SciPy sparse matrix from/to SparseDataFrame

Pandas now supports creating sparse dataframes directly from `scipy.sparse.spmatrix` instances. See the *documentation* for more information. (GH4343)

All sparse formats are supported, but matrices that are not in `COOrdinate` format will be converted, copying data as needed.

```
from scipy.sparse import csr_matrix
arr = np.random.random(size=(1000, 5))
arr[arr < .9] = 0
sp_arr = csr_matrix(arr)
sp_arr
sdf = pd.SparseDataFrame(sp_arr)
sdf
```

To convert a `SparseDataFrame` back to sparse SciPy matrix in COO format, you can use:

```
sdf.to_coo()
```

Excel output for styled DataFrames

Experimental support has been added to export `DataFrame.style` formats to Excel using the `openpyxl` engine. (GH15530)

For example, after running the following, `styled.xlsx` renders as below:

```
In [45]: np.random.seed(24)

In [46]: df = pd.DataFrame({'A': np.linspace(1, 10, 10)})

In [47]: df = pd.concat([df, pd.DataFrame(np.random.RandomState(24).randn(10, 4),
   ....:                                  columns=list('BCDE'))],
   ....:                axis=1)
   ....:

In [48]: df.iloc[0, 2] = np.nan

In [49]: df
Out[49]:
      A         B         C         D         E
0   1.0  1.329212       NaN -0.316280 -0.990810
1   2.0 -1.070816 -1.438713  0.564417  0.295722
2   3.0 -1.626404  0.219565  0.678805  1.889273
3   4.0  0.961538  0.104011 -0.481165  0.850229
4   5.0  1.453425  1.057737  0.165562  0.515018
5   6.0 -1.336936  0.562861  1.392855 -0.063328
6   7.0  0.121668  1.207603 -0.002040  1.627796
7   8.0  0.354493  1.037528 -0.385684  0.519818
8   9.0  1.686583 -1.325963  1.428984 -2.089354
9  10.0 -0.129820  0.631523 -0.586538  0.290720

[10 rows x 5 columns]

In [50]: styled = (df.style
   ....:           .applymap(lambda val: 'color: %s' % 'red' if val < 0 else 'black')
```

(continues on next page)

(continued from previous page)

```
   ....:              .highlight_max())
   ....:

In [51]: styled.to_excel('styled.xlsx', engine='openpyxl')
```

	A	B	C	D	E	F
1		A	B	C	D	E
2	0	1	1.329212		-0.31628	-0.99081
3	1	2	-1.070816	-1.438713	0.564417	0.295722
4	2	3	-1.626404	0.219565	0.678805	1.889273
5	3	4	0.961538	0.104011	-0.481165	0.850229
6	4	5	1.453425	1.057737	0.165562	0.515018
7	5	6	-1.336936	0.562861	1.392855	-0.063328
8	6	7	0.121668	1.207603	-0.00204	1.627796
9	7	8	0.354493	1.037528	-0.385684	0.519818
10	8	9	1.686583	-1.325963	1.428984	-2.089354
11	9	10	-0.12982	0.631523	-0.586538	0.29072

See the *Style documentation* for more detail.

IntervalIndex

pandas has gained an `IntervalIndex` with its own dtype, `interval` as well as the `Interval` scalar type. These allow first-class support for interval notation, specifically as a return type for the categories in `cut()` and `qcut()`. The `IntervalIndex` allows some unique indexing, see the *docs*. (GH7640, GH8625)

> **Warning:** These indexing behaviors of the IntervalIndex are provisional and may change in a future version of pandas. Feedback on usage is welcome.

Previous behavior:

The returned categories were strings, representing Intervals

```
In [1]: c = pd.cut(range(4), bins=2)

In [2]: c
Out[2]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3], (1.5, 3]]
Categories (2, object): [(-0.003, 1.5] < (1.5, 3]]

In [3]: c.categories
Out[3]: Index(['(-0.003, 1.5]', '(1.5, 3]'], dtype='object')
```

New behavior:

```
In [52]: c = pd.cut(range(4), bins=2)

In [53]: c
Out[53]:
```

(continues on next page)

(continued from previous page)

```
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]

In [54]: c.categories
Out[54]:
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]],
              closed='right',
              dtype='interval[float64]')
```

Furthermore, this allows one to bin *other* data with these same bins, with `NaN` representing a missing value similar to other dtypes.

```
In [55]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[55]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
```

An `IntervalIndex` can also be used in `Series` and `DataFrame` as the index.

```
In [56]: df = pd.DataFrame({'A': range(4),
   ....:                    'B': pd.cut([0, 3, 1, 1], bins=c.categories)
   ....:                    }).set_index('B')
   ....:

In [57]: df
Out[57]:
               A
B
(-0.003, 1.5]  0
(1.5, 3.0]     1
(-0.003, 1.5]  2
(-0.003, 1.5]  3

[4 rows x 1 columns]
```

Selecting via a specific interval:

```
In [58]: df.loc[pd.Interval(1.5, 3.0)]
Out[58]:
A    1
Name: (1.5, 3.0], Length: 1, dtype: int64
```

Selecting via a scalar value that is contained *in* the intervals.

```
In [59]: df.loc[0]
Out[59]:
               A
B
(-0.003, 1.5]  0
(-0.003, 1.5]  2
(-0.003, 1.5]  3

[3 rows x 1 columns]
```

