Bug fixes

Reshaping

- Bug in `DataFrame.groupby()` with `Grouper` when there is a time change (DST) and grouping frequency is `'1d'` (GH24972)

Visualization

- Fixed the warning for implicitly registered matplotlib converters not showing. See *Restore Matplotlib datetime converter registration* for more (GH24963).

Other

- Fixed AttributeError when printing a DataFrame's HTML repr after accessing the IPython config object (GH25036)

Contributors

A total of 7 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Alex Buchkovsky
- Roman Yurchak
- h-vetinari
- jbrockmendel
- Jeremy Schendel
- Joris Van den Bossche
- Tom Augspurger

5.3.3 What's new in 0.24.0 (January 25, 2019)

Warning: The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See Dropping Python 2.7 for more details.

This is a major release from 0.23.4 and includes a number of API changes, new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- *Optional Integer NA Support*
- *New APIs for accessing the array backing a Series or Index*
- *A new top-level method for creating arrays*
- *Store Interval and Period data in a Series or DataFrame*
- *Support for joining on two MultiIndexes*

Check the *API Changes* and *deprecations* before updating.

These are the changes in pandas 0.24.0. See *Release Notes* for a full changelog including other versions of pandas.

Enhancements

Optional integer NA support

Pandas has gained the ability to hold integer dtypes with missing values. This long requested feature is enabled through the use of *extension types*.

Note: IntegerArray is currently experimental. Its API or implementation may change without warning.

We can construct a `Series` with the specified dtype. The dtype string `Int64` is a pandas `ExtensionDtype`. Specifying a list or array using the traditional missing value marker of `np.nan` will infer to integer dtype. The display of the `Series` will also use the `NaN` to indicate missing values in string outputs. (GH20700, GH20747, GH22441, GH21789, GH22346)

```
In [1]: s = pd.Series([1, 2, np.nan], dtype='Int64')

In [2]: s
Out[2]:
0       1
1       2
2    <NA>
Length: 3, dtype: Int64
```

Operations on these dtypes will propagate `NaN` as other pandas operations.

```
# arithmetic
In [3]: s + 1
Out[3]:
0       2
1       3
2    <NA>
Length: 3, dtype: Int64

# comparison
In [4]: s == 1
Out[4]:
0     True
1    False
2     <NA>
Length: 3, dtype: boolean

# indexing
In [5]: s.iloc[1:3]
Out[5]:
1       2
2    <NA>
Length: 2, dtype: Int64

# operate with other dtypes
In [6]: s + s.iloc[1:3].astype('Int8')
Out[6]:
0    <NA>
1       4
2    <NA>
Length: 3, dtype: Int64
```

(continues on next page)

(continued from previous page)

```
# coerce when needed
In [7]: s + 0.01
Out[7]:
0    1.01
1    2.01
2     NaN
Length: 3, dtype: float64
```

These dtypes can operate as part of a `DataFrame`.

```
In [8]: df = pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab')})

In [9]: df
Out[9]:
      A  B  C
0     1  1  a
1     2  1  a
2  <NA>  3  b

[3 rows x 3 columns]

In [10]: df.dtypes
Out[10]:
A     Int64
B     int64
C    object
Length: 3, dtype: object
```

These dtypes can be merged, reshaped, and casted.

```
In [11]: pd.concat([df[['A']], df[['B', 'C']]], axis=1).dtypes
Out[11]:
A     Int64
B     int64
C    object
Length: 3, dtype: object

In [12]: df['A'].astype(float)
Out[12]:
0    1.0
1    2.0
2    NaN
Name: A, Length: 3, dtype: float64
```

Reduction and groupby operations such as `sum` work.

```
In [13]: df.sum()
Out[13]:
A      3
B      5
C    aab
Length: 3, dtype: object

In [14]: df.groupby('B').A.sum()
Out[14]:
B
```

(continues on next page)

(continued from previous page)

```
1    3
3    0
Name: A, Length: 2, dtype: Int64
```

Warning: The Integer NA support currently uses the capitalized dtype version, e.g. `Int8` as compared to the traditional `int8`. This may be changed at a future date.

See *Nullable integer data type* for more.

Accessing the values in a Series or Index

`Series.array` and *`Index.array`* have been added for extracting the array backing a `Series` or `Index`. (GH19954, GH23623)

```
In [15]: idx = pd.period_range('2000', periods=4)

In [16]: idx.array
Out[16]:
<PeriodArray>
['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04']
Length: 4, dtype: period[D]

In [17]: pd.Series(idx).array
Out[17]:
<PeriodArray>
['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04']
Length: 4, dtype: period[D]
```

Historically, this would have been done with `series.values`, but with `.values` it was unclear whether the returned value would be the actual array, some transformation of it, or one of pandas custom arrays (like `Categorical`). For example, with *`PeriodIndex`*, `.values` generates a new ndarray of period objects each time.

```
In [18]: idx.values
Out[18]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
       Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)

In [19]: id(idx.values)
Out[19]: 139993251009120

In [20]: id(idx.values)
Out[20]: 139995379933952
```

If you need an actual NumPy array, use *`Series.to_numpy()`* or *`Index.to_numpy()`*.

```
In [21]: idx.to_numpy()
Out[21]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
       Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)

In [22]: pd.Series(idx).to_numpy()
```

(continues on next page)

(continued from previous page)

```
Out[22]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
       Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)
```

For Series and Indexes backed by normal NumPy arrays, *Series.array* will return a new *arrays.PandasArray*, which is a thin (no-copy) wrapper around a numpy.ndarray. *PandasArray* isn't especially useful on its own, but it does provide the same interface as any extension array defined in pandas or by a third-party library.

```
In [23]: ser = pd.Series([1, 2, 3])

In [24]: ser.array
Out[24]:
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64

In [25]: ser.to_numpy()
Out[25]: array([1, 2, 3])
```

We haven't removed or deprecated *Series.values* or *DataFrame.values*, but we highly recommend and using `.array` or `.to_numpy()` instead.

