(continued from previous page)

```
                       B
foo
2013-01-01 09:00:00  NaN
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  NaN

[5 rows x 1 columns]
```

Using the time-specification generates variable windows for this sparse data.

```
In [20]: dft.rolling('2s').sum()
Out[20]: 
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

[5 rows x 1 columns]
```

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a DataFrame.

```
In [21]: dft = dft.reset_index()

In [22]: dft
Out[22]: 
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  2.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0

[5 rows x 2 columns]

In [23]: dft.rolling('2s', on='foo').sum()
Out[23]: 
                  foo    B
0 2013-01-01 09:00:00  0.0
1 2013-01-01 09:00:02  1.0
2 2013-01-01 09:00:03  3.0
3 2013-01-01 09:00:05  NaN
4 2013-01-01 09:00:06  4.0

[5 rows x 2 columns]
```

`read_csv` has improved support for duplicate column names

Duplicate column names are now supported in `read_csv()` whether they are in the file or passed in as the `names` parameter (GH7160, GH9424)

```
In [24]: data = '0,1,2\n3,4,5'

In [25]: names = ['a', 'b', 'a']
```

Previous behavior:

```
In [2]: pd.read_csv(StringIO(data), names=names)
Out[2]:
   a  b  a
0  2  1  2
1  5  4  5
```

The first `a` column contained the same data as the second `a` column, when it should have contained the values `[0, 3]`.

New behavior:

```
In [26]: pd.read_csv(StringIO(data), names=names)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-a095135d9435> in <module>
----> 1 pd.read_csv(StringIO(data), names=names)

/pandas-release/pandas/pandas/io/parsers.py in parser_f(filepath_or_buffer, sep,␣
↪delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols,␣
↪dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows,␣
↪skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines,
↪ parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_
↪dates, iterator, chunksize, compression, thousands, decimal, lineterminator,␣
↪quotechar, quoting, doublequote, escapechar, comment, encoding, dialect, error_bad_
↪lines, warn_bad_lines, delim_whitespace, low_memory, memory_map, float_precision)
    674         )
    675 
--> 676         return _read(filepath_or_buffer, kwds)
    677 
    678     parser_f.__name__ = name

/pandas-release/pandas/pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
    443 
    444     # Check for duplicates in names.
--> 445     _validate_names(kwds.get("names", None))
    446 
    447     # Create the parser.

/pandas-release/pandas/pandas/io/parsers.py in _validate_names(names)
    411     if names is not None:
    412         if len(names) != len(set(names)):
--> 413             raise ValueError("Duplicate names are not allowed.")
    414 
    415 

ValueError: Duplicate names are not allowed.
```

read_csv supports parsing Categorical directly

The *read_csv()* function now supports parsing a Categorical column when specified as a dtype (GH10153). Depending on the structure of the data, this can result in a faster parse time and lower memory usage compared to converting to Categorical after parsing. See the io *docs here*.

```
In [27]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [28]: pd.read_csv(StringIO(data))
Out[28]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

[3 rows x 3 columns]

In [29]: pd.read_csv(StringIO(data)).dtypes
Out[29]:
col1    object
col2    object
col3     int64
Length: 3, dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[30]:
col1    category
col2    category
col3    category
Length: 3, dtype: object
```

Individual columns can be parsed as a Categorical using a dict specification

```
In [31]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[31]:
col1    category
col2      object
col3       int64
Length: 3, dtype: object
```

Note: The resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the *to_numeric()* function, or as appropriate, another converter such as *to_datetime()*.

```
In [32]: df = pd.read_csv(StringIO(data), dtype='category')

In [33]: df.dtypes
Out[33]:
col1    category
col2    category
col3    category
Length: 3, dtype: object

In [34]: df['col3']
Out[34]:
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2    3
Name: col3, Length: 3, dtype: category
Categories (3, object): [1, 2, 3]

In [35]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [36]: df['col3']
Out[36]:
0    1
1    2
2    3
Name: col3, Length: 3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Categorical concatenation

- A function `union_categoricals()` has been added for combining categoricals, see *Unioning Categoricals* (GH13361, GH13763, GH13846, GH14173)

```
In [37]: from pandas.api.types import union_categoricals

In [38]: a = pd.Categorical(["b", "c"])

In [39]: b = pd.Categorical(["a", "b"])

In [40]: union_categoricals([a, b])
Out[40]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

- `concat` and `append` now can concat `category` dtypes with different `categories` as `object` dtype (GH13524)

```
In [41]: s1 = pd.Series(['a', 'b'], dtype='category')

In [42]: s2 = pd.Series(['b', 'c'], dtype='category')
```

Previous behavior:

```
In [1]: pd.concat([s1, s2])
ValueError: incompatible categories in categorical concat
```

New behavior:

```
In [43]: pd.concat([s1, s2])
Out[43]:
0    a
1    b
0    b
1    c
Length: 4, dtype: object
```