Other enhancements

- DataFrame.rolling() now accepts the parameter closed='right'|'left'|'both'|'neither' to choose the rolling window-endpoint closedness. See the *documentation* (GH13965)
- Integration with the feather-format, including a new top-level pd.read_feather() and DataFrame.to_feather() method, see *here*.
- Series.str.replace() now accepts a callable, as replacement, which is passed to re.sub (GH15055)
- Series.str.replace() now accepts a compiled regular expression as a pattern (GH15446)
- Series.sort_index accepts parameters kind and na_position (GH13589, GH14444)
- DataFrame and DataFrame.groupby() have gained a nunique() method to count the distinct values over an axis (GH14336, GH15197).
- DataFrame has gained a melt() method, equivalent to pd.melt(), for unpivoting from a wide to long format (GH12640).
- pd.read_excel() now preserves sheet order when using sheetname=None (GH9930)
- Multiple offset aliases with decimal points are now supported (e.g. 0.5min is parsed as 30s) (GH8419)
- .isnull() and .notnull() have been added to Index object to make them more consistent with the Series API (GH15300)
- New UnsortedIndexError (subclass of KeyError) raised when indexing/slicing into an unsorted MultiIndex (GH11897). This allows differentiation between errors due to lack of sorting or an incorrect key. See *here*
- MultiIndex has gained a .to_frame() method to convert to a DataFrame (GH12397)
- pd.cut and pd.qcut now support datetime64 and timedelta64 dtypes (GH14714, GH14798)
- pd.qcut has gained the duplicates='raise'|'drop' option to control whether to raise on duplicated edges (GH7751)
- Series provides a to_excel method to output Excel files (GH8825)
- The usecols argument in pd.read_csv() now accepts a callable function as a value (GH14154)
- The skiprows argument in pd.read_csv() now accepts a callable function as a value (GH10882)
- The nrows and chunksize arguments in pd.read_csv() are supported if both are passed (GH6774, GH15755)
- DataFrame.plot now prints a title above each subplot if suplots=True and title is a list of strings (GH14753)
- DataFrame.plot can pass the matplotlib 2.0 default color cycle as a single string as color parameter, see here. (GH15516)
- Series.interpolate() now supports timedelta as an index type with method='time' (GH6424)
- Addition of a level keyword to DataFrame/Series.rename to rename labels in the specified level of a MultiIndex (GH4160).
- DataFrame.reset_index() will now interpret a tuple index.name as a key spanning across levels of columns, if this is a MultiIndex (GH16164)
- Timedelta.isoformat method added for formatting Timedeltas as an ISO 8601 duration. See the *Timedelta docs* (GH15136)
- .select_dtypes() now allows the string datetimetz to generically select datetimes with tz (GH14910)

- The .to_latex() method will now accept multicolumn and multirow arguments to use the accompanying LaTeX enhancements
- pd.merge_asof() gained the option direction='backward'|'forward'|'nearest' (GH14887)
- Series/DataFrame.asfreq() have gained a fill_value parameter, to fill missing values (GH3715).
- Series/DataFrame.resample.asfreq have gained a fill_value parameter, to fill missing values during resampling (GH3715).
- *pandas.util.hash_pandas_object()* has gained the ability to hash a MultiIndex (GH15224)
- Series/DataFrame.squeeze() have gained the axis parameter. (GH15339)
- DataFrame.to_excel() has a new freeze_panes parameter to turn on Freeze Panes when exporting to Excel (GH15160)
- pd.read_html() will parse multiple header rows, creating a MultiIndex header. (GH13434).
- HTML table output skips colspan or rowspan attribute if equal to 1. (GH15403)
- *pandas.io.formats.style.Styler* template now has blocks for easier extension, see the *example notebook* (GH15649)
- *Styler.render()* now accepts **kwargs to allow user-defined variables in the template (GH15649)
- Compatibility with Jupyter notebook 5.0; MultiIndex column labels are left-aligned and MultiIndex row-labels are top-aligned (GH15379)
- TimedeltaIndex now has a custom date-tick formatter specifically designed for nanosecond level precision (GH8711)
- pd.api.types.union_categoricals gained the ignore_ordered argument to allow ignoring the ordered attribute of unioned categoricals (GH13410). See the *categorical union docs* for more information.
- DataFrame.to_latex() and DataFrame.to_string() now allow optional header aliases. (GH15536)
- Re-enable the parse_dates keyword of pd.read_excel() to parse string columns as dates (GH14326)
- Added .empty property to subclasses of Index. (GH15270)
- Enabled floor division for Timedelta and TimedeltaIndex (GH15828)
- pandas.io.json.json_normalize() gained the option errors='ignore'|'raise'; the default is errors='raise' which is backward compatible. (GH14583)
- pandas.io.json.json_normalize() with an empty list will return an empty DataFrame (GH15534)
- pandas.io.json.json_normalize() has gained a sep option that accepts str to separate joined fields; the default is ".", which is backward compatible. (GH14883)
- *MultiIndex.remove_unused_levels()* has been added to facilitate *removing unused levels*. (GH15694)
- pd.read_csv() will now raise a ParserError error whenever any parsing error occurs (GH15913, GH15925)
- pd.read_csv() now supports the error_bad_lines and warn_bad_lines arguments for the Python parser (GH15925)
- The display.show_dimensions option can now also be used to specify whether the length of a Series should be shown in its repr (GH7117).

