pandas: powerful Python data analysis toolkit

Release 0.24.0

Wes McKinney& PyData Development Team

Jan 25, 2019

CONTENTS

Date: Jan 25, 2019 **Version**: 0.24.0

Download documentation: PDF Version | Zipped HTML

Useful links: Binary Installers | Source Repository | Issues & Ideas | Q&A Support | Mailing List

`pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

See the *Package overview* for more detail about what's in the library.

CHAPTER

ONE

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 *Plan for dropping Python 2.7* for more.

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.

1.1 Enhancements

1.1.1 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]:
```

(continues on next page)

(continued from previous page)

```
0      1
1      2
2    NaN
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    NaN
Length: 3, dtype: Int64

# comparison
In [4]: s == 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:
0     True
1    False
2    False
Length: 3, dtype: bool

# indexing
In [5]: s.iloc[1:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
1      2
2    NaN
Length: 2, dtype: Int64

# operate with other dtypes
In [6]: s + s.iloc[1:3].astype('Int8')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1      4
2    NaN
Length: 3, dtype: Int64

# coerce when needed
In [7]: s + 0.01
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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
```

(continues on next page)

(continued from previous page)

```
1    2  1  a
2  NaN  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
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.

1.1.2 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
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<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)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪139878025578576

In [20]: id(idx.values)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪139878052163872
```

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()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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]
```

(continues on next page)

(continued from previous page)

```
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.

1.1.3 `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, NaN]
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]:
<PandasArray>
[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]:
<PandasArray>
[1.0, 2.0, nan]
Length: 3, dtype: float64
```

1.1.4 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
↪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
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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]],
              closed='right',
              dtype='interval[int64]')

In [37]: pser.array
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<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.

1.1.5 Joining with two multi-indexes

DataFrame.merge() and *DataFrame.join()* can now be used to join multi-indexed `Dataframe` instances on the overlaping 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'])
   ....:
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]
```

1.1.6 `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]]
```

1.1.7 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 0x7f37dc4beef0>
```

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

1.1.8 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([i for i in 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'})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
           N
AB New 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]
```

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

1.1.9 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)

1.2 Backwards incompatible API changes

Pandas 0.24.0 includes a number of API breaking changes.

1.2.1 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).

1.2.2 *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:
```

(continues on next page)

(continued from previous page)

```
   ...:     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'
```

1.2.3 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)
```

(continues on next page)

(continued from previous page)

```
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.

1.2.4 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)
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]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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)
```

1.2.5 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 [64]: p = pd.Period('2017-01-01', 'D')

In [65]: pi = pd.PeriodIndex([p])

In [66]: pd.Series(pi).dt.end_time[0]
Out[66]: Timestamp('2017-01-01 23:59:59.999999999')

In [67]: p.end_time
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[67]: Timestamp('2017-01-01␣
↪23:59:59.999999999')
```

1.2.6 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 [68]: 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 [69]: ser.unique()
Out[69]:
<DatetimeArray>
['2000-01-01 00:00:00+00:00']
Length: 1, dtype: datetime64[ns, UTC]
```

1.2.7 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 [70]: s = pd.Series([0, 0, 1, 1, 1], dtype='Sparse[int]')

In [71]: s.sparse.density
Out[71]: 0.6
```

1.2.8 `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 [72]: type(pd.get_dummies(df, sparse=True))
Out[72]: pandas.core.frame.DataFrame

In [73]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[73]: pandas.core.sparse.frame.SparseDataFrame
```

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.

1.2.9 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 [74]: df = pd.DataFrame({'a': [1, 2], 'b': [0.5, 0.75]}, index=['A', 'A'])

In [75]: df
Out[75]:
   a     b
A  1  0.50
A  2  0.75

[2 rows x 2 columns]

In [76]: df.to_dict(orient='index')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\---------------------
↪---------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-76-f5309a7c6adb> in <module>
----> 1 df.to_dict(orient='index')

/pandas/pandas/core/frame.py in to_dict(self, orient, into)
   1304             if not self.index.is_unique:
   1305                 raise ValueError(
-> 1306                     "DataFrame index must be unique for orient='index'."
   1307                 )
   1308             return into_c((t[0], dict(zip(self.columns, t[1:])))

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

1.2.10 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')

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

In [78]: tic = pd.offsets.Hour(n=2)
```

(continues on next page)

(continued from previous page)

```
In [79]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[79]: True
```

1.2.11 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 [80]: june = pd.Period('June 2018')

In [81]: april = pd.Period('April 2018')

In [82]: june - april
Out[82]: <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 [83]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [84]: pi - pi[0]
Out[84]: Index([<0 * MonthEnds>, <MonthEnd>, <2 * MonthEnds>], dtype='object')
```

1.2.12 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 [85]: df = pd.DataFrame([pd.Timedelta(days=1)])

In [86]: df
Out[86]:
```

(continues on next page)

(continued from previous page)

```
       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'
```

1.2.13 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 [87]: arr = np.arange(6).reshape(3, 2)

In [88]: df = pd.DataFrame(arr)

In [89]: df
Out[89]:
   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
```

(continues on next page)

(continued from previous page)

```
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 [90]: df == arr[[0], :]
Out[90]:
       0      1
0   True   True
1  False  False
2  False  False

[3 rows x 2 columns]

In [91]: df + arr[[0], :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9
↪
   0  1
0  0  2
1  2  4
2  4  6

[3 rows x 2 columns]
```

```
# Comparison operations and arithmetic operations both broadcast.
In [92]: df == (1, 2)
Out[92]:
```

(continues on next page)

(continued from previous page)

```
       0      1
0  False  False
1  False  False
2  False  False

[3 rows x 2 columns]

In [93]: df + (1, 2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9
↪
   0  1
0  1  3
1  3  5
2  5  7

[3 rows x 2 columns]
```

```
# Comparison operations and arithmetic opeartions 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
```

1.2.14 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 [94]: arr = np.arange(6).reshape(3, 2)

In [95]: df = pd.DataFrame(arr)

In [96]: df
Out[96]:
   0  1
0  0  1
1  2  3
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 [97]: df + arr[[0], :]   # 1 row, 2 columns
Out[97]:
   0  1
0  0  2
1  2  4
2  4  6

[3 rows x 2 columns]

In [98]: df + arr[:, [1]]   # 1 column, 3 rows
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[98]:
   0   1
0  1   2
1  5   6
2  9  10

[3 rows x 2 columns]
```

1.2.15 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
```

1.2.16 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 [99]: s = pd.Series([0, 1, np.nan])

In [100]: 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
```

(continues on next page)

(continued from previous page)

```
2    NaN
dtype: float64
```

New Behavior

```
In [101]: pd.concat([s, c])
Out[101]:
0      0
1      1
2    NaN
0      0
1      1
2    NaN
Length: 6, dtype: object
```

1.2.17 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)

1.2.18 Other API Changes

- A newly constructed empty *DataFrame* with integer as the dtype will now only be cast to float64 if index is specified (GH22858)
- *Series.str.cat()* will now raise if others is a set (GH23009)
- Passing scalar values to *DatetimeIndex* or *TimedeltaIndex* will now raise TypeError instead of ValueError (GH23539)
- max_rows and max_cols parameters removed from HTMLFormatter since truncation is handled by DataFrameFormatter (GH23818)
- *read_csv()* will now raise a ValueError if a column with missing values is declared as having dtype bool (GH20591)
- The column order of the resultant *DataFrame* from *MultiIndex.to_frame()* is now guaranteed to match the *MultiIndex.names* order. (GH22420)
- Incorrectly passing a *DatetimeIndex* to *MultiIndex.from_tuples()*, rather than a sequence of tuples, now raises a TypeError rather than a ValueError (GH24024)
- pd.offsets.generate_range() argument time_rule has been removed; use offset instead (GH24157)

- In 0.23.x, pandas would raise a `ValueError` on a merge of a numeric column (e.g. `int` dtyped column) and an `object` dtyped column (GH9780). We have re-enabled the ability to merge `object` and other dtypes; pandas will still raise on a merge between a numeric and an `object` dtyped column that is composed only of strings (GH21681)
- Accessing a level of a `MultiIndex` with a duplicate name (e.g. in *get_level_values()*) now raises a `ValueError` instead of a `KeyError` (GH21678).
- Invalid construction of `IntervalDtype` will now always raise a `TypeError` rather than a `ValueError` if the subdtype is invalid (GH21185)
- Trying to reindex a `DataFrame` with a non unique `MultiIndex` now raises a `ValueError` instead of an `Exception` (GH21770)
- *Index* subtraction will attempt to operate element-wise instead of raising `TypeError` (GH19369)
- *pandas.io.formats.style.Styler* supports a `number-format` property when using *to_excel()* (GH22015)
- *DataFrame.corr()* and *Series.corr()* now raise a `ValueError` along with a helpful error message instead of a `KeyError` when supplied with an invalid method (GH22298)
- `shift()` will now always return a copy, instead of the previous behaviour of returning self when shifting by 0 (GH22397)
- *DataFrame.set_index()* now gives a better (and less frequent) KeyError, raises a `ValueError` for incorrect types, and will not fail on duplicate column names with `drop=True`. (GH22484)
- Slicing a single row of a DataFrame with multiple ExtensionArrays of the same type now preserves the dtype, rather than coercing to object (GH22784)
- `DateOffset` attribute *_cacheable* and method *_should_cache* have been removed (GH23118)
- *Series.searchsorted()*, when supplied a scalar value to search for, now returns a scalar instead of an array (GH23801).
- `Categorical.searchsorted()`, when supplied a scalar value to search for, now returns a scalar instead of an array (GH23466).
- `Categorical.searchsorted()` now raises a `KeyError` rather that a `ValueError`, if a searched for key is not found in its categories (GH23466).
- *Index.hasnans()* and *Series.hasnans()* now always return a python boolean. Previously, a python or a numpy boolean could be returned, depending on circumstances (GH23294).
- The order of the arguments of *DataFrame.to_html()* and *DataFrame.to_string()* is rearranged to be consistent with each other. (GH23614)
- `CategoricalIndex.reindex()` now raises a `ValueError` if the target index is non-unique and not equal to the current index. It previously only raised if the target index was not of a categorical dtype (GH23963).
- *Series.to_list()* and *Index.to_list()* are now aliases of `Series.tolist` respectively `Index.tolist` (GH8826)
- The result of `SparseSeries.unstack` is now a *DataFrame* with sparse values, rather than a `SparseDataFrame` (GH24372).
- *DatetimeIndex* and *TimedeltaIndex* no longer ignore the dtype precision. Passing a non-nanosecond resolution dtype will raise a `ValueError` (GH24753)

1.3 Extension Type Changes

Equality and Hashability

Pandas now requires that extension dtypes be hashable (i.e. the respective `ExtensionDtype` objects; hashability is not a requirement for the values of the corresponding `ExtensionArray`). The base class implements a default `__eq__` and `__hash__`. If you have a parametrized dtype, you should update the `ExtensionDtype._metadata` tuple to match the signature of your `__init__` method. See *`pandas.api.extensions.ExtensionDtype`* for more (GH22476).

New and changed methods

- `dropna()` has been added (GH21185)
- `repeat()` has been added (GH24349)
- The `ExtensionArray` constructor, `_from_sequence` now take the keyword arg `copy=False` (GH21185)
- *`pandas.api.extensions.ExtensionArray.shift()`* added as part of the basic `ExtensionArray` interface (GH22387).
- `searchsorted()` has been added (GH24350)
- Support for reduction operations such as `sum`, `mean` via opt-in base class method override (GH22762)
- `ExtensionArray.isna()` is allowed to return an `ExtensionArray` (GH22325).

Dtype changes

- `ExtensionDtype` has gained the ability to instantiate from string dtypes, e.g. `decimal` would instantiate a registered `DecimalDtype`; furthermore the `ExtensionDtype` has gained the method `construct_array_type` (GH21185)
- Added `ExtensionDtype._is_numeric` for controlling whether an extension dtype is considered numeric (GH22290).
- Added `pandas.api.types.register_extension_dtype()` to register an extension type with pandas (GH22664)
- Updated the `.type` attribute for `PeriodDtype`, `DatetimeTZDtype`, and `IntervalDtype` to be instances of the dtype (`Period`, `Timestamp`, and `Interval` respectively) (GH22938)

Operator support

A `Series` based on an `ExtensionArray` now supports arithmetic and comparison operators (GH19577). There are two approaches for providing operator support for an `ExtensionArray`:

1. Define each of the operators on your `ExtensionArray` subclass.
2. Use an operator implementation from pandas that depends on operators that are already defined on the underlying elements (scalars) of the `ExtensionArray`.

See the *ExtensionArray Operator Support* documentation section for details on both ways of adding operator support.

Other changes

- A default repr for *`pandas.api.extensions.ExtensionArray`* is now provided (GH23601).
- `ExtensionArray._formatting_values()` is deprecated. Use `ExtensionArray._formatter` instead. (GH23601)
- An `ExtensionArray` with a boolean dtype now works correctly as a boolean indexer. *`pandas.api.types.is_bool_dtype()`* now properly considers them boolean (GH22326)

Bug Fixes

- Bug in *Series.get()* for Series using ExtensionArray and integer index (GH21257)
- *shift()* now dispatches to ExtensionArray.shift() (GH22386)
- *Series.combine()* works correctly with *ExtensionArray* inside of *Series* (GH20825)
- *Series.combine()* with scalar argument now works for any function type (GH21248)
- *Series.astype()* and *DataFrame.astype()* now dispatch to ExtensionArray.astype() (GH21185).
- Slicing a single row of a DataFrame with multiple ExtensionArrays of the same type now preserves the dtype, rather than coercing to object (GH22784)
- Bug when concatenating multiple Series with different extension dtypes not casting to object dtype (GH22994)
- Series backed by an ExtensionArray now work with *util.hash_pandas_object()* (GH23066)
- *DataFrame.stack()* no longer converts to object dtype for DataFrames where each column has the same extension dtype. The output Series will have the same dtype as the columns (GH23077).
- *Series.unstack()* and *DataFrame.unstack()* no longer convert extension arrays to object-dtype ndarrays. Each column in the output DataFrame will now have the same dtype as the input (GH23077).
- Bug when grouping Dataframe.groupby() and aggregating on ExtensionArray it was not returning the actual ExtensionArray dtype (GH23227).
- Bug in *pandas.merge()* when merging on an extension array-backed column (GH23020).

1.4 Deprecations

- MultiIndex.labels has been deprecated and replaced by *MultiIndex.codes*. The functionality is unchanged. The new name better reflects the natures of these codes and makes the MultiIndex API more similar to the API for *CategoricalIndex* (GH13443). As a consequence, other uses of the name labels in MultiIndex have also been deprecated and replaced with codes:
  - You should initialize a MultiIndex instance using a parameter named codes rather than labels.
  - MultiIndex.set_labels has been deprecated in favor of *MultiIndex.set_codes()*.
  - For method MultiIndex.copy(), the labels parameter has been deprecated and replaced by a codes parameter.
- *DataFrame.to_stata()*, *read_stata()*, StataReader and StataWriter have deprecated the encoding argument. The encoding of a Stata dta file is determined by the file type and cannot be changed (GH21244)
- *MultiIndex.to_hierarchical()* is deprecated and will be removed in a future version (GH21613)
- *Series.ptp()* is deprecated. Use numpy.ptp instead (GH21614)
- *Series.compress()* is deprecated. Use Series[condition] instead (GH18262)
- The signature of *Series.to_csv()* has been uniformed to that of *DataFrame.to_csv()*: the name of the first argument is now path_or_buf, the order of subsequent arguments has changed, the header argument now defaults to True. (GH19715)
- *Categorical.from_codes()* has deprecated providing float values for the codes argument. (GH21767)

- *pandas.read_table()* is deprecated. Instead, use *read_csv()* passing `sep='\t'` if necessary (GH21948)
- *Series.str.cat()* has deprecated using arbitrary list-likes *within* list-likes. A list-like container may still contain many `Series`, `Index` or 1-dimensional `np.ndarray`, or alternatively, only scalar values. (GH21950)
- `FrozenNDArray.searchsorted()` has deprecated the `v` parameter in favor of `value` (GH14645)
- `DatetimeIndex.shift()` and `PeriodIndex.shift()` now accept `periods` argument instead of `n` for consistency with *Index.shift()* and *Series.shift()*. Using `n` throws a deprecation warning (GH22458, GH22912)
- The `fastpath` keyword of the different Index constructors is deprecated (GH23110).
- *Timestamp.tz_localize()*, *DatetimeIndex.tz_localize()*, and *Series.tz_localize()* have deprecated the `errors` argument in favor of the `nonexistent` argument (GH8917)
- The class `FrozenNDArray` has been deprecated. When unpickling, `FrozenNDArray` will be unpickled to `np.ndarray` once this class is removed (GH9031)
- The methods *DataFrame.update()* and *Panel.update()* have deprecated the `raise_conflict=False|True` keyword in favor of `errors='ignore'|'raise'` (GH23585)
- The methods *Series.str.partition()* and *Series.str.rpartition()* have deprecated the `pat` keyword in favor of `sep` (GH22676)
- Deprecated the `nthreads` keyword of *pandas.read_feather()* in favor of `use_threads` to reflect the changes in `pyarrow>=0.11.0`. (GH23053)
- *pandas.read_excel()* has deprecated accepting `usecols` as an integer. Please pass in a list of ints from 0 to `usecols` inclusive instead (GH23527)
- Constructing a *TimedeltaIndex* from data with `datetime64`-dtyped data is deprecated, will raise `TypeError` in a future version (GH23539)
- Constructing a *DatetimeIndex* from data with `timedelta64`-dtyped data is deprecated, will raise `TypeError` in a future version (GH23675)
- The `keep_tz=False` option (the default) of the `keep_tz` keyword of *DatetimeIndex.to_series()* is deprecated (GH17832).
- Timezone converting a tz-aware `datetime.datetime` or *Timestamp* with *Timestamp* and the `tz` argument is now deprecated. Instead, use *Timestamp.tz_convert()* (GH23579)
- *pandas.api.types.is_period()* is deprecated in favor of `pandas.api.types.is_period_dtype` (GH23917)
- *pandas.api.types.is_datetimetz()* is deprecated in favor of `pandas.api.types.is_datetime64tz` (GH23917)
- Creating a *TimedeltaIndex*, *DatetimeIndex*, or *PeriodIndex* by passing range arguments *start*, *end*, and *periods* is deprecated in favor of *timedelta_range()*, *date_range()*, or *period_range()* (GH23919)
- Passing a string alias like `'datetime64[ns, UTC]'` as the `unit` parameter to *DatetimeTZDtype* is deprecated. Use *DatetimeTZDtype.construct_from_string* instead (GH23990).
- The `skipna` parameter of *infer_dtype()* will switch to `True` by default in a future version of pandas (GH17066, GH24050)
- In *Series.where()* with Categorical data, providing an `other` that is not present in the categories is deprecated. Convert the categorical to a different dtype or add the `other` to the categories first (GH24077).

- *Series.clip_lower()*, *Series.clip_upper()*, *DataFrame.clip_lower()* and *DataFrame.clip_upper()* are deprecated and will be removed in a future version. Use `Series.clip(lower=threshold)`, `Series.clip(upper=threshold)` and the equivalent `DataFrame` methods (GH24203)
- *Series.nonzero()* is deprecated and will be removed in a future version (GH18262)
- Passing an integer to *Series.fillna()* and *DataFrame.fillna()* with `timedelta64[ns]` dtypes is deprecated, will raise `TypeError` in a future version. Use `obj.fillna(pd.Timedelta(...))` instead (GH24694)
- `Series.cat.categorical`, `Series.cat.name` and `Sersies.cat.index` have been deprecated. Use the attributes on `Series.cat` or `Series` directly. (GH24751).
- Passing a dtype without a precision like `np.dtype('datetime64')` or `timedelta64` to *Index*, *DatetimeIndex* and *TimedeltaIndex* is now deprecated. Use the nanosecond-precision dtype instead (GH24753).

1.4.1 Integer Addition/Subtraction with Datetimes and Timedeltas is Deprecated

In the past, users could—in some cases—add or subtract integers or integer-dtype arrays from *Timestamp*, *DatetimeIndex* and *TimedeltaIndex*.

This usage is now deprecated. Instead add or subtract integer multiples of the object's `freq` attribute (GH21939, GH23878).

Previous Behavior:

```
In [5]: ts = pd.Timestamp('1994-05-06 12:15:16', freq=pd.offsets.Hour())
In [6]: ts + 2
Out[6]: Timestamp('1994-05-06 14:15:16', freq='H')

In [7]: tdi = pd.timedelta_range('1D', periods=2)
In [8]: tdi - np.array([2, 1])
Out[8]: TimedeltaIndex(['-1 days', '1 days'], dtype='timedelta64[ns]', freq=None)

In [9]: dti = pd.date_range('2001-01-01', periods=2, freq='7D')
In [10]: dti + pd.Index([1, 2])
Out[10]: DatetimeIndex(['2001-01-08', '2001-01-22'], dtype='datetime64[ns]',␣
↪freq=None)
```

New Behavior:

```
In [102]: ts = pd.Timestamp('1994-05-06 12:15:16', freq=pd.offsets.Hour())

In [103]: ts + 2 * ts.freq
Out[103]: Timestamp('1994-05-06 14:15:16', freq='H')

In [104]: tdi = pd.timedelta_range('1D', periods=2)

In [105]: tdi - np.array([2 * tdi.freq, 1 * tdi.freq])
Out[105]: TimedeltaIndex(['-1 days', '1 days'], dtype='timedelta64[ns]', freq=None)

In [106]: dti = pd.date_range('2001-01-01', periods=2, freq='7D')

In [107]: dti + pd.Index([1 * dti.freq, 2 * dti.freq])
Out[107]: DatetimeIndex(['2001-01-08', '2001-01-22'], dtype='datetime64[ns]',␣
↪freq=None)
```

1.4.2 Passing Integer data and a timezone to DatetimeIndex

The behavior of *DatetimeIndex* when passed integer data and a timezone is changing in a future version of pandas. Previously, these were interpreted as wall times in the desired timezone. In the future, these will be interpreted as wall times in UTC, which are then converted to the desired timezone (GH24559).

The default behavior remains the same, but issues a warning:

```
In [3]: pd.DatetimeIndex([946684800000000000], tz="US/Central")
/bin/ipython:1: FutureWarning:
    Passing integer-dtype data and a timezone to DatetimeIndex. Integer values
    will be interpreted differently in a future version of pandas. Previously,
    these were viewed as datetime64[ns] values representing the wall time
    *in the specified timezone*. In the future, these will be viewed as
    datetime64[ns] values representing the wall time *in UTC*. This is similar
    to a nanosecond-precision UNIX epoch. To accept the future behavior, use

        pd.to_datetime(integer_data, utc=True).tz_convert(tz)

    To keep the previous behavior, use

        pd.to_datetime(integer_data).tz_localize(tz)

 #!/bin/python3
 Out[3]: DatetimeIndex(['2000-01-01 00:00:00-06:00'], dtype='datetime64[ns, US/
↪Central]', freq=None)
```

As the warning message explains, opt in to the future behavior by specifying that the integer values are UTC, and then converting to the final timezone:

```
In [108]: pd.to_datetime([946684800000000000], utc=True).tz_convert('US/Central')
Out[108]: DatetimeIndex(['1999-12-31 18:00:00-06:00'], dtype='datetime64[ns, US/
↪Central]', freq=None)
```

The old behavior can be retained with by localizing directly to the final timezone:

```
In [109]: pd.to_datetime([946684800000000000]).tz_localize('US/Central')
Out[109]: DatetimeIndex(['2000-01-01 00:00:00-06:00'], dtype='datetime64[ns, US/
↪Central]', freq=None)
```

1.4.3 Converting Timezone-Aware Series and Index to NumPy Arrays

The conversion from a *Series* or *Index* with timezone-aware datetime data will change to preserve timezones by default (GH23569).

NumPy doesn't have a dedicated dtype for timezone-aware datetimes. In the past, converting a *Series* or *DatetimeIndex* with timezone-aware datatimes would convert to a NumPy array by

1. converting the tz-aware data to UTC
2. dropping the timezone-info
3. returning a `numpy.ndarray` with `datetime64[ns]` dtype

Future versions of pandas will preserve the timezone information by returning an object-dtype NumPy array where each value is a *Timestamp* with the correct timezone attached

```
In [110]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))

In [111]: ser
Out[111]:
0   2000-01-01 00:00:00+01:00
1   2000-01-02 00:00:00+01:00
Length: 2, dtype: datetime64[ns, CET]
```

The default behavior remains the same, but issues a warning

```
In [8]: np.asarray(ser)
/bin/ipython:1: FutureWarning: Converting timezone-aware DatetimeArray to timezone-
↪naive
      ndarray with 'datetime64[ns]' dtype. In the future, this will return an ndarray
      with 'object' dtype where each element is a 'pandas.Timestamp' with the correct
↪'tz'.

        To accept the future behavior, pass 'dtype=object'.
        To keep the old behavior, pass 'dtype="datetime64[ns]"'.
  #!/bin/python3
Out[8]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
      dtype='datetime64[ns]')
```

The previous or future behavior can be obtained, without any warnings, by specifying the `dtype`

Previous Behavior

```
In [112]: np.asarray(ser, dtype='datetime64[ns]')
Out[112]: array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],␣
↪dtype='datetime64[ns]')
```

Future Behavior

```
# New behavior
In [113]: np.asarray(ser, dtype=object)
Out[113]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')], dtype=object)
```

Or by using *Series.to_numpy()*

```
In [114]: ser.to_numpy()
Out[114]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')], dtype=object)