Semi-month offsets

Pandas has gained new frequency offsets, `SemiMonthEnd` ('SM') and `SemiMonthBegin` ('SMS'). These provide date offsets anchored (by default) to the 15th and end of month, and 15th and 1st of month respectively. (GH1543)

```
In [44]: from pandas.tseries.offsets import SemiMonthEnd, SemiMonthBegin
```

SemiMonthEnd:

```
In [45]: pd.Timestamp('2016-01-01') + SemiMonthEnd()
Out[45]: Timestamp('2016-01-15 00:00:00')

In [46]: pd.date_range('2015-01-01', freq='SM', periods=4)
Out[46]: DatetimeIndex(['2015-01-15', '2015-01-31', '2015-02-15', '2015-02-28'],
↪dtype='datetime64[ns]', freq='SM-15')
```

SemiMonthBegin:

```
In [47]: pd.Timestamp('2016-01-01') + SemiMonthBegin()
Out[47]: Timestamp('2016-01-15 00:00:00')

In [48]: pd.date_range('2015-01-01', freq='SMS', periods=4)
Out[48]: DatetimeIndex(['2015-01-01', '2015-01-15', '2015-02-01', '2015-02-15'],
↪dtype='datetime64[ns]', freq='SMS-15')
```

Using the anchoring suffix, you can also specify the day of month to use instead of the 15th.

```
In [49]: pd.date_range('2015-01-01', freq='SMS-16', periods=4)
Out[49]: DatetimeIndex(['2015-01-01', '2015-01-16', '2015-02-01', '2015-02-16'],
↪dtype='datetime64[ns]', freq='SMS-16')

In [50]: pd.date_range('2015-01-01', freq='SM-14', periods=4)
Out[50]: DatetimeIndex(['2015-01-14', '2015-01-31', '2015-02-14', '2015-02-28'],
↪dtype='datetime64[ns]', freq='SM-14')
```

New Index methods

The following methods and options are added to `Index`, to be more consistent with the `Series` and `DataFrame` API.

`Index` now supports the `.where()` function for same shape indexing (GH13170)

```
In [51]: idx = pd.Index(['a', 'b', 'c'])

In [52]: idx.where([True, False, True])
Out[52]: Index(['a', nan, 'c'], dtype='object')
```

`Index` now supports `.dropna()` to exclude missing values (GH6194)

```
In [53]: idx = pd.Index([1, 2, np.nan, 4])

In [54]: idx.dropna()
Out[54]: Float64Index([1.0, 2.0, 4.0], dtype='float64')
```

For `MultiIndex`, values are dropped if any level is missing by default. Specifying `how='all'` only drops values where all levels are missing.

```
In [55]: midx = pd.MultiIndex.from_arrays([[1, 2, np.nan, 4],
   ....:                                    [1, 2, np.nan, np.nan]])
   ....:

In [56]: midx
Out[56]:
MultiIndex([(1.0, 1.0),
            (2.0, 2.0),
            (nan, nan),
            (4.0, nan)],
           )

In [57]: midx.dropna()
Out[57]:
MultiIndex([(1, 1),
            (2, 2)],
           )

In [58]: midx.dropna(how='all')
Out[58]:
MultiIndex([(1, 1.0),
            (2, 2.0),
            (4, nan)],
           )
```

`Index` now supports `.str.extractall()` which returns a `DataFrame`, see the *docs here* (GH10008, GH13156)

```
In [59]: idx = pd.Index(["a1a2", "b1", "c1"])

In [60]: idx.str.extractall(r"[ab](?P<digit>\d)")
Out[60]:
        digit
  match
0 0         1
  1         2
1 0         1

[3 rows x 1 columns]
```

`Index.astype()` now accepts an optional boolean argument `copy`, which allows optional copying if the requirements on dtype are satisfied (GH13209)

Google BigQuery Enhancements

- The *read_gbq()* method has gained the `dialect` argument to allow users to specify whether to use BigQuery's legacy SQL or BigQuery's standard SQL. See the docs for more details (GH13615).
- The *to_gbq()* method now allows the DataFrame column order to differ from the destination table schema (GH11359).

Fine-grained numpy errstate

Previous versions of pandas would permanently silence numpy's ufunc error handling when `pandas` was imported. Pandas did this in order to silence the warnings that would arise from using numpy ufuncs on missing data, which are usually represented as `NaN` s. Unfortunately, this silenced legitimate warnings arising in non-pandas code in the application. Starting with 0.19.0, pandas will use the `numpy.errstate` context manager to silence these warnings in a more fine-grained manner, only around where these operations are actually used in the pandas code base. (GH13109, GH13145)

After upgrading pandas, you may see *new* `RuntimeWarnings` being issued from your code. These are likely legitimate, and the underlying cause likely existed in the code when using previous versions of pandas that simply silenced the warning. Use numpy.errstate around the source of the `RuntimeWarning` to control how these conditions are handled.

`get_dummies` now returns integer dtypes

The `pd.get_dummies` function now returns dummy-encoded columns as small integers, rather than floats (GH8725). This should provide an improved memory footprint.