- `parallel_coordinates()` has gained a `sort_labels` keyword argument that sorts class labels and the colors assigned to them (GH15908)
- Options added to allow one to turn on/off using `bottleneck` and `numexpr`, see *here* (GH16157)
- `DataFrame.style.bar()` now accepts two more options to further customize the bar chart. Bar alignment is set with `align='left'|'mid'|'zero'`, the default is "left", which is backward compatible; You can now pass a list of `color=[color_negative, color_positive]`. (GH14757)

Backwards incompatible API changes

Possible incompatibility for HDF5 formats created with pandas < 0.13.0

`pd.TimeSeries` was deprecated officially in 0.17.0, though has already been an alias since 0.13.0. It has been dropped in favor of `pd.Series`. (GH15098).

This *may* cause HDF5 files that were created in prior versions to become unreadable if `pd.TimeSeries` was used. This is most likely to be for pandas < 0.13.0. If you find yourself in this situation. You can use a recent prior version of pandas to read in your HDF5 files, then write them out again after applying the procedure below.

```
In [2]: s = pd.TimeSeries([1, 2, 3], index=pd.date_range('20130101', periods=3))

In [3]: s
Out[3]:
2013-01-01    1
2013-01-02    2
2013-01-03    3
Freq: D, dtype: int64

In [4]: type(s)
Out[4]: pandas.core.series.TimeSeries

In [5]: s = pd.Series(s)

In [6]: s
Out[6]:
2013-01-01    1
2013-01-02    2
2013-01-03    3
Freq: D, dtype: int64

In [7]: type(s)
Out[7]: pandas.core.series.Series
```

Map on Index types now return other Index types

`map` on an `Index` now returns an `Index`, not a numpy array (GH12766)

```
In [60]: idx = pd.Index([1, 2])

In [61]: idx
Out[61]: Int64Index([1, 2], dtype='int64')

In [62]: mi = pd.MultiIndex.from_tuples([(1, 2), (2, 4)])
```

(continues on next page)

(continued from previous page)

```
In [63]: mi
Out[63]:
MultiIndex([(1, 2),
            (2, 4)],
           )
```

Previous behavior:

```
In [5]: idx.map(lambda x: x * 2)
Out[5]: array([2, 4])

In [6]: idx.map(lambda x: (x, x * 2))
Out[6]: array([(1, 2), (2, 4)], dtype=object)

In [7]: mi.map(lambda x: x)
Out[7]: array([(1, 2), (2, 4)], dtype=object)

In [8]: mi.map(lambda x: x[0])
Out[8]: array([1, 2])
```

New behavior:

```
In [64]: idx.map(lambda x: x * 2)
Out[64]: Int64Index([2, 4], dtype='int64')

In [65]: idx.map(lambda x: (x, x * 2))
Out[65]:
MultiIndex([(1, 2),
            (2, 4)],
           )

In [66]: mi.map(lambda x: x)
Out[66]:
MultiIndex([(1, 2),
            (2, 4)],
           )

In [67]: mi.map(lambda x: x[0])
Out[67]: Int64Index([1, 2], dtype='int64')
```

`map` on a `Series` with `datetime64` values may return `int64` dtypes rather than `int32`

```
In [68]: s = pd.Series(pd.date_range('2011-01-02T00:00', '2011-01-02T02:00', freq='H')
   ....:                .tz_localize('Asia/Tokyo'))
   ....:

In [69]: s
Out[69]:
0   2011-01-02 00:00:00+09:00
1   2011-01-02 01:00:00+09:00
2   2011-01-02 02:00:00+09:00
Length: 3, dtype: datetime64[ns, Asia/Tokyo]
```

Previous behavior:

```
In [9]: s.map(lambda x: x.hour)
Out[9]:
```

(continues on next page)

(continued from previous page)

```
0    0
1    1
2    2
dtype: int32
```

New behavior:

```
In [70]: s.map(lambda x: x.hour)
Out[70]:
0    0
1    1
2    2
Length: 3, dtype: int64
```

Accessing datetime fields of Index now return Index

The datetime-related attributes (see *here* for an overview) of `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex` previously returned numpy arrays. They will now return a new `Index` object, except in the case of a boolean field, where the result will still be a boolean ndarray. (GH15022)

Previous behaviour:

```
In [1]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [2]: idx.hour
Out[2]: array([ 0, 10, 20,  6, 16], dtype=int32)
```

New behavior:

```
In [71]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [72]: idx.hour
Out[72]: Int64Index([0, 10, 20, 6, 16], dtype='int64')
```

This has the advantage that specific `Index` methods are still available on the result. On the other hand, this might have backward incompatibilities: e.g. compared to numpy arrays, `Index` objects are not mutable. To get the original ndarray, you can always convert explicitly using `np.asarray(idx.hour)`.

pd.unique will now be consistent with extension types

In prior versions, using *`Series.unique()`* and *`pandas.unique()`* on `Categorical` and tz-aware data-types would yield different return types. These are now made consistent. (GH15903)

- Datetime tz-aware

  Previous behaviour:

```
# Series
In [5]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
   ...:            pd.Timestamp('20160101', tz='US/Eastern')]).unique()
Out[5]: array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],␣
↪dtype=object)

In [6]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
```

(continues on next page)

(continued from previous page)

```
   ...:                        pd.Timestamp('20160101', tz='US/Eastern')]))
Out[6]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')

# Index
In [7]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
   ...:           pd.Timestamp('20160101', tz='US/Eastern')]).unique()
Out[7]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/
↪Eastern]', freq=None)

In [8]: pd.unique([pd.Timestamp('20160101', tz='US/Eastern'),
   ...:            pd.Timestamp('20160101', tz='US/Eastern')])
Out[8]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')
```