In [115]: ser.to_numpy(dtype="datetime64[ns]")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'], dtype=
↪'datetime64[ns]')
```

All the above applies to a *DatetimeIndex* with tz-aware values as well.

1.5 Removal of prior version deprecations/changes

- The `LongPanel` and `WidePanel` classes have been removed (GH10892)
- *`Series.repeat()`* has renamed the `reps` argument to `repeats` (GH14645)
- Several private functions were removed from the (non-public) module `pandas.core.common` (GH22001)
- Removal of the previously deprecated module `pandas.core.datetools` (GH14105, GH14094)
- Strings passed into *`DataFrame.groupby()`* that refer to both column and index levels will raise a `ValueError` (GH14432)
- *`Index.repeat()`* and `MultiIndex.repeat()` have renamed the `n` argument to `repeats` (GH14645)
- The `Series` constructor and `.astype` method will now raise a `ValueError` if timestamp dtypes are passed in without a unit (e.g. `np.datetime64`) for the `dtype` parameter (GH15987)
- Removal of the previously deprecated `as_indexer` keyword completely from `str.match()` (GH22356, GH6581)
- The modules `pandas.types`, `pandas.computation`, and `pandas.util.decorators` have been removed (GH16157, GH16250)
- Removed the `pandas.formats.style` shim for *`pandas.io.formats.style.Styler`* (GH16059)
- `pandas.pnow`, `pandas.match`, `pandas.groupby`, `pd.get_store`, `pd.Expr`, and `pd.Term` have been removed (GH15538, GH15940)
- `Categorical.searchsorted()` and *`Series.searchsorted()`* have renamed the `v` argument to `value` (GH14645)
- `pandas.parser`, `pandas.lib`, and `pandas.tslib` have been removed (GH15537)
- *`Index.searchsorted()`* have renamed the `key` argument to `value` (GH14645)
- `DataFrame.consolidate` and `Series.consolidate` have been removed (GH15501)
- Removal of the previously deprecated module `pandas.json` (GH19944)
- The module `pandas.tools` has been removed (GH15358, GH16005)
- *`SparseArray.get_values()`* and *`SparseArray.to_dense()`* have dropped the `fill` parameter (GH14686)
- `DataFrame.sortlevel` and `Series.sortlevel` have been removed (GH15099)
- `SparseSeries.to_dense()` has dropped the `sparse_only` parameter (GH14686)
- *`DataFrame.astype()`* and *`Series.astype()`* have renamed the `raise_on_error` argument to `errors` (GH14967)
- `is_sequence`, `is_any_int_dtype`, and `is_floating_dtype` have been removed from `pandas.api.types` (GH16163, GH16189)

1.6 Performance Improvements

- Slicing Series and DataFrames with an monotonically increasing *`CategoricalIndex`* is now very fast and has speed comparable to slicing with an `Int64Index`. The speed increase is both when indexing by label (using .loc) and position(.iloc) (GH20395) Slicing a monotonically increasing *`CategoricalIndex`* itself (i.e. `ci[1000:2000]`) shows similar speed improvements as above (GH21659)

- Improved performance of *CategoricalIndex.equals()* when comparing to another *CategoricalIndex* (GH24023)
- Improved performance of *Series.describe()* in case of numeric dtpyes (GH21274)
- Improved performance of *pandas.core.groupby.GroupBy.rank()* when dealing with tied rankings (GH21237)
- Improved performance of *DataFrame.set_index()* with columns consisting of *Period* objects (GH21582, GH21606)
- Improved performance of *Series.at()* and *Index.get_value()* for Extension Arrays values (e.g. *Categorical*) (GH24204)
- Improved performance of membership checks in *Categorical* and *CategoricalIndex* (i.e. `x in cat`-style checks are much faster). `CategoricalIndex.contains()` is likewise much faster (GH21369, GH21508)
- Improved performance of *HDFStore.groups()* (and dependent functions like *HDFStore.keys()*. (i.e. `x in store` checks are much faster) (GH21372)
- Improved the performance of *pandas.get_dummies()* with `sparse=True` (GH21997)
- Improved performance of `IndexEngine.get_indexer_non_unique()` for sorted, non-unique indexes (GH9466)
- Improved performance of `PeriodIndex.unique()` (GH23083)
- Improved performance of *concat()* for *Series* objects (GH23404)
- Improved performance of *DatetimeIndex.normalize()* and *Timestamp.normalize()* for timezone naive or UTC datetimes (GH23634)
- Improved performance of *DatetimeIndex.tz_localize()* and various `DatetimeIndex` attributes with dateutil UTC timezone (GH23772)
- Fixed a performance regression on Windows with Python 3.7 of *read_csv()* (GH23516)
- Improved performance of *Categorical* constructor for `Series` objects (GH23814)
- Improved performance of *where()* for Categorical data (GH24077)
- Improved performance of iterating over a *Series*. Using *DataFrame.itertuples()* now creates iterators without internally allocating lists of all elements (GH20783)
- Improved performance of *Period* constructor, additionally benefitting `PeriodArray` and `PeriodIndex` creation (GH24084, GH24118)
- Improved performance of tz-aware `DatetimeArray` binary operations (GH24491)

1.7 Bug Fixes

1.7.1 Categorical

- Bug in *Categorical.from_codes()* where `NaN` values in `codes` were silently converted to `0` (GH21767). In the future this will raise a `ValueError`. Also changes the behavior of `.from_codes([1.1, 2.0])`.
- Bug in `Categorical.sort_values()` where `NaN` values were always positioned in front regardless of `na_position` value. (GH22556).

- Bug when indexing with a boolean-valued `Categorical`. Now a boolean-valued `Categorical` is treated as a boolean mask (GH22665)
- Constructing a *CategoricalIndex* with empty values and boolean categories was raising a `ValueError` after a change to dtype coercion (GH22702).
- Bug in `Categorical.take()` with a user-provided `fill_value` not encoding the `fill_value`, which could result in a `ValueError`, incorrect results, or a segmentation fault (GH23296).
- In *Series.unstack()*, specifying a `fill_value` not present in the categories now raises a `TypeError` rather than ignoring the `fill_value` (GH23284)
- Bug when resampling *DataFrame.resample()* and aggregating on categorical data, the categorical dtype was getting lost. (GH23227)
- Bug in many methods of the `.str`-accessor, which always failed on calling the `CategoricalIndex.str` constructor (GH23555, GH23556)
- Bug in *Series.where()* losing the categorical dtype for categorical data (GH24077)
- Bug in `Categorical.apply()` where `NaN` values could be handled unpredictably. They now remain unchanged (GH24241)
- Bug in *Categorical* comparison methods incorrectly raising `ValueError` when operating against a *DataFrame* (GH24630)
- Bug in `Categorical.set_categories()` where setting fewer new categories with `rename=True` caused a segmentation fault (GH24675)

1.7.2 Datetimelike

- Fixed bug where two `DateOffset` objects with different `normalize` attributes could evaluate as equal (GH21404)
- Fixed bug where *Timestamp.resolution()* incorrectly returned 1-microsecond `timedelta` instead of 1-nanosecond *Timedelta* (GH21336, GH21365)
- Bug in *to_datetime()* that did not consistently return an *Index* when `box=True` was specified (GH21864)
- Bug in *DatetimeIndex* comparisons where string comparisons incorrectly raises `TypeError` (GH22074)
- Bug in *DatetimeIndex* comparisons when comparing against `timedelta64[ns]` dtyped arrays; in some cases `TypeError` was incorrectly raised, in others it incorrectly failed to raise (GH22074)
- Bug in *DatetimeIndex* comparisons when comparing against object-dtyped arrays (GH22074)
- Bug in *DataFrame* with `datetime64[ns]` dtype addition and subtraction with `Timedelta`-like objects (GH22005, GH22163)
- Bug in *DataFrame* with `datetime64[ns]` dtype addition and subtraction with `DateOffset` objects returning an `object` dtype instead of `datetime64[ns]` dtype (GH21610, GH22163)
- Bug in *DataFrame* with `datetime64[ns]` dtype comparing against `NaT` incorrectly (GH22242, GH22163)
- Bug in *DataFrame* with `datetime64[ns]` dtype subtracting `Timestamp`-like object incorrectly returned `datetime64[ns]` dtype instead of `timedelta64[ns]` dtype (GH8554, GH22163)
- Bug in *DataFrame* with `datetime64[ns]` dtype subtracting `np.datetime64` object with non-nanosecond unit failing to convert to nanoseconds (GH18874, GH22163)
- Bug in *DataFrame* comparisons against `Timestamp`-like objects failing to raise `TypeError` for inequality checks with mismatched types (GH8932, GH22163)

- Bug in *DataFrame* with mixed dtypes including `datetime64[ns]` incorrectly raising `TypeError` on equality comparisons (GH13128, GH22163)
- Bug in *DataFrame.values* returning a *DatetimeIndex* for a single-column `DataFrame` with tz-aware datetime values. Now a 2-D `numpy.ndarray` of *Timestamp* objects is returned (GH24024)
- Bug in *DataFrame.eq()* comparison against `NaT` incorrectly returning `True` or `NaN` (GH15697, GH22163)
- Bug in *DatetimeIndex* subtraction that incorrectly failed to raise `OverflowError` (GH22492, GH22508)
- Bug in *DatetimeIndex* incorrectly allowing indexing with `Timedelta` object (GH20464)
- Bug in *DatetimeIndex* where frequency was being set if original frequency was `None` (GH22150)
- Bug in rounding methods of *DatetimeIndex* (*round()*, *ceil()*, *floor()*) and *Timestamp* (*round()*, *ceil()*, *floor()*) could give rise to loss of precision (GH22591)
- Bug in *to_datetime()* with an *Index* argument that would drop the `name` from the result (GH21697)
- Bug in *PeriodIndex* where adding or subtracting a `timedelta` or `Tick` object produced incorrect results (GH22988)
- Bug in the *Series* repr with period-dtype data missing a space before the data (GH23601)
- Bug in *date_range()* when decrementing a start date to a past end date by a negative frequency (GH23270)
- Bug in *Series.min()* which would return `NaN` instead of `NaT` when called on a series of `NaT` (GH23282)
- Bug in *Series.combine_first()* not properly aligning categoricals, so that missing values in `self` where not filled by valid values from `other` (GH24147)
- Bug in *DataFrame.combine()* with datetimelike values raising a TypeError (GH23079)
- Bug in *date_range()* with frequency of `Day` or higher where dates sufficiently far in the future could wrap around to the past instead of raising `OutOfBoundsDatetime` (GH14187)
- Bug in *period_range()* ignoring the frequency of `start` and `end` when those are provided as *Period* objects (GH20535).
- Bug in *PeriodIndex* with attribute `freq.n` greater than 1 where adding a `DateOffset` object would return incorrect results (GH23215)
- Bug in *Series* that interpreted string indices as lists of characters when setting datetimelike values (GH23451)
- Bug in *DataFrame* when creating a new column from an ndarray of *Timestamp* objects with timezones creating an object-dtype column, rather than datetime with timezone (GH23932)
- Bug in *Timestamp* constructor which would drop the frequency of an input *Timestamp* (GH22311)
- Bug in *DatetimeIndex* where calling `np.array(dtindex, dtype=object)` would incorrectly return an array of `long` objects (GH23524)
- Bug in *Index* where passing a timezone-aware *DatetimeIndex* and *dtype=object* would incorrectly raise a `ValueError` (GH23524)
- Bug in *Index* where calling `np.array(dtindex, dtype=object)` on a timezone-naive *DatetimeIndex* would return an array of `datetime` objects instead of *Timestamp* objects, potentially losing nanosecond portions of the timestamps (GH23524)
- Bug in `Categorical.__setitem__` not allowing setting with another `Categorical` when both are undordered and have the same categories, but in a different order (GH24142)
- Bug in *date_range()* where using dates with millisecond resolution or higher could return incorrect values or the wrong number of values in the index (GH24110)

- Bug in *DatetimeIndex* where constructing a *DatetimeIndex* from a *Categorical* or *CategoricalIndex* would incorrectly drop timezone information (GH18664)
- Bug in *DatetimeIndex* and *TimedeltaIndex* where indexing with `Ellipsis` would incorrectly lose the index's `freq` attribute (GH21282)
- Clarified error message produced when passing an incorrect `freq` argument to *DatetimeIndex* with `NaT` as the first entry in the passed data (GH11587)
- Bug in *to_datetime()* where `box` and `utc` arguments were ignored when passing a *DataFrame* or `dict` of unit mappings (GH23760)
- Bug in *Series.dt* where the cache would not update properly after an in-place operation (GH24408)
- Bug in *PeriodIndex* where comparisons against an array-like object with length 1 failed to raise `ValueError` (GH23078)
- Bug in `DatetimeIndex.astype()`, `PeriodIndex.astype()` and `TimedeltaIndex.astype()` ignoring the sign of the `dtype` for unsigned integer dtypes (GH24405).
- Fixed bug in *Series.max()* with `datetime64[ns]`-dtype failing to return `NaT` when nulls are present and `skipna=False` is passed (GH24265)
- Bug in *to_datetime()* where arrays of `datetime` objects containing both timezone-aware and timezone-naive `datetimes` would fail to raise `ValueError` (GH24569)
- Bug in *to_datetime()* with invalid datetime format doesn't coerce input to `NaT` even if `errors='coerce'` (GH24763)

1.7.3 Timedelta

- Bug in *DataFrame* with `timedelta64[ns]` dtype division by `Timedelta`-like scalar incorrectly returning `timedelta64[ns]` dtype instead of `float64` dtype (GH20088, GH22163)
- Bug in adding a *Index* with object dtype to a *Series* with `timedelta64[ns]` dtype incorrectly raising (GH22390)
- Bug in multiplying a *Series* with numeric dtype against a `timedelta` object (GH22390)
- Bug in *Series* with numeric dtype when adding or subtracting an an array or `Series` with `timedelta64` dtype (GH22390)
- Bug in *Index* with numeric dtype when multiplying or dividing an array with dtype `timedelta64` (GH22390)
- Bug in *TimedeltaIndex* incorrectly allowing indexing with `Timestamp` object (GH20464)
- Fixed bug where subtracting *Timedelta* from an object-dtyped array would raise `TypeError` (GH21980)
- Fixed bug in adding a *DataFrame* with all-*timedelta64[ns]* dtypes to a *DataFrame* with all-integer dtypes returning incorrect results instead of raising `TypeError` (GH22696)
- Bug in *TimedeltaIndex* where adding a timezone-aware datetime scalar incorrectly returned a timezone-naive *DatetimeIndex* (GH23215)
- Bug in *TimedeltaIndex* where adding `np.timedelta64('NaT')` incorrectly returned an all-`NaT` *DatetimeIndex* instead of an all-`NaT` *TimedeltaIndex* (GH23215)
- Bug in *Timedelta* and *to_timedelta()* have inconsistencies in supported unit string (GH21762)
- Bug in *TimedeltaIndex* division where dividing by another *TimedeltaIndex* raised `TypeError` instead of returning a *Float64Index* (GH23829, GH22631)

- Bug in *TimedeltaIndex* comparison operations where comparing against non-`Timedelta`-like objects would raise `TypeError` instead of returning all-`False` for `__eq__` and all-`True` for `__ne__` (GH24056)
- Bug in *Timedelta* comparisons when comparing with a `Tick` object incorrectly raising `TypeError` (GH24710)

1.7.4 Timezones

- Bug in *Index.shift()* where an `AssertionError` would raise when shifting across DST (GH8616)
- Bug in *Timestamp* constructor where passing an invalid timezone offset designator (`Z`) would not raise a `ValueError` (GH8910)
- Bug in *Timestamp.replace()* where replacing at a DST boundary would retain an incorrect offset (GH7825)
- Bug in *Series.replace()* with `datetime64[ns, tz]` data when replacing `NaT` (GH11792)
- Bug in *Timestamp* when passing different string date formats with a timezone offset would produce different timezone offsets (GH12064)
- Bug when comparing a tz-naive *Timestamp* to a tz-aware *DatetimeIndex* which would coerce the *DatetimeIndex* to tz-naive (GH12601)
- Bug in *Series.truncate()* with a tz-aware *DatetimeIndex* which would cause a core dump (GH9243)
- Bug in *Series* constructor which would coerce tz-aware and tz-naive *Timestamp* to tz-aware (GH13051)
- Bug in *Index* with `datetime64[ns, tz]` dtype that did not localize integer data correctly (GH20964)
- Bug in *DatetimeIndex* where constructing with an integer and tz would not localize correctly (GH12619)
- Fixed bug where *DataFrame.describe()* and *Series.describe()* on tz-aware datetimes did not show *first* and *last* result (GH21328)
- Bug in *DatetimeIndex* comparisons failing to raise `TypeError` when comparing timezone-aware `DatetimeIndex` against `np.datetime64` (GH22074)
- Bug in `DataFrame` assignment with a timezone-aware scalar (GH19843)
- Bug in *DataFrame.asof()* that raised a `TypeError` when attempting to compare tz-naive and tz-aware timestamps (GH21194)
- Bug when constructing a *DatetimeIndex* with *Timestamp* constructed with the `replace` method across DST (GH18785)
- Bug when setting a new value with *DataFrame.loc()* with a *DatetimeIndex* with a DST transition (GH18308, GH20724)
- Bug in *Index.unique()* that did not re-localize tz-aware dates correctly (GH21737)
- Bug when indexing a *Series* with a DST transition (GH21846)
- Bug in *DataFrame.resample()* and *Series.resample()* where an `AmbiguousTimeError` or `NonExistentTimeError` would raise if a timezone aware timeseries ended on a DST transition (GH19375, GH10117)
- Bug in *DataFrame.drop()* and *Series.drop()* when specifying a tz-aware Timestamp key to drop from a *DatetimeIndex* with a DST transition (GH21761)
- Bug in *DatetimeIndex* constructor where `NaT` and `dateutil.tz.tzlocal` would raise an `OutOfBoundsDatetime` error (GH23807)

- Bug in `DatetimeIndex.tz_localize()` and `Timestamp.tz_localize()` with `dateutil.tz.tzlocal` near a DST transition that would return an incorrectly localized datetime (GH23807)
- Bug in `Timestamp` constructor where a `dateutil.tz.tzutc` timezone passed with a `datetime.datetime` argument would be converted to a `pytz.UTC` timezone (GH23807)
- Bug in `to_datetime()` where `utc=True` was not respected when specifying a `unit` and `errors='ignore'` (GH23758)
- Bug in `to_datetime()` where `utc=True` was not respected when passing a `Timestamp` (GH24415)
- Bug in `DataFrame.any()` returns wrong value when `axis=1` and the data is of datetimelike type (GH23070)
- Bug in `DatetimeIndex.to_period()` where a timezone aware index was converted to UTC first before creating `PeriodIndex` (GH22905)
- Bug in `DataFrame.tz_localize()`, `DataFrame.tz_convert()`, `Series.tz_localize()`, and `Series.tz_convert()` where `copy=False` would mutate the original argument inplace (GH6326)
- Bug in `DataFrame.max()` and `DataFrame.min()` with `axis=1` where a `Series` with `NaN` would be returned when all columns contained the same timezone (GH10390)

1.7.5 Offsets

- Bug in `FY5253` where date offsets could incorrectly raise an `AssertionError` in arithmetic operatons (GH14774)
- Bug in `DateOffset` where keyword arguments `week` and `milliseconds` were accepted and ignored. Passing these will now raise `ValueError` (GH19398)
- Bug in adding `DateOffset` with `DataFrame` or `PeriodIndex` incorrectly raising `TypeError` (GH23215)
- Bug in comparing `DateOffset` objects with non-DateOffset objects, particularly strings, raising `ValueError` instead of returning `False` for equality checks and `True` for not-equal checks (GH23524)

1.7.6 Numeric

- Bug in `Series` `__rmatmul__` doesn't support matrix vector multiplication (GH21530)
- Bug in `factorize()` fails with read-only array (GH12813)
- Fixed bug in `unique()` handled signed zeros inconsistently: for some inputs 0.0 and -0.0 were treated as equal and for some inputs as different. Now they are treated as equal for all inputs (GH21866)
- Bug in `DataFrame.agg()`, `DataFrame.transform()` and `DataFrame.apply()` where, when supplied with a list of functions and `axis=1` (e.g. `df.apply(['sum', 'mean'], axis=1)`), a `TypeError` was wrongly raised. For all three methods such calculation are now done correctly. (GH16679).
- Bug in `Series` comparison against datetime-like scalars and arrays (GH22074)
- Bug in `DataFrame` multiplication between boolean dtype and integer returning `object` dtype instead of integer dtype (GH22047, GH22163)
- Bug in `DataFrame.apply()` where, when supplied with a string argument and additional positional or keyword arguments (e.g. `df.apply('sum', min_count=1)`), a `TypeError` was wrongly raised (GH22376)
- Bug in `DataFrame.astype()` to extension dtype may raise `AttributeError` (GH22578)

- Bug in *DataFrame* with `timedelta64[ns]` dtype arithmetic operations with `ndarray` with integer dtype incorrectly treating the narray as `timedelta64[ns]` dtype (GH23114)
- Bug in *Series.rpow()* with object dtype `NaN` for `1 ** NA` instead of `1` (GH22922).
- *Series.agg()* can now handle numpy NaN-aware methods like `numpy.nansum()` (GH19629)
- Bug in *Series.rank()* and *DataFrame.rank()* when `pct=True` and more than 2^{24} rows are present resulted in percentages greater than 1.0 (GH18271)
- Calls such as *DataFrame.round()* with a non-unique *CategoricalIndex()* now return expected data. Previously, data would be improperly duplicated (GH21809).
- Added `log10`, *floor* and *ceil* to the list of supported functions in *DataFrame.eval()* (GH24139, GH24353)
- Logical operations `&, |, ^` between *Series* and *Index* will no longer raise `ValueError` (GH22092)
- Checking PEP 3141 numbers in *is_scalar()* function returns `True` (GH22903)
- Reduction methods like *Series.sum()* now accept the default value of `keepdims=False` when called from a NumPy ufunc, rather than raising a `TypeError`. Full support for `keepdims` has not been implemented (GH24356).

1.7.7 Conversion

- Bug in *DataFrame.combine_first()* in which column types were unexpectedly converted to float (GH20699)
- Bug in *DataFrame.clip()* in which column types are not preserved and casted to float (GH24162)
- Bug in *DataFrame.clip()* when order of columns of dataframes doesn't match, result observed is wrong in numeric values (GH20911)
- Bug in *DataFrame.astype()* where converting to an extension dtype when duplicate column names are present causes a `RecursionError` (GH24704)

1.7.8 Strings

- Bug in `Index.str.partition()` was not nan-safe (GH23558).
- Bug in `Index.str.split()` was not nan-safe (GH23677).
- Bug *Series.str.contains()* not respecting the `na` argument for a `Categorical` dtype `Series` (GH22158)
- Bug in `Index.str.cat()` when the result contained only `NaN` (GH24044)

1.7.9 Interval

- Bug in the *IntervalIndex* constructor where the `closed` parameter did not always override the inferred `closed` (GH19370)
- Bug in the `IntervalIndex` repr where a trailing comma was missing after the list of intervals (GH20611)
- Bug in *Interval* where scalar arithmetic operations did not retain the `closed` value (GH22313)
- Bug in *IntervalIndex* where indexing with datetime-like values raised a `KeyError` (GH20636)
- Bug in `IntervalTree` where data containing `NaN` triggered a warning and resulted in incorrect indexing queries with *IntervalIndex* (GH23352)

1.7.10 Indexing

- Bug in `DataFrame.ne()` fails if columns contain column name "dtype" (GH22383)
- The traceback from a `KeyError` when asking `.loc` for a single missing label is now shorter and more clear (GH21557)
- `PeriodIndex` now emits a `KeyError` when a malformed string is looked up, which is consistent with the behavior of `DatetimeIndex` (GH22803)
- When `.ix` is asked for a missing integer label in a `MultiIndex` with a first level of integer type, it now raises a `KeyError`, consistently with the case of a flat `Int64Index`, rather than falling back to positional indexing (GH21593)
- Bug in `Index.reindex()` when reindexing a tz-naive and tz-aware `DatetimeIndex` (GH8306)
- Bug in `Series.reindex()` when reindexing an empty series with a `datetime64[ns, tz]` dtype (GH20869)
- Bug in `DataFrame` when setting values with `.loc` and a timezone aware `DatetimeIndex` (GH11365)
- `DataFrame.__getitem__` now accepts dictionaries and dictionary keys as list-likes of labels, consistently with `Series.__getitem__` (GH21294)
- Fixed `DataFrame[np.nan]` when columns are non-unique (GH21428)
- Bug when indexing `DatetimeIndex` with nanosecond resolution dates and timezones (GH11679)
- Bug where indexing with a Numpy array containing negative values would mutate the indexer (GH21867)
- Bug where mixed indexes wouldn't allow integers for `.at` (GH19860)
- `Float64Index.get_loc` now raises `KeyError` when boolean key passed. (GH19087)
- Bug in `DataFrame.loc()` when indexing with an `IntervalIndex` (GH19977)
- `Index` no longer mangles `None`, `NaN` and `NaT`, i.e. they are treated as three different keys. However, for numeric Index all three are still coerced to a `NaN` (GH22332)
- Bug in `scalar in Index` if scalar is a float while the `Index` is of integer dtype (GH22085)
- Bug in `MultiIndex.set_levels()` when levels value is not subscriptable (GH23273)
- Bug where setting a timedelta column by `Index` causes it to be casted to double, and therefore lose precision (GH23511)
- Bug in `Index.union()` and `Index.intersection()` where name of the `Index` of the result was not computed correctly for certain cases (GH9943, GH9862)
- Bug in `Index` slicing with boolean `Index` may raise `TypeError` (GH22533)
- Bug in `PeriodArray.__setitem__` when accepting slice and list-like value (GH23978)
- Bug in `DatetimeIndex`, `TimedeltaIndex` where indexing with `Ellipsis` would lose their `freq` attribute (GH21282)
- Bug in `iat` where using it to assign an incompatible value would create a new column (GH23236)

1.7.11 Missing

- Bug in `DataFrame.fillna()` where a `ValueError` would raise when one column contained a `datetime64[ns, tz]` dtype (GH15522)

- Bug in `Series.hasnans()` that could be incorrectly cached and return incorrect answers if null elements are introduced after an initial call (GH19700)
- `Series.isin()` now treats all NaN-floats as equal also for `np.object`-dtype. This behavior is consistent with the behavior for float64 (GH22119)
- `unique()` no longer mangles NaN-floats and the `NaT`-object for `np.object`-dtype, i.e. `NaT` is no longer coerced to a NaN-value and is treated as a different entity. (GH22295)
- `DataFrame` and `Series` now properly handle numpy masked arrays with hardened masks. Previously, constructing a DataFrame or Series from a masked array with a hard mask would create a pandas object containing the underlying value, rather than the expected NaN. (GH24574)
- Bug in `DataFrame` constructor where `dtype` argument was not honored when handling numpy masked record arrays. (GH24874)

1.7.12 MultiIndex

- Bug in `io.formats.style.Styler.applymap()` where `subset=` with `MultiIndex` slice would reduce to `Series` (GH19861)
- Removed compatibility for `MultiIndex` pickles prior to version 0.8.0; compatibility with `MultiIndex` pickles from version 0.13 forward is maintained (GH21654)
- `MultiIndex.get_loc_level()` (and as a consequence, `.loc` on a `Series` or `DataFrame` with a `MultiIndex` index) will now raise a `KeyError`, rather than returning an empty `slice`, if asked a label which is present in the `levels` but is unused (GH22221)
- `MultiIndex` has gained the `MultiIndex.from_frame()`, it allows constructing a `MultiIndex` object from a `DataFrame` (GH22420)
- Fix `TypeError` in Python 3 when creating `MultiIndex` in which some levels have mixed types, e.g. when some labels are tuples (GH15457)

1.7.13 I/O

- Bug in `read_csv()` in which a column specified with `CategoricalDtype` of boolean categories was not being correctly coerced from string values to booleans (GH20498)
- Bug in `read_csv()` in which unicode column names were not being properly recognized with Python 2.x (GH13253)
- Bug in `DataFrame.to_sql()` when writing timezone aware data (`datetime64[ns, tz]` dtype) would raise a `TypeError` (GH9086)
- Bug in `DataFrame.to_sql()` where a naive `DatetimeIndex` would be written as `TIMESTAMP WITH TIMEZONE` type in supported databases, e.g. PostgreSQL (GH23510)
- Bug in `read_excel()` when `parse_cols` is specified with an empty dataset (GH9208)
- `read_html()` no longer ignores all-whitespace `<tr>` within `<thead>` when considering the `skiprows` and `header` arguments. Previously, users had to decrease their `header` and `skiprows` values on such tables to work around the issue. (GH21641)
- `read_excel()` will correctly show the deprecation warning for previously deprecated `sheetname` (GH17994)
- `read_csv()` and `read_table()` will throw `UnicodeError` and not coredump on badly encoded strings (GH22748)

- *read_csv()* will correctly parse timezone-aware datetimes (GH22256)
- Bug in *read_csv()* in which memory management was prematurely optimized for the C engine when the data was being read in chunks (GH23509)
- Bug in *read_csv()* in unnamed columns were being improperly identified when extracting a multi-index (GH23687)
- *read_sas()* will parse numbers in sas7bdat-files that have width less than 8 bytes correctly. (GH21616)
- *read_sas()* will correctly parse sas7bdat files with many columns (GH22628)
- *read_sas()* will correctly parse sas7bdat files with data page types having also bit 7 set (so page type is 128 + 256 = 384) (GH16615)
- Bug in *read_sas()* in which an incorrect error was raised on an invalid file format. (GH24548)
- Bug in `detect_client_encoding()` where potential `IOError` goes unhandled when importing in a mod_wsgi process due to restricted access to stdout. (GH21552)
- Bug in *DataFrame.to_html()* with `index=False` misses truncation indicators (`...`) on truncated DataFrame (GH15019, GH22783)
- Bug in *DataFrame.to_html()* with `index=False` when both columns and row index are `MultiIndex` (GH22579)
- Bug in *DataFrame.to_html()* with `index_names=False` displaying index name (GH22747)
- Bug in *DataFrame.to_html()* with `header=False` not displaying row index names (GH23788)
- Bug in *DataFrame.to_html()* with `sparsify=False` that caused it to raise `TypeError` (GH22887)
- Bug in *DataFrame.to_string()* that broke column alignment when `index=False` and width of first column's values is greater than the width of first column's header (GH16839, GH13032)
- Bug in *DataFrame.to_string()* that caused representations of *DataFrame* to not take up the whole window (GH22984)
- Bug in *DataFrame.to_csv()* where a single level MultiIndex incorrectly wrote a tuple. Now just the value of the index is written (GH19589).
- `HDFStore` will raise `ValueError` when the `format` kwarg is passed to the constructor (GH13291)
- Bug in *HDFStore.append()* when appending a *DataFrame* with an empty string column and `min_itemsize` < 8 (GH12242)
- Bug in *read_csv()* in which memory leaks occurred in the C engine when parsing `NaN` values due to insufficient cleanup on completion or error (GH21353)
- Bug in *read_csv()* in which incorrect error messages were being raised when `skipfooter` was passed in along with `nrows`, `iterator`, or `chunksize` (GH23711)
- Bug in *read_csv()* in which *MultiIndex* index names were being improperly handled in the cases when they were not provided (GH23484)
- Bug in *read_csv()* in which unnecessary warnings were being raised when the dialect's values conflicted with the default arguments (GH23761)
- Bug in *read_html()* in which the error message was not displaying the valid flavors when an invalid one was provided (GH23549)
- Bug in *read_excel()* in which extraneous header names were extracted, even though none were specified (GH11733)
- Bug in *read_excel()* in which column names were not being properly converted to string sometimes in Python 2.x (GH23874)

- Bug in `read_excel()` in which `index_col=None` was not being respected and parsing index columns anyway (GH18792, GH20480)
- Bug in `read_excel()` in which `usecols` was not being validated for proper column names when passed in as a string (GH20480)
- Bug in `DataFrame.to_dict()` when the resulting dict contains non-Python scalars in the case of numeric data (GH23753)
- `DataFrame.to_string()`, `DataFrame.to_html()`, `DataFrame.to_latex()` will correctly format output when a string is passed as the `float_format` argument (GH21625, GH22270)
- Bug in `read_csv()` that caused it to raise `OverflowError` when trying to use 'inf' as `na_value` with integer index column (GH17128)
- Bug in `read_csv()` that caused the C engine on Python 3.6+ on Windows to improperly read CSV filenames with accented or special characters (GH15086)
- Bug in `read_fwf()` in which the compression type of a file was not being properly inferred (GH22199)
- Bug in `pandas.io.json.json_normalize()` that caused it to raise `TypeError` when two consecutive elements of `record_path` are dicts (GH22706)
- Bug in `DataFrame.to_stata()`, `pandas.io.stata.StataWriter` and `pandas.io.stata.StataWriter117` where a exception would leave a partially written and invalid dta file (GH23573)
- Bug in `DataFrame.to_stata()` and `pandas.io.stata.StataWriter117` that produced invalid files when using strLs with non-ASCII characters (GH23573)
- Bug in `HDFStore` that caused it to raise `ValueError` when reading a Dataframe in Python 3 from fixed format written in Python 2 (GH24510)
- Bug in `DataFrame.to_string()` and more generally in the floating `repr` formatter. Zeros were not trimmed if `inf` was present in a columns while it was the case with NA values. Zeros are now trimmed as in the presence of NA (GH24861).
- Bug in the `repr` when truncating the number of columns and having a wide last column (GH24849).

1.7.14 Plotting

- Bug in `DataFrame.plot.scatter()` and `DataFrame.plot.hexbin()` caused x-axis label and ticklabels to disappear when colorbar was on in IPython inline backend (GH10611, GH10678, and GH20455)
- Bug in plotting a Series with datetimes using `matplotlib.axes.Axes.scatter()` (GH22039)
- Bug in `DataFrame.plot.bar()` caused bars to use multiple colors instead of a single one (GH20585)
- Bug in validating color parameter caused extra color to be appended to the given color array. This happened to multiple plotting functions using matplotlib. (GH20726)

1.7.15 Groupby/Resample/Rolling

- Bug in `pandas.core.window.Rolling.min()` and `pandas.core.window.Rolling.max()` with `closed='left'`, a datetime-like index and only one entry in the series leading to segfault (GH24718)
- Bug in `pandas.core.groupby.GroupBy.first()` and `pandas.core.groupby.GroupBy.last()` with `as_index=False` leading to the loss of timezone information (GH15884)
- Bug in `DateFrame.resample()` when downsampling across a DST boundary (GH8531)
- Bug in date anchoring for `DateFrame.resample()` with offset `Day` when n > 1 (GH24127)

- Bug where `ValueError` is wrongly raised when calling `count()` method of a `SeriesGroupBy` when the grouping variable only contains NaNs and numpy version < 1.13 (GH21956).
- Multiple bugs in *pandas.core.window.Rolling.min()* with `closed='left'` and a datetime-like index leading to incorrect results and also segfault. (GH21704)
- Bug in *pandas.core.resample.Resampler.apply()* when passing postiional arguments to applied func (GH14615).
- Bug in *Series.resample()* when passing `numpy.timedelta64` to `loffset` kwarg (GH7687).
- Bug in *pandas.core.resample.Resampler.asfreq()* when frequency of `TimedeltaIndex` is a subperiod of a new frequency (GH13022).
- Bug in `pandas.core.groupby.SeriesGroupBy.mean()` when values were integral but could not fit inside of int64, overflowing instead. (GH22487)
- `pandas.core.groupby.RollingGroupby.agg()` and `pandas.core.groupby.ExpandingGroupby.agg()` now support multiple aggregation functions as parameters (GH15072)
- Bug in *DataFrame.resample()* and *Series.resample()* when resampling by a weekly offset (`'W'`) across a DST transition (GH9119, GH21459)
- Bug in *DataFrame.expanding()* in which the `axis` argument was not being respected during aggregations (GH23372)
- Bug in *pandas.core.groupby.GroupBy.transform()* which caused missing values when the input function can accept a *DataFrame* but renames it (GH23455).
- Bug in *pandas.core.groupby.GroupBy.nth()* where column order was not always preserved (GH20760)
- Bug in *pandas.core.groupby.GroupBy.rank()* with `method='dense'` and `pct=True` when a group has only one member would raise a `ZeroDivisionError` (GH23666).
- Calling *pandas.core.groupby.GroupBy.rank()* with empty groups and `pct=True` was raising a `ZeroDivisionError` (GH22519)
- Bug in *DataFrame.resample()* when resampling `NaT` in `TimeDeltaIndex` (GH13223).
- Bug in *DataFrame.groupby()* did not respect the `observed` argument when selecting a column and instead always used `observed=False` (GH23970)
- Bug in `pandas.core.groupby.SeriesGroupBy.pct_change()` or *pandas.core.groupby.DataFrameGroupBy.pct_change()* would previously work across groups when calculating the percent change, where it now correctly works per group (GH21200, GH21235).
- Bug preventing hash table creation with very large number (2^32) of rows (GH22805)
- Bug in groupby when grouping on categorical causes `ValueError` and incorrect grouping if `observed=True` and `nan` is present in categorical column (GH24740, GH21151).

1.7.16 Reshaping

- Bug in *pandas.concat()* when joining resampled DataFrames with timezone aware index (GH13783)
- Bug in *pandas.concat()* when joining only *Series* the *names* argument of *concat* is no longer ignored (GH23490)
- Bug in *Series.combine_first()* with `datetime64[ns, tz]` dtype which would return tz-naive result (GH21469)
- Bug in *Series.where()* and *DataFrame.where()* with `datetime64[ns, tz]` dtype (GH21546)

- Bug in *DataFrame.where()* with an empty DataFrame and empty cond having non-bool dtype (GH21947)
- Bug in *Series.mask()* and *DataFrame.mask()* with list conditionals (GH21891)
- Bug in *DataFrame.replace()* raises RecursionError when converting OutOfBounds datetime64[ns, tz] (GH20380)
- *pandas.core.groupby.GroupBy.rank()* now raises a ValueError when an invalid value is passed for argument na_option (GH22124)
- Bug in *get_dummies()* with Unicode attributes in Python 2 (GH22084)
- Bug in *DataFrame.replace()* raises RecursionError when replacing empty lists (GH22083)
- Bug in *Series.replace()* and *DataFrame.replace()* when dict is used as the to_replace value and one key in the dict is is another key's value, the results were inconsistent between using integer key and using string key (GH20656)
- Bug in *DataFrame.drop_duplicates()* for empty DataFrame which incorrectly raises an error (GH20516)
- Bug in *pandas.wide_to_long()* when a string is passed to the stubnames argument and a column name is a substring of that stubname (GH22468)
- Bug in *merge()* when merging datetime64[ns, tz] data that contained a DST transition (GH18885)
- Bug in *merge_asof()* when merging on float values within defined tolerance (GH22981)
- Bug in *pandas.concat()* when concatenating a multicolumn DataFrame with tz-aware data against a DataFrame with a different number of columns (GH22796)
- Bug in *merge_asof()* where confusing error message raised when attempting to merge with missing values (GH23189)
- Bug in *DataFrame.nsmallest()* and *DataFrame.nlargest()* for dataframes that have a *MultiIndex* for columns (GH23033).
- Bug in *pandas.melt()* when passing column names that are not present in DataFrame (GH23575)
- Bug in *DataFrame.append()* with a *Series* with a dateutil timezone would raise a TypeError (GH23682)
- Bug in *Series* construction when passing no data and dtype=str (GH22477)
- Bug in *cut()* with bins as an overlapping IntervalIndex where multiple bins were returned per item instead of raising a ValueError (GH23980)
- Bug in *pandas.concat()* when joining Series datetimetz with Series category would lose timezone (GH23816)
- Bug in *DataFrame.join()* when joining on partial MultiIndex would drop names (GH20452).
- *DataFrame.nlargest()* and *DataFrame.nsmallest()* now returns the correct n values when keep != 'all' also when tied on the first columns (GH22752)
- Constructing a DataFrame with an index argument that wasn't already an instance of Index was broken (GH22227).
- Bug in *DataFrame* prevented list subclasses to be used to construction (GH21226)
- Bug in *DataFrame.unstack()* and *DataFrame.pivot_table()* returning a missleading error message when the resulting DataFrame has more elements than int32 can handle. Now, the error message is improved, pointing towards the actual problem (GH20601)
- Bug in *DataFrame.unstack()* where a ValueError was raised when unstacking timezone aware values (GH18338)

- Bug in *DataFrame.stack()* where timezone aware values were converted to timezone naive values (GH19420)
- Bug in *merge_asof()* where a `TypeError` was raised when `by_col` were timezone aware values (GH21184)
- Bug showing an incorrect shape when throwing error during `DataFrame` construction. (GH20742)

1.7.17 Sparse

- Updating a boolean, datetime, or timedelta column to be Sparse now works (GH22367)
- Bug in *Series.to_sparse()* with Series already holding sparse data not constructing properly (GH22389)
- Providing a `sparse_index` to the SparseArray constructor no longer defaults the na-value to `np.nan` for all dtypes. The correct na_value for `data.dtype` is now used.
- Bug in `SparseArray.nbytes` under-reporting its memory usage by not including the size of its sparse index.
- Improved performance of *Series.shift()* for non-NA `fill_value`, as values are no longer converted to a dense array.
- Bug in `DataFrame.groupby` not including `fill_value` in the groups for non-NA `fill_value` when grouping by a sparse column (GH5078)
- Bug in unary inversion operator (`~`) on a `SparseSeries` with boolean values. The performance of this has also been improved (GH22835)
- Bug in `SparseArary.unique()` not returning the unique values (GH19595)
- Bug in `SparseArray.nonzero()` and `SparseDataFrame.dropna()` returning shifted/incorrect results (GH21172)
- Bug in *DataFrame.apply()* where dtypes would lose sparseness (GH23744)
- Bug in *concat()* when concatenating a list of *Series* with all-sparse values changing the `fill_value` and converting to a dense Series (GH24371)

1.7.18 Style

- *background_gradient()* now takes a `text_color_threshold` parameter to automatically lighten the text color based on the luminance of the background color. This improves readability with dark background colors without the need to limit the background colormap range. (GH21258)
- *background_gradient()* now also supports tablewise application (in addition to rowwise and columnwise) with `axis=None` (GH15204)
- *bar()* now also supports tablewise application (in addition to rowwise and columnwise) with `axis=None` and setting clipping range with `vmin` and `vmax` (GH21548 and GH21526). `NaN` values are also handled properly.

1.7.19 Build Changes

- Building pandas for development now requires `cython >= 0.28.2` (GH21688)
- Testing pandas now requires `hypothesis>=3.58`. You can find the Hypothesis docs here, and a pandas-specific introduction *in the contributing guide*. (GH22280)
- Building pandas on macOS now targets minimum macOS 10.9 if run on macOS 10.9 or above (GH23424)

1.7.20 Other

- Bug where C variables were declared with external linkage causing import errors if certain other C libraries were imported before Pandas. (GH24113)

1.8 Contributors

A total of 337 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- AJ Dyka +
- AJ Pryor, Ph.D +
- Aaron Critchley
- Adam Hooper
- Adam J. Stewart
- Adam Kim
- Adam Klimont +
- Addison Lynch +
- Alan Hogue +
- Alex Radu +
- Alex Rychyk
- Alex Strick van Linschoten +
- Alex Volkov +
- Alexander Buchkovsky
- Alexander Hess +
- Alexander Ponomaroff +
- Allison Browne +
- Aly Sivji
- Andrew
- Andrew Gross +
- Andrew Spott +
- Andy +
- Aniket uttam +
- Anjali2019 +
- Anjana S +
- Antti Kaihola +
- Anudeep Tubati +
- Arjun Sharma +
- Armin Varshokar

- Artem Bogachev
- ArtinSarraf +
- Barry Fitzgerald +
- Bart Aelterman +
- Ben James +
- Ben Nelson +
- Benjamin Grove +
- Benjamin Rowell +
- Benoit Paquet +
- Boris Lau +
- Brett Naul
- Brian Choi +
- C.A.M. Gerlach +
- Carl Johan +
- Chalmer Lowe
- Chang She
- Charles David +
- Cheuk Ting Ho
- Chris
- Chris Roberts +
- Christopher Whelan
- Chu Qing Hao +
- Da Cheezy Mobsta +
- Damini Satya
- Daniel Himmelstein
- Daniel Saxton +
- Darcy Meyer +
- DataOmbudsman
- David Arcos
- David Krych
- Dean Langsam +
- Diego Argueta +
- Diego Torres +
- Dobatymo +
- Doug Latornell +
- Dr. Irv

- Dylan Dmitri Gray +
- Eric Boxer +
- Eric Chea
- Erik +
- Erik Nilsson +
- Fabian Haase +
- Fabian Retkowski
- Fabien Aulaire +
- Fakabbir Amin +
- Fei Phoon +
- Fernando Margueirat +
- Florian Müller +
- Fábio Rosado +
- Gabe Fernando
- Gabriel Reid +
- Giftlin Rajaiah
- Gioia Ballin +
- Gjelt
- Gosuke Shibahara +
- Graham Inggs
- Guillaume Gay
- Guillaume Lemaitre +
- Hannah Ferchland
- Haochen Wu
- Hubert +
- HubertKl +
- HyunTruth +
- Iain Barr
- Ignacio Vergara Kausel +
- Irv Lustig +
- IsvenC +
- Jacopo Rota
- Jakob Jarmar +
- James Bourbeau +
- James Myatt +
- James Winegar +

- Jan Rudolph
- Jared Groves +
- Jason Kiley +
- Javad Noorbakhsh +
- Jay Offerdahl +
- Jeff Reback
- Jeongmin Yu +
- Jeremy Schendel
- Jerod Estapa +
- Jesper Dramsch +
- Jim Jeon +
- Joe Jevnik
- Joel Nothman
- Joel Ostblom +
- Jordi Contestí
- Jorge López Fueyo +
- Joris Van den Bossche
- Jose Quinones +
- Jose Rivera-Rubio +
- Josh
- Jun +
- Justin Zheng +
- Kaiqi Dong +
- Kalyan Gokhale
- Kang Yoosam +
- Karl Dunkle Werner +
- Karmanya Aggarwal +
- Kevin Markham +
- Kevin Sheppard
- Kimi Li +
- Koustav Samaddar +
- Krishna +
- Kristian Holsheimer +
- Ksenia Gueletina +
- Kyle Prestel +
- LJ +

- LeakedMemory +
- Li Jin +
- Licht Takeuchi
- Luca Donini +
- Luciano Viola +
- Mak Sze Chun +
- Marc Garcia
- Marius Potgieter +
- Mark Sikora +
- Markus Meier +
- Marlene Silva Marchena +
- Martin Babka +
- MatanCohe +
- Mateusz Woś +
- Mathew Topper +
- Matt Boggess +
- Matt Cooper +
- Matt Williams +
- Matthew Gilbert
- Matthew Roeschke
- Max Kanter
- Michael Odintsov
- Michael Silverstein +
- Michael-J-Ward +
- Mickaël Schoentgen +
- Miguel Sánchez de León Peque +
- Ming Li
- Mitar
- Mitch Negus
- Monson Shao +
- Moonsoo Kim +
- Mortada Mehyar
- Myles Braithwaite
- Nehil Jain +
- Nicholas Musolino +
- Nicolas Dickreuter +

- Nikhil Kumar Mengani +
- Nikoleta Glynatsi +
- Ondrej Kokes
- Pablo Ambrosio +
- Pamela Wu +
- Parfait G +
- Patrick Park +
- Paul
- Paul Ganssle
- Paul Reidy
- Paul van Mulbregt +
- Phillip Cloud
- Pietro Battiston
- Piyush Aggarwal +
- Prabakaran Kumaresshan +
- Pulkit Maloo
- Pyry Kovanen
- Rajib Mitra +
- Redonnet Louis +
- Rhys Parry +
- Rick +
- Robin
- Roei.r +
- RomainSa +
- Roman Imankulov +
- Roman Yurchak +
- Ruijing Li +
- Ryan +
- Ryan Nazareth +
- Rüdiger Busche +
- SEUNG HOON, SHIN +
- Sandrine Pataut +
- Sangwoong Yoon
- Santosh Kumar +
- Saurav Chakravorty +
- Scott McAllister +

- Sean Chan +
- Shadi Akiki +
- Shengpu Tang +
- Shirish Kadam +
- Simon Hawkins +
- Simon Riddell +
- Simone Basso
- Sinhrks
- Soyoun(Rose) Kim +
- Srinivas Reddy Thatiparthy ( ) +
- Stefaan Lippens +
- Stefano Cianciulli
- Stefano Miccoli +
- Stephen Childs
- Stephen Pascoe
- Steve Baker +
- Steve Cook +
- Steve Dower +
- Stéphan Taljaard +
- Sumin Byeon +
- Sören +
- Tamas Nagy +
- Tanya Jain +
- Tarbo Fukazawa
- Thein Oo +
- Thiago Cordeiro da Fonseca +
- Thierry Moisan
- Thiviyan Thanapalasingam +
- Thomas Lentali +
- Tim D. Smith +
- Tim Swast
- Tom Augspurger
- Tomasz Kluczkowski +
- Tony Tao +
- Triple0 +
- Troels Nielsen +

- Tuhin Mahmud +
- Tyler Reddy +
- Uddeshya Singh
- Uwe L. Korn +
- Vadym Barda +
- Varad Gunjal +
- Victor Maryama +
- Victor Villas
- Vincent La
- Vitória Helena +
- Vu Le
- Vyom Jain +
- Weiwen Gu +
- Wenhuan
- Wes Turner
- Wil Tan +
- William Ayd
- Yeojin Kim +
- Yitzhak Andrade +
- Yuecheng Wu +
- Yuliya Dovzhenko +
- Yury Bayda +
- Zac Hatfield-Dodds +
- aberres +
- aeltanawy +
- ailchau +
- alimcmaster1
- alphaCTzo7G +
- amphy +
- araraonline +
- azure-pipelines[bot] +
- benarthur91 +
- bk521234 +
- cgangwar11 +
- chris-b1
- cxl923cc +

- dahlbaek +
- dannyhyunkim +
- darke-spirits +
- david-liu-brattle-1
- davidmvalente +
- deflatSOCO
- doosik_bae +
- dylanchase +
- eduardo naufel schettino +
- euri10 +
- evangelineliu +
- fengyqf +
- fjdiod
- fl4p +
- fleimgruber +
- gfyoung
- h-vetinari
- harisbal +
- henriqueribeiro +
- himanshu awasthi
- hongshaoyang +
- igorfassen +
- jalazbe +
- jbrockmendel
- jh-wu +
- justinchan23 +
- louispotok
- marcosrullan +
- miker985
- nicolab100 +
- nprad
- nsuresh +
- ottiP
- pajachiet +
- raguiar2 +
- ratijas +

- realead +
- robbuckley +
- saurav2608 +
- sideeye +
- ssikdar1
- svenharris +
- syutbai +
- testvinder +
- thatneat
- tmnhat2001
- tomascassidy +
- tomneep
- topper-123
- vkk800 +
- winlu +
- ym-pett +
- yrhooke +
- ywpark1 +
- zertrin
- zhezherun +

CHAPTER

TWO

INSTALLATION

The easiest way to install pandas is to install it as part of the Anaconda distribution, a cross platform distribution for data analysis and scientific computing. This is the recommended installation method for most users.

Instructions for installing from source, PyPI, ActivePython, various Linux distributions, or a development version are also provided.

2.1 Plan for dropping Python 2.7

The Python core team plans to stop supporting Python 2.7 on January 1st, 2020. In line with NumPy's plans, all pandas releases through December 31, 2018 will support Python 2.

The 0.24.x feature release will be the last release to support Python 2. The released package will continue to be available on PyPI and through conda.

> Starting **January 1, 2019**, all new feature releases (> 0.24) will be Python 3 only.

If there are people interested in continued support for Python 2.7 past December 31, 2018 (either backporting bug fixes or funding) please reach out to the maintainers on the issue tracker.

For more information, see the Python 3 statement and the Porting to Python 3 guide.

2.2 Python version support

Officially Python 2.7, 3.5, 3.6, and 3.7.

2.3 Installing pandas

2.3.1 Installing with Anaconda

Installing pandas and the rest of the NumPy and SciPy stack can be a little difficult for inexperienced users.

The simplest way to install not only pandas, but Python and the most popular packages that make up the SciPy stack (IPython, NumPy, Matplotlib, …) is with Anaconda, a cross-platform (Linux, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

After running the installer, the user will have access to pandas and the rest of the SciPy stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for Anaconda can be found here.

A full list of the packages available as part of the Anaconda distribution can be found here.

Another advantage to installing Anaconda is that you don't need admin rights to install it. Anaconda can install in the user's home directory, which makes it trivial to delete Anaconda if you decide (just delete that folder).

2.3.2 Installing with Miniconda

The previous section outlined how to get pandas installed as part of the Anaconda distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with Miniconda may be a better solution.

Conda is the package manager that the Anaconda distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

Miniconda allows you to create a minimal self contained Python installation, and then use the Conda command to install additional packages.

First you will need Conda to be installed and downloading and running the Miniconda will do this for you. The installer can be found here

The next step is to create a new conda environment. A conda environment is like a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:

```
source activate name_of_my_env
```

On Windows the command is:

```
activate name_of_my_env
```

The final step required is to install pandas. This can be done with the following command:

```
conda install pandas
```

To install a specific pandas version:

```
conda install pandas=0.20.3
```

To install other packages, IPython for example:

```
conda install ipython
```

To install the full Anaconda distribution:

```
conda install anaconda
```

If you need packages that are available to pip but not conda, then install pip, and then use pip to install those packages:

```
conda install pip
pip install django
```

2.3.3 Installing from PyPI

pandas can be installed via pip from PyPI.

```
pip install pandas
```

2.3.4 Installing with ActivePython

Installation instructions for ActivePython can be found here. Versions 2.7 and 3.5 include pandas.

2.3.5 Installing using your Linux distribution's package manager.

The commands in this table will install pandas for Python 3 from your distribution. To install pandas for Python 2, you may need to use the `python-pandas` package.

Distribution	Status	Download / Repository Link	Install method
Debian	stable	official Debian repository	`sudo apt-get install python3-pandas`
Debian & Ubuntu	unstable (latest packages)	NeuroDebian	`sudo apt-get install python3-pandas`
Ubuntu	stable	official Ubuntu repository	`sudo apt-get install python3-pandas`
OpenSuse	stable	OpenSuse Repository	`zypper in python3-pandas`
Fedora	stable	official Fedora repository	`dnf install python3-pandas`
Centos/RHEL	stable	EPEL repository	`yum install python3-pandas`

However, the packages in the linux package managers are often a few versions behind, so to get the newest version of pandas, it's recommended to install using the `pip` or `conda` methods described above.

2.3.6 Installing from source

See the *contributing guide* for complete instructions on building from the git source tree. Further, see *creating a development environment* if you wish to create a *pandas* development environment.

2.4 Running the test suite

pandas is equipped with an exhaustive set of unit tests, covering about 97% of the code base as of this writing. To run it on your machine to verify that everything is working (and that you have all of the dependencies, soft and hard, installed), make sure you have pytest >= 3.6 and Hypothesis >= 3.58, then run:

```
>>> pd.test()
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-
↪packages\pandas
============================= test session starts =============================
```

(continues on next page)

(continued from previous page)

```
platform win32 -- Python 3.6.2, pytest-3.6.0, py-1.4.34, pluggy-0.4.0
rootdir: C:\Users\TP\Documents\Python\pandasdev\pandas, inifile: setup.cfg
collected 12145 items / 3 skipped

..................................................................S......
........S................................................................
.........................................................................

=================== 12130 passed, 12 skipped in 368.339 seconds =====================
```

2.5 Dependencies

- setuptools: 24.2.0 or higher
- NumPy: 1.12.0 or higher
- python-dateutil: 2.5.0 or higher
- pytz

2.5.1 Recommended Dependencies

- numexpr: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups. If installed, must be Version 2.6.1 or higher.
- bottleneck: for accelerating certain types of `nan` evaluations. `bottleneck` uses specialized cython routines to achieve large speedups. If installed, must be Version 1.2.0 or higher.

Note: You are highly encouraged to install these libraries, as they provide speed improvements, especially when working with large data sets.

2.5.2 Optional Dependencies

- Cython: Only necessary to build development version. Version 0.28.2 or higher.
- SciPy: miscellaneous statistical functions, Version 0.18.1 or higher
- xarray: pandas like handling for > 2 dims, needed for converting Panels to xarray objects. Version 0.7.0 or higher is recommended.
- PyTables: necessary for HDF5-based storage, Version 3.4.2 or higher
- pyarrow (>= 0.9.0): necessary for feather-based storage.
- Apache Parquet, either pyarrow (>= 0.7.0) or fastparquet (>= 0.2.1) for parquet-based storage. The snappy and brotli are available for compression support.
- SQLAlchemy: for SQL database support. Version 0.8.1 or higher recommended. Besides SQLAlchemy, you also need a database specific driver. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs. Some common drivers are:
  - psycopg2: for PostgreSQL
  - pymysql: for MySQL.

  - SQLite: for SQLite, this is included in Python's standard library by default.
- matplotlib: for plotting, Version 2.0.0 or higher.
- For Excel I/O:
  - xlrd/xlwt: Excel reading (xlrd), version 1.0.0 or higher required, and writing (xlwt)
  - openpyxl: openpyxl version 2.4.0 for writing .xlsx files (xlrd >= 0.9.0)
  - XlsxWriter: Alternative Excel writer
- Jinja2: Template engine for conditional HTML formatting.
- s3fs: necessary for Amazon S3 access (s3fs >= 0.0.7).
- blosc: for msgpack compression using `blosc`
- gcsfs: necessary for Google Cloud Storage access (gcsfs >= 0.1.0).
- One of qtpy (requires PyQt or PySide), PyQt5, PyQt4, pygtk, xsel, or xclip: necessary to use *`read_clipboard()`*. Most package managers on Linux distributions will have `xclip` and/or `xsel` immediately available for installation.
- pandas-gbq: for Google BigQuery I/O. (pandas-gbq >= 0.8.0)
- Backports.lzma: Only for Python 2, for writing to and/or reading from an xz compressed DataFrame in CSV; Python 3 support is built into the standard library.
- One of the following combinations of libraries is needed to use the top-level *`read_html()`* function:

  Changed in version 0.23.0.

  Note: If using BeautifulSoup4 a minimum version of 4.2.1 is required

  - BeautifulSoup4 and html5lib (Any recent version of html5lib is okay.)
  - BeautifulSoup4 and lxml
  - BeautifulSoup4 and html5lib and lxml
  - Only lxml, although see *HTML Table Parsing* for reasons as to why you should probably **not** take this approach.

  Warning:

  - if you install BeautifulSoup4 you must install either lxml or html5lib or both. *`read_html()`* will **not** work with *only* BeautifulSoup4 installed.
  - You are highly encouraged to read *HTML Table Parsing gotchas*. It explains issues surrounding the installation and usage of the above three libraries.

  Note:

  - if you're on a system with `apt-get` you can do

    ```
    sudo apt-get build-dep python-lxml
    ```

    to get the necessary dependencies for installation of lxml. This will prevent further headaches down the line.

Note: Without the optional dependencies, many useful features will not work. Hence, it is highly recommended that you install these. A packaged distribution like Anaconda, ActivePython (version 2.7 or 3.5), or Enthought Canopy may be worth considering.

CHAPTER

THREE

GETTING STARTED

3.1 Package overview

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis / manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, `Series` (1-dimensional) and `DataFrame` (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, `DataFrame` provides everything that R's `data.frame` provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let *Series*, *DataFrame*, etc. automatically align the data for you in computations
- Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets
- Intuitive **merging** and **joining** data sets
- Flexible **reshaping** and pivoting of data sets
- **Hierarchical** labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**

- **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

3.1.1 Data Structures

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

Why more than one data structure?

The best way to think about the pandas data structures is as flexible containers for lower dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using ndarrays to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing functions; axes are considered more or less equivalent (except when C- or Fortran-contiguousness matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set there is likely to be a “right” way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the DataFrame thus results in more readable code:

```
for col in df.columns:
    series = df[col]
    # do something with series
```

3.1.2 Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to **favor immutability** where sensible.

3.1.3 Getting Support

The first stop for pandas issues and ideas is the Github Issue Tracker. If you have a general question, pandas community experts can answer through Stack Overflow.

3.1.4 Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to all of our contributors.

If you're interested in contributing, please visit the *contributing guide*.

pandas is a NumFOCUS sponsored project. This will help ensure the success of development of pandas as a world-class open-source project, and makes it possible to donate to the project.

3.1.5 Project Governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in Project Governance documents. The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).

3.1.6 Development Team

The list of the Core Team members and more detailed information can be found on the people's page of the governance repo.

3.1.7 Institutional Partners

The information about current institutional partners can be found on pandas website page.

3.1.8 License

```
BSD 3-Clause License

Copyright (c) 2008-2012, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData␣
↪Development Team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.
```

(continues on next page)

(continued from previous page)

```
* Neither the name of the copyright holder nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

{{ header }}

3.2 10 Minutes to pandas

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*.

Customarily, we import as follows:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

3.2.1 Object Creation

See the *Data Structure Intro section*.

Creating a `Series` by passing a list of values, letting pandas create a default integer index:

```
In [3]: s = pd.Series([1, 3, 5, np.nan, 6, 8])

In [4]: s
Out[4]: 
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a `DataFrame` by passing a NumPy array, with a datetime index and labeled columns:

```
In [5]: dates = pd.date_range('20130101', periods=6)

In [6]: dates
Out[6]: 
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
```

(continues on next page)

(continued from previous page)

```
               dtype='datetime64[ns]', freq='D')

In [7]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))

In [8]: df
Out[8]: 
                   A         B         C         D
2013-01-01  0.768029  0.582268 -1.947263 -0.200429
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-03 -0.238464 -0.486944  0.015596  1.403566
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343
2013-01-05  1.092525  1.164866 -0.846108  0.152275
2013-01-06  1.001105  0.071785 -1.067411 -1.500314
```

Creating a `DataFrame` by passing a dict of objects that can be converted to series-like.

```
In [9]: df2 = pd.DataFrame({'A': 1.,
   ...:                     'B': pd.Timestamp('20130102'),
   ...:                     'C': pd.Series(1, index=list(range(4)), dtype='float32'),
   ...:                     'D': np.array([3] * 4, dtype='int32'),
   ...:                     'E': pd.Categorical(["test", "train", "test", "train"]),
   ...:                     'F': 'foo'})
   ...: 

In [10]: df2
Out[10]: 
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

The columns of the resulting `DataFrame` have different *dtypes*.

```
In [11]: df2.dtypes
Out[11]: 
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

If you're using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here's a subset of the attributes that will be completed:

```
In [12]: df2.<TAB>  # noqa: E225, E999
df2.A                  df2.bool
df2.abs                df2.boxplot
df2.add                df2.C
df2.add_prefix         df2.clip
df2.add_suffix         df2.clip_lower
df2.align              df2.clip_upper
df2.all                df2.columns
df2.any                df2.combine
```

(continues on next page)

(continued from previous page)

```
df2.append              df2.combine_first
df2.apply               df2.compound
df2.applymap            df2.consolidate
df2.D
```

As you can see, the columns A, B, C, and D are automatically tab completed. E is there as well; the rest of the attributes have been truncated for brevity.

3.2.2 Viewing Data

See the *Basics section*.

Here is how to view the top and bottom rows of the frame:

```
In [13]: df.head()
Out[13]: 
                   A         B         C         D
2013-01-01  0.768029  0.582268 -1.947263 -0.200429
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-03 -0.238464 -0.486944  0.015596  1.403566
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343
2013-01-05  1.092525  1.164866 -0.846108  0.152275

In [14]: df.tail(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343
2013-01-05  1.092525  1.164866 -0.846108  0.152275
2013-01-06  1.001105  0.071785 -1.067411 -1.500314
```

Display the index, columns:

```
In [15]: df.index
Out[15]: 
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [16]: df.columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['A', 'B', 'C', 'D'], dtype='object')
```

`DataFrame.to_numpy()` gives a NumPy representation of the underlying data. Note that his can be an expensive operation when your `DataFrame` has columns with different data types, which comes down to a fundamental difference between pandas and NumPy: **NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column**. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. This may end up being `object`, which requires casting every value to a Python object.

For `df`, our `DataFrame` of all floating-point values, `DataFrame.to_numpy()` is fast and doesn't require copying data.

```
In [17]: df.to_numpy()
Out[17]: 
```

(continues on next page)

(continued from previous page)

```
array([[ 0.76802879,  0.58226818, -1.94726304, -0.20042915],
       [-0.48838847, -0.00378929, -0.5903971 , -1.22647701],
       [-0.23846363, -0.48694401,  0.01559601,  1.40356558],
       [-0.27492472,  0.82333823, -0.76168133, -1.35734316],
       [ 1.09252516,  1.16486632, -0.84610785,  0.15227547],
       [ 1.00110534,  0.07178548, -1.06741091, -1.50031353]])
```

For `df2`, the `DataFrame` with multiple dtypes, `DataFrame.to_numpy()` is relatively expensive.

```
In [18]: df2.to_numpy()
Out[18]:
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']], dtype=object)
```

Note: `DataFrame.to_numpy()` does *not* include the index or column labels in the output.

`describe()` shows a quick statistic summary of your data:

```
In [19]: df.describe()
Out[19]:
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.309980  0.358587 -0.866211 -0.454787
std    0.718350  0.607389  0.644072  1.130481
min   -0.488388 -0.486944 -1.947263 -1.500314
25%   -0.265809  0.015104 -1.012085 -1.324627
50%    0.264783  0.327027 -0.803895 -0.713453
75%    0.942836  0.763071 -0.633218  0.064099
max    1.092525  1.164866  0.015596  1.403566
```

Transposing your data:

```
In [20]: df.T
Out[20]:
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A    0.768029   -0.488388   -0.238464   -0.274925    1.092525    1.001105
B    0.582268   -0.003789   -0.486944    0.823338    1.164866    0.071785
C   -1.947263   -0.590397    0.015596   -0.761681   -0.846108   -1.067411
D   -0.200429   -1.226477    1.403566   -1.357343    0.152275   -1.500314
```

Sorting by an axis:

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
                   D         C         B         A
2013-01-01 -0.200429 -1.947263  0.582268  0.768029
2013-01-02 -1.226477 -0.590397 -0.003789 -0.488388
2013-01-03  1.403566  0.015596 -0.486944 -0.238464
2013-01-04 -1.357343 -0.761681  0.823338 -0.274925
2013-01-05  0.152275 -0.846108  1.164866  1.092525
2013-01-06 -1.500314 -1.067411  0.071785  1.001105
```

Sorting by values:

```
In [22]: df.sort_values(by='B')
Out[22]:
                   A         B         C         D
2013-01-03 -0.238464 -0.486944  0.015596  1.403566
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-06  1.001105  0.071785 -1.067411 -1.500314
2013-01-01  0.768029  0.582268 -1.947263 -0.200429
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343
2013-01-05  1.092525  1.164866 -0.846108  0.152275
```

3.2.3 Selection

Note: While standard Python / Numpy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, `.at`, `.iat`, `.loc` and `.iloc`.

See the indexing documentation *Indexing and Selecting Data* and *MultiIndex / Advanced Indexing*.

Getting

Selecting a single column, which yields a `Series`, equivalent to `df.A`:

```
In [23]: df['A']
Out[23]:
2013-01-01    0.768029
2013-01-02   -0.488388
2013-01-03   -0.238464
2013-01-04   -0.274925
2013-01-05    1.092525
2013-01-06    1.001105
Freq: D, Name: A, dtype: float64
```

Selecting via `[]`, which slices the rows.

```
In [24]: df[0:3]
Out[24]:
                   A         B         C         D
2013-01-01  0.768029  0.582268 -1.947263 -0.200429
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-03 -0.238464 -0.486944  0.015596  1.403566

In [25]: df['20130102':'20130104']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-03 -0.238464 -0.486944  0.015596  1.403566
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343
```

Selection by Label

See more in *Selection by Label*.