See *Dtypes* and *Attributes and Underlying Data* for more.

pandas.array: a new top-level method for creating arrays

A new top-level method *array()* has been added for creating 1-dimensional arrays (GH22860). This can be used to create any *extension array*, including extension arrays registered by 3rd party libraries. See the *dtypes docs* for more on extension arrays.

```
In [26]: pd.array([1, 2, np.nan], dtype='Int64')
Out[26]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64

In [27]: pd.array(['a', 'b', 'c'], dtype='category')
Out[27]:
[a, b, c]
Categories (3, object): [a, b, c]
```

Passing data for which there isn't dedicated extension type (e.g. float, integer, etc.) will return a new *arrays.PandasArray*, which is just a thin (no-copy) wrapper around a numpy.ndarray that satisfies the pandas extension array interface.

```
In [28]: pd.array([1, 2, 3])
Out[28]:
<IntegerArray>
[1, 2, 3]
Length: 3, dtype: Int64
```

On their own, a *PandasArray* isn't a very useful object. But if you need write low-level code that works generically for any *ExtensionArray*, *PandasArray* satisfies that need.

Notice that by default, if no `dtype` is specified, the dtype of the returned array is inferred from the data. In particular, note that the first example of `[1, 2, np.nan]` would have returned a floating-point array, since `NaN` is a float.

```
In [29]: pd.array([1, 2, np.nan])
Out[29]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

Storing Interval and Period data in Series and DataFrame

Interval and *Period* data may now be stored in a *Series* or *DataFrame*, in addition to an *IntervalIndex* and *PeriodIndex* like previously (GH19453, GH22862).

```
In [30]: ser = pd.Series(pd.interval_range(0, 5))

In [31]: ser
Out[31]:
0    (0, 1]
1    (1, 2]
2    (2, 3]
3    (3, 4]
4    (4, 5]
Length: 5, dtype: interval

In [32]: ser.dtype
Out[32]: interval[int64]
```

For periods:

```
In [33]: pser = pd.Series(pd.period_range("2000", freq="D", periods=5))

In [34]: pser
Out[34]:
0    2000-01-01
1    2000-01-02
2    2000-01-03
3    2000-01-04
4    2000-01-05
Length: 5, dtype: period[D]

In [35]: pser.dtype
Out[35]: period[D]
```

Previously, these would be cast to a NumPy array with object dtype. In general, this should result in better performance when storing an array of intervals or periods in a *Series* or column of a *DataFrame*.

Use *Series.array* to extract the underlying array of intervals or periods from the `Series`:

```
In [36]: ser.array
Out[36]:
<IntervalArray>
[(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]
Length: 5, closed: right, dtype: interval[int64]

In [37]: pser.array
```

(continues on next page)

(continued from previous page)

```
Out[37]:
<PeriodArray>
['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04', '2000-01-05']
Length: 5, dtype: period[D]
```

These return an instance of *arrays.IntervalArray* or *arrays.PeriodArray*, the new extension arrays that back interval and period data.

Warning: For backwards compatibility, *Series.values* continues to return a NumPy array of objects for Interval and Period data. We recommend using *Series.array* when you need the array of data stored in the `Series`, and *Series.to_numpy()* when you know you need a NumPy array.

See *Dtypes* and *Attributes and Underlying Data* for more.

Joining with two multi-indexes

DataFrame.merge() and *DataFrame.join()* can now be used to join multi-indexed `Dataframe` instances on the overlapping index levels (GH6360)

See the *Merge, join, and concatenate* documentation section.

```
In [38]: index_left = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
   ....:                                         ('K1', 'X2')],
   ....:                                        names=['key', 'X'])
   ....:

In [39]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']}, index=index_left)
   ....:

In [40]: index_right = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   ....:                                          ('K2', 'Y2'), ('K2', 'Y3')],
   ....:                                         names=['key', 'Y'])
   ....:

In [41]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']}, index=index_right)
   ....:

In [42]: left.join(right)
Out[42]:
            A   B   C   D
key X  Y
K0  X0 Y0  A0  B0  C0  D0
    X1 Y0  A1  B1  C0  D0
K1  X2 Y1  A2  B2  C1  D1

[3 rows x 4 columns]
```

For earlier versions this can be done using the following.

```
In [43]: pd.merge(left.reset_index(), right.reset_index(),
   ....:          on=['key'], how='inner').set_index(['key', 'X', 'Y'])
   ....:
```

(continues on next page)

(continued from previous page)

```
Out[43]:
             A   B   C   D
key X  Y
K0  X0 Y0  A0  B0  C0  D0
    X1 Y0  A1  B1  C0  D0
K1  X2 Y1  A2  B2  C1  D1

[3 rows x 4 columns]
```

`read_html` Enhancements

`read_html()` previously ignored `colspan` and `rowspan` attributes. Now it understands them, treating them as sequences of cells with the same value. (GH17054)

```
In [44]: result = pd.read_html("""
   ....:   <table>
   ....:     <thead>
   ....:       <tr>
   ....:         <th>A</th><th>B</th><th>C</th>
   ....:       </tr>
   ....:     </thead>
   ....:     <tbody>
   ....:       <tr>
   ....:         <td colspan="2">1</td><td>2</td>
   ....:       </tr>
   ....:     </tbody>
   ....:   </table>""")
   ....:
```

Previous behavior:

```
In [13]: result
Out [13]:
[   A  B   C
 0  1  2 NaN]
```

New behavior:

```
In [45]: result
Out[45]:
[   A  B  C
 0  1  1  2

 [1 rows x 3 columns]]
```

New `Styler.pipe()` method

The *Styler* class has gained a *pipe()* method. This provides a convenient way to apply users' predefined styling functions, and can help reduce "boilerplate" when using DataFrame styling functionality repeatedly within a notebook. (GH23229)

```
In [46]: df = pd.DataFrame({'N': [1250, 1500, 1750], 'X': [0.25, 0.35, 0.50]})

In [47]: def format_and_align(styler):
   ....:     return (styler.format({'N': '{:,}', 'X': '{:.1%}'})
   ....:                   .set_properties(**{'text-align': 'right'}))
   ....:

In [48]: df.style.pipe(format_and_align).set_caption('Summary of results.')
Out[48]: <pandas.io.formats.style.Styler at 0x7f52bc1e5890>
```

Similar methods already exist for other classes in pandas, including *DataFrame.pipe()*, *GroupBy.pipe()*, and *Resampler.pipe()*.

Renaming names in a MultiIndex

DataFrame.rename_axis() now supports `index` and `columns` arguments and *Series.rename_axis()* supports `index` argument (GH19978).

This change allows a dictionary to be passed so that some of the names of a `MultiIndex` can be changed.

Example:

```
In [49]: mi = pd.MultiIndex.from_product([list('AB'), list('CD'), list('EF')],
   ....:                                 names=['AB', 'CD', 'EF'])
   ....:

In [50]: df = pd.DataFrame(list(range(len(mi))), index=mi, columns=['N'])

In [51]: df
Out[51]:
           N
AB CD EF
A  C  E    0
      F    1
   D  E    2
      F    3
B  C  E    4
      F    5
   D  E    6
      F    7

[8 rows x 1 columns]

In [52]: df.rename_axis(index={'CD': 'New'})
Out[52]:
            N
AB New EF
A  C   E    0
       F    1
   D   E    2
```

(continues on next page)

(continued from previous page)

