(continued from previous page)

```
1    5    77    84
2   10    96    65
```

current behavior:

```
In [9]: df.groupby(ts, as_index=False).max()
Out[9]:
   jim  joe
0   72   83
1   77   84
2   96   65

[3 rows x 2 columns]
```

- `groupby` will not erroneously exclude columns if the column name conflicts with the grouper name (GH8112):

```
In [10]: df = pd.DataFrame({'jim': range(5), 'joe': range(5, 10)})

In [11]: df
Out[11]:
   jim  joe
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9

[5 rows x 2 columns]

In [12]: gr = df.groupby(df['jim'] < 2)
```

previous behavior (excludes 1st column from output):

```
In [4]: gr.apply(sum)
Out[4]:
       joe
jim
False   24
True    11
```

current behavior:

```
In [13]: gr.apply(sum)
Out[13]:
       jim  joe
jim
False    9   24
True     1   11

[2 rows x 2 columns]
```

- Support for slicing with monotonic decreasing indexes, even if `start` or `stop` is not found in the index (GH7860):

```
In [14]: s = pd.Series(['a', 'b', 'c', 'd'], [4, 3, 2, 1])
```

(continues on next page)

(continued from previous page)

```
In [15]: s
Out[15]:
4    a
3    b
2    c
1    d
Length: 4, dtype: object
```

previous behavior:

```
In [8]: s.loc[3.5:1.5]
KeyError: 3.5
```

current behavior:

```
In [16]: s.loc[3.5:1.5]
Out[16]:
3    b
2    c
Length: 2, dtype: object
```

- `io.data.Options` has been fixed for a change in the format of the Yahoo Options page (GH8612), (GH8741)

Note: As a result of a change in Yahoo's option page layout, when an expiry date is given, `Options` methods now return data for a single expiry date. Previously, methods returned all data for the selected month.

The `month` and `year` parameters have been undeprecated and can be used to get all options data for a given month.

If an expiry date that is not valid is given, data for the next expiry after the given date is returned.

Option data frames are now saved on the instance as `callsYYMMDD` or `putsYYMMDD`. Previously they were saved as `callsMMYY` and `putsMMYY`. The next expiry is saved as `calls` and `puts`.

New features:

  - The expiry parameter can now be a single date or a list-like object containing dates.
  - A new property `expiry_dates` was added, which returns all available expiry dates.

Current behavior:

```
In [17]: from pandas.io.data import Options

In [18]: aapl = Options('aapl', 'yahoo')

In [19]: aapl.get_call_data().iloc[0:5, 0:1]
Out[19]:
                                             Last
Strike Expiry     Type Symbol
80     2014-11-14 call AAPL141114C00080000  29.05
84     2014-11-14 call AAPL141114C00084000  24.80
85     2014-11-14 call AAPL141114C00085000  24.05
86     2014-11-14 call AAPL141114C00086000  22.76
87     2014-11-14 call AAPL141114C00087000  21.74
```

(continues on next page)

(continued from previous page)

```
In [20]: aapl.expiry_dates
Out[20]:
[datetime.date(2014, 11, 14),
 datetime.date(2014, 11, 22),
 datetime.date(2014, 11, 28),
 datetime.date(2014, 12, 5),
 datetime.date(2014, 12, 12),
 datetime.date(2014, 12, 20),
 datetime.date(2015, 1, 17),
 datetime.date(2015, 2, 20),
 datetime.date(2015, 4, 17),
 datetime.date(2015, 7, 17),
 datetime.date(2016, 1, 15),
 datetime.date(2017, 1, 20)]

In [21]: aapl.get_near_stock_price(expiry=aapl.expiry_dates[0:3]).iloc[0:5, 0:1]
Out[21]:
                                              Last
Strike Expiry     Type Symbol
109    2014-11-22 call AAPL141122C00109000  1.48
       2014-11-28 call AAPL141128C00109000  1.79
110    2014-11-14 call AAPL141114C00110000  0.55
       2014-11-22 call AAPL141122C00110000  1.02
       2014-11-28 call AAPL141128C00110000  1.32
```

- pandas now also registers the `datetime64` dtype in matplotlib's units registry to plot such values as datetimes. This is activated once pandas is imported. In previous versions, plotting an array of `datetime64` values will have resulted in plotted integer values. To keep the previous behaviour, you can do `del matplotlib.units.registry[np.datetime64]` (GH8614).

Enhancements

- `concat` permits a wider variety of iterables of pandas objects to be passed as the first parameter (GH8645):

```
In [17]: from collections import deque

In [18]: df1 = pd.DataFrame([1, 2, 3])

In [19]: df2 = pd.DataFrame([4, 5, 6])
```

previous behavior:

```
In [7]: pd.concat(deque((df1, df2)))
TypeError: first argument must be a list-like of pandas objects, you passed an␣
↪object of type "deque"
```

current behavior:

```
In [20]: pd.concat(deque((df1, df2)))
Out[20]:
   0
0  1
1  2
2  3
0  4
```

(continues on next page)