Previous behavior:

```
In [1]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes

Out[1]:
a    float64
b    float64
c    float64
dtype: object
```

New behavior:

```
In [61]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[61]:
a    uint8
b    uint8
c    uint8
Length: 3, dtype: object
```

Downcast values to smallest possible dtype in `to_numeric`

`pd.to_numeric()` now accepts a `downcast` parameter, which will downcast the data if possible to smallest specified numerical dtype (GH13352)

```
In [62]: s = ['1', 2, 3]

In [63]: pd.to_numeric(s, downcast='unsigned')
Out[63]: array([1, 2, 3], dtype=uint8)

In [64]: pd.to_numeric(s, downcast='integer')
Out[64]: array([1, 2, 3], dtype=int8)
```

pandas development API

As part of making pandas API more uniform and accessible in the future, we have created a standard sub-package of pandas, `pandas.api` to hold public API's. We are starting by exposing type introspection functions in `pandas.api.types`. More sub-packages and officially sanctioned API's will be published in future versions of pandas (GH13147, GH13634)

The following are now part of this API:

```
In [65]: import pprint

In [66]: from pandas.api import types

In [67]: funcs = [f for f in dir(types) if not f.startswith('_')]

In [68]: pprint.pprint(funcs)
['CategoricalDtype',
 'DatetimeTZDtype',
 'IntervalDtype',
 'PeriodDtype',
 'infer_dtype',
 'is_array_like',
 'is_bool',
 'is_bool_dtype',
 'is_categorical',
 'is_categorical_dtype',
 'is_complex',
 'is_complex_dtype',
 'is_datetime64_any_dtype',
 'is_datetime64_dtype',
 'is_datetime64_ns_dtype',
 'is_datetime64tz_dtype',
 'is_dict_like',
 'is_dtype_equal',
 'is_extension_array_dtype',
 'is_extension_type',
 'is_file_like',
 'is_float',
 'is_float_dtype',
 'is_hashable',
 'is_int64_dtype',
 'is_integer',
 'is_integer_dtype',
 'is_interval',
 'is_interval_dtype',
 'is_iterator',
 'is_list_like',
 'is_named_tuple',
 'is_number',
 'is_numeric_dtype',
 'is_object_dtype',
 'is_period_dtype',
 'is_re',
 'is_re_compilable',
 'is_scalar',
 'is_signed_integer_dtype',
 'is_sparse',
 'is_string_dtype',
```

(continues on next page)

(continued from previous page)

```
'is_timedelta64_dtype',
'is_timedelta64_ns_dtype',
'is_unsigned_integer_dtype',
'pandas_dtype',
'union_categoricals']
```

Note: Calling these functions from the internal module `pandas.core.common` will now show a `DeprecationWarning` (GH13990)

Other enhancements

- `Timestamp` can now accept positional and keyword parameters similar to `datetime.datetime()` (GH10758, GH11630)

```
In [69]: pd.Timestamp(2012, 1, 1)
Out[69]: Timestamp('2012-01-01 00:00:00')

In [70]: pd.Timestamp(year=2012, month=1, day=1, hour=8, minute=30)
Out[70]: Timestamp('2012-01-01 08:30:00')
```

- The `.resample()` function now accepts a `on=` or `level=` parameter for resampling on a datetimelike column or `MultiIndex` level (GH13500)

```
In [71]: df = pd.DataFrame({'date': pd.date_range('2015-01-01', freq='W',␣
↪periods=5),
   ....:                    'a': np.arange(5)},
   ....:                   index=pd.MultiIndex.from_arrays([[1, 2, 3, 4, 5],
   ....:                                                    pd.date_range('2015-
↪01-01',
   ....:                                                                  freq='W
↪',
   ....:                                                                  ␣
↪periods=5)
   ....:                                                    ], names=['v', 'd']))
   ....:

In [72]: df
Out[72]:
                   date  a
v d
1 2015-01-04 2015-01-04  0
2 2015-01-11 2015-01-11  1
3 2015-01-18 2015-01-18  2
4 2015-01-25 2015-01-25  3
5 2015-02-01 2015-02-01  4

[5 rows x 2 columns]

In [73]: df.resample('M', on='date').sum()
Out[73]:
            a
date
2015-01-31  6
```

(continues on next page)

(continued from previous page)

```
2015-02-28  4

[2 rows x 1 columns]

In [74]: df.resample('M', level='d').sum()
Out[74]:
            a
d
2015-01-31  6
2015-02-28  4

[2 rows x 1 columns]
```