```
        F   3
B  C    E   4
        F   5
   D    E   6
        F   7

[8 rows x 1 columns]
```

See the *Advanced documentation on renaming* for more details.

Other enhancements

- *merge()* now directly allows merge between objects of type DataFrame and named Series, without the need to convert the Series object into a DataFrame beforehand (GH21220)
- ExcelWriter now accepts mode as a keyword argument, enabling append to existing workbooks when using the openpyxl engine (GH3441)
- FrozenList has gained the .union() and .difference() methods. This functionality greatly simplifies groupby's that rely on explicitly excluding certain columns. See *Splitting an object into groups* for more information (GH15475, GH15506).
- *DataFrame.to_parquet()* now accepts index as an argument, allowing the user to override the engine's default behavior to include or omit the dataframe's indexes from the resulting Parquet file. (GH20768)
- *read_feather()* now accepts columns as an argument, allowing the user to specify which columns should be read. (GH24025)
- *DataFrame.corr()* and *Series.corr()* now accept a callable for generic calculation methods of correlation, e.g. histogram intersection (GH22684)
- *DataFrame.to_string()* now accepts decimal as an argument, allowing the user to specify which decimal separator should be used in the output. (GH23614)
- *DataFrame.to_html()* now accepts render_links as an argument, allowing the user to generate HTML with links to any URLs that appear in the DataFrame. See the *section on writing HTML* in the IO docs for example usage. (GH2679)
- *pandas.read_csv()* now supports pandas extension types as an argument to dtype, allowing the user to use pandas extension types when reading CSVs. (GH23228)
- The *shift()* method now accepts *fill_value* as an argument, allowing the user to specify a value which will be used instead of NA/NaT in the empty periods. (GH15486)
- *to_datetime()* now supports the %Z and %z directive when passed into format (GH13486)
- *Series.mode()* and *DataFrame.mode()* now support the dropna parameter which can be used to specify whether NaN/NaT values should be considered (GH17534)
- *DataFrame.to_csv()* and *Series.to_csv()* now support the compression keyword when a file handle is passed. (GH21227)
- *Index.droplevel()* is now implemented also for flat indexes, for compatibility with *MultiIndex* (GH21115)
- *Series.droplevel()* and *DataFrame.droplevel()* are now implemented (GH20342)
- Added support for reading from/writing to Google Cloud Storage via the gcsfs library (GH19454, GH23094)

- *DataFrame.to_gbq()* and *read_gbq()* signature and documentation updated to reflect changes from the Pandas-GBQ library version 0.8.0. Adds a `credentials` argument, which enables the use of any kind of google-auth credentials. (GH21627, GH22557, GH23662)
- New method *HDFStore.walk()* will recursively walk the group hierarchy of an HDF5 file (GH10932)
- *read_html()* copies cell data across `colspan` and `rowspan`, and it treats all-`th` table rows as headers if `header` kwarg is not given and there is no `thead` (GH17054)
- *Series.nlargest()*, *Series.nsmallest()*, *DataFrame.nlargest()*, and *DataFrame.nsmallest()* now accept the value `"all"` for the `keep` argument. This keeps all ties for the nth largest/smallest value (GH16818)
- *IntervalIndex* has gained the *set_closed()* method to change the existing `closed` value (GH21670)
- *to_csv()*, *to_csv()*, *to_json()*, and *to_json()* now support `compression='infer'` to infer compression based on filename extension (GH15008). The default compression for `to_csv`, `to_json`, and `to_pickle` methods has been updated to `'infer'` (GH22004).
- *DataFrame.to_sql()* now supports writing `TIMESTAMP WITH TIME ZONE` types for supported databases. For databases that don't support timezones, datetime data will be stored as timezone unaware local timestamps. See the *Datetime data types* for implications (GH9086).
- *to_timedelta()* now supports iso-formated timedelta strings (GH21877)
- *Series* and *DataFrame* now support `Iterable` objects in the constructor (GH2193)
- *DatetimeIndex* has gained the *DatetimeIndex.timetz* attribute. This returns the local time with timezone information. (GH21358)
- *round()*, *ceil()*, and *floor()* for *DatetimeIndex* and *Timestamp* now support an `ambiguous` argument for handling datetimes that are rounded to ambiguous times (GH18946) and a `nonexistent` argument for handling datetimes that are rounded to nonexistent times. See *Nonexistent times when localizing* (GH22647)
- The result of *resample()* is now iterable similar to `groupby()` (GH15314).
- *Series.resample()* and *DataFrame.resample()* have gained the *pandas.core.resample.Resampler.quantile()* (GH15023).
- *DataFrame.resample()* and *Series.resample()* with a *PeriodIndex* will now respect the `base` argument in the same fashion as with a *DatetimeIndex*. (GH23882)
- *pandas.api.types.is_list_like()* has gained a keyword `allow_sets` which is `True` by default; if `False`, all instances of `set` will not be considered "list-like" anymore (GH23061)
- *Index.to_frame()* now supports overriding column name(s) (GH22580).
- *Categorical.from_codes()* now can take a `dtype` parameter as an alternative to passing `categories` and `ordered` (GH24398).
- New attribute `__git_version__` will return git commit sha of current build (GH21295).
- Compatibility with Matplotlib 3.0 (GH22790).
- Added *Interval.overlaps()*, *arrays.IntervalArray.overlaps()*, and *IntervalIndex.overlaps()* for determining overlaps between interval-like objects (GH21998)
- *read_fwf()* now accepts keyword `infer_nrows` (GH15138).
- *to_parquet()* now supports writing a `DataFrame` as a directory of parquet files partitioned by a subset of the columns when `engine = 'pyarrow'` (GH23283)

- *Timestamp.tz_localize()*, *DatetimeIndex.tz_localize()*, and *Series.tz_localize()* have gained the `nonexistent` argument for alternative handling of nonexistent times. See *Nonexistent times when localizing* (GH8917, GH24466)
- *Index.difference()*, *Index.intersection()*, *Index.union()*, and *Index.symmetric_difference()* now have an optional `sort` parameter to control whether the results should be sorted if possible (GH17839, GH24471)
- *read_excel()* now accepts `usecols` as a list of column names or callable (GH18273)
- *MultiIndex.to_flat_index()* has been added to flatten multiple levels into a single-level *Index* object.
- *DataFrame.to_stata()* and `pandas.io.stata.StataWriter117` can write mixed sting columns to Stata strl format (GH23633)
- *DataFrame.between_time()* and *DataFrame.at_time()* have gained the `axis` parameter (GH8839)
- *DataFrame.to_records()* now accepts `index_dtypes` and `column_dtypes` parameters to allow different data types in stored column and index records (GH18146)
- *IntervalIndex* has gained the *is_overlapping* attribute to indicate if the `IntervalIndex` contains any overlapping intervals (GH23309)
- *pandas.DataFrame.to_sql()* has gained the `method` argument to control SQL insertion clause. See the *insertion method* section in the documentation. (GH8953)
- *DataFrame.corrwith()* now supports Spearman's rank correlation, Kendall's tau as well as callable correlation methods. (GH21925)
- *DataFrame.to_json()*, *DataFrame.to_csv()*, *DataFrame.to_pickle()*, and other export methods now support tilde(~) in path argument. (GH23473)

Backwards incompatible API changes

Pandas 0.24.0 includes a number of API breaking changes.

Increased minimum versions for dependencies

We have updated our minimum supported versions of dependencies (GH21242, GH18742, GH23774, GH24767). If installed, we now require:

Package	Minimum Version	Required
numpy	1.12.0	X
bottleneck	1.2.0	
fastparquet	0.2.1	
matplotlib	2.0.0	
numexpr	2.6.1	
pandas-gbq	0.8.0	
pyarrow	0.9.0	
pytables	3.4.2	
scipy	0.18.1	
xlrd	1.0.0	
pytest (dev)	3.6	

Additionally we no longer depend on `feather-format` for feather based storage and replaced it with references to `pyarrow` (GH21639 and GH23053).

os.linesep is used for `line_terminator` of `DataFrame.to_csv`

DataFrame.to_csv() now uses `os.linesep()` rather than `'\n'` for the default line terminator (GH20353). This change only affects when running on Windows, where `'\r\n'` was used for line terminator even when `'\n'` was passed in `line_terminator`.

Previous behavior on Windows:

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
   ...:                      "string_with_crlf": ["a\r\nbc"]})

In [2]: # When passing file PATH to to_csv,
   ...: # line_terminator does not work, and csv is saved with '\r\n'.
   ...: # Also, this converts all '\n's in the data to '\r\n'.
   ...: data.to_csv("test.csv", index=False, line_terminator='\n')

In [3]: with open("test.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\r\nbc","a\r\r\nbc"\r\n'

In [4]: # When passing file OBJECT with newline option to
   ...: # to_csv, line_terminator works.
   ...: with open("test2.csv", mode='w', newline='\n') as f:
   ...:     data.to_csv(f, index=False, line_terminator='\n')

In [5]: with open("test2.csv", mode='rb') as f:
   ...:     print(f.read())
Out[5]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'
```