(continued from previous page)

```
1   5
2   6

[6 rows x 1 columns]
```

- Represent `MultiIndex` labels with a dtype that utilizes memory based on the level size. In prior versions, the memory usage was a constant 8 bytes per element in each level. In addition, in prior versions, the *reported* memory usage was incorrect as it didn't show the usage for the memory occupied by the underling data array. (GH8456)

```
In [21]: dfi = pd.DataFrame(1, index=pd.MultiIndex.from_product([['a'],
   ....:                          range(1000)]), columns=['A'])
   ....:
```

previous behavior:

```
# this was underreported in prior versions
In [1]: dfi.memory_usage(index=True)
Out[1]:
Index    8000 # took about 24008 bytes in < 0.15.1
A        8000
dtype: int64
```

current behavior:

```
In [22]: dfi.memory_usage(index=True)
Out[22]:
Index    52080
A         8000
Length: 2, dtype: int64
```

- Added Index properties *is_monotonic_increasing* and *is_monotonic_decreasing* (GH8680).
- Added option to select columns when importing Stata files (GH7935)
- Qualify memory usage in `DataFrame.info()` by adding + if it is a lower bound (GH8578)
- Raise errors in certain aggregation cases where an argument such as `numeric_only` is not handled (GH8592).
- Added support for 3-character ISO and non-standard country codes in `io.wb.download()` (GH8482)
- World Bank data requests now will warn/raise based on an `errors` argument, as well as a list of hard-coded country codes and the World Bank's JSON response. In prior versions, the error messages didn't look at the World Bank's JSON response. Problem-inducing input were simply dropped prior to the request. The issue was that many good countries were cropped in the hard-coded approach. All countries will work now, but some bad countries will raise exceptions because some edge cases break the entire response. (GH8482)
- Added option to `Series.str.split()` to return a `DataFrame` rather than a `Series` (GH8428)
- Added option to `df.info(null_counts=None|True|False)` to override the default display options and force showing of the null-counts (GH8701)

Bug fixes

- Bug in unpickling of a CustomBusinessDay object (GH8591)
- Bug in coercing Categorical to a records array, e.g. df.to_records() (GH8626)
- Bug in Categorical not created properly with Series.to_frame() (GH8626)
- Bug in coercing in astype of a Categorical of a passed pd.Categorical (this now raises TypeError correctly), (GH8626)
- Bug in cut/qcut when using Series and retbins=True (GH8589)
- Bug in writing Categorical columns to an SQL database with to_sql (GH8624).
- Bug in comparing Categorical of datetime raising when being compared to a scalar datetime (GH8687)
- Bug in selecting from a Categorical with .iloc (GH8623)
- Bug in groupby-transform with a Categorical (GH8623)
- Bug in duplicated/drop_duplicates with a Categorical (GH8623)
- Bug in Categorical reflected comparison operator raising if the first argument was a numpy array scalar (e.g. np.int64) (GH8658)
- Bug in Panel indexing with a list-like (GH8710)
- Compat issue is DataFrame.dtypes when options.mode.use_inf_as_null is True (GH8722)
- Bug in read_csv, dialect parameter would not take a string (GH8703)
- Bug in slicing a MultiIndex level with an empty-list (GH8737)
- Bug in numeric index operations of add/sub with Float/Index Index with numpy arrays (GH8608)
- Bug in setitem with empty indexer and unwanted coercion of dtypes (GH8669)
- Bug in ix/loc block splitting on setitem (manifests with integer-like dtypes, e.g. datetime64) (GH8607)
- Bug when doing label based indexing with integers not found in the index for non-unique but monotonic indexes (GH8680).
- Bug when indexing a Float64Index with np.nan on numpy 1.7 (GH8980).
- Fix shape attribute for MultiIndex (GH8609)
- Bug in GroupBy where a name conflict between the grouper and columns would break groupby operations (GH7115, GH8112)
- Fixed a bug where plotting a column y and specifying a label would mutate the index name of the original DataFrame (GH8494)
- Fix regression in plotting of a DatetimeIndex directly with matplotlib (GH8614).
- Bug in date_range where partially-specified dates would incorporate current date (GH6961)
- Bug in Setting by indexer to a scalar value with a mixed-dtype *Panel4d* was failing (GH8702)
- Bug where DataReader's would fail if one of the symbols passed was invalid. Now returns data for valid symbols and np.nan for invalid (GH8494)
- Bug in get_quote_yahoo that wouldn't allow non-float return values (GH5229).

Contributors

A total of 23 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Staple +
- Andrew Rosenfeld
- Anton I. Sipos
- Artemy Kolchinsky
- Bill Letson +
- Dave Hughes +
- David Stephens
- Guillaume Horel +
- Jeff Reback
- Joris Van den Bossche
- Kevin Sheppard
- Nick Stahl +
- Sanghee Kim +
- Stephan Hoyer
- Tom Augspurger
- TomAugspurger
- WANG Aiyong +
- behzad nouri
- immerrr
- jnmclarty
- jreback
- pallav-fdsi +
- unutbu

5.12.3 v0.15.0 (October 18, 2014)

This is a major release from 0.14.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

> **Warning:** pandas >= 0.15.0 will no longer support compatibility with NumPy versions < 1.7.0. If you want to use the latest versions of pandas, please upgrade to NumPy >= 1.7.0 (GH7711)

- Highlights include:
  - The `Categorical` type was integrated as a first-class pandas type, see *here*
  - New scalar type `Timedelta`, and a new index type `TimedeltaIndex`, see *here*