- The `.get_credentials()` method of `GbqConnector` can now first try to fetch the application default credentials. See the docs for more details (GH13577).
- The `.tz_localize()` method of `DatetimeIndex` and `Timestamp` has gained the `errors` keyword, so you can potentially coerce nonexistent timestamps to `NaT`. The default behavior remains to raising a `NonExistentTimeError` (GH13057)
- `.to_hdf/read_hdf()` now accept path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path (GH11773)
- The `pd.read_csv()` with `engine='python'` has gained support for the `decimal` (GH12933), `na_filter` (GH13321) and the `memory_map` option (GH13381).
- Consistent with the Python API, `pd.read_csv()` will now interpret `+inf` as positive infinity (GH13274)
- The `pd.read_html()` has gained support for the `na_values`, `converters`, `keep_default_na` options (GH13461)
- `Categorical.astype()` now accepts an optional boolean argument `copy`, effective when dtype is categorical (GH13209)
- `DataFrame` has gained the `.asof()` method to return the last non-NaN values according to the selected subset (GH13358)
- The `DataFrame` constructor will now respect key ordering if a list of `OrderedDict` objects are passed in (GH13304)
- `pd.read_html()` has gained support for the `decimal` option (GH12907)
- `Series` has gained the properties `.is_monotonic`, `.is_monotonic_increasing`, `.is_monotonic_decreasing`, similar to `Index` (GH13336)
- `DataFrame.to_sql()` now allows a single value as the SQL type for all columns (GH11886).
- `Series.append` now supports the `ignore_index` option (GH13677)
- `.to_stata()` and `StataWriter` can now write variable labels to Stata dta files using a dictionary to make column names to labels (GH13535, GH13536)
- `.to_stata()` and `StataWriter` will automatically convert `datetime64[ns]` columns to Stata format `%tc`, rather than raising a `ValueError` (GH12259)
- `read_stata()` and `StataReader` raise with a more explicit error message when reading Stata files with repeated value labels when `convert_categoricals=True` (GH13923)
- `DataFrame.style` will now render sparsified MultiIndexes (GH11655)
- `DataFrame.style` will now show column level names (e.g. `DataFrame.columns.names`) (GH13775)

- `DataFrame` has gained support to re-order the columns based on the values in a row using `df.sort_values(by='...', axis=1)` (GH10806)

```
In [75]: df = pd.DataFrame({'A': [2, 7], 'B': [3, 5], 'C': [4, 8]},
   ....:                   index=['row1', 'row2'])
   ....:

In [76]: df
Out[76]:
      A  B  C
row1  2  3  4
row2  7  5  8

[2 rows x 3 columns]

In [77]: df.sort_values(by='row2', axis=1)
Out[77]:
      B  A  C
row1  3  2  4
row2  5  7  8

[2 rows x 3 columns]
```

- Added documentation to *I/O* regarding the perils of reading in columns with mixed dtypes and how to handle it (GH13746)
- *`to_html()`* now has a `border` argument to control the value in the opening `<table>` tag. The default is the value of the `html.border` option, which defaults to 1. This also affects the notebook HTML repr, but since Jupyter's CSS includes a border-width attribute, the visual effect is the same. (GH11563).
- Raise `ImportError` in the sql functions when `sqlalchemy` is not installed and a connection string is used (GH11920).
- Compatibility with matplotlib 2.0. Older versions of pandas should also work with matplotlib 2.0 (GH13333)
- `Timestamp`, `Period`, `DatetimeIndex`, `PeriodIndex` and `.dt` accessor have gained a `.is_leap_year` property to check whether the date belongs to a leap year. (GH13727)
- `astype()` will now accept a dict of column name to data types mapping as the `dtype` argument. (GH12086)
- The `pd.read_json` and `DataFrame.to_json` has gained support for reading and writing json lines with `lines` option see *Line delimited json* (GH9180)
- *`read_excel()`* now supports the true_values and false_values keyword arguments (GH13347)
- `groupby()` will now accept a scalar and a single-element list for specifying `level` on a non-`MultiIndex` grouper. (GH13907)
- Non-convertible dates in an excel date column will be returned without conversion and the column will be `object` dtype, rather than raising an exception (GH10001).
- `pd.Timedelta(None)` is now accepted and will return `NaT`, mirroring `pd.Timestamp` (GH13687)
- `pd.read_stata()` can now handle some format 111 files, which are produced by SAS when generating Stata dta files (GH11526)
- `Series` and `Index` now support `divmod` which will return a tuple of series or indices. This behaves like a standard binary operator with regards to broadcasting rules (GH14208).

API changes

`Series.tolist()` will now return Python types

`Series.tolist()` will now return Python types in the output, mimicking NumPy `.tolist()` behavior (GH10904)

```
In [78]: s = pd.Series([1, 2, 3])
```

Previous behavior:

```
In [7]: type(s.tolist()[0])
Out[7]:
 <class 'numpy.int64'>
```

New behavior:

```
In [79]: type(s.tolist()[0])
Out[79]: int
```

`Series` operators for different indexes

Following `Series` operators have been changed to make all operators consistent, including `DataFrame` (GH1134, GH4581, GH13538)

- `Series` comparison operators now raise `ValueError` when `index` are different.
- `Series` logical operators align both `index` of left and right hand side.

> **Warning:** Until 0.18.1, comparing `Series` with the same length, would succeed even if the `.index` are different (the result ignores `.index`). As of 0.19.0, this will raises `ValueError` to be more strict. This section also describes how to keep previous behavior or align different indexes, using the flexible comparison methods like `.eq`.

