284]: array([0.3506, 0.4779, 0.4825, 0.9197, 0.5019])
```

2.2.20 Index objects

The pandas *`Index`* class and its subclasses can be viewed as implementing an *ordered multiset*. Duplicates are allowed. However, if you try to convert an *`Index`* object with duplicate entries into a `set`, an exception will be raised.

`Index` also provides the infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create an *`Index`* directly is to pass a `list` or other sequence to *`Index`*:

```
In [285]: index = pd.Index(['e', 'd', 'a', 'b'])

In [286]: index
Out[286]: Index(['e', 'd', 'a', 'b'], dtype='object')

In [287]: 'd' in index
Out[287]: True
```

You can also pass a `name` to be stored in the index:

```
In [288]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')

In [289]: index.name
Out[289]: 'something'
```

The name, if set, will be shown in the console display:

```
In [290]: index = pd.Index(list(range(5)), name='rows')

In [291]: columns = pd.Index(['A', 'B', 'C'], name='cols')

In [292]: df = pd.DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [293]: df
Out[293]:
cols          A         B         C
rows
0      1.295989  0.185778  0.436259
1      0.678101  0.311369 -0.528378
2     -0.674808 -1.103529 -0.656157
3      1.889957  2.076651 -1.102192
4     -1.211795 -0.791746  0.634724

In [294]: df['A']
Out[294]:
rows
0    1.295989
1    0.678101
2   -0.674808
3    1.889957
4   -1.211795
Name: A, dtype: float64
```

Setting metadata

Indexes are "mostly immutable", but it is possible to set and change their metadata, like the index `name` (or, for `MultiIndex`, `levels` and `codes`).

You can use the `rename`, `set_names`, `set_levels`, and `set_codes` to set these attributes directly. They default to returning a copy; however, you can specify `inplace=True` to have the data change in place.

See *Advanced Indexing* for usage of MultiIndexes.

```
In [295]: ind = pd.Index([1, 2, 3])

In [296]: ind.rename("apple")
Out[296]: Int64Index([1, 2, 3], dtype='int64', name='apple')

In [297]: ind
Out[297]: Int64Index([1, 2, 3], dtype='int64')

In [298]: ind.set_names(["apple"], inplace=True)

In [299]: ind.name = "bob"
```

(continues on next page)