New behavior:

```
# Series, returns an array of Timestamp tz-aware
In [73]: pd.Series([pd.Timestamp(r'20160101', tz=r'US/Eastern'),
   ....:            pd.Timestamp(r'20160101', tz=r'US/Eastern')]).unique()
   ....:
Out[73]:
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

In [74]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:           pd.Timestamp('20160101', tz='US/Eastern')]))
   ....:
Out[74]:
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

# Index, returns a DatetimeIndex
In [75]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:           pd.Timestamp('20160101', tz='US/Eastern')]).unique()
   ....:
Out[75]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/
↪Eastern]', freq=None)

In [76]: pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:                     pd.Timestamp('20160101', tz='US/Eastern')]))
   ....:
Out[76]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/
↪Eastern]', freq=None)
```

- Categoricals

  Previous behaviour:

```
In [1]: pd.Series(list('baabc'), dtype='category').unique()
Out[1]:
[b, a, c]
Categories (3, object): [b, a, c]

In [2]: pd.unique(pd.Series(list('baabc'), dtype='category'))
Out[2]: array(['b', 'a', 'c'], dtype=object)
```

New behavior:

```
# returns a Categorical
In [77]: pd.Series(list('baabc'), dtype='category').unique()
Out[77]:
[b, a, c]
Categories (3, object): [b, a, c]

In [78]: pd.unique(pd.Series(list('baabc'), dtype='category'))
Out[78]:
[b, a, c]
Categories (3, object): [b, a, c]
```

S3 file handling

pandas now uses s3fs for handling S3 connections. This shouldn't break any code. However, since `s3fs` is not a required dependency, you will need to install it separately, like `boto` in prior versions of pandas. (GH11915).

Partial string indexing changes

DatetimeIndex Partial String Indexing now works as an exact match, provided that string resolution coincides with index resolution, including a case when both are seconds (GH14826). See *Slice vs. Exact Match* for details.

```
In [79]: df = pd.DataFrame({'a': [1, 2, 3]}, pd.DatetimeIndex(['2011-12-31 23:59:59',
   ....:                                                       '2012-01-01 00:00:00',
   ....:                                                       '2012-01-01 00:00:01
↪']))
   ....:
```

Previous behavior:

```
In [4]: df['2011-12-31 23:59:59']
Out[4]:
                       a
2011-12-31 23:59:59  1

In [5]: df['a']['2011-12-31 23:59:59']
Out[5]:
2011-12-31 23:59:59    1
Name: a, dtype: int64
```

New behavior:

```
In [4]: df['2011-12-31 23:59:59']
KeyError: '2011-12-31 23:59:59'

In [5]: df['a']['2011-12-31 23:59:59']
Out[5]: 1
```

Concat of different float dtypes will not automatically upcast

Previously, `concat` of multiple objects with different `float` dtypes would automatically upcast results to a dtype of `float64`. Now the smallest acceptable dtype will be used (GH13247)

```
In [80]: df1 = pd.DataFrame(np.array([1.0], dtype=np.float32, ndmin=2))

In [81]: df1.dtypes
Out[81]:
0    float32
Length: 1, dtype: object

In [82]: df2 = pd.DataFrame(np.array([np.nan], dtype=np.float32, ndmin=2))

In [83]: df2.dtypes
Out[83]:
0    float32
Length: 1, dtype: object
```

Previous behavior:

```
In [7]: pd.concat([df1, df2]).dtypes
Out[7]:
0    float64
dtype: object
```

New behavior:

```
In [84]: pd.concat([df1, df2]).dtypes
Out[84]:
0    float32
Length: 1, dtype: object
```

Pandas Google BigQuery support has moved

pandas has split off Google BigQuery support into a separate package `pandas-gbq`. You can `conda install pandas-gbq -c conda-forge` or `pip install pandas-gbq` to get it. The functionality of *`read_gbq()`* and *`DataFrame.to_gbq()`* remain the same with the currently released version of `pandas-gbq=0.1.4`. Documentation is now hosted here (GH15347)

Memory usage for Index is more accurate

In previous versions, showing `.memory_usage()` on a pandas structure that has an index, would only include actual index values and not include structures that facilitated fast indexing. This will generally be different for `Index` and `MultiIndex` and less-so for other index types. (GH15237)

Previous behavior:

```
In [8]: index = pd.Index(['foo', 'bar', 'baz'])

In [9]: index.memory_usage(deep=True)
Out[9]: 180

In [10]: index.get_loc('foo')
```

(continues on next page)

(continued from previous page)

```
Out[10]: 0

In [11]: index.memory_usage(deep=True)
Out[11]: 180
```

New behavior:

```
In [8]: index = pd.Index(['foo', 'bar', 'baz'])

In [9]: index.memory_usage(deep=True)
Out[9]: 180

In [10]: index.get_loc('foo')
Out[10]: 0

In [11]: index.memory_usage(deep=True)
Out[11]: 260
```

DataFrame.sort_index changes

In certain cases, calling `.sort_index()` on a MultiIndexed DataFrame would return the *same* DataFrame without seeming to sort. This would happen with a `lexsorted`, but non-monotonic levels. (GH15622, GH15687, GH14015, GH13431, GH15797)

This is *unchanged* from prior versions, but shown for illustration purposes:

```
In [85]: df = pd.DataFrame(np.arange(6), columns=['value'],
   ....:                    index=pd.MultiIndex.from_product([list('BA'), range(3)]))
   ....:

In [86]: df
Out[86]:
     value
B 0      0
  1      1
  2      2
A 0      3
  1      4
  2      5

[6 rows x 1 columns]
```

```
In [87]: df.index.is_lexsorted()
Out[87]: False

In [88]: df.index.is_monotonic
Out[88]: False
```