For getting a cross section using a label:

```
In [26]: df.loc[dates[0]]
Out[26]:
A    0.768029
B    0.582268
C   -1.947263
D   -0.200429
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label:

```
In [27]: df.loc[:, ['A', 'B']]
Out[27]:
                   A         B
2013-01-01  0.768029  0.582268
2013-01-02 -0.488388 -0.003789
2013-01-03 -0.238464 -0.486944
2013-01-04 -0.274925  0.823338
2013-01-05  1.092525  1.164866
2013-01-06  1.001105  0.071785
```

Showing label slicing, both endpoints are *included*:

```
In [28]: df.loc['20130102':'20130104', ['A', 'B']]
Out[28]:
                   A         B
2013-01-02 -0.488388 -0.003789
2013-01-03 -0.238464 -0.486944
2013-01-04 -0.274925  0.823338
```

Reduction in the dimensions of the returned object:

```
In [29]: df.loc['20130102', ['A', 'B']]
Out[29]:
A   -0.488388
B   -0.003789
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], 'A']
Out[30]: 0.7680287850661025
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [31]: df.at[dates[0], 'A']
Out[31]: 0.7680287850661025
```

Selection by Position

See more in *Selection by Position*.

Select via the position of the passed integers:

```
In [32]: df.iloc[3]
Out[32]:
A   -0.274925
B    0.823338
C   -0.761681
D   -1.357343
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python:

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
                   A         B
2013-01-04 -0.274925  0.823338
2013-01-05  1.092525  1.164866
```

By lists of integer position locations, similar to the numpy/python style:

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
                   A         C
2013-01-02 -0.488388 -0.590397
2013-01-03 -0.238464  0.015596
2013-01-05  1.092525 -0.846108
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
                   A         B         C         D
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477
2013-01-03 -0.238464 -0.486944  0.015596  1.403566
```

For slicing columns explicitly:

```
In [36]: df.iloc[:, 1:3]
Out[36]:
                   B         C
2013-01-01  0.582268 -1.947263
2013-01-02 -0.003789 -0.590397
2013-01-03 -0.486944  0.015596
2013-01-04  0.823338 -0.761681
2013-01-05  1.164866 -0.846108
2013-01-06  0.071785 -1.067411
```

For getting a value explicitly:

```
In [37]: df.iloc[1, 1]
Out[37]: -0.0037892926810494899
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [38]: df.iat[1, 1]
Out[38]: -0.0037892926810494899
```

Boolean Indexing

Using a single column's values to select data.

```
In [39]: df[df.A > 0]
Out[39]:
                   A         B         C         D
2013-01-01  0.768029  0.582268 -1.947263 -0.200429
2013-01-05  1.092525  1.164866 -0.846108  0.152275
2013-01-06  1.001105  0.071785 -1.067411 -1.500314
```

Selecting values from a DataFrame where a boolean condition is met.

```
In [40]: df[df > 0]
Out[40]:
                   A         B         C         D
2013-01-01  0.768029  0.582268       NaN       NaN
2013-01-02       NaN       NaN       NaN       NaN
2013-01-03       NaN       NaN  0.015596  1.403566
2013-01-04       NaN  0.823338       NaN       NaN
2013-01-05  1.092525  1.164866       NaN  0.152275
2013-01-06  1.001105  0.071785       NaN       NaN
```

Using the `isin()` method for filtering:

```
In [41]: df2 = df.copy()

In [42]: df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']

In [43]: df2
Out[43]:
                   A         B         C         D      E
2013-01-01  0.768029  0.582268 -1.947263 -0.200429    one
2013-01-02 -0.488388 -0.003789 -0.590397 -1.226477    one
2013-01-03 -0.238464 -0.486944  0.015596  1.403566    two
2013-01-04 -0.274925  0.823338 -0.761681 -1.357343  three
2013-01-05  1.092525  1.164866 -0.846108  0.152275   four
2013-01-06  1.001105  0.071785 -1.067411 -1.500314  three

In [44]: df2[df2['E'].isin(['two', 'four'])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D     E
2013-01-03 -0.238464 -0.486944  0.015596  1.403566   two
2013-01-05  1.092525  1.164866 -0.846108  0.152275  four
```

Setting

Setting a new column automatically aligns the data by the indexes.

```
In [45]: s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range('20130102',␣
↪periods=6))

In [46]: s1
Out[46]:
2013-01-02    1
```

(continues on next page)

(continued from previous page)

```
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

In [47]: df['F'] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], 'A'] = 0
```

Setting values by position:

```
In [49]: df.iat[0, 1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, 'D'] = np.array([5] * len(df))
```

The result of the prior setting operations.

```
In [51]: df
Out[51]: 
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -1.947263  5  NaN
2013-01-02 -0.488388 -0.003789 -0.590397  5  1.0
2013-01-03 -0.238464 -0.486944  0.015596  5  2.0
2013-01-04 -0.274925  0.823338 -0.761681  5  3.0
2013-01-05  1.092525  1.164866 -0.846108  5  4.0
2013-01-06  1.001105  0.071785 -1.067411  5  5.0
```

A `where` operation with setting.

```
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]: 
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -1.947263 -5  NaN
2013-01-02 -0.488388 -0.003789 -0.590397 -5 -1.0
2013-01-03 -0.238464 -0.486944 -0.015596 -5 -2.0
2013-01-04 -0.274925 -0.823338 -0.761681 -5 -3.0
2013-01-05 -1.092525 -1.164866 -0.846108 -5 -4.0
2013-01-06 -1.001105 -0.071785 -1.067411 -5 -5.0
```

3.2.4 Missing Data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the *Missing Data section*.

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data.

```
In [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])

In [56]: df1.loc[dates[0]:dates[1], 'E'] = 1

In [57]: df1
Out[57]:
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000 -1.947263  5  NaN  1.0
2013-01-02 -0.488388 -0.003789 -0.590397  5  1.0  1.0
2013-01-03 -0.238464 -0.486944  0.015596  5  2.0  NaN
2013-01-04 -0.274925  0.823338 -0.761681  5  3.0  NaN
```

To drop any rows that have missing data.

```
In [58]: df1.dropna(how='any')
Out[58]:
                   A         B         C  D    F    E
2013-01-02 -0.488388 -0.003789 -0.590397  5  1.0  1.0
```

Filling missing data.

```
In [59]: df1.fillna(value=5)
Out[59]:
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000 -1.947263  5  5.0  1.0
2013-01-02 -0.488388 -0.003789 -0.590397  5  1.0  1.0
2013-01-03 -0.238464 -0.486944  0.015596  5  2.0  5.0
2013-01-04 -0.274925  0.823338 -0.761681  5  3.0  5.0
```

To get the boolean mask where values are `nan`.

```
In [60]: pd.isna(df1)
Out[60]:
                A      B      C      D      F      E
2013-01-01  False  False  False  False   True  False
2013-01-02  False  False  False  False  False  False
2013-01-03  False  False  False  False  False   True
2013-01-04  False  False  False  False  False   True
```

3.2.5 Operations

See the *Basic section on Binary Ops*.

Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A    0.181976
B    0.261543
C   -0.866211
D    5.000000
```

(continues on next page)

(continued from previous page)

```
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    0.763184
2013-01-02    0.983485
2013-01-03    1.258038
2013-01-04    1.557346
2013-01-05    2.082257
2013-01-06    2.001096
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension.

```
In [63]: s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)

In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
Freq: D, dtype: float64

In [65]: df.sub(s, axis='index')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C    D    F
2013-01-01       NaN       NaN       NaN  NaN  NaN
2013-01-02       NaN       NaN       NaN  NaN  NaN
2013-01-03 -1.238464 -1.486944 -0.984404  4.0  1.0
2013-01-04 -3.274925 -2.176662 -3.761681  2.0  0.0
2013-01-05 -3.907475 -3.835134 -5.846108  0.0 -1.0
2013-01-06       NaN       NaN       NaN  NaN  NaN
```

Apply

Applying functions to the data:

```
In [66]: df.apply(np.cumsum)
Out[66]:
                   A         B         C   D     F
2013-01-01  0.000000  0.000000 -1.947263   5   NaN
2013-01-02 -0.488388 -0.003789 -2.537660  10   1.0
2013-01-03 -0.726852 -0.490733 -2.522064  15   3.0
2013-01-04 -1.001777  0.332605 -3.283745  20   6.0
2013-01-05  0.090748  1.497471 -4.129853  25  10.0
2013-01-06  1.091854  1.569257 -5.197264  30  15.0
```

(continues on next page)

(continued from previous page)

```
In [67]: df.apply(lambda x: x.max() - x.min())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    1.580914
B    1.651810
C    1.962859
D    0.000000
F    4.000000
dtype: float64
```

Histogramming

See more at *Histogramming and Discretization*.

```
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))

In [69]: s
Out[69]:
0    1
1    5
2    0
3    3
4    4
5    3
6    3
7    0
8    4
9    6
dtype: int64

In [70]: s.value_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[70]
↪
3    3
4    2
0    2
6    1
5    1
1    1
dtype: int64
```

String Methods

Series is equipped with a set of string processing methods in the *str* attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in *str* generally uses regular expressions by default (and in some cases always uses them). See more at *Vectorized String Methods*.

```
In [71]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [72]: s.str.lower()
Out[72]:
0       a
1       b
```

(continues on next page)

(continued from previous page)

```
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

3.2.6 Merge

Concat

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the *Merging section*.

Concatenating pandas objects together with `concat()`:

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]:
          0         1         2         3
0  0.631496  1.236626 -0.222839 -0.498957
1  0.728976 -1.077731  0.448808 -1.028403
2 -0.024724 -0.503176  0.354288 -0.077188
3  0.249369 -0.040457 -0.258803  0.702885
4  1.732037  0.821932  1.224690 -0.516889
5 -1.993549  0.803785  1.809205 -0.920181
6  0.161959 -0.176957  1.987065  1.847001
7 -1.546473 -0.210836  2.254850 -0.241549
8  0.711518  2.195299 -0.226047 -1.539275
9  0.586976 -1.431953 -1.090199 -0.385503

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]

In [76]: pd.concat(pieces)
Out[76]:
          0         1         2         3
0  0.631496  1.236626 -0.222839 -0.498957
1  0.728976 -1.077731  0.448808 -1.028403
2 -0.024724 -0.503176  0.354288 -0.077188
3  0.249369 -0.040457 -0.258803  0.702885
4  1.732037  0.821932  1.224690 -0.516889
5 -1.993549  0.803785  1.809205 -0.920181
6  0.161959 -0.176957  1.987065  1.847001
7 -1.546473 -0.210836  2.254850 -0.241549
8  0.711518  2.195299 -0.226047 -1.539275
9  0.586976 -1.431953 -1.090199 -0.385503
```

Join

SQL style merges. See the *Database style joining* section.

```
In [77]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [78]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [79]: left
Out[79]:
   key  lval
0  foo     1
1  foo     2

In [80]: right
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[80]:
   key  rval
0  foo     4
1  foo     5

In [81]: pd.merge(left, right, on='key')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Another example that can be given is:

```
In [82]: left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})

In [83]: right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

In [84]: left
Out[84]:
   key  lval
0  foo     1
1  bar     2

In [85]: right
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[85]:
   key  rval
0  foo     4
1  bar     5

In [86]: pd.merge(left, right, on='key')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

Append

Append rows to a dataframe. See the *Appending* section.

```
In [87]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])

In [88]: df
Out[88]:
          A         B         C         D
0  0.199412  2.294588  0.585654  0.509685
1  1.011544  0.835444  0.390855  0.402837
2 -1.413492  0.714828 -0.869333  1.118509
3 -0.645973 -0.436889  0.534250  1.425380
4 -0.422522  0.025048 -0.012703  0.878407
5  1.501616 -0.805316  0.134207 -1.160595
6  0.385089 -1.324173  1.203969  0.081638
7 -0.936488 -0.801764  0.053552  0.757385

In [89]: s = df.iloc[3]

In [90]: df.append(s, ignore_index=True)
Out[90]:
          A         B         C         D
0  0.199412  2.294588  0.585654  0.509685
1  1.011544  0.835444  0.390855  0.402837
2 -1.413492  0.714828 -0.869333  1.118509
3 -0.645973 -0.436889  0.534250  1.425380
4 -0.422522  0.025048 -0.012703  0.878407
5  1.501616 -0.805316  0.134207 -1.160595
6  0.385089 -1.324173  1.203969  0.081638
7 -0.936488 -0.801764  0.053552  0.757385
8 -0.645973 -0.436889  0.534250  1.425380
```

3.2.7 Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

See the *Grouping section*.

```
In [91]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ....:                          'foo', 'bar', 'foo', 'foo'],
   ....:                    'B': ['one', 'one', 'two', 'three',
   ....:                          'two', 'two', 'one', 'three'],
   ....:                    'C': np.random.randn(8),
   ....:                    'D': np.random.randn(8)})
   ....:

In [92]: df
Out[92]:
     A      B         C         D
0  foo    one  0.842386  1.568896
1  bar    one -0.012934 -0.124184
2  foo    two  0.180837  0.401103
3  bar  three  0.782382 -0.478031
4  foo    two -1.097331 -0.361864
```

(continues on next page)

(continued from previous page)

```
5  bar    two -0.955556 -1.633040
6  foo    one  0.103884  0.465614
7  foo  three  0.023263 -0.562763
```

Grouping and then applying the `sum()` function to the resulting groups.

```
In [93]: df.groupby('A').sum()
Out[93]:
            C         D
A
bar -0.186108 -2.235255
foo  0.053039  1.510985
```

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum` function.

```
In [94]: df.groupby(['A', 'B']).sum()
Out[94]:
                  C         D
A   B
bar one   -0.012934 -0.124184
    three  0.782382 -0.478031
    two   -0.955556 -1.633040
foo one    0.946270  2.034510
    three  0.023263 -0.562763
    two   -0.916494  0.039239
```

3.2.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

Stack

```
In [95]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
   ....:                      'foo', 'foo', 'qux', 'qux'],
   ....:                     ['one', 'two', 'one', 'two',
   ....:                      'one', 'two', 'one', 'two']]))
   ....:

In [96]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [97]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [98]: df2 = df[:4]

In [99]: df2
Out[99]:
                     A         B
first second
bar   one    -1.068846  1.566019
      two     1.802849  0.369416
baz   one    -1.179039  1.256525
      two    -1.275606 -0.219552
```

The `stack()` method "compresses" a level in the DataFrame's columns.

```
In [100]: stacked = df2.stack()

In [101]: stacked
Out[101]:
first  second
bar    one     A   -1.068846
               B    1.566019
       two     A    1.802849
               B    0.369416
baz    one     A   -1.179039
               B    1.256525
       two     A   -1.275606
               B   -0.219552
dtype: float64
```

With a "stacked" DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack()` is `unstack()`, which by default unstacks the **last level**:

```
In [102]: stacked.unstack()
Out[102]:
                     A         B
first second
bar   one    -1.068846  1.566019
      two     1.802849  0.369416
baz   one    -1.179039  1.256525
      two    -1.275606 -0.219552

In [103]: stacked.unstack(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
second        one       two
first
bar   A -1.068846  1.802849
      B  1.566019  0.369416
baz   A -1.179039 -1.275606
      B  1.256525 -0.219552

In [104]: stacked.unstack(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
first          bar       baz
second
one    A -1.068846 -1.179039
       B  1.566019  1.256525
two    A  1.802849 -1.275606
       B  0.369416 -0.219552
```

Pivot Tables

See the section on *Pivot Tables*.

```
In [105]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
   .....:                    'B': ['A', 'B', 'C'] * 4,
   .....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
   .....:                    'D': np.random.randn(12),
   .....:                    'E': np.random.randn(12)})
```

(continues on next page)

(continued from previous page)

```
   .....:

In [106]: df
Out[106]: 
        A  B    C         D         E
0     one  A  foo  1.364009 -0.877730
1     one  B  foo  0.547870 -0.529527
2     two  C  foo -1.058456 -1.636664
3   three  A  bar -1.296174 -0.057538
4     one  B  bar -0.073298 -0.569279
5     one  C  bar  0.293082  0.870111
6     two  A  foo -0.449975  1.734457
7   three  B  foo -1.229194 -0.102787
8     one  C  foo  0.087445 -0.720873
9     one  A  bar  1.276478  0.012643
10    two  B  bar  0.045970 -1.164115
11  three  C  bar  0.607962  0.863238
```

We can produce pivot tables from this data very easily:

```
In [107]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[107]: 
C             bar       foo
A     B                    
one   A  1.276478  1.364009
      B -0.073298  0.547870
      C  0.293082  0.087445
three A -1.296174       NaN
      B       NaN -1.229194
      C  0.607962       NaN
two   A       NaN -0.449975
      B  0.045970       NaN
      C       NaN -1.058456
```

3.2.9 Time Series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the *Time Series section*.

```
In [108]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [109]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [110]: ts.resample('5Min').sum()
Out[110]: 
2012-01-01    24384
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [111]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [112]: ts = pd.Series(np.random.randn(len(rng)), rng)
```

(continues on next page)

(continued from previous page)

```
In [113]: ts
Out[113]:
2012-03-06    0.600390
2012-03-07   -0.296105
2012-03-08   -0.645278
2012-03-09   -0.229840
2012-03-10   -0.448017
Freq: D, dtype: float64

In [114]: ts_utc = ts.tz_localize('UTC')

In [115]: ts_utc
Out[115]:
2012-03-06 00:00:00+00:00    0.600390
2012-03-07 00:00:00+00:00   -0.296105
2012-03-08 00:00:00+00:00   -0.645278
2012-03-09 00:00:00+00:00   -0.229840
2012-03-10 00:00:00+00:00   -0.448017
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [116]: ts_utc.tz_convert('US/Eastern')
Out[116]:
2012-03-05 19:00:00-05:00    0.600390
2012-03-06 19:00:00-05:00   -0.296105
2012-03-07 19:00:00-05:00   -0.645278
2012-03-08 19:00:00-05:00   -0.229840
2012-03-09 19:00:00-05:00   -0.448017
Freq: D, dtype: float64
```

Converting between time span representations:

```
In [117]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [118]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [119]: ts
Out[119]:
2012-01-31    1.314444
2012-02-29   -0.744069
2012-03-31   -1.273368
2012-04-30    0.906342
2012-05-31    0.938490
Freq: M, dtype: float64

In [120]: ps = ts.to_period()

In [121]: ps
Out[121]:
2012-01    1.314444
2012-02   -0.744069
2012-03   -1.273368
2012-04    0.906342
2012-05    0.938490
Freq: M, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [122]: ps.to_timestamp()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01    1.314444
2012-02-01   -0.744069
2012-03-01   -1.273368
2012-04-01    0.906342
2012-05-01    0.938490
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [123]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [124]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [125]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [126]: ts.head()
Out[126]:
1990-03-01 09:00   -0.200122
1990-06-01 09:00    0.693220
1990-09-01 09:00   -0.349043
1990-12-01 09:00   -0.466658
1991-03-01 09:00    1.524515
Freq: H, dtype: float64
```

3.2.10 Categoricals

pandas can include categorical data in a `DataFrame`. For full docs, see the *categorical introduction* and the *API documentation*.

```
In [127]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
   .....:                    "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
   .....:
```

Convert the raw grades to a categorical data type.

```
In [128]: df["grade"] = df["raw_grade"].astype("category")

In [129]: df["grade"]
Out[129]:
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

Rename the categories to more meaningful names (assigning to `Series.cat.categories` is inplace!).

```
In [130]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

Reorder the categories and simultaneously add the missing categories (methods under `Series .cat` return a new `Series` by default).

```
In [131]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                                "good", "very good"])
   .....:

In [132]: df["grade"]
Out[132]: 
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

Sorting is per order in the categories, not lexical order.

```
In [133]: df.sort_values(by="grade")
Out[133]: 
   id raw_grade      grade
5   6         e   very bad
1   2         b       good
2   3         b       good
0   1         a  very good
3   4         a  very good
4   5         a  very good
```

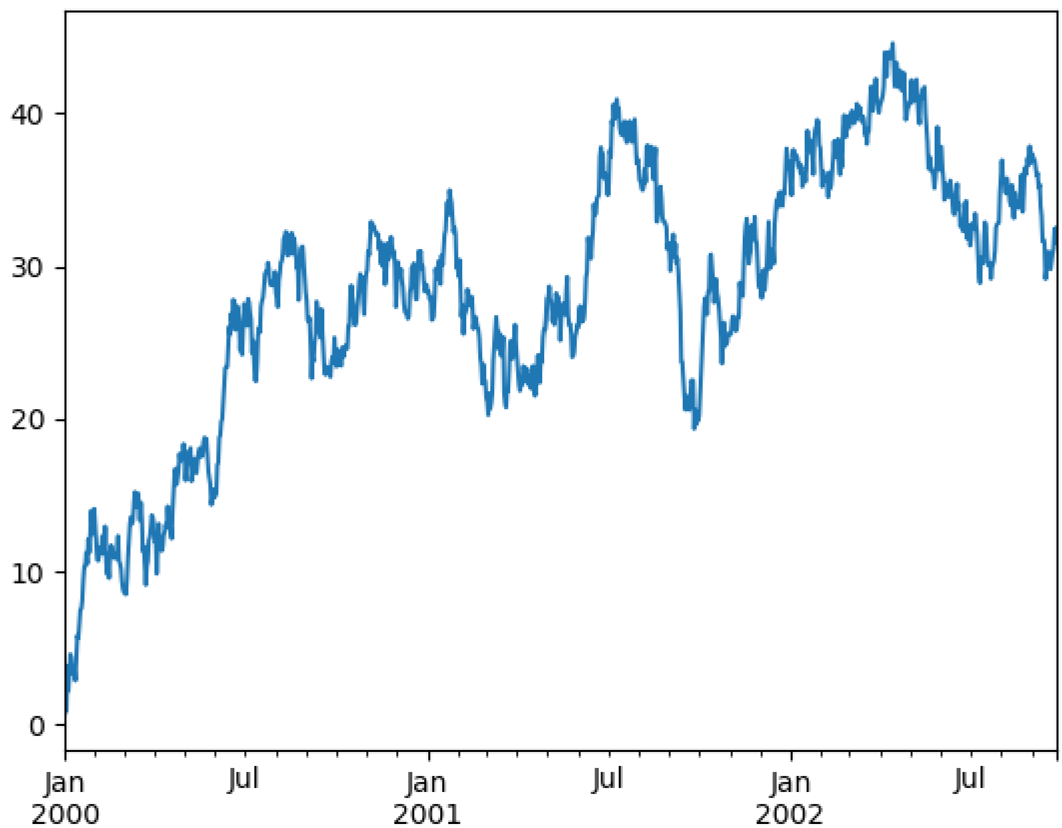
Grouping by a categorical column also shows empty categories.

```
In [134]: df.groupby("grade").size()
Out[134]: 
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

3.2.11 Plotting

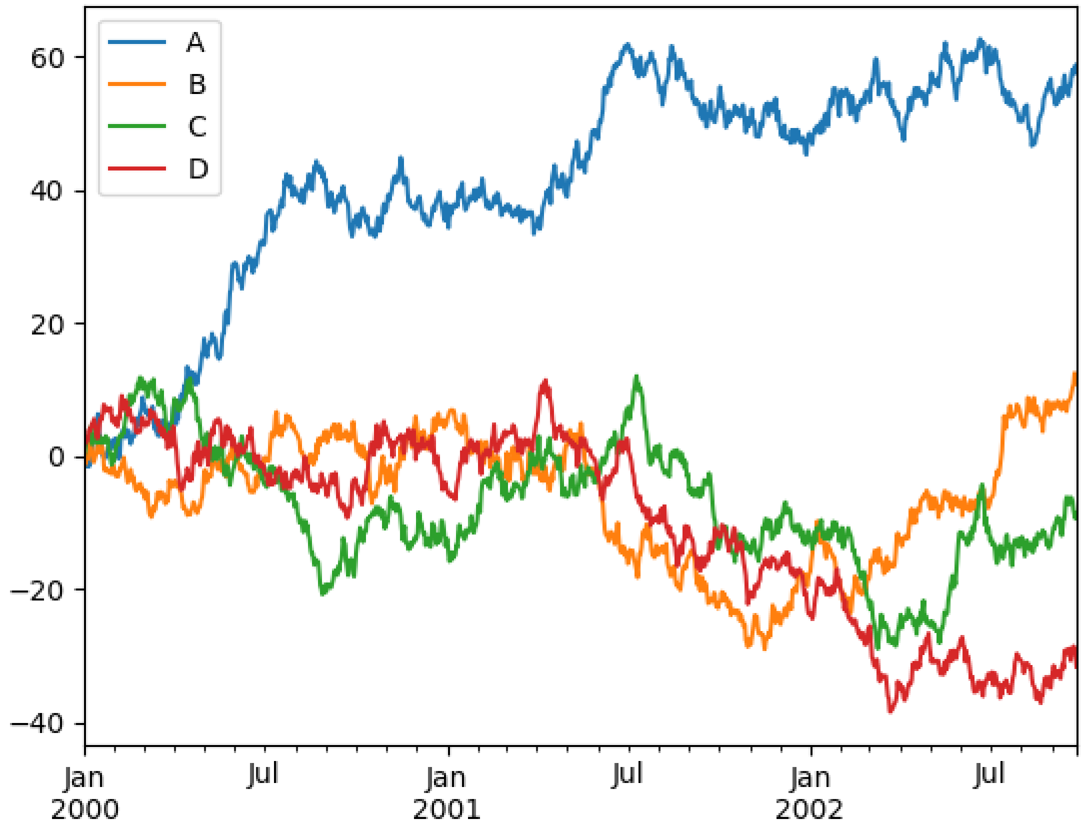
See the *Plotting* docs.

```
In [135]: ts = pd.Series(np.random.randn(1000),
   .....:                index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [136]: ts = ts.cumsum()

In [137]: ts.plot()
Out[137]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8dc65aae10>
```



On a DataFrame, the `plot()` method is a convenience to plot all of the columns with labels:

```
In [138]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
   .....:                   columns=['A', 'B', 'C', 'D'])
   .....:

In [139]: df = df.cumsum()

In [140]: plt.figure()
Out[140]: <Figure size 640x480 with 0 Axes>

In [141]: df.plot()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[141]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f8dc6415dd8>

In [142]: plt.legend(loc='best')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.legend.Legend at 0x7f8dc5d532e8>
```



3.2.12 Getting Data In/Out

CSV

Writing to a csv file.

```
In [143]: df.to_csv('foo.csv')
```

Reading from a csv file.

```
In [144]: pd.read_csv('foo.csv')
Out[144]:
     Unnamed: 0          A          B          C          D
0    2000-01-01  -1.659011   0.713900   0.022399   0.890292
1    2000-01-02  -0.974291   0.142524   0.678732   2.461393
2    2000-01-03  -0.019561  -1.196159   1.683430   2.637489
3    2000-01-04  -1.484574  -0.002263   2.380332   2.136855
4    2000-01-05  -1.097367  -0.369024   2.721312   2.681373
5    2000-01-06  -1.633227   0.812824   2.609760   3.931478
6    2000-01-07  -0.792972   2.071979   3.602726   3.732689
7    2000-01-08   0.506783   1.669192   4.398826   4.151986
8    2000-01-09  -0.113122   0.729355   1.760579   4.856881
```

(continues on next page)

(continued from previous page)

```
9    2000-01-10   1.979568  -0.769143   1.977370   4.420570
10   2000-01-11   2.517245  -0.936821   3.016042   5.556133
11   2000-01-12   3.441000  -0.412886   3.088331   5.036865
12   2000-01-13   3.776233   0.165964   3.494834   3.498450
13   2000-01-14   4.702290   1.294818   1.924773   2.655767
14   2000-01-15   6.292178   1.740214   2.306281   2.706634
15   2000-01-16   3.412076   1.549127   3.077083   3.496413
16   2000-01-17   3.021144   0.482532   2.140421   5.153778
17   2000-01-18   2.562915   0.950441   1.932492   5.705523
18   2000-01-19   0.468108   1.412616   2.130569   5.480885
19   2000-01-20  -0.055932   1.227314   2.557387   6.683110
20   2000-01-21  -0.002119  -1.537245   1.970671   7.706905
21   2000-01-22   0.309592  -2.260373   2.694873   6.699953
22   2000-01-23  -0.305449  -1.837849   2.987405   7.336121
23   2000-01-24   0.336863  -2.136834   2.119003   6.978600
24   2000-01-25   0.868558  -2.262168   2.602056   7.118693
25   2000-01-26   1.366189  -2.513342   0.751068   6.730901
26   2000-01-27   1.485792  -2.057377   0.639817   6.930417
27   2000-01-28   1.090456  -2.667635  -0.605780   7.223304
28   2000-01-29   2.266351  -1.930239  -1.351490   5.588364
29   2000-01-30   1.443513  -1.332835   0.057098   5.140447
..          ...        ...        ...        ...        ...
970  2002-08-28  55.019032   8.240932 -12.803911 -34.593062
971  2002-08-29  55.945681   6.430458 -13.889503 -33.232133
972  2002-08-30  55.824126   6.093906 -13.379117 -33.446813
973  2002-08-31  56.606539   6.745277 -13.438253 -34.812811
974  2002-09-01  54.555489   7.617210 -12.069331 -33.586094
975  2002-09-02  54.741712   7.973922 -11.550093 -32.322140
976  2002-09-03  54.324332   7.139671 -10.689092 -31.465478
977  2002-09-04  53.220408   7.400727 -10.874569 -32.015850
978  2002-09-05  52.663166   7.277322  -9.965704 -33.168278
979  2002-09-06  53.498462   7.823665  -9.893813 -32.612398
980  2002-09-07  52.958550   8.243191  -9.188644 -32.243882
981  2002-09-08  54.552911   8.022240  -9.890197 -32.543317
982  2002-09-09  55.208332   7.487259  -9.521482 -32.943181
983  2002-09-10  54.966418   7.187150 -10.688797 -33.392390
984  2002-09-11  53.701551   6.932330 -11.968916 -31.902797
985  2002-09-12  54.006298   7.608374 -10.938544 -31.179489
986  2002-09-13  56.359682   8.187546  -9.328342 -31.878881
987  2002-09-14  57.130009   8.082406  -9.308174 -29.202203
988  2002-09-15  55.737759   7.192985  -6.973547 -29.166748
989  2002-09-16  54.645679   6.687280  -6.192009 -29.057392
990  2002-09-17  54.336067   7.804372  -7.055312 -29.359995
991  2002-09-18  54.047158   8.304871  -7.341700 -29.799458
992  2002-09-19  55.628570   8.430707  -6.578065 -28.892582
993  2002-09-20  55.629679   8.778814  -6.222834 -30.663677
994  2002-09-21  57.284243   9.492078  -6.341743 -29.005189
995  2002-09-22  57.934048  10.241479  -6.787524 -29.344235
996  2002-09-23  57.237177  10.717876  -8.156142 -28.535133
997  2002-09-24  58.434254  12.433318  -8.905875 -29.240876
998  2002-09-25  58.228671  11.500508  -9.451469 -30.126563
999  2002-09-26  58.670623  10.919977  -8.716128 -31.622134

[1000 rows x 5 columns]
```

HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store.

```
In [145]: df.to_hdf('foo.h5', 'df')
```

Reading from a HDF5 Store.

```
In [146]: pd.read_hdf('foo.h5', 'df')
Out[146]:
                    A          B          C          D
2000-01-01  -1.659011   0.713900   0.022399   0.890292
2000-01-02  -0.974291   0.142524   0.678732   2.461393
2000-01-03  -0.019561  -1.196159   1.683430   2.637489
2000-01-04  -1.484574  -0.002263   2.380332   2.136855
2000-01-05  -1.097367  -0.369024   2.721312   2.681373
2000-01-06  -1.633227   0.812824   2.609760   3.931478
2000-01-07  -0.792972   2.071979   3.602726   3.732689
2000-01-08   0.506783   1.669192   4.398826   4.151986
2000-01-09  -0.113122   0.729355   1.760579   4.856881
2000-01-10   1.979568  -0.769143   1.977370   4.420570
2000-01-11   2.517245  -0.936821   3.016042   5.556133
2000-01-12   3.441000  -0.412886   3.088331   5.036865
2000-01-13   3.776233   0.165964   3.494834   3.498450
2000-01-14   4.702290   1.294818   1.924773   2.655767
2000-01-15   6.292178   1.740214   2.306281   2.706634
2000-01-16   3.412076   1.549127   3.077083   3.496413
2000-01-17   3.021144   0.482532   2.140421   5.153778
2000-01-18   2.562915   0.950441   1.932492   5.705523
2000-01-19   0.468108   1.412616   2.130569   5.480885
2000-01-20  -0.055932   1.227314   2.557387   6.683110
2000-01-21  -0.002119  -1.537245   1.970671   7.706905
2000-01-22   0.309592  -2.260373   2.694873   6.699953
2000-01-23  -0.305449  -1.837849   2.987405   7.336121
2000-01-24   0.336863  -2.136834   2.119003   6.978600
2000-01-25   0.868558  -2.262168   2.602056   7.118693
2000-01-26   1.366189  -2.513342   0.751068   6.730901
2000-01-27   1.485792  -2.057377   0.639817   6.930417
2000-01-28   1.090456  -2.667635  -0.605780   7.223304
2000-01-29   2.266351  -1.930239  -1.351490   5.588364
2000-01-30   1.443513  -1.332835   0.057098   5.140447
...               ...        ...        ...        ...
2002-08-28  55.019032   8.240932 -12.803911 -34.593062
2002-08-29  55.945681   6.430458 -13.889503 -33.232133
2002-08-30  55.824126   6.093906 -13.379117 -33.446813
2002-08-31  56.606539   6.745277 -13.438253 -34.812811
2002-09-01  54.555489   7.617210 -12.069331 -33.586094
2002-09-02  54.741712   7.973922 -11.550093 -32.322140
2002-09-03  54.324332   7.139671 -10.689092 -31.465478
2002-09-04  53.220408   7.400727 -10.874569 -32.015850
2002-09-05  52.663166   7.277322  -9.965704 -33.168278
2002-09-06  53.498462   7.823665  -9.893813 -32.612398
2002-09-07  52.958550   8.243191  -9.188644 -32.243882
2002-09-08  54.552911   8.022240  -9.890197 -32.543317
2002-09-09  55.208332   7.487259  -9.521482 -32.943181
2002-09-10  54.966418   7.187150 -10.688797 -33.392390
```

(continues on next page)

(continued from previous page)

```
2002-09-11  53.701551   6.932330 -11.968916 -31.902797
2002-09-12  54.006298   7.608374 -10.938544 -31.179489
2002-09-13  56.359682   8.187546  -9.328342 -31.878881
2002-09-14  57.130009   8.082406  -9.308174 -29.202203
2002-09-15  55.737759   7.192985  -6.973547 -29.166748
2002-09-16  54.645679   6.687280  -6.192009 -29.057392
2002-09-17  54.336067   7.804372  -7.055312 -29.359995
2002-09-18  54.047158   8.304871  -7.341700 -29.799458
2002-09-19  55.628570   8.430707  -6.578065 -28.892582
2002-09-20  55.629679   8.778814  -6.222834 -30.663677
2002-09-21  57.284243   9.492078  -6.341743 -29.005189
2002-09-22  57.934048  10.241479  -6.787524 -29.344235
2002-09-23  57.237177  10.717876  -8.156142 -28.535133
2002-09-24  58.434254  12.433318  -8.905875 -29.240876
2002-09-25  58.228671  11.500508  -9.451469 -30.126563
2002-09-26  58.670623  10.919977  -8.716128 -31.622134

[1000 rows x 4 columns]
```

Excel

Reading and writing to *MS Excel*.