  - New datetimelike properties accessor `.dt` for Series, see *Datetimelike Properties*
  - New DataFrame default display for `df.info()` to include memory usage, see *Memory Usage*
  - `read_csv` will now by default ignore blank lines when parsing, see *here*
  - API change in using Indexes in set operations, see *here*
  - Enhancements in the handling of timezones, see *here*
  - A lot of improvements to the rolling and expanding moment functions, see *here*
  - Internal refactoring of the `Index` class to no longer sub-class `ndarray`, see *Internal Refactoring*
  - dropping support for `PyTables` less than version 3.0.0, and `numexpr` less than version 2.1 (GH7990)
  - Split indexing documentation into *Indexing and Selecting Data* and *MultiIndex / Advanced Indexing*
  - Split out string methods documentation into *Working with Text Data*
- Check the *API Changes* and *deprecations* before updating
- *Other Enhancements*
- *Performance Improvements*
- *Bug Fixes*

Warning: In 0.15.0 `Index` has internally been refactored to no longer sub-class `ndarray` but instead subclass `PandasObject`, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (See the *Internal Refactoring*)

Warning: The refactoring in `Categorical` changed the two argument constructor from "codes/labels and levels" to "values and levels (now called 'categories')". This can lead to subtle bugs. If you use `Categorical` directly, please audit your code before updating to this pandas version and change it to use the `from_codes()` constructor. See more on `Categorical` *here*

New features

Categoricals in Series/DataFrame

`Categorical` can now be included in *Series* and *DataFrames* and gained new methods to manipulate. Thanks to Jan Schulz for much of this API/implementation. (GH3943, GH5313, GH5314, GH7444, GH7839, GH7848, GH7864, GH7914, GH7768, GH8006, GH3678, GH8075, GH8076, GH8143, GH8453, GH8518).

For full docs, see the *categorical introduction* and the *API documentation*.

```
In [1]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
   ...:                    "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
   ...:

In [2]: df["grade"] = df["raw_grade"].astype("category")

In [3]: df["grade"]
Out[3]:
0    a
```

(continues on next page)

(continued from previous page)

```
1    b
2    b
3    a
4    a
5    e
Name: grade, Length: 6, dtype: category
Categories (3, object): [a, b, e]

# Rename the categories
In [4]: df["grade"].cat.categories = ["very good", "good", "very bad"]

# Reorder the categories and simultaneously add the missing categories
In [5]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad",
   ...:                                               "medium", "good", "very good"])
   ...:

In [6]: df["grade"]
Out[6]:
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, Length: 6, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]

In [7]: df.sort_values("grade")
Out[7]:
   id raw_grade      grade
5   6         e   very bad
1   2         b       good
2   3         b       good
0   1         a  very good
3   4         a  very good
4   5         a  very good

[6 rows x 3 columns]

In [8]: df.groupby("grade").size()
Out[8]:
grade
very bad     1
bad          0
medium       0
good         2
very good    3
Length: 5, dtype: int64
```

- `pandas.core.group_agg` and `pandas.core.factor_agg` were removed. As an alternative, construct a dataframe and use `df.groupby(<group>).agg(<func>)`.
- Supplying "codes/labels and levels" to the *`Categorical`* constructor is not supported anymore. Supplying two arguments to the constructor is now interpreted as "values and levels (now called 'categories')". Please change your code to use the *`from_codes()`* constructor.
- The `Categorical.labels` attribute was renamed to `Categorical.codes` and is read only. If you want to manipulate codes, please use one of the *API methods on Categoricals*.

- The `Categorical.levels` attribute is renamed to `Categorical.categories`.

TimedeltaIndex/Scalar

We introduce a new scalar type `Timedelta`, which is a subclass of `datetime.timedelta`, and behaves in a similar manner, but allows compatibility with `np.timedelta64` types as well as a host of custom representation, parsing, and attributes. This type is very similar to how `Timestamp` works for `datetimes`. It is a nice-API box for the type. See the *docs*. (GH3009, GH4533, GH8209, GH8187, GH8190, GH7869, GH7661, GH8345, GH8471)

> **Warning:** `Timedelta` scalars (and `TimedeltaIndex`) component fields are *not the same* as the component fields on a `datetime.timedelta` object. For example, `.seconds` on a `datetime.timedelta` object returns the total number of seconds combined between `hours`, `minutes` and `seconds`. In contrast, the pandas `Timedelta` breaks out hours, minutes, microseconds and nanoseconds separately.
>
> ```
> # Timedelta accessor
> In [9]: tds = pd.Timedelta('31 days 5 min 3 sec')
>
> In [10]: tds.minutes
> Out[10]: 5L
>
> In [11]: tds.seconds
> Out[11]: 3L
>
> # datetime.timedelta accessor
> # this is 5 minutes * 60 + 3 seconds
> In [12]: tds.to_pytimedelta().seconds
> Out[12]: 303
> ```
>
> **Note**: this is no longer true starting from v0.16.0, where full compatibility with `datetime.timedelta` is introduced. See the *0.16.0 whatsnew entry*

> **Warning:** Prior to 0.15.0 `pd.to_timedelta` would return a `Series` for list-like/Series input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for Series input, and `Timedelta` for scalar input.
>
> The arguments to `pd.to_timedelta` are now `(arg,unit='ns',box=True,coerce=False)`, previously were `(arg,box=True,unit='ns')` as these are more logical.