As a result, `Series` and `DataFrame` operators behave as below:

Arithmetic operators

Arithmetic operators align both `index` (no changes).

```
In [80]: s1 = pd.Series([1, 2, 3], index=list('ABC'))

In [81]: s2 = pd.Series([2, 2, 2], index=list('ABD'))

In [82]: s1 + s2
Out[82]:
A    3.0
B    4.0
C    NaN
D    NaN
Length: 4, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [83]: df1 = pd.DataFrame([1, 2, 3], index=list('ABC'))

In [84]: df2 = pd.DataFrame([2, 2, 2], index=list('ABD'))

In [85]: df1 + df2
Out[85]:
     0
A  3.0
B  4.0
C  NaN
D  NaN

[4 rows x 1 columns]
```

Comparison operators

Comparison operators raise `ValueError` when `.index` are different.

Previous behavior (`Series`):

`Series` compared values ignoring the `.index` as long as both had the same length:

```
In [1]: s1 == s2
Out[1]:
A    False
B     True
C    False
dtype: bool
```

New behavior (`Series`):

```
In [2]: s1 == s2
Out[2]:
ValueError: Can only compare identically-labeled Series objects
```

Note: To achieve the same result as previous versions (compare values based on locations ignoring `.index`), compare both `.values`.

```
In [86]: s1.values == s2.values
Out[86]: array([False,  True, False])
```

If you want to compare `Series` aligning its `.index`, see flexible comparison methods section below:

```
In [87]: s1.eq(s2)
Out[87]:
A    False
B     True
C    False
D    False
Length: 4, dtype: bool
```

Current behavior (`DataFrame`, no change):

```
In [3]: df1 == df2
Out[3]:
ValueError: Can only compare identically-labeled DataFrame objects
```

Logical operators

Logical operators align both `.index` of left and right hand side.

Previous behavior (`Series`), only left hand side `index` was kept:

```
In [4]: s1 = pd.Series([True, False, True], index=list('ABC'))
In [5]: s2 = pd.Series([True, True, True], index=list('ABD'))
In [6]: s1 & s2
Out[6]:
A     True
B    False
C    False
dtype: bool
```

New behavior (`Series`):

```
In [88]: s1 = pd.Series([True, False, True], index=list('ABC'))

In [89]: s2 = pd.Series([True, True, True], index=list('ABD'))

In [90]: s1 & s2
Out[90]:
A     True
B    False
C    False
D    False
Length: 4, dtype: bool
```

Note: `Series` logical operators fill a `NaN` result with `False`.

Note: To achieve the same result as previous versions (compare values based on only left hand side index), you can use `reindex_like`:

```
In [91]: s1 & s2.reindex_like(s1)
Out[91]:
A     True
B    False
C    False
Length: 3, dtype: bool
```

Current behavior (`DataFrame`, no change):

```
In [92]: df1 = pd.DataFrame([True, False, True], index=list('ABC'))

In [93]: df2 = pd.DataFrame([True, True, True], index=list('ABD'))

In [94]: df1 & df2
```

(continues on next page)

(continued from previous page)

```
Out[94]:
       0
A   True
B  False
C  False
D  False

[4 rows x 1 columns]
```

Flexible comparison methods

Series flexible comparison methods like eq, ne, le, lt, ge and gt now align both index. Use these operators if you want to compare two Series which has the different index.

```
In [95]: s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [96]: s2 = pd.Series([2, 2, 2], index=['b', 'c', 'd'])

In [97]: s1.eq(s2)
Out[97]:
a    False
b     True
c    False
d    False
Length: 4, dtype: bool

In [98]: s1.ge(s2)
Out[98]:
a    False
b     True
c     True
d    False
Length: 4, dtype: bool
```

Previously, this worked the same as comparison operators (see above).

Series type promotion on assignment

A Series will now correctly promote its dtype for assignment with incompat values to the current dtype (GH13234)

```
In [99]: s = pd.Series()
```

Previous behavior:

```
In [2]: s["a"] = pd.Timestamp("2016-01-01")

In [3]: s["b"] = 3.0
TypeError: invalid type promotion
```

New behavior:

```
In [100]: s["a"] = pd.Timestamp("2016-01-01")
```

(continues on next page)

(continued from previous page)

```
In [101]: s["b"] = 3.0

In [102]: s
Out[102]:
a    2016-01-01 00:00:00
b                      3
Length: 2, dtype: object

In [103]: s.dtype
Out[103]: dtype('O')
```

`.to_datetime()` changes

Previously if `.to_datetime()` encountered mixed integers/floats and strings, but no datetimes with `errors='coerce'` it would convert all to `NaT`.