New behavior on Windows:

Passing `line_terminator` explicitly, set thes `line terminator` to that character.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
   ...:                      "string_with_crlf": ["a\r\nbc"]})

In [2]: data.to_csv("test.csv", index=False, line_terminator='\n')

In [3]: with open("test.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'
```

On Windows, the value of `os.linesep` is `'\r\n'`, so if `line_terminator` is not set, `'\r\n'` is used for line terminator.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
   ...:                      "string_with_crlf": ["a\r\nbc"]})

In [2]: data.to_csv("test.csv", index=False)

In [3]: with open("test.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\nbc","a\r\nbc"\r\n'
```

For file objects, specifying `newline` is not sufficient to set the line terminator. You must pass in the `line_terminator` explicitly, even in this case.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
   ...:                      "string_with_crlf": ["a\r\nbc"]})

In [2]: with open("test2.csv", mode='w', newline='\n') as f:
   ...:     data.to_csv(f, index=False)

In [3]: with open("test2.csv", mode='rb') as f:
   ...:     print(f.read())
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\nbc","a\r\nbc"\r\n'
```

Proper handling of *np.NaN* in a string data-typed column with the Python engine

There was bug in *read_excel()* and *read_csv()* with the Python engine, where missing values turned to `'nan'` with `dtype=str` and `na_filter=True`. Now, these missing values are converted to the string missing indicator, `np.nan`. (GH20377)

Previous behavior:

```
In [5]: data = 'a,b,c\n1,,3\n4,5,6'
In [6]: df = pd.read_csv(StringIO(data), engine='python', dtype=str, na_filter=True)
In [7]: df.loc[0, 'b']
Out[7]:
'nan'
```

New behavior:

```
In [53]: data = 'a,b,c\n1,,3\n4,5,6'

In [54]: df = pd.read_csv(StringIO(data), engine='python', dtype=str, na_filter=True)

In [55]: df.loc[0, 'b']
Out[55]: nan
```

Notice how we now instead output `np.nan` itself instead of a stringified form of it.

Parsing datetime strings with timezone offsets

Previously, parsing datetime strings with UTC offsets with *to_datetime()* or *DatetimeIndex* would automatically convert the datetime to UTC without timezone localization. This is inconsistent from parsing the same datetime string with *Timestamp* which would preserve the UTC offset in the `tz` attribute. Now, *to_datetime()* preserves the UTC offset in the `tz` attribute when all the datetime strings have the same UTC offset (GH17697, GH11736, GH22457)

Previous behavior:

```
In [2]: pd.to_datetime("2015-11-18 15:30:00+05:30")
Out[2]: Timestamp('2015-11-18 10:00:00')

In [3]: pd.Timestamp("2015-11-18 15:30:00+05:30")
Out[3]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')

# Different UTC offsets would automatically convert the datetimes to UTC (without a␣
↪UTC timezone)
```

(continues on next page)

(continued from previous page)

```
In [4]: pd.to_datetime(["2015-11-18 15:30:00+05:30", "2015-11-18 16:30:00+06:30"])
Out[4]: DatetimeIndex(['2015-11-18 10:00:00', '2015-11-18 10:00:00'], dtype=
↪'datetime64[ns]', freq=None)
```

New behavior:

```
In [56]: pd.to_datetime("2015-11-18 15:30:00+05:30")
Out[56]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')

In [57]: pd.Timestamp("2015-11-18 15:30:00+05:30")
Out[57]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330)')
```

Parsing datetime strings with the same UTC offset will preserve the UTC offset in the `tz`

```
In [58]: pd.to_datetime(["2015-11-18 15:30:00+05:30"] * 2)
Out[58]: DatetimeIndex(['2015-11-18 15:30:00+05:30', '2015-11-18 15:30:00+05:30'],␣
↪dtype='datetime64[ns, pytz.FixedOffset(330)]', freq=None)
```

Parsing datetime strings with different UTC offsets will now create an Index of `datetime.datetime` objects with different UTC offsets

```
In [59]: idx = pd.to_datetime(["2015-11-18 15:30:00+05:30",
   ....:                       "2015-11-18 16:30:00+06:30"])
   ....:

In [60]: idx
Out[60]: Index([2015-11-18 15:30:00+05:30, 2015-11-18 16:30:00+06:30], dtype='object')

In [61]: idx[0]
Out[61]: datetime.datetime(2015, 11, 18, 15, 30, tzinfo=tzoffset(None, 19800))

In [62]: idx[1]
Out[62]: datetime.datetime(2015, 11, 18, 16, 30, tzinfo=tzoffset(None, 23400))
```

Passing `utc=True` will mimic the previous behavior but will correctly indicate that the dates have been converted to UTC

```
In [63]: pd.to_datetime(["2015-11-18 15:30:00+05:30",
   ....:                 "2015-11-18 16:30:00+06:30"], utc=True)
   ....:
Out[63]: DatetimeIndex(['2015-11-18 10:00:00+00:00', '2015-11-18 10:00:00+00:00'],␣
↪dtype='datetime64[ns, UTC]', freq=None)
```

Parsing mixed-timezones with `read_csv()`

`read_csv()` no longer silently converts mixed-timezone columns to UTC (GH24987).

Previous behavior

```
>>> import io
>>> content = """\
... a
... 2000-01-01T00:00:00+05:00
... 2000-01-01T00:00:00+06:00"""
>>> df = pd.read_csv(io.StringIO(content), parse_dates=['a'])
```

(continues on next page)

(continued from previous page)