Writing to an excel file.

```
In [147]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

Reading from an excel file.

```
In [148]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
Out[148]:
    Unnamed: 0         A         B         C         D
0   2000-01-01 -1.659011  0.713900  0.022399  0.890292
1   2000-01-02 -0.974291  0.142524  0.678732  2.461393
2   2000-01-03 -0.019561 -1.196159  1.683430  2.637489
3   2000-01-04 -1.484574 -0.002263  2.380332  2.136855
4   2000-01-05 -1.097367 -0.369024  2.721312  2.681373
5   2000-01-06 -1.633227  0.812824  2.609760  3.931478
6   2000-01-07 -0.792972  2.071979  3.602726  3.732689
7   2000-01-08  0.506783  1.669192  4.398826  4.151986
8   2000-01-09 -0.113122  0.729355  1.760579  4.856881
9   2000-01-10  1.979568 -0.769143  1.977370  4.420570
10  2000-01-11  2.517245 -0.936821  3.016042  5.556133
11  2000-01-12  3.441000 -0.412886  3.088331  5.036865
12  2000-01-13  3.776233  0.165964  3.494834  3.498450
13  2000-01-14  4.702290  1.294818  1.924773  2.655767
14  2000-01-15  6.292178  1.740214  2.306281  2.706634
15  2000-01-16  3.412076  1.549127  3.077083  3.496413
16  2000-01-17  3.021144  0.482532  2.140421  5.153778
17  2000-01-18  2.562915  0.950441  1.932492  5.705523
18  2000-01-19  0.468108  1.412616  2.130569  5.480885
19  2000-01-20 -0.055932  1.227314  2.557387  6.683110
20  2000-01-21 -0.002119 -1.537245  1.970671  7.706905
21  2000-01-22  0.309592 -2.260373  2.694873  6.699953
22  2000-01-23 -0.305449 -1.837849  2.987405  7.336121
```

(continues on next page)

(continued from previous page)

```
23  2000-01-24   0.336863  -2.136834   2.119003   6.978600
24  2000-01-25   0.868558  -2.262168   2.602056   7.118693
25  2000-01-26   1.366189  -2.513342   0.751068   6.730901
26  2000-01-27   1.485792  -2.057377   0.639817   6.930417
27  2000-01-28   1.090456  -2.667635  -0.605780   7.223304
28  2000-01-29   2.266351  -1.930239  -1.351490   5.588364
29  2000-01-30   1.443513  -1.332835   0.057098   5.140447
..         ...        ...        ...        ...        ...
970 2002-08-28  55.019032   8.240932 -12.803911 -34.593062
971 2002-08-29  55.945681   6.430458 -13.889503 -33.232133
972 2002-08-30  55.824126   6.093906 -13.379117 -33.446813
973 2002-08-31  56.606539   6.745277 -13.438253 -34.812811
974 2002-09-01  54.555489   7.617210 -12.069331 -33.586094
975 2002-09-02  54.741712   7.973922 -11.550093 -32.322140
976 2002-09-03  54.324332   7.139671 -10.689092 -31.465478
977 2002-09-04  53.220408   7.400727 -10.874569 -32.015850
978 2002-09-05  52.663166   7.277322  -9.965704 -33.168278
979 2002-09-06  53.498462   7.823665  -9.893813 -32.612398
980 2002-09-07  52.958550   8.243191  -9.188644 -32.243882
981 2002-09-08  54.552911   8.022240  -9.890197 -32.543317
982 2002-09-09  55.208332   7.487259  -9.521482 -32.943181
983 2002-09-10  54.966418   7.187150 -10.688797 -33.392390
984 2002-09-11  53.701551   6.932330 -11.968916 -31.902797
985 2002-09-12  54.006298   7.608374 -10.938544 -31.179489
986 2002-09-13  56.359682   8.187546  -9.328342 -31.878881
987 2002-09-14  57.130009   8.082406  -9.308174 -29.202203
988 2002-09-15  55.737759   7.192985  -6.973547 -29.166748
989 2002-09-16  54.645679   6.687280  -6.192009 -29.057392
990 2002-09-17  54.336067   7.804372  -7.055312 -29.359995
991 2002-09-18  54.047158   8.304871  -7.341700 -29.799458
992 2002-09-19  55.628570   8.430707  -6.578065 -28.892582
993 2002-09-20  55.629679   8.778814  -6.222834 -30.663677
994 2002-09-21  57.284243   9.492078  -6.341743 -29.005189
995 2002-09-22  57.934048  10.241479  -6.787524 -29.344235
996 2002-09-23  57.237177  10.717876  -8.156142 -28.535133
997 2002-09-24  58.434254  12.433318  -8.905875 -29.240876
998 2002-09-25  58.228671  11.500508  -9.451469 -30.126563
999 2002-09-26  58.670623  10.919977  -8.716128 -31.622134

[1000 rows x 5 columns]
```

3.2.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

```
>>> if pd.Series([False, True, False]):
...     print("I was true")
Traceback
    ...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *Comparisons* for an explanation and what to do.

See *Gotchas* as well.

3.3 Essential Basic Functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

```
In [1]: index = pd.date_range('1/1/2000', periods=8)

In [2]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   ...:                   columns=['A', 'B', 'C'])
   ...:

In [4]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   ...:               major_axis=pd.date_range('1/1/2000', periods=5),
   ...:               minor_axis=['A', 'B', 'C', 'D'])
   ...:
```

3.3.1 Head and Tail

To view a small sample of a Series or DataFrame object, use the *head()* and *tail()* methods. The default number of elements to display is five, but you may pass a custom number.

```
In [5]: long_series = pd.Series(np.random.randn(1000))

In [6]: long_series.head()
Out[6]:
0   -2.211372
1    0.974466
2   -2.006747
3   -0.410001
4   -0.078638
dtype: float64

In [7]: long_series.tail(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]:
↪
997   -0.196166
998    0.380733
999   -0.275874
dtype: float64
```

3.3.2 Attributes and Underlying Data

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- **Axis labels**
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*
  - **Panel**: *items*, *major_axis*, and *minor_axis*

Note, **these attributes can be safely assigned to**!

```
In [8]: df[:2]
Out[8]:
                   A         B         C
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929

In [9]: df.columns = [x.lower() for x in df.columns]

In [10]: df
Out[10]:
                   a         b         c
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929
2000-01-03  1.071804  0.721555 -0.706771
2000-01-04 -1.039575  0.271860 -0.424972
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427
2000-01-07  0.524988  0.404705  0.577046
2000-01-08 -1.715002 -1.039268 -0.370647
```

Pandas objects (*Index*, *Series*, *DataFrame*) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may *extend* NumPy's type system to add support for custom arrays (see *dtypes*).

To get the actual data inside a *Index* or *Series*, use the `.array` property

```
In [11]: s.array
Out[11]:
<PandasArray>
[ 0.46911229990718628, -0.28286334432866328,  -1.5090585031735124,
  -1.1356323710171934,   1.2121120250208506]
Length: 5, dtype: float64

In [12]: s.index.array
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<PandasArray>
['a', 'b', 'c', 'd', 'e']
Length: 5, dtype: object
```

array will always be an *ExtensionArray*. The exact details of what an *ExtensionArray* is and why pandas uses them is a bit beyond the scope of this introduction. See *dtypes* for more.

If you know you need a NumPy array, use *to_numpy()* or `numpy.asarray()`.

```
In [13]: s.to_numpy()
Out[13]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])

In [14]: np.asarray(s)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: array([ 0.4691,
↪ -0.2829, -1.5091, -1.1356,  1.2121])
```

When the Series or Index is backed by an *ExtensionArray*, *to_numpy()* may involve copying data and coercing values. See *dtypes* for more.

to_numpy() gives some control over the `dtype` of the resulting `numpy.ndarray`. For example, consider datetimes with timezones. NumPy doesn't have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype `numpy.ndarray` with *Timestamp* objects, each with the correct `tz`
2. A `datetime64[ns]` -dtype `numpy.ndarray`, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with `dtype=object`

```
In [15]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))

In [16]: ser.to_numpy(dtype=object)
Out[16]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')], dtype=object)
```

Or thrown away with `dtype='datetime64[ns]'`

```
In [17]: ser.to_numpy(dtype="datetime64[ns]")
Out[17]: array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],␣
↪dtype='datetime64[ns]')
```

Getting the "raw data" inside a *DataFrame* is possibly a bit more complex. When your `DataFrame` only has a single data type for all the columns, *DataFrame.to_numpy()* will return the underlying data:

```
In [18]: df.to_numpy()
Out[18]:
array([[-0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949],
       [ 1.0718,  0.7216, -0.7068],
       [-1.0396,  0.2719, -0.425 ],
       [ 0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 ],
       [-1.715 , -1.0393, -0.3706]])
```

If a DataFrame or Panel contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame's columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

Note: When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

In the past, pandas recommended *Series.values* or *DataFrame.values* for extracting the data from a Series or DataFrame. You'll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an *extension type*, it's unclear whether *Series.values* returns a NumPy array or the extension array. *Series.array* will always return an *ExtensionArray*, and will never copy data. *Series.to_numpy()* will always return a NumPy array, potentially at the cost of copying / coercing values.
2. When your DataFrame contains a mixture of data types, *DataFrame.values* may involve copying data and coercing values to a common dtype, a relatively expensive operation. *DataFrame.to_numpy()*, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

3.3.3 Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have `nans`.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

Operation	0.11.0 (ms)	Prior Version (ms)	Ratio to Prior
`df1 > df2`	13.32	125.35	0.1063
`df1 * df2`	21.71	36.63	0.5928
`df1 + df2`	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section *Recommended Dependencies* for more installation info.

These are both enabled to be used by default, you can control this by setting the options:

New in version 0.20.0.

```
pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)
```

3.3.4 Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

Matching / broadcasting behavior

DataFrame has the methods *`add()`*, *`sub()`*, *`mul()`*, *`div()`* and related functions *`radd()`*, *`rsub()`*, ... for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [19]: df = pd.DataFrame({
   ....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   ....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   ....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   ....:

In [20]: df
Out[20]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830
```

(continues on next page)

(continued from previous page)

```
In [21]: row = df.iloc[1]

In [22]: column = df['two']

In [23]: df.sub(row, axis='columns')
Out[23]:
        one       two     three
a  1.757280 -2.688953       NaN
b  0.000000  0.000000  0.000000
c  1.153738 -0.121396 -0.402113
d       NaN  0.507782 -2.065853

In [24]: df.sub(row, axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  1.757280 -2.688953       NaN
b  0.000000  0.000000  0.000000
c  1.153738 -0.121396 -0.402113
d       NaN  0.507782 -2.065853

In [25]: df.sub(column, axis='index')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one  two     three
a  3.043851  0.0       NaN
b -1.402381  0.0 -0.650888
c -0.127247  0.0 -0.931605
d       NaN  0.0 -3.224524

In [26]: df.sub(column, axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one  two     three
a  3.043851  0.0       NaN
b -1.402381  0.0 -0.650888
c -0.127247  0.0 -0.931605
d       NaN  0.0 -3.224524
```

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

```
In [27]: dfmi = df.copy()

In [28]: dfmi.index = pd.MultiIndex.from_tuples([(1, 'a'), (1, 'b'),
   ....:                                         (1, 'c'), (2, 'a')],
   ....:                                        names=['first', 'second'])
   ....:

In [29]: dfmi.sub(column, axis=0, level='second')
Out[29]:
                   one       two     three
first second
1     a       3.043851  0.000000       NaN
      b      -1.402381  0.000000 -0.650888
      c      -0.127247  0.000000 -0.931605
2     a            NaN  3.196734 -0.027789
```

With Panel, describing the matching behavior is a bit more difficult, so the arithmetic methods instead (and perhaps confusingly?) give you the option to specify the *broadcast axis*. For example, suppose we wished to demean the data over a particular axis. This can be accomplished by taking the mean over an axis and broadcasting over the same axis:

```
In [30]: major_mean = wp.mean(axis='major')

In [31]: major_mean
Out[31]:
      Item1     Item2
A -0.378069  0.675929
B -0.241429 -0.018080
C -0.597702  0.129006
D  0.204005  0.245570

In [32]: wp.sub(major_mean, axis='major')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

And similarly for `axis="items"` and `axis="minor"`.

Note: I could be convinced to make the **axis** argument in the DataFrame methods match the broadcasting behavior of Panel. Though it would require a transition period so users can change their code...

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [33]: s = pd.Series(np.arange(10))

In [34]: s
Out[34]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64

In [35]: div, rem = divmod(s, 3)

In [36]: div
Out[36]:
0    0
1    0
2    0
3    1
4    1
```

(continues on next page)

(continued from previous page)

```
5    1
6    2
7    2
8    2
9    3
dtype: int64

In [37]: rem
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[37]
↪
0    0
1    1
2    2
3    0
4    1
5    2
6    0
7    1
8    2
9    0
dtype: int64

In [38]: idx = pd.Index(np.arange(10))

In [39]: idx
Out[39]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [40]: div, rem = divmod(idx, 3)

In [41]: div
Out[41]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [42]: rem
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[42]:␣
↪Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

We can also do elementwise `divmod()`:

```
In [43]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

In [44]: div
Out[44]:
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64

In [45]: rem
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[45]
↪
```

(continues on next page)

(continued from previous page)

```
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int64
```

Missing data / operations with fill values

In Series and DataFrame, the arithmetic functions have the option of inputting a *fill_value*, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using `fillna` if you wish).

```
In [46]: df
Out[46]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [47]: df2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  1.400810 -1.643041  1.000000
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [48]: df + df2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  2.801620 -3.286083       NaN
b -0.712940  2.091822  0.790046
c  1.594536  1.849030 -0.014180
d       NaN  3.107386 -3.341661

In [49]: df.add(df2, fill_value=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  2.801620 -3.286083  1.000000
b -0.712940  2.091822  0.790046
c  1.594536  1.849030 -0.014180
d       NaN  3.107386 -3.341661
```

Flexible Comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

```
In [50]: df.gt(df2)
Out[50]:
     one    two  three
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [51]: df2.ne(df)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     one    two  three
a  False  False   True
b  False  False  False
c  False  False  False
d   True  False  False
```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These `boolean` objects can be used in indexing operations, see the section on *Boolean indexing*.

Boolean Reductions

You can apply the reductions: *empty*, *any()*, *all()*, and *bool()* to provide a way to summarize a boolean result.

```
In [52]: (df > 0).all()
Out[52]:
one      False
two      False
three    False
dtype: bool

In [53]: (df > 0).any()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[53]:
one      True
two      True
three    True
dtype: bool
```

You can reduce to a final boolean value.

```
In [54]: (df > 0).any().any()
Out[54]: True
```

You can test if a pandas object is empty, via the *empty* property.

```
In [55]: df.empty
Out[55]: False

In [56]: pd.DataFrame(columns=list('ABC')).empty
\\\\\\\\\\\\\\\\Out[56]: True
```

To evaluate single-element pandas objects in a boolean context, use the method *bool()*:

```
In [57]: pd.Series([True]).bool()
Out[57]: True

In [58]: pd.Series([False]).bool()
\\\\\\\\\\\\\\Out[58]: False

In [59]: pd.DataFrame([[True]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[59]: True

In [60]: pd.DataFrame([[False]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[60]: False
```

Warning: You might be tempted to do the following:

```
>>> if df:
...     pass
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.:

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.
↪all().
```

See *gotchas* for a more detailed discussion.

Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df + df` and `df * 2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df + df == df * 2).all()`. But in fact, this expression is False:

```
In [61]: df + df == df * 2
Out[61]:
     one   two  three
a   True  True  False
b   True  True   True
c   True  True   True
d  False  True   True

In [62]: (df + df == df * 2).all()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df + df == df * 2` contains some False values! This is because NaNs do not compare as equals:

```
In [63]: np.nan == np.nan
Out[63]: False
```

So, NDFrames (such as Series, DataFrames, and Panels) have an *equals()* method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [64]: (df + df).equals(df * 2)
Out[64]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [65]: df1 = pd.DataFrame({'col': ['foo', 0, np.nan]})

In [66]: df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2, 1, 0])

In [67]: df1.equals(df2)
Out[67]: False

In [68]: df1.equals(df2.sort_index())
\\\\\\\\\\\\\\\Out[68]: True
```

Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [69]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
Out[69]:
0     True
1    False
2    False
dtype: bool

In [70]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[70]: array([ True, False,␣
↪False], dtype=bool)
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [71]: pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
Out[71]:
0     True
1     True
2    False
dtype: bool

In [72]: pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[72]:
0     True
1     True
2    False
dtype: bool
```

Trying to compare `Index` or `Series` objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [73]: np.array([1, 2, 3]) == np.array([2])
Out[73]: array([False,  True, False], dtype=bool)
```

or it can return False if broadcasting can not be done:

```
In [74]: np.array([1, 2, 3]) == np.array([1, 2])
Out[74]: False
```

Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of "higher quality". However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [75]: df1 = pd.DataFrame({'A': [1., np.nan, 3., 5., np.nan],
   ....:                     'B': [np.nan, 2., 3., np.nan, 6.]})
   ....:

In [76]: df2 = pd.DataFrame({'A': [5., 2., 4., np.nan, 3., 7.],
   ....:                     'B': [np.nan, np.nan, 3., 4., 6., 8.]})
   ....:

In [77]: df1
Out[77]:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  3.0
3  5.0  NaN
4  NaN  6.0

In [78]: df2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[78]:␣
↪
     A    B
0  5.0  NaN
1  2.0  NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [79]: df1.combine_first(df2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
```

(continues on next page)

(continued from previous page)

```
4  3.0  6.0
5  7.0  8.0
```

General DataFrame Combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```
In [80]: def combiner(x, y):
   ....:     np.where(pd.isna(x), y, x)
   ....: df1.combine(df2, combiner)
   ....:
Out[80]:
     A    B
0  NaN  NaN
1  NaN  NaN
2  NaN  NaN
3  NaN  NaN
4  NaN  NaN
5  NaN  NaN
```

3.3.5 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*, and *Panel*. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like *ndarray.{sum, std, …}*, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: “index” (axis=0, default), “columns” (axis=1)
- **Panel**: “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

```
In [81]: df
Out[81]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [82]: df.mean(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
one      0.613869
two      0.470270
three   -0.427633
dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [83]: df.mean(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a   -0.121116
b    0.361488
c    0.571564
d   -0.058569
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

```
In [84]: df.sum(0, skipna=False)
Out[84]:
one           NaN
two      1.881078
three         NaN
dtype: float64

In [85]: df.sum(axis=1, skipna=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[85]:␣
↪
a   -0.242232
b    1.084464
c    1.714693
d   -0.117137
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [86]: ts_stand = (df - df.mean()) / df.std()

In [87]: ts_stand.std()
Out[87]:
one      1.0
two      1.0
three    1.0
dtype: float64

In [88]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [89]: xs_stand.std(1)
Out[89]:
a    1.0
b    1.0
c    1.0
d    1.0
dtype: float64
```

Note that methods like *cumsum()* and *cumprod()* preserve the location of `NaN` values. This is somewhat different from *expanding()* and *rolling()*. For more details please see *this note*.

```
In [90]: df.cumsum()
Out[90]:
        one       two     three
```

(continues on next page)

(continued from previous page)

```
a  1.400810 -1.643041       NaN
b  1.044340 -0.597130  0.395023
c  1.841608  0.327385  0.387933
d       NaN  1.881078 -1.282898
```

Here is a quick reference summary table of common functions. Each also takes an optional `level` parameter which applies only if the object has a *hierarchical index*.

Function	Description
`count`	Number of non-NA observations
`sum`	Sum of values
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Bessel-corrected sample standard deviation
`var`	Unbiased variance
`sem`	Standard error of the mean
`skew`	Sample skewness (3rd moment)
`kurt`	Sample kurtosis (4th moment)
`quantile`	Sample quantile (value at %)
`cumsum`	Cumulative sum
`cumprod`	Cumulative product
`cummax`	Cumulative maximum
`cummin`	Cumulative minimum

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```
In [91]: np.mean(df['one'])
Out[91]: 0.6138692844180106

In [92]: np.mean(df['one'].to_numpy())
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[92]: nan
```

`Series.nunique()` will return the number of unique non-NA values in a Series:

```
In [93]: series = pd.Series(np.random.randn(500))

In [94]: series[20:500] = np.nan

In [95]: series[10:20] = 5

In [96]: series.nunique()
Out[96]: 11
```

Summarizing data: describe

There is a convenient *describe()* function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [97]: series = pd.Series(np.random.randn(1000))

In [98]: series[::2] = np.nan

In [99]: series.describe()
Out[99]:
count    500.000000
mean      -0.020695
std        1.011840
min       -2.683763
25%       -0.709297
50%       -0.070211
75%        0.712856
max        3.160915
dtype: float64

In [100]: frame = pd.DataFrame(np.random.randn(1000, 5),
   .....:                      columns=['a', 'b', 'c', 'd', 'e'])
   .....:

In [101]: frame.iloc[::2] = np.nan

In [102]: frame.describe()
Out[102]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.026515    0.022952   -0.047307   -0.052551    0.011210
std      1.016752    0.980046    1.020837    1.008271    1.006726
min     -3.000951   -2.637901   -3.303099   -3.159200   -3.188821
25%     -0.647623   -0.593587   -0.709906   -0.691338   -0.689176
50%      0.047578   -0.026675   -0.029655   -0.032769   -0.015775
75%      0.723946    0.771931    0.603753    0.667044    0.652221
max      2.740139    2.752332    3.004229    2.728702    3.240991
```

You can select specific percentiles to include in the output:

```
In [103]: series.describe(percentiles=[.05, .25, .75, .95])
Out[103]:
count    500.000000
mean      -0.020695
std        1.011840
min       -2.683763
5%        -1.641337
25%       -0.709297
50%       -0.070211
75%        0.712856
95%        1.699176
max        3.160915
dtype: float64
```

By default, the median is always included.

For a non-numerical Series object, *describe()* will give a simple summary of the number of unique values and most frequently occurring values:

```
In [104]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [105]: s.describe()
Out[105]:
count      9
unique     4
top        a
freq       5
dtype: object
```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [106]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [107]: frame.describe()
Out[107]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

This behavior can be controlled by providing a list of types as `include`/`exclude` arguments. The special value `all` can also be used:

```
In [108]: frame.describe(include=['object'])
Out[108]:
          a
count     4
unique    2
top     Yes
freq      2

In [109]: frame.describe(include=['number'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[109]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000

In [110]: frame.describe(include='all')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b
count     4  4.000000
unique    2       NaN
top     Yes       NaN
```

(continues on next page)

(continued from previous page)

```
freq      2       NaN
mean    NaN  1.500000
std     NaN  1.290994
min     NaN  0.000000
25%     NaN  0.750000
50%     NaN  1.500000
75%     NaN  2.250000
max     NaN  3.000000
```

That feature relies on *select_dtypes*. Refer to there for details about accepted inputs.

Index of Min/Max Values

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [111]: s1 = pd.Series(np.random.randn(5))

In [112]: s1
Out[112]:
0   -0.068822
1   -1.129788
2   -0.269798
3   -0.375580
4    0.513381
dtype: float64

In [113]: s1.idxmin(), s1.idxmax()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪(1, 4)

In [114]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

In [115]: df1
Out[115]:
          A         B         C
0  0.333329 -0.910090 -1.321220
1  2.111424  1.701169  0.858336
2 -0.608055 -2.082155 -0.069618
3  1.412817 -0.562658  0.770042
4  0.373294 -0.965381 -1.607840

In [116]: df1.idxmin(axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    2
B    2
C    4
dtype: int64

In [117]: df1.idxmax(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    A
1    A
```

(continues on next page)

(continued from previous page)

```
2    C
3    A
4    A
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, *idxmin()* and *idxmax()* return the first matching index:

```
In [118]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [119]: df3
Out[119]: 
     A
e  2.0
d  1.0
c  1.0
b  3.0
a  NaN

In [120]: df3['A'].idxmin()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[120]: 'd'
```

Note: idxmin and idxmax are called argmin and argmax in NumPy.

Value counts (histogramming) / Mode

The *value_counts()* Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [121]: data = np.random.randint(0, 7, size=50)

In [122]: data
Out[122]: 
array([6, 4, 1, 3, 4, 4, 4, 6, 5, 2, 6, 1, 0, 4, 3, 2, 5, 3, 4, 0, 5, 3, 0,
       1, 5, 0, 1, 5, 3, 4, 1, 2, 3, 2, 4, 6, 1, 4, 3, 5, 2, 1, 2, 4, 1, 6,
       3, 6, 3, 3])

In [123]: s = pd.Series(data)

In [124]: s.value_counts()
Out[124]: 
4    10
3    10
1     8
6     6
5     6
2     6
0     4
dtype: int64

In [125]: pd.value_counts(data)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[125]: 
↪
```

(continues on next page)

(continued from previous page)

```
4    10
3    10
1     8
6     6
5     6
2     6
0     4
dtype: int64
```

Similarly, you can get the most frequently occurring value(s) (the mode) of the values in a Series or DataFrame:

```
In [126]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])

In [127]: s5.mode()
Out[127]: 
0    3
1    7
dtype: int64

In [128]: df5 = pd.DataFrame({"A": np.random.randint(0, 7, size=50),
   .....:                     "B": np.random.randint(-10, 15, size=50)})
   .....: 

In [129]: df5.mode()
Out[129]: 
   A  B
0  0 -9
```

Discretization and quantiling

Continuous values can be discretized using the *cut()* (bins based on values) and *qcut()* (bins based on sample quantiles) functions:

```
In [130]: arr = np.random.randn(20)

In [131]: factor = pd.cut(arr, 4)

In [132]: factor
Out[132]: 
[(1.27, 2.31], (0.231, 1.27], (-0.809, 0.231], (-1.853, -0.809], (1.27, 2.31], ...,
↪(0.231, 1.27], (-0.809, 0.231], (-1.853, -0.809], (1.27, 2.31], (0.231, 1.27]]
Length: 20
Categories (4, interval[float64]): [(-1.853, -0.809] < (-0.809, 0.231] < (0.231, 1.
↪27] < (1.27, 2.31]]

In [133]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [134]: factor
Out[134]: 
[(1, 5], (0, 1], (-1, 0], (-5, -1], (1, 5], ..., (1, 5], (-1, 0], (-5, -1], (1, 5],
↪(0, 1]]
Length: 20
Categories (4, interval[int64]): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]
```

qcut() computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [135]: arr = np.random.randn(30)

In [136]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [137]: factor
Out[137]:
[(-2.219, -0.669], (-0.669, 0.00453], (0.367, 2.369], (0.00453, 0.367], (0.367, 2.
↪369], ..., (0.00453, 0.367], (0.367, 2.369], (0.00453, 0.367], (-0.669, 0.00453],␣
↪(0.367, 2.369]]
Length: 30
Categories (4, interval[float64]): [(-2.219, -0.669] < (-0.669, 0.00453] < (0.00453,␣
↪0.367] <
                                    (0.367, 2.369]]

In [138]: pd.value_counts(factor)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
(0.367, 2.369]       8
(-2.219, -0.669]     8
(0.00453, 0.367]     7
(-0.669, 0.00453]    7
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [139]: arr = np.random.randn(20)

In [140]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [141]: factor
Out[141]:
[(0.0, inf], (-inf, 0.0], (-inf, 0.0], (-inf, 0.0], (-inf, 0.0], ..., (-inf, 0.0], (-
↪inf, 0.0], (0.0, inf], (-inf, 0.0], (-inf, 0.0]]
Length: 20
Categories (2, interval[float64]): [(-inf, 0.0] < (0.0, inf]]
```

3.3.6 Function application

To apply your own or another library's functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. *Tablewise Function Application*: `pipe()`
2. *Row or Column-wise Function Application*: `apply()`
3. *Aggregation API*: `agg()` and `transform()`
4. *Applying Elementwise Functions*: `applymap()`

Tablewise Function Application

`DataFrames` and `Series` can of course just be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method. Compare the following

```
# f, g, and h are functions taking and returning ``DataFrames``
>>> f(g(h(df), arg1=1), arg2=2, arg3=3)
```

with the equivalent

```
>>> (df.pipe(h)
...     .pipe(g, arg1=1)
...     .pipe(f, arg2=2, arg3=3))
```

Pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another library's functions in method chains, alongside pandas' methods.

In the example above, the functions `f`, `g`, and `h` each expected the `DataFrame` as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide `pipe` with a tuple of `(callable, data_keyword)`. `.pipe` will route the `DataFrame` to the argument specified in the tuple.

For example, we can fit a regression using statsmodels. Their API expects a formula first and a `DataFrame` as the second argument, `data`. We pass in the function, keyword pair `(sm.ols, 'data')` to `pipe`:

```
In [142]: import statsmodels.formula.api as sm

In [143]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [144]: (bb.query('h > 0')
   .....:    .assign(ln_h=lambda df: np.log(df.h))
   .....:    .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
   .....:    .fit()
   .....:    .summary()
   .....:  )
   .....:
Out[144]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                     hr   R-squared:                       0.685
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Fri, 25 Jan 2019   Prob (F-statistic):           3.48e-15
Time:                        16:28:07   Log-Likelihood:                -205.92
No. Observations:                  68   AIC:                             421.8
Df Residuals:                      63   BIC:                             432.9
Df Model:                           4
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept   -8484.7720   4664.146     -1.819      0.074   -1.78e+04     835.780
C(lg)[T.NL]    -2.2736      1.325     -1.716      0.091      -4.922       0.375
ln_h           -1.3542      0.875     -1.547      0.127      -3.103       0.395
year            4.2277      2.324      1.819      0.074      -0.417       8.872
g               0.1841      0.029      6.258      0.000       0.125       0.243
==============================================================================
Omnibus:                       10.875   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               17.298
Skew:                           0.537   Prob(JB):                     0.000175
Kurtosis:                       5.225   Cond. No.                     1.49e+07
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular (%>%) (read pipe) operator for R. The implementation of pipe here is quite clean and feels right at home in python. We encourage you to view the source code of *pipe()*.