Sorting works as expected

```
In [89]: df.sort_index()
Out[89]:
     value
A 0      3
```

(continues on next page)

(continued from previous page)

```
  1       4
  2       5
B 0       0
  1       1
  2       2

[6 rows x 1 columns]
```

```
In [90]: df.sort_index().index.is_lexsorted()
Out[90]: True

In [91]: df.sort_index().index.is_monotonic
Out[91]: True
```

However, this example, which has a non-monotonic 2nd level, doesn't behave as desired.

```
In [92]: df = pd.DataFrame({'value': [1, 2, 3, 4]},
   ....:                   index=pd.MultiIndex([['a', 'b'], ['bb', 'aa']],
   ....:                                       [[0, 0, 1, 1], [0, 1, 0, 1]]))
   ....:

In [93]: df
Out[93]:
      value
a bb      1
  aa      2
b bb      3
  aa      4

[4 rows x 1 columns]
```

Previous behavior:

```
In [11]: df.sort_index()
Out[11]:
      value
a bb      1
  aa      2
b bb      3
  aa      4

In [14]: df.sort_index().index.is_lexsorted()
Out[14]: True

In [15]: df.sort_index().index.is_monotonic
Out[15]: False
```

New behavior:

```
In [94]: df.sort_index()
Out[94]:
      value
a aa      2
  bb      1
b aa      4
  bb      3
```

(continues on next page)

(continued from previous page)

```
[4 rows x 1 columns]

In [95]: df.sort_index().index.is_lexsorted()
Out[95]: True

In [96]: df.sort_index().index.is_monotonic
Out[96]: True
```

Groupby describe formatting

The output formatting of `groupby.describe()` now labels the `describe()` metrics in the columns instead of the index. This format is consistent with `groupby.agg()` when applying multiple functions at once. (GH4792)

Previous behavior:

```
In [1]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})

In [2]: df.groupby('A').describe()
Out[2]:
                B
A
1 count  2.000000
  mean   1.500000
  std    0.707107
  min    1.000000
  25%    1.250000
  50%    1.500000
  75%    1.750000
  max    2.000000
2 count  2.000000
  mean   3.500000
  std    0.707107
  min    3.000000
  25%    3.250000
  50%    3.500000
  75%    3.750000
  max    4.000000

In [3]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
Out[3]:
     B
  mean       std amin amax
A
1  1.5  0.707107    1    2
2  3.5  0.707107    3    4
```

New behavior:

```
In [97]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})

In [98]: df.groupby('A').describe()
Out[98]:
      B
  count mean       std  min   25%  50%   75%  max
```

(continues on next page)

(continued from previous page)

```
A
1   2.0  1.5  0.707107  1.0  1.25  1.5  1.75  2.0
2   2.0  3.5  0.707107  3.0  3.25  3.5  3.75  4.0

[2 rows x 8 columns]

In [99]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
Out[99]:
     B
  mean       std amin amax
A
1  1.5  0.707107    1    2
2  3.5  0.707107    3    4

[2 rows x 4 columns]
```

Window binary corr/cov operations return a MultiIndex DataFrame

A binary window operation, like `.corr()` or `.cov()`, when operating on a `.rolling(..)`, `.expanding(..)`, or `.ewm(..)` object, will now return a 2-level `MultiIndexed DataFrame` rather than a `Panel`, as `Panel` is now deprecated, see *here*. These are equivalent in function, but a MultiIndexed `DataFrame` enjoys more support in pandas. See the section on *Windowed Binary Operations* for more information. (GH15677)

```
In [100]: np.random.seed(1234)

In [101]: df = pd.DataFrame(np.random.rand(100, 2),
   .....:                    columns=pd.Index(['A', 'B'], name='bar'),
   .....:                    index=pd.date_range('20160101',
   .....:                                        periods=100, freq='D', name='foo'))
   .....:

In [102]: df.tail()
Out[102]:
bar                A         B
foo
2016-04-05  0.640880  0.126205
2016-04-06  0.171465  0.737086
2016-04-07  0.127029  0.369650
2016-04-08  0.604334  0.103104
2016-04-09  0.802374  0.945553

[5 rows x 2 columns]
```

Previous behavior:

```
In [2]: df.rolling(12).corr()
Out[2]:
<class 'pandas.core.panel.Panel'>
Dimensions: 100 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: 2016-01-01 00:00:00 to 2016-04-09 00:00:00
Major_axis axis: A to B
Minor_axis axis: A to B
```

New behavior:

```
In [103]: res = df.rolling(12).corr()

In [104]: res.tail()
Out[104]:
bar                      A         B
foo        bar
2016-04-07 B    -0.132090  1.000000
2016-04-08 A     1.000000 -0.145775
           B    -0.145775  1.000000
2016-04-09 A     1.000000  0.119645
           B     0.119645  1.000000

[5 rows x 2 columns]
```

Retrieving a correlation matrix for a cross-section

```
In [105]: df.rolling(12).corr().loc['2016-04-07']
Out[105]:
bar                     A        B
foo        bar
2016-04-07 A     1.00000 -0.13209
           B    -0.13209  1.00000

[2 rows x 2 columns]
```

HDFStore where string comparison

In previous versions most types could be compared to string column in a `HDFStore` usually resulting in an invalid comparison, returning an empty result frame. These comparisons will now raise a `TypeError` (GH15492)

```
In [106]: df = pd.DataFrame({'unparsed_date': ['2014-01-01', '2014-01-01']})

In [107]: df.to_hdf('store.h5', 'key', format='table', data_columns=True)

In [108]: df.dtypes
Out[108]:
unparsed_date    object
Length: 1, dtype: object
```