```
>>> df.a
0   1999-12-31 19:00:00
1   1999-12-31 18:00:00
Name: a, dtype: datetime64[ns]
```

New behavior

```
In [64]: import io

In [65]: content = """\
   ....: a
   ....: 2000-01-01T00:00:00+05:00
   ....: 2000-01-01T00:00:00+06:00"""
   ....:

In [66]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'])

In [67]: df.a
Out[67]:
0    2000-01-01 00:00:00+05:00
1    2000-01-01 00:00:00+06:00
Name: a, Length: 2, dtype: object
```

As can be seen, the `dtype` is object; each value in the column is a string. To convert the strings to an array of datetimes, the `date_parser` argument

```
In [68]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'],
   ....:                  date_parser=lambda col: pd.to_datetime(col, utc=True))
   ....:

In [69]: df.a
Out[69]:
0   1999-12-31 19:00:00+00:00
1   1999-12-31 18:00:00+00:00
Name: a, Length: 2, dtype: datetime64[ns, UTC]
```

See *Parsing datetime strings with timezone offsets* for more.

Time values in `dt.end_time` and `to_timestamp(how='end')`

The time values in *Period* and *PeriodIndex* objects are now set to '23:59:59.999999999' when calling *Series.dt.end_time*, *Period.end_time*, *PeriodIndex.end_time*, *Period.to_timestamp()* with `how='end'`, or *PeriodIndex.to_timestamp()* with `how='end'` (GH17157)

Previous behavior:

```
In [2]: p = pd.Period('2017-01-01', 'D')
In [3]: pi = pd.PeriodIndex([p])

In [4]: pd.Series(pi).dt.end_time[0]
Out[4]: Timestamp(2017-01-01 00:00:00)

In [5]: p.end_time
Out[5]: Timestamp(2017-01-01 23:59:59.999999999)
```

New behavior:

Calling *Series.dt.end_time* will now result in a time of '23:59:59.999999999' as is the case with *Period.end_time*, for example

```
In [70]: p = pd.Period('2017-01-01', 'D')

In [71]: pi = pd.PeriodIndex([p])

In [72]: pd.Series(pi).dt.end_time[0]
Out[72]: Timestamp('2017-01-01 23:59:59.999999999')

In [73]: p.end_time
Out[73]: Timestamp('2017-01-01 23:59:59.999999999')
```

Series.unique for Timezone-Aware Data

The return type of *Series.unique()* for datetime with timezone values has changed from an numpy.ndarray of *Timestamp* objects to a *arrays.DatetimeArray* (GH24024).

```
In [74]: ser = pd.Series([pd.Timestamp('2000', tz='UTC'),
   ....:                  pd.Timestamp('2000', tz='UTC')])
   ....:
```

Previous behavior:

```
In [3]: ser.unique()
Out[3]: array([Timestamp('2000-01-01 00:00:00+0000', tz='UTC')], dtype=object)
```

New behavior:

```
In [75]: ser.unique()
Out[75]:
<DatetimeArray>
['2000-01-01 00:00:00+00:00']
Length: 1, dtype: datetime64[ns, UTC]
```

Sparse data structure refactor

SparseArray, the array backing SparseSeries and the columns in a SparseDataFrame, is now an extension array (GH21978, GH19056, GH22835). To conform to this interface and for consistency with the rest of pandas, some API breaking changes were made:

- SparseArray is no longer a subclass of numpy.ndarray. To convert a SparseArray to a NumPy array, use numpy.asarray().
- SparseArray.dtype and SparseSeries.dtype are now instances of *SparseDtype*, rather than np.dtype. Access the underlying dtype with SparseDtype.subtype.
- numpy.asarray(sparse_array) now returns a dense array with all the values, not just the non-fill-value values (GH14167)
- SparseArray.take now matches the API of *pandas.api.extensions.ExtensionArray.take()* (GH19506):
  - The default value of allow_fill has changed from False to True.
  - The out and mode parameters are now longer accepted (previously, this raised if they were specified).

  - Passing a scalar for `indices` is no longer allowed.

- The result of *`concat()`* with a mix of sparse and dense Series is a Series with sparse values, rather than a `SparseSeries`.
- `SparseDataFrame.combine` and `DataFrame.combine_first` no longer supports combining a sparse column with a dense column while preserving the sparse subtype. The result will be an object-dtype SparseArray.
- Setting `SparseArray.fill_value` to a fill value with a different dtype is now allowed.
- `DataFrame[column]` is now a *`Series`* with sparse values, rather than a `SparseSeries`, when slicing a single column with sparse values (GH23559).
- The result of *`Series.where()`* is now a `Series` with sparse values, like with other extension arrays (GH24077)

Some new warnings are issued for operations that require or are likely to materialize a large dense array:

- A *`errors.PerformanceWarning`* is issued when using fillna with a `method`, as a dense array is constructed to create the filled array. Filling with a `value` is the efficient way to fill a sparse array.
- A *`errors.PerformanceWarning`* is now issued when concatenating sparse Series with differing fill values. The fill value from the first sparse array continues to be used.

In addition to these API breaking changes, many *Performance Improvements and Bug Fixes have been made*.

Finally, a `Series.sparse` accessor was added to provide sparse-specific methods like *`Series.sparse.from_coo()`*.