Construct a scalar

```
In [9]: pd.Timedelta('1 days 06:05:01.00003')
Out[9]: Timedelta('1 days 06:05:01.000030')

In [10]: pd.Timedelta('15.5us')
Out[10]: Timedelta('0 days 00:00:00.000015')

In [11]: pd.Timedelta('1 hour 15.5us')
Out[11]: Timedelta('0 days 01:00:00.000015')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [12]: pd.Timedelta('-1us')
Out[12]: Timedelta('-1 days +23:59:59.999999')
```

(continues on next page)

(continued from previous page)

```
# a NaT
In [13]: pd.Timedelta('nan')
Out[13]: NaT
```

Access fields for a `Timedelta`

```
In [14]: td = pd.Timedelta('1 hour 3m 15.5us')

In [15]: td.seconds
Out[15]: 3780

In [16]: td.microseconds
Out[16]: 16

In [17]: td.nanoseconds
Out[17]: 500
```

Construct a `TimedeltaIndex`

```
In [18]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05',
   ....:                    np.timedelta64(2, 'D'),
   ....:                    datetime.timedelta(days=2, seconds=2)])
   ....:
Out[18]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
                '2 days 00:00:02'],
               dtype='timedelta64[ns]', freq=None)
```

Constructing a `TimedeltaIndex` with a regular range

```
In [19]: pd.timedelta_range('1 days', periods=5, freq='D')
Out[19]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↪'timedelta64[ns]', freq='D')

In [20]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')
Out[20]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00',
                '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00',
                '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00',
                '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00',
                '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00',
                '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00',
                '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00',
                '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00',
                '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00',
                '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='30T')
```

You can now use a `TimedeltaIndex` as the index of a pandas object

```
In [21]: s = pd.Series(np.arange(5),
   ....:               index=pd.timedelta_range('1 days', periods=5, freq='s'))
   ....:

In [22]: s
Out[22]:
1 days 00:00:00    0
1 days 00:00:01    1
1 days 00:00:02    2
1 days 00:00:03    3
1 days 00:00:04    4
Freq: S, Length: 5, dtype: int64
```

You can select with partial string selections

```
In [23]: s['1 day 00:00:02']
Out[23]: 2

In [24]: s['1 day':'1 day 00:00:02']
Out[24]:
1 days 00:00:00    0
1 days 00:00:01    1
1 days 00:00:02    2
Freq: S, Length: 3, dtype: int64
```

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are `NaT` preserving:

```
In [25]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [26]: tdi.tolist()
Out[26]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [27]: dti = pd.date_range('20130101', periods=3)

In [28]: dti.tolist()
Out[28]:
[Timestamp('2013-01-01 00:00:00', freq='D'),
 Timestamp('2013-01-02 00:00:00', freq='D'),
 Timestamp('2013-01-03 00:00:00', freq='D')]

In [29]: (dti + tdi).tolist()
Out[29]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [30]: (dti - tdi).tolist()
Out[30]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

- iteration of a `Series` e.g. `list(Series(...))` of `timedelta64[ns]` would prior to v0.15.0 return `np.timedelta64` for each element. These will now be wrapped in `Timedelta`.

Memory usage

Implemented methods to find memory usage of a DataFrame. See the *FAQ* for more. (GH6852).

A new display option `display.memory_usage` (see *Options and settings*) sets the default behavior of the `memory_usage` argument in the `df.info()` method. By default `display.memory_usage` is `True`.

```
In [31]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
   ....:           'complex128', 'object', 'bool']
   ....:

In [32]: n = 5000

In [33]: data = {t: np.random.randint(100, size=n).astype(t) for t in dtypes}

In [34]: df = pd.DataFrame(data)

In [35]: df['categorical'] = df['object'].astype('category')

In [36]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   int64            5000 non-null   int64
 1   float64          5000 non-null   float64
 2   datetime64[ns]   5000 non-null   datetime64[ns]
 3   timedelta64[ns]  5000 non-null   timedelta64[ns]
 4   complex128       5000 non-null   complex128
 5   object           5000 non-null   object
 6   bool             5000 non-null   bool
 7   categorical      5000 non-null   category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↪object(1), timedelta64[ns](1)
memory usage: 289.1+ KB
```

Additionally `memory_usage()` is an available method for a dataframe object which returns the memory usage of each column.

```
In [37]: df.memory_usage(index=True)
Out[37]:
Index                128
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
complex128         80000
object             40000
bool                5000
categorical        10920
Length: 9, dtype: int64
```

.dt accessor

`Series` has gained an accessor to succinctly return datetime like properties for the *values* of the Series, if its a datetime/period like Series. (GH7207) This will return a Series, indexed like the existing Series. See the *docs*

```
# datetime
In [38]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [39]: s
Out[39]:
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
Length: 4, dtype: datetime64[ns]

In [40]: s.dt.hour
Out[40]:
0    9
1    9
2    9
3    9
Length: 4, dtype: int64

In [41]: s.dt.second
Out[41]:
0    12
1    12
2    12
3    12
Length: 4, dtype: int64

In [42]: s.dt.day
Out[42]:
0    1
1    2
2    3
3    4
Length: 4, dtype: int64

In [43]: s.dt.freq
Out[43]: 'D'
```

This enables nice expressions like this:

```
In [44]: s[s.dt.day == 2]
Out[44]:
1   2013-01-02 09:10:12
Length: 1, dtype: datetime64[ns]
```

You can easily produce tz aware transformations:

```
In [45]: stz = s.dt.tz_localize('US/Eastern')

In [46]: stz
Out[46]:
0   2013-01-01 09:10:12-05:00
```

(continues on next page)

(continued from previous page)

```
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
Length: 4, dtype: datetime64[ns, US/Eastern]

In [47]: stz.dt.tz
Out[47]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

You can also chain these types of operations:

```
In [48]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[48]:
0   2013-01-01 04:10:12-05:00
1   2013-01-02 04:10:12-05:00
2   2013-01-03 04:10:12-05:00
3   2013-01-04 04:10:12-05:00
Length: 4, dtype: datetime64[ns, US/Eastern]
```

The `.dt` accessor works for period and timedelta dtypes.

```
# period
In [49]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [50]: s
Out[50]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
Length: 4, dtype: period[D]

In [51]: s.dt.year
Out[51]:
0    2013
1    2013
2    2013
3    2013
Length: 4, dtype: int64

In [52]: s.dt.day
Out[52]:
0    1
1    2
2    3
3    4
Length: 4, dtype: int64
```

```
# timedelta
In [53]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [54]: s
Out[54]:
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
3   1 days 00:00:08
```

(continues on next page)

(continued from previous page)

```
Length: 4, dtype: timedelta64[ns]

In [55]: s.dt.days
Out[55]:
0    1
1    1
2    1
3    1
Length: 4, dtype: int64

In [56]: s.dt.seconds
Out[56]:
0    5
1    6
2    7
3    8
Length: 4, dtype: int64

In [57]: s.dt.components
Out[57]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0

[4 rows x 7 columns]
```

Timezone handling improvements

- `tz_localize(None)` for tz-aware `Timestamp` and `DatetimeIndex` now removes timezone holding local time, previously this resulted in `Exception` or `TypeError` (GH7812)

```
In [58]: ts = pd.Timestamp('2014-08-01 09:00', tz='US/Eastern')

In [59]: ts
Out[59]: Timestamp('2014-08-01 09:00:00-0400', tz='US/Eastern')

In [60]: ts.tz_localize(None)
Out[60]: Timestamp('2014-08-01 09:00:00')

In [61]: didx = pd.date_range(start='2014-08-01 09:00', freq='H',
   ....:                      periods=10, tz='US/Eastern')
   ....:

In [62]: didx
Out[62]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
               '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
               '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
               '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
               '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

In [63]: didx.tz_localize(None)
```

(continues on next page)

(continued from previous page)

```
Out[63]:
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
               '2014-08-01 11:00:00', '2014-08-01 12:00:00',
               '2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')
```

- `tz_localize` now accepts the `ambiguous` keyword which allows for passing an array of bools indicating whether the date belongs in DST or not, 'NaT' for setting transition times to NaT, 'infer' for inferring DST/non-DST, and 'raise' (default) for an `AmbiguousTimeError` to be raised. See *the docs* for more details (GH7943)
- `DataFrame.tz_localize` and `DataFrame.tz_convert` now accepts an optional `level` argument for localizing a specific level of a MultiIndex (GH7846)
- `Timestamp.tz_localize` and `Timestamp.tz_convert` now raise `TypeError` in error cases, rather than `Exception` (GH8025)
- a timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone (rather than being a naive `datetime64[ns]`) as `object` dtype (GH8411)
- `Timestamp.__repr__` displays `dateutil.tz.tzoffset` info (GH7907)

Rolling/expanding moments improvements

- `rolling_min()`, `rolling_max()`, `rolling_cov()`, and `rolling_corr()` now return objects with all `NaN` when `len(arg) < min_periods <= window` rather than raising. (This makes all rolling functions consistent in this behavior). (GH7766)

  Prior to 0.15.0

  ```
  In [64]: s = pd.Series([10, 11, 12, 13])
  ```

  ```
  In [15]: pd.rolling_min(s, window=10, min_periods=5)
  ValueError: min_periods (5) must be <= window (4)
  ```

  New behavior

  ```
  In [4]: pd.rolling_min(s, window=10, min_periods=5)
  Out[4]:
  0   NaN
  1   NaN
  2   NaN
  3   NaN
  dtype: float64
  ```

- `rolling_max()`, `rolling_min()`, `rolling_sum()`, `rolling_mean()`, `rolling_median()`, `rolling_std()`, `rolling_var()`, `rolling_skew()`, `rolling_kurt()`, `rolling_quantile()`, `rolling_cov()`, `rolling_corr()`, `rolling_corr_pairwise()`, `rolling_window()`, and `rolling_apply()` with `center=True` previously would return a result of the same structure as the input `arg` with `NaN` in the final `(window-1)/2` entries.