Previous behavior:

```
In [4]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
File "<string>", line 1
  (unparsed_date > 1970-01-01 00:00:01.388552400)
                        ^
SyntaxError: invalid token
```

New behavior:

```
In [18]: ts = pd.Timestamp('2014-01-01')

In [19]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
TypeError: Cannot compare 2014-01-01 00:00:00 of
type <class 'pandas.tslib.Timestamp'> to string column
```

Index.intersection and inner join now preserve the order of the left Index

`Index.intersection()` now preserves the order of the calling `Index` (left) instead of the other `Index` (right) (GH15582). This affects inner joins, `DataFrame.join()` and `merge()`, and the `.align` method.

- `Index.intersection`

```
In [109]: left = pd.Index([2, 1, 0])

In [110]: left
Out[110]: Int64Index([2, 1, 0], dtype='int64')

In [111]: right = pd.Index([1, 2, 3])

In [112]: right
Out[112]: Int64Index([1, 2, 3], dtype='int64')
```

Previous behavior:

```
In [4]: left.intersection(right)
Out[4]: Int64Index([1, 2], dtype='int64')
```

New behavior:

```
In [113]: left.intersection(right)
Out[113]: Int64Index([2, 1], dtype='int64')
```

- `DataFrame.join` and `pd.merge`

```
In [114]: left = pd.DataFrame({'a': [20, 10, 0]}, index=[2, 1, 0])

In [115]: left
Out[115]:
    a
2  20
1  10
0   0

[3 rows x 1 columns]

In [116]: right = pd.DataFrame({'b': [100, 200, 300]}, index=[1, 2, 3])

In [117]: right
Out[117]:
     b
1  100
2  200
3  300

[3 rows x 1 columns]
```

Previous behavior:

```
In [4]: left.join(right, how='inner')
Out[4]:
   a    b
1  10  100
2  20  200
```

New behavior:

```
In [118]: left.join(right, how='inner')
Out[118]:
    a    b
2  20  200
1  10  100

[2 rows x 2 columns]
```

Pivot table always returns a DataFrame

The documentation for `pivot_table()` states that a `DataFrame` is *always* returned. Here a bug is fixed that allowed this to return a `Series` under certain circumstance. (GH4386)

```
In [119]: df = pd.DataFrame({'col1': [3, 4, 5],
   .....:                    'col2': ['C', 'D', 'E'],
   .....:                    'col3': [1, 3, 9]})
   .....:

In [120]: df
Out[120]:
   col1 col2  col3
0     3    C     1
1     4    D     3
2     5    E     9

[3 rows x 3 columns]
```

Previous behavior:

```
In [2]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[2]:
col3  col2
1     C       3
3     D       4
9     E       5
Name: col1, dtype: int64
```

New behavior:

```
In [121]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[121]:
           col1
col3 col2
1    C        3
3    D        4
9    E        5

[3 rows x 1 columns]
```

Other API changes

- `numexpr` version is now required to be >= 2.4.6 and it will not be used at all if this requisite is not fulfilled (GH15213).
- `CParserError` has been renamed to `ParserError` in `pd.read_csv()` and will be removed in the future (GH12665)
- `SparseArray.cumsum()` and `SparseSeries.cumsum()` will now always return `SparseArray` and `SparseSeries` respectively (GH12855)
- `DataFrame.applymap()` with an empty `DataFrame` will return a copy of the empty `DataFrame` instead of a `Series` (GH8222)
- `Series.map()` now respects default values of dictionary subclasses with a `__missing__` method, such as `collections.Counter` (GH15999)
- `.loc` has compat with `.ix` for accepting iterators, and NamedTuples (GH15120)
- `interpolate()` and `fillna()` will raise a `ValueError` if the `limit` keyword argument is not greater than 0. (GH9217)
- `pd.read_csv()` will now issue a `ParserWarning` whenever there are conflicting values provided by the `dialect` parameter and the user (GH14898)
- `pd.read_csv()` will now raise a `ValueError` for the C engine if the quote character is larger than than one byte (GH11592)
- `inplace` arguments now require a boolean value, else a `ValueError` is thrown (GH14189)
- `pandas.api.types.is_datetime64_ns_dtype` will now report `True` on a tz-aware dtype, similar to `pandas.api.types.is_datetime64_any_dtype`
- `DataFrame.asof()` will return a null filled `Series` instead the scalar `NaN` if a match is not found (GH15118)
- Specific support for `copy.copy()` and `copy.deepcopy()` functions on NDFrame objects (GH15444)
- `Series.sort_values()` accepts a one element list of bool for consistency with the behavior of `DataFrame.sort_values()` (GH15604)
- `.merge()` and `.join()` on `category` dtype columns will now preserve the category dtype when possible (GH10409)
- `SparseDataFrame.default_fill_value` will be 0, previously was `nan` in the return from `pd.get_dummies(..., sparse=True)` (GH15594)
- The default behaviour of `Series.str.match` has changed from extracting groups to matching the pattern. The extracting behaviour was deprecated since pandas version 0.13.0 and can be done with the `Series.str.extract` method (GH5224). As a consequence, the `as_indexer` keyword is ignored (no longer needed to specify the new behaviour) and is deprecated.
- `NaT` will now correctly report `False` for datetimelike boolean operations such as `is_month_start` (GH15781)
- `NaT` will now correctly return `np.nan` for `Timedelta` and `Period` accessors such as `days` and `quarter` (GH15782)
- `NaT` will now returns `NaT` for `tz_localize` and `tz_convert` methods (GH15830)
- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `PandasError`, if called with scalar inputs and not axes (GH15541)

- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `pandas.core.common.PandasError`, if called with scalar inputs and not axes; The exception `PandasError` is removed as well. (GH15541)
- The exception `pandas.core.common.AmbiguousIndexError` is removed as it is not referenced (GH15541)

Reorganization of the library: privacy changes

Modules privacy has changed

Some formerly public python/c/c++/cython extension modules have been moved and/or renamed. These are all removed from the public API. Furthermore, the `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are now considered to be PRIVATE. If indicated, a deprecation warning will be issued if you reference theses modules. (GH12588)

Previous Location	New Location	Deprecated
pandas.lib	pandas._libs.lib	X
pandas.tslib	pandas._libs.tslib	X
pandas.computation	pandas.core.computation	X
pandas.msgpack	pandas.io.msgpack	
pandas.index	pandas._libs.index	
pandas.algos	pandas._libs.algos	
pandas.hashtable	pandas._libs.hashtable	
pandas.indexes	pandas.core.indexes	
pandas.json	pandas._libs.json / pandas.io.json	X
pandas.parser	pandas._libs.parsers	X
pandas.formats	pandas.io.formats	
pandas.sparse	pandas.core.sparse	
pandas.tools	pandas.core.reshape	X
pandas.types	pandas.core.dtypes	X
pandas.io.sas.saslib	pandas.io.sas._sas	
pandas._join	pandas._libs.join	
pandas._hash	pandas._libs.hashing	
pandas._period	pandas._libs.period	
pandas._sparse	pandas._libs.sparse	
pandas._testing	pandas._libs.testing	
pandas._window	pandas._libs.window	

Some new subpackages are created with public functionality that is not directly exposed in the top-level namespace: `pandas.errors`, `pandas.plotting` and `pandas.testing` (more details below). Together with `pandas.api.types` and certain functions in the `pandas.io` and `pandas.tseries` submodules, these are now the public subpackages.

Further changes:

- The function *`union_categoricals()`* is now importable from `pandas.api.types`, formerly from `pandas.types.concat` (GH15998)
- The type import `pandas.tslib.NaTType` is deprecated and can be replaced by using `type(pandas.NaT)` (GH16146)
- The public functions in `pandas.tools.hashing` deprecated from that locations, but are now importable from `pandas.util` (GH16223)

- The modules in `pandas.util`: `decorators`, `print_versions`, `doctools`, `validators`, `depr_module` are now private. Only the functions exposed in `pandas.util` itself are public (GH16223)

pandas.errors

We are adding a standard public module for all pandas exceptions & warnings `pandas.errors`. (GH14800). Previously these exceptions & warnings could be imported from `pandas.core.common` or `pandas.io.common`. These exceptions and warnings will be removed from the `*.common` locations in a future release. (GH15541)

The following are now part of this API:

```
['DtypeWarning',
 'EmptyDataError',
 'OutOfBoundsDatetime',
 'ParserError',
 'ParserWarning',
 'PerformanceWarning',
 'UnsortedIndexError',
 'UnsupportedFunctionCall']
```

pandas.testing

We are adding a standard module that exposes the public testing functions in `pandas.testing` (GH9895). Those functions can be used when writing tests for functionality using pandas objects.

The following testing functions are now part of this API:

- *testing.assert_frame_equal()*
- *testing.assert_series_equal()*
- *testing.assert_index_equal()*

pandas.plotting

A new public `pandas.plotting` module has been added that holds plotting functionality that was previously in either `pandas.tools.plotting` or in the top-level namespace. See the *deprecations sections* for more details.

Other Development Changes

- Building pandas for development now requires `cython >= 0.23` (GH14831)
- Require at least 0.23 version of cython to avoid problems with character encodings (GH14699)
- Switched the test framework to use pytest (GH13097)
- Reorganization of tests directory layout (GH14854, GH15707).

Deprecations

Deprecate `.ix`

The `.ix` indexer is deprecated, in favor of the more strict `.iloc` and `.loc` indexers. `.ix` offers a lot of magic on the inference of what the user wants to do. To wit, `.ix` can decide to index *positionally* OR via *labels*, depending on the data type of the index. This has caused quite a bit of user confusion over the years. The full indexing documentation is *here*. (GH14218)

The recommended methods of indexing are:

- `.loc` if you want to *label* index
- `.iloc` if you want to *positionally* index.

Using `.ix` will now show a `DeprecationWarning` with a link to some examples of how to convert code *here*.

```
In [122]: df = pd.DataFrame({'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6]},
   .....:                   index=list('abc'))
   .....:

In [123]: df
Out[123]:
   A  B
a  1  4
b  2  5
c  3  6

[3 rows x 2 columns]
```

Previous behavior, where you wish to get the 0th and the 2nd elements from the index in the 'A' column.

```
In [3]: df.ix[[0, 2], 'A']
Out[3]:
a    1
c    3
Name: A, dtype: int64
```

Using `.loc`. Here we will select the appropriate indexes from the index, then use *label* indexing.

```
In [124]: df.loc[df.index[[0, 2]], 'A']
Out[124]:
a    1
c    3
Name: A, Length: 2, dtype: int64
```