```
In [76]: s = pd.Series([0, 0, 1, 1, 1], dtype='Sparse[int]')

In [77]: s.sparse.density
Out[77]: 0.6
```

`get_dummies()` always returns a DataFrame

Previously, when `sparse=True` was passed to *`get_dummies()`*, the return value could be either a *`DataFrame`* or a `SparseDataFrame`, depending on whether all or a just a subset of the columns were dummy-encoded. Now, a *`DataFrame`* is always returned (GH24284).

Previous behavior

The first *`get_dummies()`* returns a *`DataFrame`* because the column `A` is not dummy encoded. When just `["B", "C"]` are passed to `get_dummies`, then all the columns are dummy-encoded, and a `SparseDataFrame` was returned.

```
In [2]: df = pd.DataFrame({"A": [1, 2], "B": ['a', 'b'], "C": ['a', 'a']})

In [3]: type(pd.get_dummies(df, sparse=True))
Out[3]: pandas.core.frame.DataFrame

In [4]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
Out[4]: pandas.core.sparse.frame.SparseDataFrame
```

New behavior

Now, the return type is consistently a *`DataFrame`*.

```
In [78]: type(pd.get_dummies(df, sparse=True))
Out[78]: pandas.core.frame.DataFrame

In [79]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
Out[79]: pandas.core.frame.DataFrame
```

Note: There's no difference in memory usage between a `SparseDataFrame` and a *DataFrame* with sparse values. The memory usage will be the same as in the previous version of pandas.

Raise ValueError in `DataFrame.to_dict(orient='index')`

Bug in *DataFrame.to_dict()* raises `ValueError` when used with `orient='index'` and a non-unique index instead of losing data (GH22801)

```
In [80]: df = pd.DataFrame({'a': [1, 2], 'b': [0.5, 0.75]}, index=['A', 'A'])

In [81]: df
Out[81]:
   a     b
A  1  0.50
A  2  0.75

[2 rows x 2 columns]

In [82]: df.to_dict(orient='index')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-82-f5309a7c6adb> in <module>
----> 1 df.to_dict(orient='index')

/pandas-release/pandas/pandas/core/frame.py in to_dict(self, orient, into)
   1431         elif orient.lower().startswith("i"):
   1432             if not self.index.is_unique:
-> 1433                 raise ValueError("DataFrame index must be unique for orient=
↪'index'.")
   1434             return into_c(
   1435                 (t[0], dict(zip(self.columns, t[1:]))))

ValueError: DataFrame index must be unique for orient='index'.
```

Tick DateOffset normalize restrictions

Creating a `Tick` object (`Day`, `Hour`, `Minute`, `Second`, `Milli`, `Micro`, `Nano`) with `normalize=True` is no longer supported. This prevents unexpected behavior where addition could fail to be monotone or associative. (GH21427)

Previous behavior:

```
In [2]: ts = pd.Timestamp('2018-06-11 18:01:14')

In [3]: ts
Out[3]: Timestamp('2018-06-11 18:01:14')
```

(continues on next page)

(continued from previous page)

```
In [4]: tic = pd.offsets.Hour(n=2, normalize=True)
   ...:

In [5]: tic
Out[5]: <2 * Hours>

In [6]: ts + tic
Out[6]: Timestamp('2018-06-11 00:00:00')

In [7]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[7]: False
```

New behavior:

```
In [83]: ts = pd.Timestamp('2018-06-11 18:01:14')

In [84]: tic = pd.offsets.Hour(n=2)

In [85]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[85]: True
```

Period subtraction

Subtraction of a `Period` from another `Period` will give a `DateOffset`. instead of an integer (GH21314)

Previous behavior:

```
In [2]: june = pd.Period('June 2018')

In [3]: april = pd.Period('April 2018')

In [4]: june - april
Out [4]: 2
```

New behavior:

```
In [86]: june = pd.Period('June 2018')

In [87]: april = pd.Period('April 2018')

In [88]: june - april
Out[88]: <2 * MonthEnds>
```

Similarly, subtraction of a `Period` from a `PeriodIndex` will now return an `Index` of `DateOffset` objects instead of an `Int64Index`

Previous behavior:

```
In [2]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [3]: pi - pi[0]
Out[3]: Int64Index([0, 1, 2], dtype='int64')
```

New behavior:

```
In [89]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [90]: pi - pi[0]
Out[90]: Index([<0 * MonthEnds>, <MonthEnd>, <2 * MonthEnds>], dtype='object')
```

Addition/subtraction of `NaN` from `DataFrame`

Adding or subtracting `NaN` from a *DataFrame* column with `timedelta64[ns]` dtype will now raise a `TypeError` instead of returning all-`NaT`. This is for compatibility with `TimedeltaIndex` and `Series` behavior (GH22163)

```
In [91]: df = pd.DataFrame([pd.Timedelta(days=1)])

In [92]: df
Out[92]:
       0
0 1 days

[1 rows x 1 columns]
```

Previous behavior:

```
In [4]: df = pd.DataFrame([pd.Timedelta(days=1)])

In [5]: df - np.nan
Out[5]:
    0
0 NaT
```

New behavior:

```
In [2]: df - np.nan
...
TypeError: unsupported operand type(s) for -: 'TimedeltaIndex' and 'float'
```

DataFrame comparison operations broadcasting changes

Previously, the broadcasting behavior of *DataFrame* comparison operations (==, !=, ...) was inconsistent with the behavior of arithmetic operations (+, -, ...). The behavior of the comparison operations has been changed to match the arithmetic operations in these cases. (GH22880)

The affected cases are:

- operating against a 2-dimensional `np.ndarray` with either 1 row or 1 column will now broadcast the same way a `np.ndarray` would (GH23000).
- a list or tuple with length matching the number of rows in the *DataFrame* will now raise `ValueError` instead of operating column-by-column (GH22880.
- a list or tuple with length matching the number of columns in the *DataFrame* will now operate row-by-row instead of raising `ValueError` (GH22880).