  Now the final `(window-1)/2` entries of the result are calculated as if the input `arg` were followed by `(window-1)/2` `NaN` values (or with shrinking windows, in the case of `rolling_apply()`). (GH7925, GH8269)

  Prior behavior (note final value is `NaN`):

```
In [7]: pd.rolling_sum(Series(range(4)), window=3, min_periods=0, center=True)
Out[7]:
0      1
1      3
2      6
3    NaN
dtype: float64
```

New behavior (note final value is `5 = sum([2, 3, NaN])`):

```
In [7]: pd.rolling_sum(pd.Series(range(4)), window=3,
   ....:                 min_periods=0, center=True)
Out[7]:
0     1
1     3
2     6
3     5
dtype: float64
```

- `rolling_window()` now normalizes the weights properly in rolling mean mode (*mean=True*) so that the calculated weighted means (e.g. 'triang', 'gaussian') are distributed about the same means as those calculated without weighting (i.e. 'boxcar'). See *the note on normalization* for further details. (GH7618)

```
In [65]: s = pd.Series([10.5, 8.8, 11.4, 9.7, 9.3])
```

Behavior prior to 0.15.0:

```
In [39]: pd.rolling_window(s, window=3, win_type='triang', center=True)
Out[39]:
0          NaN
1     6.583333
2     6.883333
3     6.683333
4          NaN
dtype: float64
```

New behavior

```
In [10]: pd.rolling_window(s, window=3, win_type='triang', center=True)
Out[10]:
0        NaN
1      9.875
2     10.325
3     10.025
4        NaN
dtype: float64
```

- Removed `center` argument from all `expanding_` functions (see *list*), as the results produced when `center=True` did not make much sense. (GH7925)
- Added optional `ddof` argument to `expanding_cov()` and `rolling_cov()`. The default value of `1` is backwards-compatible. (GH8279)
- Documented the `ddof` argument to `expanding_var()`, `expanding_std()`, `rolling_var()`, and `rolling_std()`. These functions' support of a `ddof` argument (with a default value of `1`) was previously undocumented. (GH8064)
- `ewma()`, `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now interpret `min_periods` in the same manner that the `rolling_*()` and `expanding_*()` functions do: a given result entry will be

NaN if the (expanding, in this case) window does not contain at least min_periods values. The previous behavior was to set to NaN the min_periods entries starting with the first non- NaN value. (GH7977)

Prior behavior (note values start at index 2, which is min_periods after index 0 (the index of the first non-empty value)):

```
In [66]: s  = pd.Series([1, None, None, None, 2, 3])
```

```
In [51]: ewma(s, com=3., min_periods=2)
Out[51]:
0         NaN
1         NaN
2    1.000000
3    1.000000
4    1.571429
5    2.189189
dtype: float64
```

New behavior (note values start at index 4, the location of the 2nd (since min_periods=2) non-empty value):

```
In [2]: pd.ewma(s, com=3., min_periods=2)
Out[2]:
0         NaN
1         NaN
2         NaN
3         NaN
4    1.759644
5    2.383784
dtype: float64
```

- ewmstd(), ewmvol(), ewmvar(), ewmcov(), and ewmcorr() now have an optional adjust argument, just like ewma() does, affecting how the weights are calculated. The default value of adjust is True, which is backwards-compatible. See *Exponentially weighted moment functions* for details. (GH7911)
- ewma(), ewmstd(), ewmvol(), ewmvar(), ewmcov(), and ewmcorr() now have an optional ignore_na argument. When ignore_na=False (the default), missing values are taken into account in the weights calculation. When ignore_na=True (which reproduces the pre-0.15.0 behavior), missing values are ignored in the weights calculation. (GH7543)

```
In [7]: pd.ewma(pd.Series([None, 1., 8.]), com=2.)
Out[7]:
0    NaN
1    1.0
2    5.2
dtype: float64

In [8]: pd.ewma(pd.Series([1., None, 8.]), com=2.,
  ....:         ignore_na=True)  # pre-0.15.0 behavior
Out[8]:
0    1.0
1    1.0
2    5.2
dtype: float64

In [9]: pd.ewma(pd.Series([1., None, 8.]), com=2.,
  ....:         ignore_na=False)  # new default
Out[9]:
0    1.000000
```

(continues on next page)

(continued from previous page)

```
1    1.000000
2    5.846154
dtype: float64
```

Warning: By default (`ignore_na=False`) the `ewm*()` functions' weights calculation in the presence of missing values is different than in pre-0.15.0 versions. To reproduce the pre-0.15.0 calculation of weights in the presence of missing values one must specify explicitly `ignore_na=True`.

- Bug in `expanding_cov()`, `expanding_corr()`, `rolling_cov()`, `rolling_cor()`, `ewmcov()`, and `ewmcorr()` returning results with columns sorted by name and producing an error for non-unique columns; now handles non-unique columns and returns columns in original order (except for the case of two DataFrames with `pairwise=False`, where behavior is unchanged) (GH7542)
- Bug in `rolling_count()` and `expanding_*()` functions unnecessarily producing error message for zero-length data (GH8056)
- Bug in `rolling_apply()` and `expanding_apply()` interpreting `min_periods=0` as `min_periods=1` (GH8080)
- Bug in `expanding_std()` and `expanding_var()` for a single value producing a confusing error message (GH7900)
- Bug in `rolling_std()` and `rolling_var()` for a single value producing `0` rather than `NaN` (GH7900)
- Bug in `ewmstd()`, `ewmvol()`, `ewmvar()`, and `ewmcov()` calculation of de-biasing factors when `bias=False` (the default). Previously an incorrect constant factor was used, based on `adjust=True`, `ignore_na=True`, and an infinite number of observations. Now a different factor is used for each entry, based on the actual weights (analogous to the usual `N/(N-1)` factor). In particular, for a single point a value of `NaN` is returned when `bias=False`, whereas previously a value of (approximately) `0` was returned.