Previous behavior:

```
In [2]: pd.to_datetime([1, 'foo'], errors='coerce')
Out[2]: DatetimeIndex(['NaT', 'NaT'], dtype='datetime64[ns]', freq=None)
```

Current behavior:

This will now convert integers/floats with the default unit of `ns`.

```
In [104]: pd.to_datetime([1, 'foo'], errors='coerce')
Out[104]: DatetimeIndex(['1970-01-01 00:00:00.000000001', 'NaT'], dtype=
↪'datetime64[ns]', freq=None)
```

Bug fixes related to `.to_datetime()`:

- Bug in `pd.to_datetime()` when passing integers or floats, and no `unit` and `errors='coerce'` (GH13180).
- Bug in `pd.to_datetime()` when passing invalid data types (e.g. bool); will now respect the `errors` keyword (GH13176)
- Bug in `pd.to_datetime()` which overflowed on `int8`, and `int16` dtypes (GH13451)
- Bug in `pd.to_datetime()` raise `AttributeError` with `NaN` and the other string is not valid when `errors='ignore'` (GH12424)
- Bug in `pd.to_datetime()` did not cast floats correctly when `unit` was specified, resulting in truncated datetime (GH13834)

Merging changes

Merging will now preserve the dtype of the join keys (GH8596)

```
In [105]: df1 = pd.DataFrame({'key': [1], 'v1': [10]})

In [106]: df1
Out[106]:
   key  v1
0    1  10
```

(continues on next page)

(continued from previous page)

```
[1 rows x 2 columns]

In [107]: df2 = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [108]: df2
Out[108]:
   key  v1
0    1  20
1    2  30

[2 rows x 2 columns]
```

Previous behavior:

```
In [5]: pd.merge(df1, df2, how='outer')
Out[5]:
   key    v1
0  1.0  10.0
1  1.0  20.0
2  2.0  30.0

In [6]: pd.merge(df1, df2, how='outer').dtypes
Out[6]:
key     float64
v1      float64
dtype: object
```

New behavior:

We are able to preserve the join keys

```
In [109]: pd.merge(df1, df2, how='outer')
Out[109]:
   key  v1
0    1  10
1    1  20
2    2  30

[3 rows x 2 columns]

In [110]: pd.merge(df1, df2, how='outer').dtypes
Out[110]:
key     int64
v1      int64
Length: 2, dtype: object
```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast, which is unchanged from previous.

```
In [111]: pd.merge(df1, df2, how='outer', on='key')
Out[111]:
   key  v1_x  v1_y
0    1  10.0    20
1    2   NaN    30

[2 rows x 3 columns]
```

(continues on next page)

(continued from previous page)

```
In [112]: pd.merge(df1, df2, how='outer', on='key').dtypes
Out[112]:
key        int64
v1_x     float64
v1_y       int64
Length: 3, dtype: object
```

.describe() changes

Percentile identifiers in the index of a `.describe()` output will now be rounded to the least precision that keeps them distinct (GH13104)

```
In [113]: s = pd.Series([0, 1, 2, 3, 4])

In [114]: df = pd.DataFrame([0, 1, 2, 3, 4])
```

Previous behavior:

The percentiles were rounded to at most one decimal place, which could raise `ValueError` for a data frame if the percentiles were duplicated.

```
In [3]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[3]:
count     5.000000
mean      2.000000
std       1.581139
min       0.000000
0.0%      0.000400
0.1%      0.002000
0.1%      0.004000
50%       2.000000
99.9%     3.996000
100.0%    3.998000
100.0%    3.999600
max       4.000000
dtype: float64

In [4]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[4]:
...
ValueError: cannot reindex from a duplicate axis
```

New behavior:

```
In [115]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[115]:
count     5.000000
mean      2.000000
std       1.581139
min       0.000000
0.01%     0.000400
0.05%     0.002000
0.1%      0.004000
50%       2.000000
```

(continues on next page)

(continued from previous page)

```
99.9%      3.996000
99.95%     3.998000
99.99%     3.999600
max        4.000000
Length: 12, dtype: float64

In [116]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
Out[116]:
               0
count   5.000000
mean    2.000000
std     1.581139
min     0.000000
0.01%   0.000400
0.05%   0.002000
0.1%    0.004000
50%     2.000000
99.9%   3.996000
99.95%  3.998000
99.99%  3.999600
max     4.000000

[12 rows x 1 columns]
```

Furthermore:

- Passing duplicated `percentiles` will now raise a `ValueError`.
- Bug in `.describe()` on a DataFrame with a mixed-dtype column index, which would previously raise a `TypeError` (GH13288)

`Period` changes

`PeriodIndex` now has `period` dtype

`PeriodIndex` now has its own `period` dtype. The `period` dtype is a pandas extension dtype like `category` or the *timezone aware dtype* (`datetime64[ns, tz]`) (GH13941). As a consequence of this change, `PeriodIndex` no longer has an integer dtype:

Previous behavior:

```
In [1]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')

In [2]: pi
Out[2]: PeriodIndex(['2016-08-01'], dtype='int64', freq='D')

In [3]: pd.api.types.is_integer_dtype(pi)
Out[3]: True

In [4]: pi.dtype
Out[4]: dtype('int64')
```