```
In [93]: arr = np.arange(6).reshape(3, 2)
```

(continues on next page)

(continued from previous page)

```
In [94]: df = pd.DataFrame(arr)

In [95]: df
Out[95]:
   0  1
0  0  1
1  2  3
2  4  5

[3 rows x 2 columns]
```

Previous behavior:

```
In [5]: df == arr[[0], :]
    ...: # comparison previously broadcast where arithmetic would raise
Out[5]:
       0      1
0   True   True
1  False  False
2  False  False
In [6]: df + arr[[0], :]
...
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (1, 2)

In [7]: df == (1, 2)
    ...: # length matches number of columns;
    ...: # comparison previously raised where arithmetic would broadcast
...
ValueError: Invalid broadcasting comparison [(1, 2)] with block values
In [8]: df + (1, 2)
Out[8]:
   0  1
0  1  3
1  3  5
2  5  7

In [9]: df == (1, 2, 3)
    ...:  # length matches number of rows
    ...:  # comparison previously broadcast where arithmetic would raise
Out[9]:
       0      1
0  False   True
1   True  False
2  False  False
In [10]: df + (1, 2, 3)
...
ValueError: Unable to coerce to Series, length must be 2: given 3
```

New behavior:

```
# Comparison operations and arithmetic operations both broadcast.
In [96]: df == arr[[0], :]
Out[96]:
       0      1
0   True   True
1  False  False
2  False  False
```

(continues on next page)

(continued from previous page)

```
[3 rows x 2 columns]

In [97]: df + arr[[0], :]
Out[97]:
   0  1
0  0  2
1  2  4
2  4  6

[3 rows x 2 columns]
```

```
# Comparison operations and arithmetic operations both broadcast.
In [98]: df == (1, 2)
Out[98]:
       0      1
0  False  False
1  False  False
2  False  False

[3 rows x 2 columns]

In [99]: df + (1, 2)
Out[99]:
   0  1
0  1  3
1  3  5
2  5  7

[3 rows x 2 columns]
```

```
# Comparison operations and arithmetic operations both raise ValueError.
In [6]: df == (1, 2, 3)
...
ValueError: Unable to coerce to Series, length must be 2: given 3

In [7]: df + (1, 2, 3)
...
ValueError: Unable to coerce to Series, length must be 2: given 3
```

DataFrame arithmetic operations broadcasting changes

DataFrame arithmetic operations when operating with 2-dimensional `np.ndarray` objects now broadcast in the same way as `np.ndarray` broadcast. (GH23000)

```
In [100]: arr = np.arange(6).reshape(3, 2)

In [101]: df = pd.DataFrame(arr)

In [102]: df
Out[102]:
   0  1
0  0  1
1  2  3
```

(continues on next page)

(continued from previous page)

```
2  4  5

[3 rows x 2 columns]
```

Previous behavior:

```
In [5]: df + arr[[0], :]   # 1 row, 2 columns
...
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (1, 2)
In [6]: df + arr[:, [1]]   # 1 column, 3 rows
...
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (3, 1)
```

New behavior:

```
In [103]: df + arr[[0], :]   # 1 row, 2 columns
Out[103]:
   0  1
0  0  2
1  2  4
2  4  6

[3 rows x 2 columns]

In [104]: df + arr[:, [1]]   # 1 column, 3 rows
Out[104]:
   0   1
0  1   2
1  5   6
2  9  10

[3 rows x 2 columns]
```

Series and Index data-dtype incompatibilities

`Series` and `Index` constructors now raise when the data is incompatible with a passed `dtype=` (GH15832)

Previous behavior:

```
In [4]: pd.Series([-1], dtype="uint64")
Out [4]:
0    18446744073709551615
dtype: uint64
```

New behavior:

```
In [4]: pd.Series([-1], dtype="uint64")
Out [4]:
...
OverflowError: Trying to coerce negative values to unsigned integers
```

Concatenation Changes

Calling *pandas.concat()* on a Categorical of ints with NA values now causes them to be processed as objects when concatenating with anything other than another Categorical of ints (GH19214)

```
In [105]: s = pd.Series([0, 1, np.nan])

In [106]: c = pd.Series([0, 1, np.nan], dtype="category")
```

Previous behavior

```
In [3]: pd.concat([s, c])
Out[3]:
0    0.0
1    1.0
2    NaN
0    0.0
1    1.0
2    NaN
dtype: float64
```

New behavior

```
In [107]: pd.concat([s, c])
Out[107]:
0      0
1      1
2    NaN
0      0
1      1
2    NaN
Length: 6, dtype: object
```

Datetimelike API changes

- For *DatetimeIndex* and *TimedeltaIndex* with non-None freq attribute, addition or subtraction of integer-dtyped array or Index will return an object of the same class (GH19959)
- DateOffset objects are now immutable. Attempting to alter one of these will now raise AttributeError (GH21341)
- *PeriodIndex* subtraction of another PeriodIndex will now return an object-dtype *Index* of DateOffset objects instead of raising a TypeError (GH20049)
- *cut()* and *qcut()* now returns a *DatetimeIndex* or *TimedeltaIndex* bins when the input is datetime or timedelta dtype respectively and retbins=True (GH19891)
- *DatetimeIndex.to_period()* and *Timestamp.to_period()* will issue a warning when timezone information will be lost (GH21333)
- PeriodIndex.tz_convert() and PeriodIndex.tz_localize() have been removed (GH21781)