  For example, consider the following pre-0.15.0 results for `ewmvar(..., bias=False)`, and the corresponding debiasing factors:

```
In [67]: s = pd.Series([1., 2., 0., 4.])
```

```
In [89]: ewmvar(s, com=2., bias=False)
Out[89]:
0   -2.775558e-16
1    3.000000e-01
2    9.556787e-01
3    3.585799e+00
dtype: float64

In [90]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
Out[90]:
0    1.25
1    1.25
2    1.25
3    1.25
dtype: float64
```

Note that entry `0` is approximately 0, and the debiasing factors are a constant 1.25. By comparison, the following 0.15.0 results have a `NaN` for entry `0`, and the debiasing factors are decreasing (towards 1.25):

```
In [14]: pd.ewmvar(s, com=2., bias=False)
Out[14]:
0         NaN
1    0.500000
2    1.210526
3    4.089069
dtype: float64

In [15]: pd.ewmvar(s, com=2., bias=False) / pd.ewmvar(s, com=2., bias=True)
Out[15]:
0         NaN
1    2.083333
2    1.583333
3    1.425439
dtype: float64
```

See *Exponentially weighted moment functions* for details. (GH7912)

Improvements in the sql io module

- Added support for a `chunksize` parameter to `to_sql` function. This allows DataFrame to be written in chunks and avoid packet-size overflow errors (GH8062).
- Added support for a `chunksize` parameter to `read_sql` function. Specifying this argument will return an iterator through chunks of the query result (GH2908).
- Added support for writing `datetime.date` and `datetime.time` object columns with `to_sql` (GH6932).
- Added support for specifying a `schema` to read from/write to with `read_sql_table` and `to_sql` (GH7441, GH7952). For example:

```
df.to_sql('table', engine, schema='other_schema')  # noqa F821
pd.read_sql_table('table', engine, schema='other_schema')  # noqa F821
```

- Added support for writing `NaN` values with `to_sql` (GH2754).
- Added support for writing datetime64 columns with `to_sql` for all database flavors (GH7103).

Backwards incompatible API changes

Breaking changes

API changes related to `Categorical` (see *here* for more details):

- The `Categorical` constructor with two arguments changed from "codes/labels and levels" to "values and levels (now called 'categories')". This can lead to subtle bugs. If you use *`Categorical`* directly, please audit your code by changing it to use the *`from_codes()`* constructor.

  An old function call like (prior to 0.15.0):

```
pd.Categorical([0,1,0,2,1], levels=['a', 'b', 'c'])
```

  will have to adapted to the following to keep the same behaviour:

```
In [2]: pd.Categorical.from_codes([0,1,0,2,1], categories=['a', 'b', 'c'])
Out[2]:
[a, b, a, c, b]
Categories (3, object): [a, b, c]
```

API changes related to the introduction of the `Timedelta` scalar (see *above* for more details):

- Prior to 0.15.0 *to_timedelta()* would return a `Series` for list-like/Series input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for Series input, and `Timedelta` for scalar input.

For API changes related to the rolling and expanding functions, see detailed overview *above*.

Other notable API changes:

- Consistency when indexing with `.loc` and a list-like indexer when no values are found.

```
In [68]: df = pd.DataFrame([['a'], ['b']], index=[1, 2])

In [69]: df
Out[69]:
   0
1  a
2  b

[2 rows x 1 columns]
```

In prior versions there was a difference in these two constructs:

  - `df.loc[[3]]` would return a frame reindexed by 3 (with all `np.nan` values)
  - `df.loc[[3],:]` would raise `KeyError`.

Both will now raise a `KeyError`. The rule is that *at least 1* indexer must be found when using a list-like and `.loc` (GH7999)

Furthermore in prior versions these were also different:

  - `df.loc[[1,3]]` would return a frame reindexed by [1,3]
  - `df.loc[[1,3],:]` would raise `KeyError`.

Both will now return a frame reindex by [1,3]. E.g.

```
In [3]: df.loc[[1, 3]]
Out[3]:
     0
1    a
3  NaN

In [4]: df.loc[[1, 3], :]
Out[4]:
     0
1    a
3  NaN
```

This can also be seen in multi-axis indexing with a `Panel`.

```
>>> p = pd.Panel(np.arange(2 * 3 * 4).reshape(2, 3, 4),
...              items=['ItemA', 'ItemB'],
...              major_axis=[1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
...                 minor_axis=['A', 'B', 'C', 'D'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemB
Major_axis axis: 1 to 3
Minor_axis axis: A to D
```

The following would raise `KeyError` prior to 0.15.0:

```
In [5]:
Out[5]:
   ItemA  ItemD
1      3    NaN
2      7    NaN
3     11    NaN
```

Furthermore, `.loc` will raise If no values are found in a MultiIndex with a list-like indexer:

```
In [70]: s = pd.Series(np.arange(3, dtype='int64'),
   ....:               index=pd.MultiIndex.from_product([['A'],
   ....:                                                 ['foo', 'bar', 'baz']],
   ....:                                                 names=['one', 'two'])
   ....:               ).sort_index()
   ....:

In [71]: s
Out[71]:
one  two
A    bar    1
     baz    2
     foo    0
Length: 3, dtype: int64

In [72]: try:
   ....:     s.loc[['D']]
   ....: except KeyError as e:
   ....:     print("KeyError: " + str(e))
   ....:
```

- Assigning values to `None` now considers the dtype when choosing an 'empty' value (GH7941).