Row or Column-wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame using the *apply()* method, which, like the descriptive statistics methods, takes an optional axis argument:

```
In [145]: df.apply(np.mean)
Out[145]:
one      0.613869
two      0.470270
three   -0.427633
dtype: float64

In [146]: df.apply(np.mean, axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[146]:␣
↪
a   -0.121116
b    0.361488
c    0.571564
d   -0.058569
dtype: float64

In [147]: df.apply(lambda x: x.max() - x.min())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
one      1.757280
two      3.196734
three    2.065853
dtype: float64

In [148]: df.apply(np.cumsum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  1.400810 -1.643041       NaN
b  1.044340 -0.597130  0.395023
c  1.841608  0.327385  0.387933
d       NaN  1.881078 -1.282898

In [149]: df.apply(np.exp)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a  4.058485  0.193391       NaN
```

(continues on next page)

(continued from previous page)

```
b  0.700143  2.845991  1.484418
c  2.219469  2.520646  0.992935
d       NaN  4.728902  0.188091
```

The *apply()* method will also dispatch on a string method name.

```
In [150]: df.apply('mean')
Out[150]:
one      0.613869
two      0.470270
three   -0.427633
dtype: float64

In [151]: df.apply('mean', axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[151]:␣
↪
a   -0.121116
b    0.361488
c    0.571564
d   -0.058569
dtype: float64
```

The return type of the function passed to *apply()* affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-likes return values expand (or not) to a `DataFrame`.

apply() combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [152]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [153]: tsdf.apply(lambda x: x.idxmax())
Out[153]:
A   2000-06-10
B   2001-07-04
C   2002-08-09
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the *apply()* method. For instance, consider the following function you would like to apply:

```
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [154]: tsdf
Out[154]:
                   A         B         C
2000-01-01 -0.652077 -0.239118  0.841272
2000-01-02  0.130224  0.347505 -0.385666
2000-01-03 -1.700237 -0.925899  0.199564
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.339319 -0.978307  0.689492
2000-01-09  0.601495 -0.630417 -1.040079
2000-01-10  1.511723 -0.427952 -0.400154

In [155]: tsdf.apply(pd.Series.interpolate)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C
2000-01-01 -0.652077 -0.239118  0.841272
2000-01-02  0.130224  0.347505 -0.385666
2000-01-03 -1.700237 -0.925899  0.199564
2000-01-04 -1.292326 -0.936380  0.297550
2000-01-05 -0.884415 -0.946862  0.395535
2000-01-06 -0.476503 -0.957344  0.493521
2000-01-07 -0.068592 -0.967825  0.591507
2000-01-08  0.339319 -0.978307  0.689492
2000-01-09  0.601495 -0.630417 -1.040079
2000-01-10  1.511723 -0.427952 -0.400154
```

Finally, `apply()` takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

Aggregation API

New in version 0.20.0.

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see *groupby API*, the *window functions API*, and the *resample API*. The entry point for aggregation is `DataFrame.aggregate()`, or the alias `DataFrame.agg()`.

We will use a similar starting frame from above:

```
In [156]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [157]: tsdf.iloc[3:7] = np.nan

In [158]: tsdf
Out[158]:
                   A         B         C
2000-01-01  0.396575 -0.364907  0.051290
2000-01-02 -0.310517 -0.369093 -0.353151
2000-01-03 -0.522441  1.659115 -0.272364
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.057890  1.148901  0.011528
2000-01-09 -0.155578  0.742150  0.107324
2000-01-10  0.531797  0.080254  0.833297
```

Using a single function is equivalent to `apply()`. You can also pass named methods as strings. These will return a `Series` of the aggregated output:

```
In [159]: tsdf.agg(np.sum)
Out[159]: 
A   -0.118055
B    2.896420
C    0.377923
dtype: float64

In [160]: tsdf.agg('sum')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[160]: 
A   -0.118055
B    2.896420
C    0.377923
dtype: float64

# these are equivalent to a ``.sum()`` because we are aggregating
# on a single function
In [161]: tsdf.sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A   -0.118055
B    2.896420
C    0.377923
dtype: float64
```

Single aggregations on a `Series` this will return a scalar value:

```
In [162]: tsdf.A.agg('sum')
Out[162]: -0.11805495013260869
```

Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting `DataFrame`. These are naturally named from the aggregation function.

```
In [163]: tsdf.agg(['sum'])
Out[163]: 
            A        B         C
sum -0.118055  2.89642  0.377923
```

Multiple functions yield multiple rows:

```
In [164]: tsdf.agg(['sum', 'mean'])
Out[164]: 
             A         B         C
sum  -0.118055  2.896420  0.377923
mean -0.019676  0.482737  0.062987
```

On a `Series`, multiple functions return a `Series`, indexed by the function names:

```
In [165]: tsdf.A.agg(['sum', 'mean'])
Out[165]:
sum    -0.118055
mean   -0.019676
Name: A, dtype: float64
```

Passing a `lambda` function will yield a `<lambda>` named row:

```
In [166]: tsdf.A.agg(['sum', lambda x: x.mean()])
Out[166]:
sum         -0.118055
<lambda>    -0.019676
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [167]: def mymean(x):
   .....:     return x.mean()
   .....:

In [168]: tsdf.A.agg(['sum', mymean])
Out[168]:
sum       -0.118055
mymean    -0.019676
Name: A, dtype: float64
```

Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to `DataFrame.agg` allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an `OrderedDict` instead to guarantee ordering.

```
In [169]: tsdf.agg({'A': 'mean', 'B': 'sum'})
Out[169]:
A   -0.019676
B    2.896420
dtype: float64
```

Passing a list-like will generate a `DataFrame` output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be `NaN`:

```
In [170]: tsdf.agg({'A': ['mean', 'min'], 'B': 'sum'})
Out[170]:
             A        B
mean -0.019676      NaN
min  -0.522441      NaN
sum        NaN  2.89642
```

Mixed Dtypes

When presented with mixed dtypes that cannot aggregate, `.agg` will only take the valid aggregations. This is similar to how groupby `.agg` works.

```
In [171]: mdf = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [1., 2., 3.],
   .....:                     'C': ['foo', 'bar', 'baz'],
   .....:                     'D': pd.date_range('20130101', periods=3)})
   .....:

In [172]: mdf.dtypes
Out[172]:
A             int64
B           float64
C            object
D    datetime64[ns]
dtype: object
```

```
In [173]: mdf.agg(['min', 'sum'])
Out[173]:
     A    B          C          D
min  1  1.0        bar 2013-01-01
sum  6  6.0  foobarbaz        NaT
```

Custom describe

With `.agg()` is it possible to easily create a custom describe function, similar to the built in *describe function*.

```
In [174]: from functools import partial

In [175]: q_25 = partial(pd.Series.quantile, q=0.25)

In [176]: q_25.__name__ = '25%'

In [177]: q_75 = partial(pd.Series.quantile, q=0.75)

In [178]: q_75.__name__ = '75%'

In [179]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
Out[179]:
               A         B         C
count   6.000000  6.000000  6.000000
mean   -0.019676  0.482737  0.062987
std     0.408577  0.836785  0.420419
min    -0.522441 -0.369093 -0.353151
25%    -0.271782 -0.253617 -0.201391
median -0.106734  0.411202  0.031409
75%     0.282958  1.047213  0.093315
max     0.531797  1.659115  0.833297
```

Transform API

New in version 0.20.0.

The *transform()* method returns an object that is indexed the same (same size) as the original. This API allows you to provide *multiple* operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

```
In [180]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   .....:                     index=pd.date_range('1/1/2000', periods=10))
   .....:

In [181]: tsdf.iloc[3:7] = np.nan

In [182]: tsdf
Out[182]:
                   A         B         C
2000-01-01 -1.219234 -1.652700 -0.698277
2000-01-02  1.858653 -0.738520  0.630364
2000-01-03 -0.112596  1.525897  1.364225
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.527790 -1.715506  0.387274
2000-01-09 -0.569341  0.569386  0.134136
2000-01-10 -0.413993 -0.862280  0.662690
```

Transform the entire frame. `.transform()` allows input functions as: a NumPy function, a string function name or a user defined function.

```
In [183]: tsdf.transform(np.abs)
Out[183]:
                   A         B         C
2000-01-01  1.219234  1.652700  0.698277
2000-01-02  1.858653  0.738520  0.630364
2000-01-03  0.112596  1.525897  1.364225
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.527790  1.715506  0.387274
2000-01-09  0.569341  0.569386  0.134136
2000-01-10  0.413993  0.862280  0.662690

In [184]: tsdf.transform('abs')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C
2000-01-01  1.219234  1.652700  0.698277
2000-01-02  1.858653  0.738520  0.630364
2000-01-03  0.112596  1.525897  1.364225
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.527790  1.715506  0.387274
2000-01-09  0.569341  0.569386  0.134136
2000-01-10  0.413993  0.862280  0.662690

In [185]: tsdf.transform(lambda x: x.abs())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C
2000-01-01  1.219234  1.652700  0.698277
```

(continues on next page)

(continued from previous page)

```
2000-01-02  1.858653  0.738520  0.630364
2000-01-03  0.112596  1.525897  1.364225
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.527790  1.715506  0.387274
2000-01-09  0.569341  0.569386  0.134136
2000-01-10  0.413993  0.862280  0.662690
```

Here *transform()* received a single function; this is equivalent to a ufunc application.

```
In [186]: np.abs(tsdf)
Out[186]:
                   A         B         C
2000-01-01  1.219234  1.652700  0.698277
2000-01-02  1.858653  0.738520  0.630364
2000-01-03  0.112596  1.525897  1.364225
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.527790  1.715506  0.387274
2000-01-09  0.569341  0.569386  0.134136
2000-01-10  0.413993  0.862280  0.662690
```

Passing a single function to `.transform()` with a `Series` will yield a single `Series` in return.

```
In [187]: tsdf.A.transform(np.abs)
Out[187]:
2000-01-01    1.219234
2000-01-02    1.858653
2000-01-03    0.112596
2000-01-04         NaN
2000-01-05         NaN
2000-01-06         NaN
2000-01-07         NaN
2000-01-08    0.527790
2000-01-09    0.569341
2000-01-10    0.413993
Freq: D, Name: A, dtype: float64
```

Transform with multiple functions

Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [188]: tsdf.transform([np.abs, lambda x: x + 1])
Out[188]:
                   A                   B                   C
            absolute  <lambda>  absolute  <lambda>  absolute  <lambda>
2000-01-01  1.219234 -0.219234  1.652700 -0.652700  0.698277  0.301723
2000-01-02  1.858653  2.858653  0.738520  0.261480  0.630364  1.630364
2000-01-03  0.112596  0.887404  1.525897  2.525897  1.364225  2.364225
2000-01-04       NaN       NaN       NaN       NaN       NaN       NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-05       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  0.527790  0.472210  1.715506 -0.715506  0.387274  1.387274
2000-01-09  0.569341  0.430659  0.569386  1.569386  0.134136  1.134136
2000-01-10  0.413993  0.586007  0.862280  0.137720  0.662690  1.662690
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [189]: tsdf.A.transform([np.abs, lambda x: x + 1])
Out[189]:
            absolute  <lambda>
2000-01-01  1.219234 -0.219234
2000-01-02  1.858653  2.858653
2000-01-03  0.112596  0.887404
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.527790  0.472210
2000-01-09  0.569341  0.430659
2000-01-10  0.413993  0.586007
```

Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [190]: tsdf.transform({'A': np.abs, 'B': lambda x: x + 1})
Out[190]:
                   A         B
2000-01-01  1.219234 -0.652700
2000-01-02  1.858653  0.261480
2000-01-03  0.112596  2.525897
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.527790 -0.715506
2000-01-09  0.569341  1.569386
2000-01-10  0.413993  0.137720
```

Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms.

```
In [191]: tsdf.transform({'A': np.abs, 'B': [lambda x: x + 1, 'sqrt']})
Out[191]:
                   A         B
            absolute  <lambda>      sqrt
2000-01-01  1.219234 -0.652700       NaN
2000-01-02  1.858653  0.261480       NaN
2000-01-03  0.112596  2.525897  1.235272
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
```

(continues on next page)

(continued from previous page)

```
2000-01-08  0.527790 -0.715506       NaN
2000-01-09  0.569341  1.569386  0.754577
2000-01-10  0.413993  0.137720       NaN
```

Applying Elementwise Functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods *applymap()* on DataFrame and analogously *map()* on Series accept any Python function taking a single value and returning a single value. For example:

```
In [192]: df4
Out[192]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [193]: def f(x):
   .....:     return len(str(x))
   .....:

In [194]: df4['one'].map(f)
Out[194]:
a    18
b    19
c    18
d     3
Name: one, dtype: int64

In [195]: df4.applymap(f)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[195]:
   one  two  three
a   18   19      3
b   19   18     19
c   18   18     21
d    3   18     19
```

Series.map() has an additional feature; it can be used to easily "link" or "map" values defined by a secondary series. This is closely related to *merging/joining functionality*:

```
In [196]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'],
   .....:               index=['a', 'b', 'c', 'd', 'e'])
   .....:

In [197]: t = pd.Series({'six': 6., 'seven': 7.})

In [198]: s
Out[198]:
a      six
b    seven
c      six
d    seven
e      six
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [199]: s.map(t)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[199]:␣
↪
a    6.0
b    7.0
c    6.0
d    7.0
e    6.0
dtype: float64
```

3.3.7 Reindexing and altering labels

`reindex()` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To *reindex* means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, **fill** data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [200]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [201]: s
Out[201]:
a   -0.368437
b   -0.036473
c    0.774830
d   -0.310545
e    0.709717
dtype: float64

In [202]: s.reindex(['e', 'b', 'f', 'd'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2
↪
e    0.709717
b   -0.036473
f         NaN
d   -0.310545
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [203]: df
Out[203]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830
```

(continues on next page)

(continued from previous page)

```
In [204]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      three       two       one
c -0.007090  0.924515  0.797268
f       NaN       NaN       NaN
b  0.395023  1.045911 -0.356470
```

You may also use `reindex` with an `axis` keyword:

```
In [205]: df.reindex(['c', 'f', 'b'], axis='index')
Out[205]:
        one       two     three
c  0.797268  0.924515 -0.007090
f       NaN       NaN       NaN
b -0.356470  1.045911  0.395023
```

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [206]: rs = s.reindex(df.index)

In [207]: rs
Out[207]:
a   -0.368437
b   -0.036473
c    0.774830
d   -0.310545
dtype: float64

In [208]: rs.index is df.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[208]:␣
↪True
```

This means that the reindexed Series's index is the same Python object as the DataFrame's index.

New in version 0.21.0.

DataFrame.reindex() also supports an "axis-style" calling convention, where you specify a single `labels` argument and the `axis` it applies to.

```
In [209]: df.reindex(['c', 'f', 'b'], axis='index')
Out[209]:
        one       two     three
c  0.797268  0.924515 -0.007090
f       NaN       NaN       NaN
b -0.356470  1.045911  0.395023

In [210]: df.reindex(['three', 'two', 'one'], axis='columns')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      three       two       one
a       NaN -1.643041  1.400810
b  0.395023  1.045911 -0.356470
c -0.007090  0.924515  0.797268
d -1.670830  1.553693       NaN
```

See also:

MultiIndex / Advanced Indexing is an even more concise way of doing reindexing.

Note: When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the *reindex_like()* method is available to make this simpler:

```
In [211]: df2
Out[211]:
        one       two
a  1.400810 -1.643041
b -0.356470  1.045911
c  0.797268  0.924515

In [212]: df3
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪
        one       two
a  0.786941 -1.752170
b -0.970339  0.936783
c  0.183399  0.815387

In [213]: df.reindex_like(df2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two
a  1.400810 -1.643041
b -0.356470  1.045911
c  0.797268  0.924515
```

Aligning objects with each other with `align`

The *align()* method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [214]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

(continues on next page)

(continued from previous page)

```
In [215]: s1 = s[:4]

In [216]: s2 = s[1:]

In [217]: s1.align(s2)
Out[217]:
(a   -0.610263
 b   -0.170883
 c    0.367255
 d    0.273860
 e         NaN
 dtype: float64, a         NaN
 b   -0.170883
 c    0.367255
 d    0.273860
 e    0.314782
 dtype: float64)

In [218]: s1.align(s2, join='inner')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
(b   -0.170883
 c    0.367255
 d    0.273860
 dtype: float64, b   -0.170883
 c    0.367255
 d    0.273860
 dtype: float64)

In [219]: s1.align(s2, join='left')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
(a   -0.610263
 b   -0.170883
 c    0.367255
 d    0.273860
 dtype: float64, a         NaN
 b   -0.170883
 c    0.367255
 d    0.273860
 dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [220]: df.align(df2, join='inner')
Out[220]:
(        one       two
 a  1.400810 -1.643041
 b -0.356470  1.045911
 c  0.797268  0.924515,         one       two
 a  1.400810 -1.643041
 b -0.356470  1.045911
 c  0.797268  0.924515)
```

You can also pass an `axis` option to only align on the specified axis:

```
In [221]: df.align(df2, join='inner', axis=0)
Out[221]:
(        one       two     three
 a  1.400810 -1.643041       NaN
 b -0.356470  1.045911  0.395023
 c  0.797268  0.924515 -0.007090,          one       two
 a  1.400810 -1.643041
 b -0.356470  1.045911
 c  0.797268  0.924515)
```

If you pass a Series to *DataFrame.align()*, you can choose to align both objects either on the DataFrame's index or columns using the `axis` argument:

```
In [222]: df.align(df2.iloc[0], axis=1)
Out[222]:
(        one     three       two
 a  1.400810       NaN -1.643041
 b -0.356470  0.395023  1.045911
 c  0.797268 -0.007090  0.924515
 d       NaN -1.670830  1.553693, one      1.400810
 three         NaN
 two     -1.643041
 Name: a, dtype: float64)
```

Filling while reindexing

reindex() takes an optional parameter `method` which is a filling method chosen from the following table:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward
nearest	Fill from the nearest index value

We illustrate these fill methods on a simple Series:

```
In [223]: rng = pd.date_range('1/3/2000', periods=8)

In [224]: ts = pd.Series(np.random.randn(8), index=rng)

In [225]: ts2 = ts[[0, 3, 6]]

In [226]: ts
Out[226]:
2000-01-03   -0.082578
2000-01-04    0.768554
2000-01-05    0.398842
2000-01-06   -0.357956
2000-01-07    0.156403
2000-01-08   -1.347564
2000-01-09    0.253506
2000-01-10    1.228964
Freq: D, dtype: float64

In [227]: ts2
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-03   -0.082578
2000-01-06   -0.357956
2000-01-09    0.253506
dtype: float64

In [228]: ts2.reindex(ts.index)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-03   -0.082578
2000-01-04         NaN
2000-01-05         NaN
2000-01-06   -0.357956
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    0.253506
2000-01-10         NaN
Freq: D, dtype: float64

In [229]: ts2.reindex(ts.index, method='ffill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-03   -0.082578
2000-01-04   -0.082578
2000-01-05   -0.082578
2000-01-06   -0.357956
2000-01-07   -0.357956
2000-01-08   -0.357956
2000-01-09    0.253506
2000-01-10    0.253506
Freq: D, dtype: float64

In [230]: ts2.reindex(ts.index, method='bfill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-03   -0.082578
2000-01-04   -0.357956
2000-01-05   -0.357956
2000-01-06   -0.357956
2000-01-07    0.253506
2000-01-08    0.253506
2000-01-09    0.253506
2000-01-10         NaN
Freq: D, dtype: float64

In [231]: ts2.reindex(ts.index, method='nearest')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-03   -0.082578
2000-01-04   -0.082578
2000-01-05   -0.357956
2000-01-06   -0.357956
2000-01-07   -0.357956
2000-01-08    0.253506
2000-01-09    0.253506
2000-01-10    0.253506
```

(continues on next page)

(continued from previous page)

```
Freq: D, dtype: float64
```

These methods require that the indexes are **ordered** increasing or decreasing.

Note that the same result could have been achieved using *fillna* (except for `method='nearest'`) or *interpolate*:

```
In [232]: ts2.reindex(ts.index).fillna(method='ffill')
Out[232]:
2000-01-03   -0.082578
2000-01-04   -0.082578
2000-01-05   -0.082578
2000-01-06   -0.357956
2000-01-07   -0.357956
2000-01-08   -0.357956
2000-01-09    0.253506
2000-01-10    0.253506
Freq: D, dtype: float64
```

`reindex()` will raise a ValueError if the index is not monotonically increasing or decreasing. `fillna()` and `interpolate()` will not perform any checks on the order of the index.

Limits on filling while reindexing

The `limit` and `tolerance` arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches:

```
In [233]: ts2.reindex(ts.index, method='ffill', limit=1)
Out[233]:
2000-01-03   -0.082578
2000-01-04   -0.082578
2000-01-05         NaN
2000-01-06   -0.357956
2000-01-07   -0.357956
2000-01-08         NaN
2000-01-09    0.253506
2000-01-10    0.253506
Freq: D, dtype: float64
```

In contrast, tolerance specifies the maximum distance between the index and indexer values:

```
In [234]: ts2.reindex(ts.index, method='ffill', tolerance='1 day')
Out[234]:
2000-01-03   -0.082578
2000-01-04   -0.082578
2000-01-05         NaN
2000-01-06   -0.357956
2000-01-07   -0.357956
2000-01-08         NaN
2000-01-09    0.253506
2000-01-10    0.253506
Freq: D, dtype: float64
```

Notice that when used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will coerced into a `Timedelta` if possible. This allows you to specify tolerance with appropriate strings.

Dropping labels from an axis

A method closely related to `reindex` is the *drop()* function. It removes a set of labels from an axis:

```
In [235]: df
Out[235]:
        one       two     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [236]: df.drop(['a', 'd'], axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090

In [237]: df.drop(['one'], axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        two     three
a -1.643041       NaN
b  1.045911  0.395023
c  0.924515 -0.007090
d  1.553693 -1.670830
```

Note that the following also works, but is a bit less obvious / clean:

```
In [238]: df.reindex(df.index.difference(['a', 'd']))
Out[238]:
        one       two     three
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
```

Renaming / mapping labels

The *rename()* method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [239]: s
Out[239]:
a   -0.610263
b   -0.170883
c    0.367255
d    0.273860
e    0.314782
dtype: float64

In [240]: s.rename(str.upper)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2
↪
A   -0.610263
B   -0.170883
C    0.367255
D    0.273860
```

(continues on next page)

(continued from previous page)

```
E    0.314782
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). A dict or Series can also be used:

```
In [241]: df.rename(columns={'one': 'foo', 'two': 'bar'},
   .....:           index={'a': 'apple', 'b': 'banana', 'd': 'durian'})
   .....:
Out[241]:
             foo       bar     three
apple   1.400810 -1.643041       NaN
banana -0.356470  1.045911  0.395023
c       0.797268  0.924515 -0.007090
durian       NaN  1.553693 -1.670830
```

If the mapping doesn't include a column/index label, it isn't renamed. Note that extra labels in the mapping don't throw an error.

New in version 0.21.0.

DataFrame.rename() also supports an "axis-style" calling convention, where you specify a single `mapper` and the `axis` to apply that mapping to.

```
In [242]: df.rename({'one': 'foo', 'two': 'bar'}, axis='columns')
Out[242]:
        foo       bar     three
a  1.400810 -1.643041       NaN
b -0.356470  1.045911  0.395023
c  0.797268  0.924515 -0.007090
d       NaN  1.553693 -1.670830

In [243]: df.rename({'a': 'apple', 'b': 'banana', 'd': 'durian'}, axis='index')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
             one       two     three
apple   1.400810 -1.643041       NaN
banana -0.356470  1.045911  0.395023
c       0.797268  0.924515 -0.007090
durian       NaN  1.553693 -1.670830
```

The *rename()* method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place.

New in version 0.18.0.

Finally, *rename()* also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [244]: s.rename("scalar-name")
Out[244]:
a   -0.610263
b   -0.170883
c    0.367255
d    0.273860
e    0.314782
Name: scalar-name, dtype: float64
```

New in version 0.24.0.

The methods *rename_axis()* and *rename_axis()* allow specific names of a *MultiIndex* to be changed (as opposed to the labels).

```
In [245]: df = pd.DataFrame({'x': [1, 2, 3, 4, 5, 6],
   .....:                    'y': [10, 20, 30, 40, 50, 60]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b', 'c'], [1,␣
↪2]],
   .....:                   names=['let', 'num']))
   .....:

In [246]: df
Out[246]:
         x   y
let num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [247]: df.rename_axis(index={'let': 'abc'})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
         x   y
abc num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [248]: df.rename_axis(index=str.upper)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
         x   y
LET NUM
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60
```

3.3.8 Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. Other data structures, like DataFrame and Panel, follow the dict-like convention of iterating over the "keys" of the objects.

In short, basic iteration (`for i in object`) produces:

- **Series**: values
- **DataFrame**: column labels
- **Panel**: item labels

Thus, for example, iterating over a DataFrame gives you the column names:

```
In [249]: df = pd.DataFrame({'col1': np.random.randn(3),
   .....:                    'col2': np.random.randn(3)}, index=['a', 'b', 'c'])
   .....:

In [250]: for col in df:
   .....:     print(col)
   .....:
col1
col2
```

Pandas objects also have the dict-like *iteritems()* method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- *iterrows()*: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- *itertuples()*: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than *iterrows()*, and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning: Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, . . .
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use *apply()* instead of iterating over the values. See the docs on *function application*.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the *enhancing performance* section for some examples of this approach.

Warning: You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```
In [251]: df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})

In [252]: for index, row in df.iterrows():
   .....:     row['a'] = 10
   .....:

In [253]: df
Out[253]:
   a  b
0  1  a
1  2  b
2  3  c
```

iteritems

Consistent with the dict-like interface, *iteritems()* iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs
- **Panel**: (item, DataFrame) pairs

For example:

```
In [254]: for item, frame in wp.iteritems():
   .....:     print(item)
   .....:     print(frame)
   .....:
Item1
                   A         B         C         D
2000-01-01 -1.157892 -1.344312  0.844885  1.075770
2000-01-02 -0.109050  1.643563 -1.469388  0.357021
2000-01-03 -0.674600 -1.776904 -0.968914 -1.294524
2000-01-04  0.413738  0.276662 -0.472035 -0.013960
2000-01-05 -0.362543 -0.006154 -0.923061  0.895717
Item2
                   A         B         C         D
2000-01-01  0.805244 -1.206412  2.565646  1.431256
2000-01-02  1.340309 -1.170299 -0.226169  0.410835
2000-01-03  0.813850  0.132003 -0.827317 -0.076467
2000-01-04 -1.187678  1.130127 -1.436737 -1.413681
2000-01-05  1.607920  1.024180  0.569605  0.875906
```

iterrows

iterrows() allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```
In [255]: for row_index, row in df.iterrows():
   .....:     print(row_index, row, sep='\n')
   .....:
0
a    1
b    a
Name: 0, dtype: object
1
a    2
b    b
Name: 1, dtype: object
2
a    3
b    c
Name: 2, dtype: object
```

Note: Because *iterrows()* returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [256]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])

In [257]: df_orig.dtypes
Out[257]:
int        int64
```

(continues on next page)

(continued from previous page)

```
float     float64
dtype: object

In [258]: row = next(df_orig.iterrows())[1]

In [259]: row
Out[259]:
int        1.0
float      1.5
Name: 0, dtype: float64
```

All values in `row`, returned as a Series, are now upcasted to floats, also the original integer value in column *x*:

```
In [260]: row['int'].dtype
Out[260]: dtype('float64')

In [261]: df_orig['int'].dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[261]: dtype('int64')
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally much faster than *iterrows()*.

For instance, a contrived way to transpose the DataFrame would be:

```
In [262]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})

In [263]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6

In [264]: print(df2.T)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\   0  1  2
x  1  2  3
y  4  5  6

In [265]: df2_t = pd.DataFrame({idx: values for idx, values in df2.iterrows()})

In [266]: print(df2_t)
   0  1  2
x  1  2  3
y  4  5  6
```

itertuples

The *itertuples()* method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

For instance:

```
In [267]: for row in df.itertuples():
   .....:     print(row)
   .....:
```

(continues on next page)

(continued from previous page)

```
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, *itertuples()* preserves the data type of the values and is generally faster as *iterrows()*.