New behavior:

```
In [117]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')

In [118]: pi
Out[118]: PeriodIndex(['2016-08-01'], dtype='period[D]', freq='D')

In [119]: pd.api.types.is_integer_dtype(pi)
Out[119]: False

In [120]: pd.api.types.is_period_dtype(pi)
Out[120]: True

In [121]: pi.dtype
Out[121]: period[D]

In [122]: type(pi.dtype)
Out[122]: pandas.core.dtypes.dtypes.PeriodDtype
```

`Period('NaT')` now returns `pd.NaT`

Previously, `Period` has its own `Period('NaT')` representation different from `pd.NaT`. Now `Period('NaT')` has been changed to return `pd.NaT`. (GH12759, GH13582)

Previous behavior:

```
In [5]: pd.Period('NaT', freq='D')
Out[5]: Period('NaT', 'D')
```

New behavior:

These result in `pd.NaT` without providing `freq` option.

```
In [123]: pd.Period('NaT')
Out[123]: NaT

In [124]: pd.Period(None)
Out[124]: NaT
```

To be compatible with `Period` addition and subtraction, `pd.NaT` now supports addition and subtraction with `int`. Previously it raised `ValueError`.

Previous behavior:

```
In [5]: pd.NaT + 1
...
ValueError: Cannot add integral value to Timestamp without freq.
```

New behavior:

```
In [125]: pd.NaT + 1
Out[125]: NaT

In [126]: pd.NaT - 1
Out[126]: NaT
```

`PeriodIndex.values` now returns array of `Period` object

`.values` is changed to return an array of `Period` objects, rather than an array of integers (GH13988).

Previous behavior:

```
In [6]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')
In [7]: pi.values
Out[7]: array([492, 493])
```

New behavior:

```
In [127]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')

In [128]: pi.values
Out[128]: array([Period('2011-01', 'M'), Period('2011-02', 'M')], dtype=object)
```

Index + / - no longer used for set operations

Addition and subtraction of the base Index type and of DatetimeIndex (not the numeric index types) previously performed set operations (set union and difference). This behavior was already deprecated since 0.15.0 (in favor using the specific `.union()` and `.difference()` methods), and is now disabled. When possible, + and − are now used for element-wise operations, for example for concatenating strings or subtracting datetimes (GH8227, GH14127).

Previous behavior:

```
In [1]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
FutureWarning: using '+' to provide set union with Indexes is deprecated, use '|' or .
↪union()
Out[1]: Index(['a', 'b', 'c'], dtype='object')
```

New behavior: the same operation will now perform element-wise addition:

```
In [129]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
Out[129]: Index(['aa', 'bc'], dtype='object')
```

Note that numeric Index objects already performed element-wise operations. For example, the behavior of adding two integer Indexes is unchanged. The base `Index` is now made consistent with this behavior.

```
In [130]: pd.Index([1, 2, 3]) + pd.Index([2, 3, 4])
Out[130]: Int64Index([3, 5, 7], dtype='int64')
```

Further, because of this change, it is now possible to subtract two DatetimeIndex objects resulting in a TimedeltaIndex:

Previous behavior:

```
In [1]: (pd.DatetimeIndex(['2016-01-01', '2016-01-02'])
   ...:  - pd.DatetimeIndex(['2016-01-02', '2016-01-03']))
FutureWarning: using '-' to provide set differences with datetimelike Indexes is␣
↪deprecated, use .difference()
Out[1]: DatetimeIndex(['2016-01-01'], dtype='datetime64[ns]', freq=None)
```

New behavior:

```
In [131]: (pd.DatetimeIndex(['2016-01-01', '2016-01-02'])
   .....:  - pd.DatetimeIndex(['2016-01-02', '2016-01-03']))
```

(continues on next page)

(continued from previous page)

```
.....:
Out[131]: TimedeltaIndex(['-1 days', '-1 days'], dtype='timedelta64[ns]', freq=None)
```

Index.difference and .symmetric_difference changes

Index.difference and Index.symmetric_difference will now, more consistently, treat NaN values as any other values. (GH13514)

```
In [132]: idx1 = pd.Index([1, 2, 3, np.nan])

In [133]: idx2 = pd.Index([0, 1, np.nan])
```

Previous behavior:

```
In [3]: idx1.difference(idx2)
Out[3]: Float64Index([nan, 2.0, 3.0], dtype='float64')

In [4]: idx1.symmetric_difference(idx2)
Out[4]: Float64Index([0.0, nan, 2.0, 3.0], dtype='float64')
```

New behavior:

```
In [134]: idx1.difference(idx2)
Out[134]: Float64Index([2.0, 3.0], dtype='float64')

In [135]: idx1.symmetric_difference(idx2)
Out[135]: Float64Index([0.0, 2.0, 3.0], dtype='float64')
```

Index.unique consistently returns Index

Index.unique() now returns unique values as an Index of the appropriate dtype. (GH13395). Previously, most Index classes returned np.ndarray, and DatetimeIndex, TimedeltaIndex and PeriodIndex returned Index to keep metadata like timezone.