  Previously, assigning to `None` in numeric containers changed the dtype to object (or errored, depending on the call). It now uses `NaN`:

```
In [73]: s = pd.Series([1, 2, 3])

In [74]: s.loc[0] = None

In [75]: s
Out[75]:
0    NaN
1    2.0
2    3.0
Length: 3, dtype: float64
```

  `NaT` is now used similarly for datetime containers.

For object containers, we now preserve `None` values (previously these were converted to `NaN` values).

```
In [76]: s = pd.Series(["a", "b", "c"])

In [77]: s.loc[0] = None

In [78]: s
Out[78]:
0    None
1       b
2       c
Length: 3, dtype: object
```

To insert a `NaN`, you must explicitly use `np.nan`. See the *docs*.

- In prior versions, updating a pandas object inplace would not reflect in other python references to this object. (GH8511, GH5104)

```
In [79]: s = pd.Series([1, 2, 3])

In [80]: s2 = s

In [81]: s += 1.5
```

Behavior prior to v0.15.0

```
# the original object
In [5]: s
Out[5]:
0    2.5
1    3.5
2    4.5
dtype: float64


# a reference to the original object
In [7]: s2
Out[7]:
0    1
1    2
2    3
dtype: int64
```

This is now the correct behavior

```
# the original object
In [82]: s
Out[82]:
0    2.5
1    3.5
2    4.5
Length: 3, dtype: float64

# a reference to the original object
In [83]: s2
Out[83]:
0    2.5
1    3.5
```

(continues on next page)

(continued from previous page)

```
2    4.5
Length: 3, dtype: float64
```

- Made both the C-based and Python engines for *read_csv* and *read_table* ignore empty lines in input as well as white space-filled lines, as long as `sep` is not white space. This is an API change that can be controlled by the keyword parameter `skip_blank_lines`. See *the docs* (GH4466)
- A timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone and inserted as `object` dtype rather than being converted to a naive `datetime64[ns]` (GH8411).
- Bug in passing a `DatetimeIndex` with a timezone that was not being retained in DataFrame construction from a dict (GH7822)

  In prior versions this would drop the timezone, now it retains the timezone, but gives a column of `object` dtype:

```
In [84]: i = pd.date_range('1/1/2011', periods=3, freq='10s', tz='US/Eastern')

In [85]: i
Out[85]:
DatetimeIndex(['2011-01-01 00:00:00-05:00', '2011-01-01 00:00:10-05:00',
               '2011-01-01 00:00:20-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='10S')

In [86]: df = pd.DataFrame({'a': i})

In [87]: df
Out[87]:
                          a
0 2011-01-01 00:00:00-05:00
1 2011-01-01 00:00:10-05:00
2 2011-01-01 00:00:20-05:00

[3 rows x 1 columns]

In [88]: df.dtypes
Out[88]:
a    datetime64[ns, US/Eastern]
Length: 1, dtype: object
```

  Previously this would have yielded a column of `datetime64` dtype, but without timezone info.

  The behaviour of assigning a column to an existing dataframe as *df['a'] = i* remains unchanged (this already returned an `object` column with a timezone).

- When passing multiple levels to `stack()`, it will now raise a `ValueError` when the levels aren't all level names or all level numbers (GH7660). See *Reshaping by stacking and unstacking*.
- Raise a `ValueError` in `df.to_hdf` with 'fixed' format, if `df` has non-unique columns as the resulting file will be broken (GH7761)
- `SettingWithCopy` raise/warnings (according to the option `mode.chained_assignment`) will now be issued when setting a value on a sliced mixed-dtype DataFrame using chained-assignment. (GH7845, GH7950)

```
In [1]: df = pd.DataFrame(np.arange(0, 9), columns=['count'])

In [2]: df['group'] = 'b'
```

(continues on next page)

(continued from previous page)

```
In [3]: df.iloc[0:5]['group'] = 'a'
/usr/local/bin/ipython:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
↪stable/indexing.html#indexing-view-versus-copy
```

- merge, DataFrame.merge, and ordered_merge now return the same type as the left argument (GH7737).
- Previously an enlargement with a mixed-dtype frame would act unlike .append which will preserve dtypes (related GH2578, GH8176):

```
In [89]: df = pd.DataFrame([[True, 1], [False, 2]],
   ....:                   columns=["female", "fitness"])
   ....:

In [90]: df
Out[90]:
   female  fitness
0    True        1
1   False        2

[2 rows x 2 columns]

In [91]: df.dtypes
Out[91]:
female      bool
fitness    int64
Length: 2, dtype: object

# dtypes are now preserved
In [92]: df.loc[2] = df.loc[1]

In [93]: df
Out[93]:
   female  fitness
0    True        1
1   False        2
2   False        2

[3 rows x 2 columns]

In [94]: df.dtypes
Out[94]:
female      bool
fitness    int64
Length: 2, dtype: object
```

- Series.to_csv() now returns a string when path=None, matching the behaviour of DataFrame.to_csv() (GH8215).
- read_hdf now raises IOError when a file that doesn't exist is passed in. Previously, a new, empty file was created, and a KeyError raised (GH7715).
- DataFrame.info() now ends its output with a newline character (GH8114)