Note: The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

3.3.9 .dt accessor

`Series` has an accessor to succinctly return datetime like properties for the *values* of the Series, if it is a datetime/period like Series. This will return a Series, indexed like the existing Series.

```
# datetime
In [268]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [269]: s
Out[269]:
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
dtype: datetime64[ns]

In [270]: s.dt.hour
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    9
1    9
2    9
3    9
dtype: int64

In [271]: s.dt.second
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    12
1    12
2    12
3    12
dtype: int64

In [272]: s.dt.day
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    2
2    3
3    4
dtype: int64
```

This enables nice expressions like this:

```
In [273]: s[s.dt.day == 2]
Out[273]:
1   2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [274]: stz = s.dt.tz_localize('US/Eastern')

In [275]: stz
Out[275]:
0   2013-01-01 09:10:12-05:00
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]

In [276]: stz.dt.tz
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

You can also chain these types of operations:

```
In [277]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[277]:
0   2013-01-01 04:10:12-05:00
1   2013-01-02 04:10:12-05:00
2   2013-01-03 04:10:12-05:00
3   2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also format datetime values as strings with `Series.dt.strftime()` which supports the same format as the standard `strftime()`.

```
# DatetimeIndex
In [278]: s = pd.Series(pd.date_range('20130101', periods=4))

In [279]: s
Out[279]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
dtype: datetime64[ns]

In [280]: s.dt.strftime('%Y/%m/%d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[280]
↪
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

```
# PeriodIndex
In [281]: s = pd.Series(pd.period_range('20130101', periods=4))
```

(continues on next page)

(continued from previous page)

```
In [282]: s
Out[282]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [283]: s.dt.strftime('%Y/%m/%d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[283]:
↪
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

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

```
# period
In [284]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [285]: s
Out[285]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [286]: s.dt.year
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[286]:
↪
0    2013
1    2013
2    2013
3    2013
dtype: int64

In [287]: s.dt.day
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    2
2    3
3    4
dtype: int64
```

```
# timedelta
In [288]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [289]: s
Out[289]:
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
```

(continues on next page)

(continued from previous page)

```
3   1 days 00:00:08
dtype: timedelta64[ns]

In [290]: s.dt.days
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    1
2    1
3    1
dtype: int64

In [291]: s.dt.seconds
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    5
1    6
2    7
3    8
dtype: int64

In [292]: s.dt.components
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0
```

Note: `Series.dt` will raise a `TypeError` if you access with a non-datetime-like values.

3.3.10 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```
In [293]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog
↪', 'cat'])

In [294]: s.str.lower()
Out[294]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them).

Please see *Vectorized String Methods* for a complete description.

3.3.11 Sorting

Pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

By Index

The *Series.sort_index()* and *DataFrame.sort_index()* methods are used to sort a pandas object by its index levels.

```
In [295]: df = pd.DataFrame({
   .....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   .....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   .....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   .....:

In [296]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
   .....:                          columns=['three', 'two', 'one'])
   .....:

In [297]: unsorted_df
Out[297]:
      three       two       one
a       NaN -0.867293  0.050162
d  1.215473 -0.051744       NaN
c -0.421091 -0.712097  0.953102
b  1.205223  0.632624 -1.534113

# DataFrame
In [298]: unsorted_df.sort_index()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      three       two       one
a       NaN -0.867293  0.050162
b  1.205223  0.632624 -1.534113
c -0.421091 -0.712097  0.953102
d  1.215473 -0.051744       NaN

In [299]: unsorted_df.sort_index(ascending=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      three       two       one
d  1.215473 -0.051744       NaN
c -0.421091 -0.712097  0.953102
b  1.205223  0.632624 -1.534113
a       NaN -0.867293  0.050162

In [300]: unsorted_df.sort_index(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one     three       two
```

(continues on next page)

(continued from previous page)

```
a  0.050162       NaN -0.867293
d       NaN  1.215473 -0.051744
c  0.953102 -0.421091 -0.712097
b -1.534113  1.205223  0.632624

# Series
In [301]: unsorted_df['three'].sort_index()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a         NaN
b    1.205223
c   -0.421091
d    1.215473
Name: three, dtype: float64
```

By Values

The `Series.sort_values()` method is used to sort a *Series* by its values. The `DataFrame.sort_values()` method is used to sort a *DataFrame* by its column or row values. The optional `by` parameter to `DataFrame.sort_values()` may used to specify one or more columns to use to determine the sorted order.

```
In [302]: df1 = pd.DataFrame({'one': [2, 1, 1, 1],
   .....:                     'two': [1, 3, 2, 4],
   .....:                     'three': [5, 4, 3, 2]})
   .....:

In [303]: df1.sort_values(by='two')
Out[303]:
   one  two  three
0    2    1      5
2    1    2      3
1    1    3      4
3    1    4      2
```

The `by` parameter can take a list of column names, e.g.:

```
In [304]: df1[['one', 'two', 'three']].sort_values(by=['one', 'two'])
Out[304]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

These methods have special treatment of NA values via the `na_position` argument:

```
In [305]: s[2] = np.nan

In [306]: s.sort_values()
Out[306]:
0       A
3    Aaba
1       B
4    Baca
6    CABA
```

(continues on next page)

(continued from previous page)

```
8      cat
7      dog
2      NaN
5      NaN
dtype: object

In [307]: s.sort_values(na_position='first')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2      NaN
5      NaN
0        A
3     Aaba
1        B
4     Baca
6     CABA
8      cat
7      dog
dtype: object
```

By Indexes and Values

New in version 0.23.0.

Strings passed as the `by` parameter to *DataFrame.sort_values()* may refer to either columns or index level names.

```
# Build MultiIndex
In [308]: idx = pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('a', 2),
   .....:                                  ('b', 2), ('b', 1), ('b', 1)])
   .....:

In [309]: idx.names = ['first', 'second']

# Build DataFrame
In [310]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
   .....:                         index=idx)
   .....:

In [311]: df_multi
Out[311]:
              A
first second
a     1       6
      2       5
      2       4
b     2       3
      1       2
      1       1
```

Sort by 'second' (index) and 'A' (column)

```
In [312]: df_multi.sort_values(by=['second', 'A'])
Out[312]:
              A
```

(continues on next page)

(continued from previous page)

```
first  second
b      1        1
       1        2
a      1        6
b      2        3
a      2        4
       2        5
```

Note: If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

searchsorted

Series has the *searchsorted()* method, which works similarly to `numpy.ndarray.searchsorted()`.

```
In [313]: ser = pd.Series([1, 2, 3])

In [314]: ser.searchsorted([0, 3])
Out[314]: array([0, 2])

In [315]: ser.searchsorted([0, 4])
\\\\\\\\\\\\\\\\\\\\\\\\Out[315]: array([0, 3])

In [316]: ser.searchsorted([1, 3], side='right')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[316]: array([1, 3])

In [317]: ser.searchsorted([1, 3], side='left')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[317]:␣
↪array([0, 2])

In [318]: ser = pd.Series([3, 1, 2])

In [319]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[319]: array([0, 2])
```

smallest / largest values

`Series` has the *nsmallest()* and *nlargest()* methods which return the smallest or largest n values. For a large `Series` this can be much faster than sorting the entire Series and calling `head(n)` on the result.

```
In [320]: s = pd.Series(np.random.permutation(10))

In [321]: s
Out[321]:
0    5
1    3
2    2
3    0
4    7
5    6
6    9
7    1
```

(continues on next page)

(continued from previous page)

```
8    4
9    8
dtype: int64

In [322]: s.sort_values()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[322
↪
3    0
7    1
2    2
1    3
8    4
0    5
5    6
4    7
9    8
6    9
dtype: int64

In [323]: s.nsmallest(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
3    0
7    1
2    2
dtype: int64

In [324]: s.nlargest(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
6    9
9    8
4    7
dtype: int64
```

`DataFrame` also has the `nlargest` and `nsmallest` methods.

```
In [325]: df = pd.DataFrame({'a': [-2, -1, 1, 10, 8, 11, -1],
   .....:                    'b': list('abdceff'),
   .....:                    'c': [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0]})
   .....:

In [326]: df.nlargest(3, 'a')
Out[326]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN

In [327]: df.nlargest(5, ['a', 'c'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[327]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN
2   1  d  4.0
6  -1  f  4.0
```

(continues on next page)

(continued from previous page)

```
In [328]: df.nsmallest(3, 'a')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0

In [329]: df.nsmallest(5, ['a', 'c'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0
2  1  d  4.0
4  8  e  NaN
```

Sorting by a MultiIndex column

You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to `by`.

```
In [330]: df1.columns = pd.MultiIndex.from_tuples([('a', 'one'),
   .....:                                          ('a', 'two'),
   .....:                                          ('b', 'three')])
   .....:

In [331]: df1.sort_values(by=('a', 'two'))
Out[331]:
    a         b
  one two three
0   2   1     5
2   1   2     3
1   1   3     4
3   1   4     2
```

3.3.12 Copying

The *copy()* method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column.
- Assigning to the `index` or `columns` attributes.
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing.

To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

3.3.13 dtypes

For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for `float`, `int`, `bool`, `timedelta64[ns]` and `datetime64[ns]` (note that NumPy does not support timezone-aware datetimes).

Pandas and third-party libraries *extend* NumPy's type system in a few places. This section describes the extensions pandas has made internally. See *Extension Types* for how to write your own extension that works with pandas. See *Extension Data Types* for a list of third-party libraries that have implemented an extension.

The following table lists all of pandas extension types. See the respective documentation sections for more on each type.

Kind of Data	Data Type	Scalar	Array	Documentation
tz-aware datetime	*DatetimeTZDtype*	*Timestamp*	*arrays.DatetimeArray*	*Time Zone Handling*
Categorical	*CategoricalDtype*	(none)	*Categorical*	*Categorical Data*
period (time spans)	*PeriodDtype*	*Period*	arrays.PeriodArray	*Time Span Representation*
sparse	*SparseDtype*	(none)	arrays.SparseArray	*Sparse data structures*
intervals	*IntervalDtype*	*Interval*	*arrays.IntervalArray*	*IntervalIndex*
nullable integer	*Int64Dtype*, ...	(none)	*arrays.IntegerArray*	*Nullable Integer Data Type*

Pandas uses the `object` dtype for storing strings.

Finally, arbitrary objects may be stored using the `object` dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See *object conversion*).

A convenient *dtypes* attribute for DataFrame returns a Series with the data type of each column.

```
In [332]: dft = pd.DataFrame({'A': np.random.rand(3),
   .....:                     'B': 1,
   .....:                     'C': 'foo',
   .....:                     'D': pd.Timestamp('20010102'),
   .....:                     'E': pd.Series([1.0] * 3).astype('float32'),
   .....:                     'F': False,
   .....:                     'G': pd.Series([1] * 3, dtype='int8')})
   .....:

In [333]: dft
Out[333]:
          A  B    C          D    E      F  G
0  0.278831  1  foo 2001-01-02  1.0  False  1
1  0.242124  1  foo 2001-01-02  1.0  False  1
2  0.078031  1  foo 2001-01-02  1.0  False  1

In [334]: dft.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
```

(continues on next page)

(continued from previous page)

```
F              bool
G              int8
dtype: object
```

On a `Series` object, use the *dtype* attribute.

```
In [335]: dft['A'].dtype
Out[335]: dtype('float64')
```

If a pandas object contains data with multiple dtypes *in a single column*, the dtype of the column will be chosen to accommodate all of the data types (`object` is the most general).

```
# these ints are coerced to floats
In [336]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[336]:
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [337]: pd.Series([1, 2, 3, 6., 'foo'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[337]: 
↪
0      1
1      2
2      3
3      6
4    foo
dtype: object
```

The number of columns of each type in a `DataFrame` can be found by calling *get_dtype_counts()*.

```
In [338]: dft.get_dtype_counts()
Out[338]:
float64          1
float32          1
int64            1
int8             1
datetime64[ns]   1
bool             1
object           1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [339]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [340]: df1
Out[340]:
          A
```

(continues on next page)

(continued from previous page)

```
0 -1.641339
1 -0.314062
2 -0.679206
3  1.178243
4  0.181790
5 -2.044248
6  1.151282
7 -1.641398

In [341]: df1.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
dtype: object

In [342]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8), dtype='float16'),
   .....:                     'B': pd.Series(np.random.randn(8)),
   .....:                     'C': pd.Series(np.array(np.random.randn(8),
   .....:                                         dtype='uint8')) })
   .....:

In [343]: df2
Out[343]:
          A         B    C
0  0.130737 -1.143729    1
1  0.289551  2.787500    0
2  0.590820 -0.708143  254
3 -0.020142 -1.512388    0
4 -1.048828 -0.243145    1
5 -0.808105 -0.650992    0
6  1.373047  2.090108    0
7 -0.254395  0.433098    0

In [344]: df2.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float16
B    float64
C      uint8
dtype: object
```

defaults

By default integer types are `int64` and float types are `float64`, *regardless* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [345]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[345]:
a    int64
dtype: object

In [346]: pd.DataFrame({'a': [1, 2]}).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[346]:
a    int64
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [347]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[347]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [348]: frame = pd.DataFrame(np.array([1, 2]))
```

upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [349]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [350]: df3
Out[350]:
          A         B      C
0 -1.510602 -1.143729    1.0
1 -0.024511  2.787500    0.0
2 -0.088385 -0.708143  254.0
3  1.158101 -1.512388    0.0
4 -0.867039 -0.243145    1.0
5 -2.852354 -0.650992    0.0
6  2.524329  2.090108    0.0
7 -1.895793  0.433098    0.0

In [351]: df3.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float64
C    float64
dtype: object
```

DataFrame.to_numpy() will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped NumPy array. This can force some *upcasting*.

```
In [352]: df3.to_numpy().dtype
Out[352]: dtype('float64')
```

astype

You can use the *astype()* method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **numpy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [353]: df3
Out[353]:
          A         B      C
0 -1.510602 -1.143729    1.0
1 -0.024511  2.787500    0.0
2 -0.088385 -0.708143  254.0
3  1.158101 -1.512388    0.0
4 -0.867039 -0.243145    1.0
5 -2.852354 -0.650992    0.0
6  2.524329  2.090108    0.0
7 -1.895793  0.433098    0.0

In [354]: df3.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float64
C    float64
dtype: object

# conversion of dtypes
In [355]: df3.astype('float32').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float32
C    float32
dtype: object
```

Convert a subset of columns to a specified type using *astype()*.

```
In [356]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [357]: dft[['a', 'b']] = dft[['a', 'b']].astype(np.uint8)

In [358]: dft
Out[358]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [359]: dft.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[359]:
a    uint8
b    uint8
c    int64
dtype: object
```

New in version 0.19.0.

Convert certain columns to a specific dtype by passing a dict to *astype()*.

```
In [360]: dft1 = pd.DataFrame({'a': [1, 0, 1], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [361]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})

In [362]: dft1
```

(continues on next page)

(continued from previous page)

```
Out[362]:
       a  b    c
0   True  4  7.0
1  False  5  8.0
2   True  6  9.0

In [363]: dft1.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[363]:␣
↪
a       bool
b      int64
c    float64
dtype: object
```

Note: When trying to convert a subset of columns to a specified type using *astype()* and *loc()*, upcasting occurs.

loc() tries to fit in what we are assigning to the current dtypes, while [] will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```
In [364]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [365]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out[365]:
a    uint8
b    uint8
dtype: object

In [366]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [367]: dft.dtypes
Out[367]:
a    int64
b    int64
c    int64
dtype: object
```

object conversion

pandas offers various functions to try to force conversion of types from the object dtype to other types. In cases where the data is already of the correct type, but stored in an object array, the *DataFrame.infer_objects()* and *Series.infer_objects()* methods can be used to soft convert to the correct type.

```
In [368]: import datetime

In [369]: df = pd.DataFrame([[1, 2],
   .....:                    ['a', 'b'],
   .....:                    [datetime.datetime(2016, 3, 2),
   .....:                     datetime.datetime(2016, 3, 2)]])
   .....:

In [370]: df = df.T

In [371]: df
```

(continues on next page)

(continued from previous page)

```
Out[371]:
   0  1                    2
0  1  a  2016-03-02 00:00:00
1  2  b  2016-03-02 00:00:00

In [372]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    object
1    object
2    object
dtype: object
```

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

```
In [373]: df.infer_objects().dtypes
Out[373]:
0              int64
1             object
2    datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- *to_numeric()* (conversion to numeric dtypes)

```
In [374]: m = ['1.1', 2, 3]

In [375]: pd.to_numeric(m)
Out[375]: array([ 1.1,  2. ,  3. ])
```

- *to_datetime()* (conversion to datetime objects)

```
In [376]: import datetime

In [377]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]

In [378]: pd.to_datetime(m)
Out[378]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]',␣
↪freq=None)
```

- *to_timedelta()* (conversion to timedelta objects)

```
In [379]: m = ['5us', pd.Timedelta('1day')]

In [380]: pd.to_timedelta(m)
Out[380]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'], dtype=
↪'timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [381]: import datetime

In [382]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [383]: pd.to_datetime(m, errors='coerce')
Out[383]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [384]: m = ['apple', 2, 3]

In [385]: pd.to_numeric(m, errors='coerce')
Out[385]: array([ nan,   2.,   3.])

In [386]: m = ['apple', pd.Timedelta('1day')]

In [387]: pd.to_timedelta(m, errors='coerce')
Out[387]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [388]: import datetime

In [389]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [390]: pd.to_datetime(m, errors='ignore')
Out[390]: Index(['apple', 2016-03-02 00:00:00], dtype='object')

In [391]: m = ['apple', 2, 3]

In [392]: pd.to_numeric(m, errors='ignore')
Out[392]: array(['apple', 2, 3], dtype=object)

In [393]: m = ['apple', pd.Timedelta('1day')]

In [394]: pd.to_timedelta(m, errors='ignore')
Out[394]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, *to_numeric()* provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [395]: m = ['1', 2, 3]

In [396]: pd.to_numeric(m, downcast='integer')   # smallest signed int dtype
Out[396]: array([1, 2, 3], dtype=int8)

In [397]: pd.to_numeric(m, downcast='signed')    # same as 'integer'
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[397]: array([1, 2, 3], dtype=int8)

In [398]: pd.to_numeric(m, downcast='unsigned')  # smallest unsigned int dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[398]:␣
↪array([1, 2, 3], dtype=uint8)

In [399]: pd.to_numeric(m, downcast='float')     # smallest float dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array([ 1.,  2.,  3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with *apply()*, we can "apply" the function over each column efficiently:

```
In [400]: import datetime

In [401]: df = pd.DataFrame([
   .....:     ['2016-07-09', datetime.datetime(2016, 3, 2)]] * 2, dtype='O')
   .....:

In [402]: df
Out[402]:
            0                    1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [403]: df.apply(pd.to_datetime)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
           0          1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [404]: df = pd.DataFrame([['1.1', 2, 3]] * 2, dtype='O')

In [405]: df
Out[405]:
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [406]: df.apply(pd.to_numeric)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[406]:
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [407]: df = pd.DataFrame([['5us', pd.Timedelta('1day')]] * 2, dtype='O')

In [408]: df
Out[408]:
     0                 1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00

In [409]: df.apply(pd.to_timedelta)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[409]:␣
↪
                0      1
0 00:00:00.000005 1 days
1 00:00:00.000005 1 days
```

gotchas

Performing selection operations on `integer` type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where `nans` are not introduced. See also *Support for integer NA*.

```
In [410]: dfi = df3.astype('int32')

In [411]: dfi['E'] = 1

In [412]: dfi
Out[412]:
   A  B    C  E
0 -1 -1    1  1
1  0  2    0  1
2  0  0  254  1
3  1 -1    0  1
4  0  0    1  1
5 -2  0    0  1
6  2  2    0  1
7 -1  0    0  1

In [413]: dfi.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    int32
B    int32
C    int32
E    int64
dtype: object

In [414]: casted = dfi[dfi > 0]

In [415]: casted
Out[415]:
     A    B      C  E
0  NaN  NaN    1.0  1
1  NaN  2.0    NaN  1
2  NaN  NaN  254.0  1
3  1.0  NaN    NaN  1
4  NaN  NaN    1.0  1
5  NaN  NaN    NaN  1
6  2.0  2.0    NaN  1
7  NaN  NaN    NaN  1

In [416]: casted.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float64
B    float64
C    float64
E      int64
dtype: object
```

While float dtypes are unchanged.

```
In [417]: dfa = df3.copy()

In [418]: dfa['A'] = dfa['A'].astype('float32')

In [419]: dfa.dtypes
Out[419]:
A    float32
```

(continues on next page)

(continued from previous page)

```
B    float64
C    float64
dtype: object

In [420]: casted = dfa[df2 > 0]

In [421]: casted
Out[421]:
          A         B      C
0 -1.510602       NaN    1.0
1 -0.024511  2.787500    NaN
2 -0.088385       NaN  254.0
3       NaN       NaN    NaN
4       NaN       NaN    1.0
5       NaN       NaN    NaN
6  2.524329  2.090108    NaN
7       NaN  0.433098    NaN

In [422]: casted.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float64
C    float64
dtype: object
```

3.3.14 Selecting columns based on `dtype`

The *select_dtypes()* method implements subsetting of columns based on their `dtype`.

First, let's create a *DataFrame* with a slew of different dtypes:

```
In [423]: df = pd.DataFrame({'string': list('abc'),
   .....:                    'int64': list(range(1, 4)),
   .....:                    'uint8': np.arange(3, 6).astype('u1'),
   .....:                    'float64': np.arange(4.0, 7.0),
   .....:                    'bool1': [True, False, True],
   .....:                    'bool2': [False, True, False],
   .....:                    'dates': pd.date_range('now', periods=3),
   .....:                    'category': pd.Series(list("ABC")).astype('category')})
   .....:

In [424]: df['tdeltas'] = df.dates.diff()

In [425]: df['uint64'] = np.arange(3, 6).astype('u8')

In [426]: df['other_dates'] = pd.date_range('20130101', periods=3)

In [427]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')

In [428]: df
Out[428]:
  string  int64  uint8  float64  bool1  bool2                      dates category␣
↪tdeltas  uint64 other_dates            tz_aware_dates
0      a      1      3      4.0   True  False 2019-01-25 16:28:09.510594        A      ␣
↪NaT       3  2013-01-01 2013-01-01 00:00:00-05:00
```

(continues on next page)

(continued from previous page)

```
1      b      2      4      5.0  False   True 2019-01-26 16:28:09.510594        B  1␣
↪days       4  2013-01-02 2013-01-02 00:00:00-05:00
2      c      3      5      6.0   True  False 2019-01-27 16:28:09.510594        C  1␣
↪days       5  2013-01-03 2013-01-03 00:00:00-05:00
```

And the dtypes:

```
In [429]: df.dtypes
Out[429]:
string                                      object
int64                                        int64
uint8                                        uint8
float64                                    float64
bool1                                         bool
bool2                                         bool
dates                               datetime64[ns]
category                                  category
tdeltas                            timedelta64[ns]
uint64                                      uint64
other_dates                         datetime64[ns]
tz_aware_dates    datetime64[ns, US/Eastern]
dtype: object
```

select_dtypes() has two parameters `include` and `exclude` that allow you to say "give me the columns *with* these dtypes" (`include`) and/or "give the columns *without* these dtypes" (`exclude`).

For example, to select `bool` columns:

```
In [430]: df.select_dtypes(include=[bool])
Out[430]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the NumPy dtype hierarchy:

```
In [431]: df.select_dtypes(include=['bool'])
Out[431]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

select_dtypes() also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [432]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[432]:
   int64  float64  bool1  bool2 tdeltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the `object` dtype:

```
In [433]: df.select_dtypes(include=['object'])
Out[433]:
  string
0      a
1      b
2      c
```

To see all the child dtypes of a generic `dtype` like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [434]: def subdtypes(dtype):
   .....:     subs = dtype.__subclasses__()
   .....:     if not subs:
   .....:         return dtype
   .....:     return [dtype, [subdtypes(dt) for dt in subs]]
   .....:
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [435]: subdtypes(np.generic)
Out[435]:
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.int64,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
       [numpy.uint8,
        numpy.uint16,
        numpy.uint32,
        numpy.uint64,
        numpy.uint64]]]],
    [numpy.inexact,
     [[numpy.floating,
       [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
      [numpy.complexfloating,
       [numpy.complex64, numpy.complex128, numpy.complex256]]]]]],
  [numpy.flexible,
   [[numpy.character, [numpy.bytes_, numpy.str_]],
    [numpy.void, [numpy.record]]]],
  numpy.bool_,
  numpy.datetime64,
  numpy.object_]]
```

Note: Pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and won't show up with the above function.

3.4 Intro to Data Structures

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

3.4.1 Series

`Series` is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

Here, `data` can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data is**:

From ndarray

If `data` is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [4]: s
Out[4]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]
↪Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -0.173215
```

(continues on next page)

(continued from previous page)

```
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

Note: pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

```
In [7]: d = {'b': 1, 'a': 0, 'c': 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

Note: When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and Pandas version >= 0.23.

If you're using Python < 3.6 or Pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a Pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {'a': 0., 'b': 1., 'c': 2.}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=['b', 'c', 'd', 'a'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Note: NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

```
In [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]: 
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.46911229990718628

In [14]: s[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: 
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64

In [15]: s[s > s.median()]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪
a    0.469112
e    1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]: np.exp(s)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Note: We will address array-based indexing like `s[[4, 3, 1]]` in *section*.

Like a NumPy array, a pandas Series has a *dtype*.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy's type system in a few places, in which case the dtype would be a `ExtensionDtype`. Some examples within pandas are *Categorical Data* and *Nullable Integer Data Type*. See *dtypes* for more.

If you need the actual array backing a `Series`, use `Series.array`.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633,  -1.5090585031735124,
  -1.1356323710171934,   1.2121120250208506]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable *automatic alignment*, for example).

`Series.array` will always be an `ExtensionArray`. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a `numpy.ndarray`. Pandas knows how to take an `ExtensionArray` and store it in a `Series` or a column of a `DataFrame`. See *dtypes* for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use `Series.to_numpy()`.

```
In [20]: s.to_numpy()
Out[20]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

Even if the Series is backed by a `ExtensionArray`, `Series.to_numpy()` will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s['a']
Out[21]: 0.4691122999071863

In [22]: s['e'] = 12.

In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: 'e' in s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True

In [25]: 'f' in s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪False
```

If a label is not contained, an exception is raised:

```
>>> s['f']
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```
In [26]: s.get('f')

In [27]: s.get('f', np.nan)
Out[27]: nan
```

See also the *section on attribute access*.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [29]: s * 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\O
↪
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [30]: np.exp(s)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a         1.598575
b         0.753623
c         0.221118
d         0.321219
e    162754.791419
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [31]: s[1:] + s[:-1]
Out[31]:
a         NaN
b   -0.565727
c   -3.018117
d   -2.271265
```

(continues on next page)

(continued from previous page)

```
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing `NaN`. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note: In general, we chose to make the default result of operations between differently indexed objects yield the **union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the **dropna** function.

Name attribute

Series can also have a `name` attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name='something')

In [33]: s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64

In [34]: s.name
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

New in version 0.18.0.

You can rename a Series with the *pandas.Series.rename()* method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that `s` and `s2` refer to different objects.

3.4.2 DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note: When the data is a dict, and columns is not specified, the DataFrame columns will be ordered by the dict's insertion order, if you are using Python version >= 3.6 and Pandas >= 0.23.

If you are using Python < 3.6 or Pandas < 0.23, and columns is not specified, the DataFrame columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   ....:      'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
   ....:

In [38]: df = pd.DataFrame(d)

In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=['d', 'b', 'a'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

Note: When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[43]: Index(['one', 'two'],␣
↪dtype='object')
```

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [44]: d = {'one': [1., 2., 3., 4.],
   ....:      'two': [4., 3., 2., 1.]}
   ....:

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

In [48]: data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

In [49]: pd.DataFrame(data)
Out[49]:
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=['first', 'second'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[50]:
        A    B         C
first   1  2.0  b'Hello'
```

(continues on next page)

(continued from previous page)

```
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=['C', 'A', 'B'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

From a list of dicts

```
In [52]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [53]: pd.DataFrame(data2)
Out[53]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=['first', 'second'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[54]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=['a', 'b'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a   b
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:               ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:               ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:               ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:               ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
   ....:
Out[56]:
       a              b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Missing Data

Much more will be said on this topic in the *Missing data* section. To construct a DataFrame with missing data, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate Constructors

DataFrame.from_dict

`DataFrame.from_dict` takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the `DataFrame` constructor except for the `orient` parameter which is `'columns'` by default, but which can be set to `'index'` in order to use the dict keys as row labels.

```
In [57]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]))
Out[57]:
   A  B
0  1  4
1  2  5
2  3  6
```

If you pass `orient='index'`, the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [58]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]),
   ....:                        orient='index', columns=['one', 'two', 'three'])
   ....:
Out[58]:
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal `DataFrame` constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [59]: data
Out[59]:
array([(1,  2., b'Hello'), (2,  3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [60]: pd.DataFrame.from_records(data, index='C')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [61]: df['one']
Out[61]:
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [62]: df['three'] = df['one'] * df['two']

In [63]: df['flag'] = df['one'] > 2

In [64]: df
Out[64]:
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [65]: del df['two']

In [66]: three = df.pop('three')

In [67]: df
Out[67]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [68]: df['foo'] = 'bar'

In [69]: df
Out[69]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [70]: df['one_trunc'] = df['one'][:2]

In [71]: df
Out[71]:
```

(continues on next page)

(continued from previous page)

```
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [72]: df.insert(1, 'bar', df['one'])

In [73]: df
Out[73]: 
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning New Columns in Method Chains

Inspired by dplyr's `mutate` verb, DataFrame has an *assign()* method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [74]: iris = pd.read_csv('data/iris.data')

In [75]: iris.head()
Out[75]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [76]: (iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength'])
   ....:      .head())
   ....: 
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [77]: iris.assign(sepal_ratio=lambda x: (x['SepalWidth'] / x['SepalLength'])).
↪head()
Out[77]: 
```

(continues on next page)

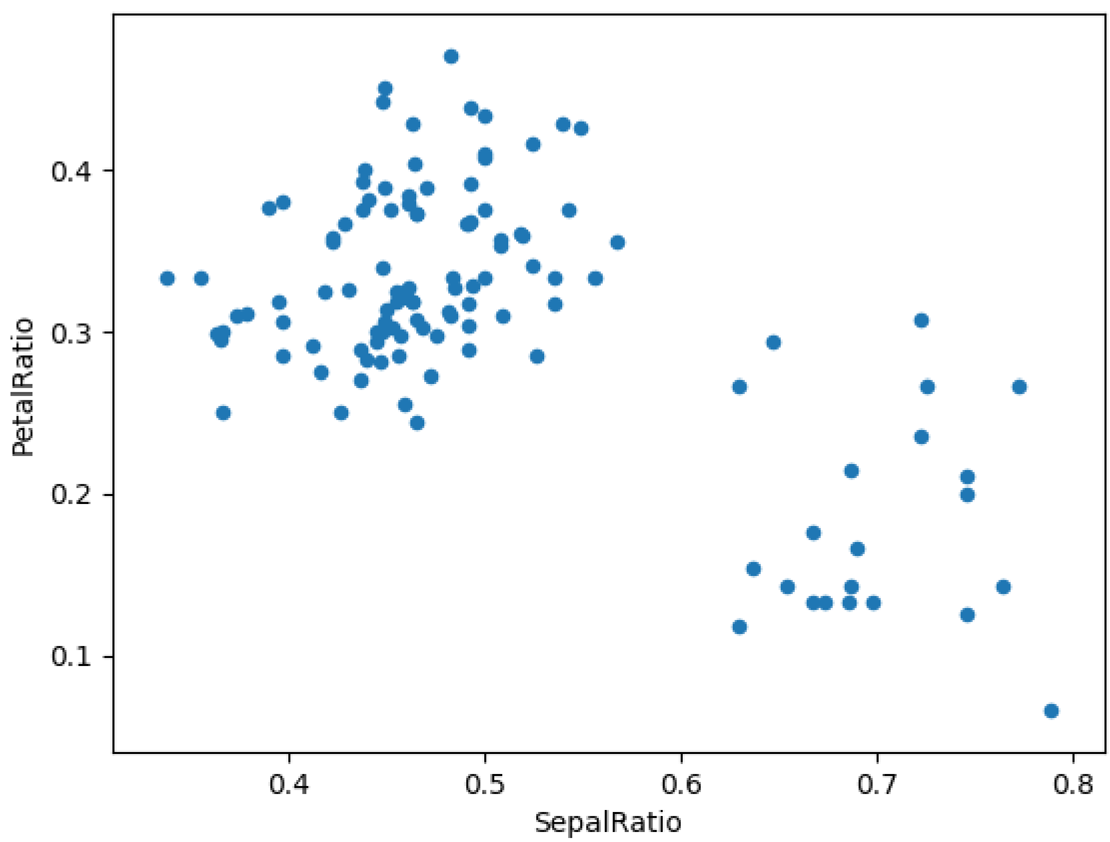
(continued from previous page)

```
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

`assign` **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand. This is common when using `assign` in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [78]: (iris.query('SepalLength > 5')
   ....:      .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:              PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
   ....:      .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
   ....:
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x7f382c50a438>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. A *copy* of the original DataFrame is returned, with the new values inserted.