Previous behavior:

```
In [1]: pd.Index([1, 2, 3]).unique()
Out[1]: array([1, 2, 3])

In [2]: pd.DatetimeIndex(['2011-01-01', '2011-01-02',
   ...:                   '2011-01-03'], tz='Asia/Tokyo').unique()
Out[2]:
DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00',
               '2011-01-03 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq=None)
```

New behavior:

```
In [136]: pd.Index([1, 2, 3]).unique()
Out[136]: Int64Index([1, 2, 3], dtype='int64')

In [137]: pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'],
   .....:                  tz='Asia/Tokyo').unique()
```

(continues on next page)

(continued from previous page)

```
   .....:
Out[137]:
DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00',
               '2011-01-03 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq=None)
```

MultiIndex constructors, groupby and set_index preserve categorical dtypes

MultiIndex.from_arrays and MultiIndex.from_product will now preserve categorical dtype in MultiIndex levels (GH13743, GH13854).

```
In [138]: cat = pd.Categorical(['a', 'b'], categories=list("bac"))

In [139]: lvl1 = ['foo', 'bar']

In [140]: midx = pd.MultiIndex.from_arrays([cat, lvl1])

In [141]: midx
Out[141]:
MultiIndex([('a', 'foo'),
            ('b', 'bar')],
           )
```

Previous behavior:

```
In [4]: midx.levels[0]
Out[4]: Index(['b', 'a', 'c'], dtype='object')

In [5]: midx.get_level_values[0]
Out[5]: Index(['a', 'b'], dtype='object')
```

New behavior: the single level is now a CategoricalIndex:

```
In [142]: midx.levels[0]
Out[142]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
↪ dtype='category')

In [143]: midx.get_level_values(0)
Out[143]: CategoricalIndex(['a', 'b'], categories=['b', 'a', 'c'], ordered=False,␣
↪dtype='category')
```

An analogous change has been made to MultiIndex.from_product. As a consequence, groupby and set_index also preserve categorical dtypes in indexes

```
In [144]: df = pd.DataFrame({'A': [0, 1], 'B': [10, 11], 'C': cat})

In [145]: df_grouped = df.groupby(by=['A', 'C']).first()

In [146]: df_set_idx = df.set_index(['A', 'C'])
```

Previous behavior:

```
In [11]: df_grouped.index.levels[1]
Out[11]: Index(['b', 'a', 'c'], dtype='object', name='C')
```

(continues on next page)

(continued from previous page)

```
In [12]: df_grouped.reset_index().dtypes
Out[12]:
A      int64
C     object
B    float64
dtype: object

In [13]: df_set_idx.index.levels[1]
Out[13]: Index(['b', 'a', 'c'], dtype='object', name='C')
In [14]: df_set_idx.reset_index().dtypes
Out[14]:
A      int64
C     object
B      int64
dtype: object
```

New behavior:

```
In [147]: df_grouped.index.levels[1]
Out[147]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
↪ name='C', dtype='category')

In [148]: df_grouped.reset_index().dtypes
Out[148]:
A       int64
C    category
B     float64
Length: 3, dtype: object

In [149]: df_set_idx.index.levels[1]
Out[149]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
↪ name='C', dtype='category')

In [150]: df_set_idx.reset_index().dtypes
Out[150]:
A       int64
C    category
B       int64
Length: 3, dtype: object
```

`read_csv` will progressively enumerate chunks

When *read_csv()* is called with `chunksize=n` and without specifying an index, each chunk used to have an independently generated index from `0` to `n-1`. They are now given instead a progressive index, starting from `0` for the first chunk, from `n` for the second, and so on, so that, when concatenated, they are identical to the result of calling *read_csv()* without the `chunksize=` argument (GH12185).

```
In [151]: data = 'A,B\n0,1\n2,3\n4,5\n6,7'
```

Previous behavior:

```
In [2]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[2]:
   A  B
```

(continues on next page)

(continued from previous page)

```
0  0  1
1  2  3
0  4  5
1  6  7
```

New behavior:

```
In [152]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
Out[152]:
   A  B
0  0  1
1  2  3
2  4  5
3  6  7

[4 rows x 2 columns]
```

Sparse Changes

These changes allow pandas to handle sparse data with more dtypes, and for work to make a smoother experience with data handling.

`int64` and `bool` support enhancements

Sparse data structures now gained enhanced support of `int64` and `bool` `dtype` (GH667, GH13849).

Previously, sparse data were `float64` dtype by default, even if all inputs were of `int` or `bool` dtype. You had to specify `dtype` explicitly to create sparse data with `int64` dtype. Also, `fill_value` had to be specified explicitly because the default was `np.nan` which doesn't appear in `int64` or `bool` data.

```
In [1]: pd.SparseArray([1, 2, 0, 0])
Out[1]:
[1.0, 2.0, 0.0, 0.0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

# specifying int64 dtype, but all values are stored in sp_values because
# fill_value default is np.nan
In [2]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[2]:
[1, 2, 0, 0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

In [3]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64, fill_value=0)
Out[3]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)
```