Using `.iloc`. Here we will get the location of the 'A' column, then use *positional* indexing to select things.

```
In [125]: df.iloc[[0, 2], df.columns.get_loc('A')]
Out[125]:
a    1
c    3
Name: A, Length: 2, dtype: int64
```

Deprecate Panel

`Panel` is deprecated and will be removed in a future version. The recommended way to represent 3-D data are with a `MultiIndex` on a `DataFrame` via the `to_frame()` or with the xarray package. Pandas provides a `to_xarray()` method to automate this conversion (GH13563).

```
In [133]: import pandas._testing as tm

In [134]: p = tm.makePanel()

In [135]: p
Out[135]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

Convert to a MultiIndex DataFrame

```
In [136]: p.to_frame()
Out[136]:
                     ItemA     ItemB     ItemC
major      minor
2000-01-03 A      0.628776 -1.409432  0.209395
           B      0.988138 -1.347533 -0.896581
           C     -0.938153  1.272395 -0.161137
           D     -0.223019 -0.591863 -1.051539
2000-01-04 A      0.186494  1.422986 -0.592886
           B     -0.072608  0.363565  1.104352
           C     -1.239072 -1.449567  0.889157
           D      2.123692 -0.414505 -0.319561
2000-01-05 A      0.952478 -2.147855 -1.473116
           B     -0.550603 -0.014752 -0.431550
           C      0.139683 -1.195524  0.288377
           D      0.122273 -1.425795 -0.619993

[12 rows x 3 columns]
```

Convert to an xarray DataArray

```
In [137]: p.to_xarray()
Out[137]:
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[[ 0.628776,  0.988138, -0.938153, -0.223019],
        [ 0.186494, -0.072608, -1.239072,  2.123692],
        [ 0.952478, -0.550603,  0.139683,  0.122273]],

       [[-1.409432, -1.347533,  1.272395, -0.591863],
        [ 1.422986,  0.363565, -1.449567, -0.414505],
        [-2.147855, -0.014752, -1.195524, -1.425795]],

       [[ 0.209395, -0.896581, -0.161137, -1.051539],
        [-0.592886,  1.104352,  0.889157, -0.319561],
        [-1.473116, -0.43155 ,  0.288377, -0.619993]]])
Coordinates:
  * items       (items) object 'ItemA' 'ItemB' 'ItemC'
```

(continues on next page)

(continued from previous page)

```
  * major_axis  (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05
  * minor_axis  (minor_axis) object 'A' 'B' 'C' 'D'
```

Deprecate groupby.agg() with a dictionary when renaming

The `.groupby(..).agg(..)`, `.rolling(..).agg(..)`, and `.resample(..).agg(..)` syntax can accept a variable of inputs, including scalars, list, and a dict of column names to scalars or lists. This provides a useful syntax for constructing multiple (potentially different) aggregations.

However, `.agg(..)` can *also* accept a dict that allows 'renaming' of the result columns. This is a complicated and confusing syntax, as well as not consistent between `Series` and `DataFrame`. We are deprecating this 'renaming' functionality.

- We are deprecating passing a dict to a grouped/rolled/resampled `Series`. This allowed one to `rename` the resulting aggregation, but this had a completely different meaning than passing a dictionary to a grouped `DataFrame`, which accepts column-to-aggregations.
- We are deprecating passing a dict-of-dicts to a grouped/rolled/resampled `DataFrame` in a similar manner.

This is an illustrative example:

```
In [126]: df = pd.DataFrame({'A': [1, 1, 1, 2, 2],
   .....:                    'B': range(5),
   .....:                    'C': range(5)})
   .....:

In [127]: df
Out[127]:
   A  B  C
0  1  0  0
1  1  1  1
2  1  2  2
3  2  3  3
4  2  4  4

[5 rows x 3 columns]
```

Here is a typical useful syntax for computing different aggregations for different columns. This is a natural, and useful syntax. We aggregate from the dict-to-list by taking the specified columns and applying the list of functions. This returns a `MultiIndex` for the columns (this is *not* deprecated).

```
In [128]: df.groupby('A').agg({'B': 'sum', 'C': 'min'})
Out[128]:
   B  C
A
1  3  0
2  7  3

[2 rows x 2 columns]
```

Here's an example of the first deprecation, passing a dict to a grouped `Series`. This is a combination aggregation & renaming:

```
In [6]: df.groupby('A').B.agg({'foo': 'count'})
FutureWarning: using a dict on a Series for aggregation
```

(continues on next page)

(continued from previous page)

```
is deprecated and will be removed in a future version

Out[6]:
   foo
A
1    3
2    2
```

You can accomplish the same operation, more idiomatically by:

```
In [129]: df.groupby('A').B.agg(['count']).rename(columns={'count': 'foo'})
Out[129]:
   foo
A
1    3
2    2

[2 rows x 1 columns]
```

Here's an example of the second deprecation, passing a dict-of-dict to a grouped `DataFrame`:

```
In [23]: (df.groupby('A')
    ...:     .agg({'B': {'foo': 'sum'}, 'C': {'bar': 'min'}})
    ...:  )
FutureWarning: using a dict with renaming is deprecated and
will be removed in a future version

Out[23]:
     B   C
   foo bar
A
1    3   0
2    7   3
```

You can accomplish nearly the same by:

```
In [130]: (df.groupby('A')
   .....:    .agg({'B': 'sum', 'C': 'min'})
   .....:    .rename(columns={'B': 'foo', 'C': 'bar'})
   .....:  )
   .....:
Out[130]:
   foo  bar
A
1    3    0
2    7    3

[2 rows x 2 columns]
```