Changed in version 0.23.0.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```
In [79]: dfa = pd.DataFrame({"A": [1, 2, 3],
   ....:                     "B": [4, 5, 6]})
   ....:

In [80]: dfa.assign(C=lambda x: x['A'] + x['B'],
   ....:            D=lambda x: x['A'] + x['C'])
   ....:
Out[80]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, `x['C']` will refer to the newly created column, that's equal to `dfa['A'] + dfa['B']`.

To write code compatible with all versions of Python, split the assignment in two.

```
In [81]: dependent = pd.DataFrame({"A": [1, 1, 1]})

In [82]: (dependent.assign(A=lambda x: x['A'] + 1)
   ....:           .assign(B=lambda x: x['A'] + 2))
   ....:
Out[82]:
   A  B
0  2  4
1  2  4
2  2  4
```

Warning: Dependent assignment maybe subtly change the behavior of your code between Python 3.6 and older versions of Python.

If you wish write code that supports versions of python before and after 3.6, you'll need to take care when passing `assign` expressions that

- Updating an existing column
- Referring to the newly updated column in the same `assign`

For example, we'll update column "A" and then refer to it when creating "B".

```
>>> dependent = pd.DataFrame({"A": [1, 1, 1]})
>>> dependent.assign(A=lambda x: x["A"] + 1, B=lambda x: x["A"] + 2)
```

For Python 3.5 and earlier the expression creating `B` refers to the "old" value of `A`, `[1, 1, 1]`. The output is then

```
   A  B
0  2  3
1  2  3
2  2  3
```

For Python 3.6 and later, the expression creating A refers to the "new" value of A, `[2, 2, 2]`, which results in

```
   A  B
0  2  4
1  2  4
2  2  4
```

Indexing / Selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [83]: df.loc['b']
Out[83]:
one              2
bar              2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object

In [84]: df.iloc[2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
one             3
bar             3
flag         True
foo           bar
one_trunc     NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the *section on indexing*. We will address the fundamentals of reindexing / conforming to new sets of labels in the *section on reindexing*.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on **both the columns and the index (row labels)**. Again, the resulting object will have the union of the column and row labels.

```
In [85]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])

In [86]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])

In [87]: df + df2
```

(continues on next page)

(continued from previous page)

```
Out[87]:
          A         B         C   D
0  0.045691 -0.014138  1.380871 NaN
1 -0.955398 -1.501007  0.037181 NaN
2 -0.662690  1.534833 -0.859691 NaN
3 -2.452949  1.237274 -0.133712 NaN
4  1.414490  1.951676 -2.320422 NaN
5 -0.494922 -1.649727 -1.084601 NaN
6 -1.047551 -0.748572 -0.805479 NaN
7       NaN       NaN       NaN NaN
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus broadcasting row-wise. For example:

```
In [88]: df - df.iloc[0]
Out[88]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

In the special case of working with time series data, and the DataFrame index also contains dates, the broadcasting will be column-wise:

```
In [89]: index = pd.date_range('1/1/2000', periods=8)

In [90]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=list('ABC'))

In [91]: df
Out[91]:
                   A         B         C
2000-01-01 -1.226825  0.769804 -1.281247
2000-01-02 -0.727707 -0.121306 -0.097883
2000-01-03  0.695775  0.341734  0.959726
2000-01-04 -1.110336 -0.619976  0.149748
2000-01-05 -0.732339  0.687738  0.176444
2000-01-06  0.403310 -0.154951  0.301624
2000-01-07 -2.179861 -1.369849 -0.954208
2000-01-08  1.462696 -1.743161 -0.826591

In [92]: type(df['A'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪pandas.core.series.Series

In [93]: df - df['A']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
            2000-01-01 00:00:00  2000-01-02 00:00:00  2000-01-03 00:00:00  2000-01-04␣
↪00:00:00  2000-01-05 00:00:00  ...  2000-01-07 00:00:00  2000-01-08 00:00:00   A  ␣
↪B   C
2000-01-01                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-02                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-03                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-04                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-05                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-06                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-07                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN
2000-01-08                  NaN                  NaN                  NaN          ␣
↪       NaN                  NaN  ...                  NaN                  NaN NaN␣
↪NaN NaN

[8 rows x 11 columns]
```

Warning:

```
df - df['A']
```

is now deprecated and will be removed in a future release. The preferred way to replicate this behavior is

```
df.sub(df['A'], axis=0)
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [94]: df * 5 + 2
Out[94]:
                   A         B         C
2000-01-01 -4.134126  5.849018 -4.406237
2000-01-02 -1.638535  1.393469  1.510587
2000-01-03  5.478873  3.708672  6.798628
2000-01-04 -3.551681 -1.099880  2.748742
2000-01-05 -1.661697  5.438692  2.882222
2000-01-06  4.016548  1.225246  3.508122
2000-01-07 -8.899303 -4.849247 -2.771039
2000-01-08  9.313480 -6.715805 -2.132955

In [95]: 1 / df
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B          C
2000-01-01 -0.815112  1.299033  -0.780489
2000-01-02 -1.374179 -8.243600 -10.216313
2000-01-03  1.437247  2.926250   1.041965
2000-01-04 -0.900628 -1.612966   6.677871
2000-01-05 -1.365487  1.454041   5.667510
2000-01-06  2.479485 -6.453662   3.315381
2000-01-07 -0.458745 -0.730007  -1.047990
2000-01-08  0.683669 -0.573671  -1.209788

In [96]: df ** 4
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                    A         B         C
2000-01-01   2.265327  0.351172  2.694833
2000-01-02   0.280431  0.000217  0.000092
2000-01-03   0.234355  0.013638  0.848376
2000-01-04   1.519910  0.147740  0.000503
2000-01-05   0.287640  0.223714  0.000969
2000-01-06   0.026458  0.000576  0.008277
2000-01-07  22.579530  3.521204  0.829033
2000-01-08   4.577374  9.233151  0.466834
```

Boolean operators work as well:

```
In [97]: df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1]}, dtype=bool)

In [98]: df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0]}, dtype=bool)

In [99]: df1 & df2
Out[99]:
       a      b
0  False  False
1  False   True
2   True  False

In [100]: df1 | df2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[100]:
      a     b
0  True  True
1  True  True
2  True  True

In [101]: df1 ^ df2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a      b
0   True   True
1   True  False
2  False   True

In [102]: -df1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a      b
```

(continues on next page)

(continued from previous page)

```
0  False   True
1   True  False
2  False  False
```

Transposing

To transpose, access the T attribute (also the transpose function), similar to an ndarray:

```
# only show the first 5 rows
In [103]: df[:5].T
Out[103]:
   2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A   -1.226825   -0.727707    0.695775   -1.110336   -0.732339
B    0.769804   -0.121306    0.341734   -0.619976    0.687738
C   -1.281247   -0.097883    0.959726    0.149748    0.176444
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on DataFrame, assuming the data within are numeric:

```
In [104]: np.exp(df)
Out[104]:
                   A         B         C
2000-01-01  0.293222  2.159342  0.277691
2000-01-02  0.483015  0.885763  0.906755
2000-01-03  2.005262  1.407386  2.610980
2000-01-04  0.329448  0.537957  1.161542
2000-01-05  0.480783  1.989212  1.192968
2000-01-06  1.496770  0.856457  1.352053
2000-01-07  0.113057  0.254145  0.385117
2000-01-08  4.317584  0.174966  0.437538

In [105]: np.asarray(df)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
array([[-1.2268,  0.7698, -1.2812],
       [-0.7277, -0.1213, -0.0979],
       [ 0.6958,  0.3417,  0.9597],
       [-1.1103, -0.62  ,  0.1497],
       [-0.7323,  0.6877,  0.1764],
       [ 0.4033, -0.155 ,  0.3016],
       [-2.1799, -1.3698, -0.9542],
       [ 1.4627, -1.7432, -0.8266]])
```

The dot method on DataFrame implements matrix multiplication:

```
In [106]: df.T.dot(df)
Out[106]:
           A         B         C
A  11.341858 -0.059772  3.007998
B  -0.059772  6.520556  2.083308
C   3.007998  2.083308  4.310549
```

Similarly, the dot method on Series implements dot product:

```
In [107]: s1 = pd.Series(np.arange(5, 10))

In [108]: s1.dot(s1)
Out[108]: 255
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics are quite different in places from a matrix.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using *info()*. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [109]: baseball = pd.read_csv('data/baseball.csv')

In [110]: print(baseball)
       id     player  year  stint team  lg   g   ab   r    h  X2b  X3b  hr   rbi   sb␣
↪  cs  bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006      2  CHN  NL  19   50   6   14    1    0   1   2.0  1.0␣
↪ 1.0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006      1  BOS  AL  31    2   0    1    0    0   0   0.0  0.0␣
↪ 0.0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..  ..  ...  ..  ...  ...  ...  ..   ...  ...␣
↪ ...  ..   ...  ...  ...  ...  ...   ...
98  89533   aloumo01  2007      1  NYN  NL  87  328  51  112   19    1  13  49.0  3.0␣
↪ 0.0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007      1  NYN  NL   8   22   1    3    1    0   0   0.0  0.0␣
↪ 0.0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [111]: baseball.info()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
id        100 non-null int64
player    100 non-null object
year      100 non-null int64
stint     100 non-null int64
team      100 non-null object
lg        100 non-null object
g         100 non-null int64
ab        100 non-null int64
r         100 non-null int64
h         100 non-null int64
X2b       100 non-null int64
X3b       100 non-null int64
hr        100 non-null int64
rbi       100 non-null float64
sb        100 non-null float64
cs        100 non-null float64
bb        100 non-null int64
so        100 non-null float64
ibb       100 non-null float64
```

(continues on next page)

(continued from previous page)

```
hbp       100 non-null float64
sh        100 non-null float64
sf        100 non-null float64
gidp      100 non-null float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.0+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

```
In [112]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [113]: pd.DataFrame(np.random.randn(3, 12))
Out[113]:
         0         1         2         3         4         5         6         7  ␣
↪  8         9        10        11
0 -0.345352  1.314232  0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441 -1.
↪236269  0.896171 -0.487602 -0.082240
1 -2.182937  0.380396  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230  0.
↪132885 -0.023688  2.410179  1.450520
2  0.206053 -0.251905 -2.213588  1.063327  1.266143  0.299368 -0.863838  0.408204 -1.
↪048089 -0.025747 -0.988387  0.094055
```

You can change how much to print on a single row by setting the `display.width` option:

```
In [114]: pd.set_option('display.width', 40)  # default is 80

In [115]: pd.DataFrame(np.random.randn(3, 12))
Out[115]:
         0         1         2         3         4         5         6         7  ␣
↪  8         9        10        11
0  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374 -0.034571 -2.
↪484478 -0.281461  0.030711  0.109121
```

(continues on next page)

(continued from previous page)

```
1  1.126203 -0.977349  1.474071 -0.064034 -1.282782  0.781836 -1.071357  0.441153  2.
↪353925  0.583787  0.221471 -0.744471
2  0.758527  1.729689 -0.964980 -0.845696 -1.340896  1.846883 -1.328865  1.682706 -1.
↪717693  0.888782  0.228440  0.901805
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [116]: datafile = {'filename': ['filename_01', 'filename_02'],
   .....:             'path': ["media/user_name/storage/folder_01/filename_01",
   .....:                      "media/user_name/storage/folder_02/filename_02"]}
   .....:

In [117]: pd.set_option('display.max_colwidth', 30)

In [118]: pd.DataFrame(datafile)
Out[118]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [119]: pd.set_option('display.max_colwidth', 100)

In [120]: pd.DataFrame(datafile)
Out[120]:
      filename                                           path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [121]: df = pd.DataFrame({'foo1': np.random.randn(5),
   .....:                    'foo2': np.random.randn(5)})
   .....:

In [122]: df
Out[122]:
       foo1      foo2
0  1.171216 -0.858447
1  0.520260  0.306996
2 -1.197071 -0.028665
3 -1.066969  0.384316
4 -0.303421  1.574159

In [123]: df.foo1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1.171216
1    0.520260
2   -1.197071
3   -1.066969
4   -0.303421
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.fo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```

3.4.3 Panel

Warning: In 0.20.0, `Panel` is deprecated and will be removed in a future version. See the section *Deprecate Panel*.

Panel is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

- **items**: axis 0, each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1, it is the **index** (rows) of each of the DataFrames
- **minor_axis**: axis 2, it is the **columns** of each of the DataFrames

Construction of Panels works about like you would expect:

From 3D ndarray with optional axis labels

```
In [124]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:                major_axis=pd.date_range('1/1/2000', periods=5),
   .....:                minor_axis=['A', 'B', 'C', 'D'])
   .....:

In [125]: wp
Out[125]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

From dict of DataFrame objects

```
In [126]: data = {'Item1': pd.DataFrame(np.random.randn(4, 3)),
   .....:         'Item2': pd.DataFrame(np.random.randn(4, 2))}
   .....:

In [127]: pd.Panel(data)
Out[127]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

Note that the values in the dict need only be **convertible to DataFrame**. Thus, they can be any of the other valid inputs to DataFrame as per above.

One helpful factory method is `Panel.from_dict`, which takes a dictionary of DataFrames as above, and the following named parameters:

Parameter	Default	Description
intersect	`False`	drops elements whose indices do not align
orient	`items`	use `minor` to use DataFrames' columns as panel items

For example, compare to the construction above:

```
In [128]: pd.Panel.from_dict(data, orient='minor')
Out[128]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: 0 to 2
Major_axis axis: 0 to 3
Minor_axis axis: Item1 to Item2
```

Orient is especially useful for mixed-type DataFrames. If you pass a dict of DataFrame objects with mixed-type columns, all of the data will get upcasted to `dtype=object` unless you pass `orient='minor'`:

```
In [129]: df = pd.DataFrame({'a': ['foo', 'bar', 'baz'],
   .....:                    'b': np.random.randn(3)})
   .....:

In [130]: df
Out[130]:
     a         b
0  foo -0.308853
1  bar -0.681087
2  baz  0.377953

In [131]: data = {'item1': df, 'item2': df}

In [132]: panel = pd.Panel.from_dict(data, orient='minor')

In [133]: panel['a']
Out[133]:
  item1 item2
0   foo   foo
1   bar   bar
2   baz   baz

In [134]: panel['b']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[134]:
      item1     item2
0 -0.308853 -0.308853
1 -0.681087 -0.681087
2  0.377953  0.377953

In [135]: panel['b'].dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
item1    float64
```

(continues on next page)

(continued from previous page)

```
item2     float64
dtype: object
```

Note: Panel, being less commonly used than Series and DataFrame, has been slightly neglected feature-wise. A number of methods and options available in DataFrame are not available in Panel.

From DataFrame using `to_panel` method

`to_panel` converts a DataFrame with a two-level index to a Panel.

```
In [136]: midx = pd.MultiIndex(levels=[['one', 'two'], ['x', 'y']],
   .....:                      codes=[[1, 1, 0, 0], [1, 0, 1, 0]])
   .....:

In [137]: df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [138]: df.to_panel()
Out[138]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: A to B
Major_axis axis: one to two
Minor_axis axis: x to y
```

Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

```
In [139]: wp['Item1']
Out[139]:
                   A         B         C         D
2000-01-01  1.588931  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707 -0.891060 -0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838

In [140]: wp['Item3'] = wp['Item1'] / wp['Item2']
```

The API for insertion and deletion is the same as for DataFrame. And as with DataFrame, if the item is a valid Python identifier, you can access it as an attribute and tab-complete it in IPython.

Transposing

A Panel can be rearranged using its `transpose` method (which does not make a copy by default unless the data are heterogeneous):

```
In [141]: wp.transpose(2, 0, 1)
Out[141]:
<class 'pandas.core.panel.Panel'>
```

(continues on next page)

(continued from previous page)

```
Dimensions: 4 (items) x 3 (major_axis) x 5 (minor_axis)
Items axis: A to D
Major_axis axis: Item1 to Item3
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

Indexing / Selection

Operation	Syntax	Result
Select item	`wp[item]`	DataFrame
Get slice at major_axis label	`wp.major_xs(val)`	DataFrame
Get slice at minor_axis label	`wp.minor_xs(val)`	DataFrame

For example, using the earlier example data, we could do:

```
In [142]: wp['Item1']
Out[142]:
                   A         B         C         D
2000-01-01  1.588931  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707 -0.891060 -0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838

In [143]: wp.major_xs(wp.major_axis[2])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      Item1     Item2     Item3
A -1.137707  0.800193 -1.421791
B -0.891060  0.782098 -1.139320
C -0.693921 -1.069094  0.649074
D  1.613616 -1.099248 -1.467927

In [144]: wp.minor_axis
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['A', 'B', 'C', 'D'], dtype='object')

In [145]: wp.minor_xs('C')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               Item1     Item2     Item3
2000-01-01  0.473424 -0.902937 -0.524316
2000-01-02  0.650776 -1.144073 -0.568824
2000-01-03 -0.693921 -1.069094  0.649074
2000-01-04 -0.496922  0.661084 -0.751678
2000-01-05 -1.561819 -1.056652  1.478083
```

Squeezing

Another way to change the dimensionality of an object is to `squeeze` a 1-len object, similar to `wp['Item1']`.

```
In [146]: wp.reindex(items=['Item1']).squeeze()
Out[146]:
```

(continues on next page)

(continued from previous page)

```
                   A         B         C         D
2000-01-01  1.588931  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707 -0.891060 -0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838

In [147]: wp.reindex(items=['Item1'], minor=['B']).squeeze()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-01    0.476720
2000-01-02   -0.284319
2000-01-03   -0.891060
2000-01-04    0.227371
2000-01-05   -1.134623
Freq: D, Name: B, dtype: float64
```

Conversion to DataFrame

A Panel can be represented in 2D form as a hierarchically indexed DataFrame. See the section *hierarchical indexing* for more on this. To convert a Panel to a DataFrame, use the `to_frame` method:

```
In [148]: panel = pd.Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
   .....:                  major_axis=pd.date_range('1/1/2000', periods=5),
   .....:                  minor_axis=['a', 'b', 'c', 'd'])
   .....:

In [149]: panel.to_frame()
Out[149]:
                       one       two     three
major      minor
2000-01-01 a      0.493672  1.219492 -1.290493
           b     -2.461467  0.062297  0.787872
           c     -1.553902 -0.110388  1.515707
           d      2.015523 -1.184357 -0.276487
2000-01-02 a     -1.833722 -0.558081 -0.223762
           b      1.771740  0.077849  1.397431
           c     -0.670027  0.629498  1.503874
           d      0.049307 -1.035260 -0.478905
2000-01-03 a     -0.521493 -0.438229 -0.135950
           b     -3.201750  0.503703 -0.730327
           c      0.792716  0.413086 -0.033277
           d      0.146111 -1.139050  0.281151
2000-01-04 a      1.903247  0.660342 -1.298915
           b     -0.747169  0.464794 -2.819487
           c     -0.309038 -0.309337 -0.851985
           d      0.393876 -0.649593 -1.106952
2000-01-05 a      1.861468  0.683758 -0.937731
           b      0.936527 -0.643834 -1.537770
           c      1.255746  0.421287  0.555759
           d     -2.655452  1.032814 -2.277282
```

3.4.4 Deprecate Panel

Over the last few years, pandas has increased in both breadth and depth, with new features, datatype support, and manipulation routines. As a result, supporting efficient indexing and functional routines for `Series`, `DataFrame` and `Panel` has contributed to an increasingly fragmented and difficult-to-understand code base.

The 3-D structure of a `Panel` is much less common for many types of data analysis, than the 1-D of the `Series` or the 2-D of the `DataFrame`. Going forward it makes sense for pandas to focus on these areas exclusively.

Oftentimes, one can simply use a MultiIndex `DataFrame` for easily working with higher dimensional data.

In addition, the `xarray` package was built from the ground up, specifically in order to support the multi-dimensional analysis that is one of `Panel` s main use cases. Here is a link to the xarray panel-transition documentation.

```
In [150]: import pandas.util.testing as tm

In [151]: p = tm.makePanel()

In [152]: p
Out[152]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 30 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-02-11 00:00:00
Minor_axis axis: A to D
```

Convert to a MultiIndex DataFrame.

```
In [153]: p.to_frame()
Out[153]:
                     ItemA     ItemB     ItemC
major      minor
2000-01-03 A     -0.390201 -1.624062 -0.605044
           B      1.562443  0.483103  0.583129
           C     -1.085663  0.768159 -0.273458
           D      0.136235 -0.021763 -0.700648
2000-01-04 A      1.207122 -0.758514  0.878404
           B      0.763264  0.061495 -0.876690
           C     -1.114738  0.225441 -0.335117
           D      0.886313 -0.047152 -1.166607
2000-01-05 A      0.178690 -0.560859 -0.921485
           B      0.162027  0.240767 -1.919354
           C     -0.058216  0.543294 -0.476268
           D     -1.350722  0.088472 -0.367236
2000-01-06 A     -1.004168 -0.589005 -0.200312
           B     -0.902704  0.782413 -0.572707
           C     -0.486768  0.771931 -1.765602
           D     -0.886348 -0.857435  1.296674
2000-01-07 A     -1.377627 -1.070678  0.522423
           B      1.106010  0.628462 -1.736484
           C      1.685148 -0.968145  0.578223
           D     -1.013316 -2.503786  0.641385
2000-01-10 A      0.499281 -1.681101  0.722511
           B     -0.199234 -0.880627 -1.335113
           C      0.112572 -1.176383  0.242697
           D      1.920906 -1.058041 -0.779432
2000-01-11 A     -1.405256  0.403776 -1.702486
           B      0.458265  0.777575 -1.244471
```

(continues on next page)

(continued from previous page)

```
          C     -1.495309 -3.192716  0.208129
          D     -0.388231 -0.657981  0.602456
2000-01-12 A     0.162565  0.609862 -0.709535
          B      0.491048 -0.779367  0.347339
...                  ...       ...       ...
2000-02-02 C    -0.303961 -0.463752 -0.288962
          D      0.104050  1.116086  0.506445
2000-02-03 A    -2.338595 -0.581967 -0.801820
          B     -0.557697 -0.033731 -0.176382
          C      0.625555 -0.055289  0.875359
          D      0.174068 -0.443915  1.626369
2000-02-04 A    -0.374279 -1.233862 -0.915751
          B      0.381353 -1.108761 -1.970108
          C     -0.059268 -0.360853 -0.614618
          D     -0.439461 -0.200491  0.429518
2000-02-07 A    -2.359958 -3.520876 -0.288156
          B      1.337122 -0.314399 -1.044208
          C      0.249698  0.728197  0.565375
          D     -0.741343  1.092633  0.013910
2000-02-08 A    -1.157886  0.516870 -1.199945
          B     -1.531095 -0.860626 -0.821179
          C      1.103949  1.326768  0.068184
          D     -0.079673 -1.675194 -0.458272
2000-02-09 A    -0.551865  0.343125 -0.072869
          B      1.331458  0.370397 -1.914267
          C     -1.087532  0.208927  0.788871
          D     -0.922875  0.437234 -1.531004
2000-02-10 A     1.592673  2.137827 -1.828740
          B     -0.571329 -1.761442 -0.826439
          C      1.998044  0.292058 -0.280343
          D      0.303638  0.388254 -0.500569
2000-02-11 A     1.559318  0.452429 -1.716981
          B     -0.026671 -0.899454  0.124808
          C     -0.244548 -2.019610  0.931536
          D     -0.917368  0.479630  0.870690

[120 rows x 3 columns]
```

Alternatively, one can convert to an xarray `DataArray`.

```
In [154]: p.to_xarray()
Out[154]:
<xarray.DataArray (items: 3, major_axis: 30, minor_axis: 4)>
array([[[-0.390201,  1.562443, -1.085663,  0.136235],
        [ 1.207122,  0.763264, -1.114738,  0.886313],
        ...,
        [ 1.592673, -0.571329,  1.998044,  0.303638],
        [ 1.559318, -0.026671, -0.244548, -0.917368]],

       [[-1.624062,  0.483103,  0.768159, -0.021763],
        [-0.758514,  0.061495,  0.225441, -0.047152],
        ...,
        [ 2.137827, -1.761442,  0.292058,  0.388254],
        [ 0.452429, -0.899454, -2.01961 ,  0.47963 ]],

       [[-0.605044,  0.583129, -0.273458, -0.700648],
        [ 0.878404, -0.87669 , -0.335117, -1.166607],
```

(continues on next page)

(continued from previous page)

```
        ...,
        [-1.82874 , -0.826439, -0.280343, -0.500569],
        [-1.716981,  0.124808,  0.931536,  0.87069 ]]])
Coordinates:
  * items       (items) object 'ItemA' 'ItemB' 'ItemC'
  * major_axis  (major_axis) datetime64[ns] 2000-01-03 2000-01-04 ... 2000-02-11
  * minor_axis  (minor_axis) object 'A' 'B' 'C' 'D'
```

You can see the full-documentation for the xarray package.

3.5 Comparison with other tools

3.5.1 Comparison with R / R libraries

Since `pandas` aims to provide a lot of the data manipulation and analysis functionality that people use R for, this page was started to provide a more detailed look at the R language and its many third party libraries as they relate to `pandas`. In comparisons with R and CRAN libraries, we care about the following things:

- **Functionality / flexibility**: what can/cannot be done with each tool
- **Performance**: how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use**: Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these R packages.

For transfer of `DataFrame` objects from `pandas` to R, one option is to use HDF5 files, see *External Compatibility* for an example.

Quick Reference

We'll start off with a quick reference guide pairing some common R operations using dplyr with pandas equivalents.

Querying, Filtering, Sampling

R	pandas
`dim(df)`	`df.shape`
`head(df)`	`df.head()`
`slice(df, 1:10)`	`df.iloc[:9]`
`filter(df, col1 == 1, col2 == 1)`	`df.query('col1 == 1 & col2 == 1')`
`df[df$col1 == 1 & df$col2 == 1,]`	`df[(df.col1 == 1) & (df.col2 == 1)]`
`select(df, col1, col2)`	`df[['col1', 'col2']]`
`select(df, col1:col3)`	`df.loc[:, 'col1':'col3']`
`select(df, -(col1:col3))`	`df.drop(cols_to_drop, axis=1)` but see[1]
`distinct(select(df, col1))`	`df[['col1']].drop_duplicates()`
`distinct(select(df, col1, col2))`	`df[['col1', 'col2']].drop_duplicates()`
`sample_n(df, 10)`	`df.sample(n=10)`
`sample_frac(df, 0.01)`	`df.sample(frac=0.01)`

Sorting

R	pandas
`arrange(df, col1, col2)`	`df.sort_values(['col1', 'col2'])`
`arrange(df, desc(col1))`	`df.sort_values('col1', ascending=False)`

Transforming

R	pandas
`select(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})['col_one']`
`rename(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})`
`mutate(df, c=a-b)`	`df.assign(c=df.a-df.b)`

Grouping and Summarizing

R	pandas
`summary(df)`	`df.describe()`
`gdf <- group_by(df, col1)`	`gdf = df.groupby('col1')`
`summarise(gdf, avg=mean(col1, na.rm=TRUE))`	`df.groupby('col1').agg({'col1': 'mean'})`
`summarise(gdf, total=sum(col1))`	`df.groupby('col1').sum()`

Base R

Slicing with R's c

R makes it easy to access `data.frame` columns by name

```
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in `pandas` is straightforward

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list('abc'))

In [2]: df[['a', 'c']]
Out[2]:
          a         c
```

(continues on next page)

[1] R's shorthand for a subrange of columns (`select(df, col1:col3)`) can be approached cleanly in pandas, if you have the list of columns, for example `df[cols[1:3]]` or `df.drop(cols[1:3])`, but doing this by column name is a bit messy.

(continued from previous page)

```
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892

In [3]: df.loc[:, ['a', 'c']]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         c
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892
```

Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list('abcdefg')

In [5]: n = 30

In [6]: columns = named + np.arange(len(named), n).tolist()

In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)

In [8]: df.iloc[:, np.r_[:10, 24:30]]
Out[8]:
           a         b         c         d         e         f         g         7  ␣
↪        8         9        24        25        26        27        28        29
0  -1.344312  0.844885  1.075770 -0.109050  1.643563 -1.469388  0.357021 -0.674600 -1.
↪776904 -0.968914 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317
1  -0.076467 -1.187678  1.130127 -1.436737 -1.413681  1.607920  1.024180  0.569605  0.
↪875906 -2.211372  0.959726 -1.110336 -0.619976  0.149748 -0.732339  0.687738
2   0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849 -0.954208  1.462696 -1.
↪743161 -0.826591  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230
3   0.132885 -0.023688  2.410179  1.450520  0.206053 -0.251905 -2.213588  1.063327  1.
↪266143  0.299368 -2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349
4   1.474071 -0.064034 -1.282782  0.781836 -1.071357  0.441153  2.353925  0.583787  0.
↪221471 -0.744471 -1.197071 -1.066969 -0.303421 -0.858447  0.306996 -0.028665
5   0.384316  1.574159  1.588931  0.476720  0.473424 -0.242861 -0.014805 -0.284319  0.
↪650776 -1.461665 -0.902937  0.068159 -0.057873 -0.368204 -1.144073  0.861209
6   0.800193  0.782098 -1.069094 -1.099248  0.255269  0.009750  0.661084  0.379319 -0.
↪008434  1.952541  0.604603  2.121453  0.597701  0.563700  0.967661 -1.057909
..       ...       ...       ...       ...       ...       ...       ...       ...  ␣
↪   ...       ...       ...       ...       ...       ...       ...       ...
```

(continues on next page)

(continued from previous page)

```
23  1.534417 -1.374226 -0.367477  0.782551  1.356489  0.981552  0.304501  0.354041 -1.
↪232756 -0.267074  0.641606 -1.690959  0.961088  0.052372  1.166439  0.407281
24  0.859275 -0.995910  0.261263  1.783442  0.380989  2.289726  0.309489  2.189028  1.
↪389045 -0.873585 -0.169076  0.840316  0.638172  0.890673 -1.949397 -0.003437
25  1.492125 -0.068190  0.681456  1.221829 -0.434352  1.204815 -0.195612  1.251683 -1.
↪040389 -0.796211  1.944517  0.042344 -0.307904  0.428572  0.880609  0.487645
26  0.725238  0.624607 -0.141185 -0.143948 -0.328162  2.095086 -0.608888 -0.926422  1.
↪872601 -2.513465 -0.846188  1.190624  0.778507  1.008500  1.424017  0.717110
27  1.262419  1.950057  0.301038 -0.933858  0.814946  0.181439 -0.110015 -2.364638 -1.
↪584814  0.307941 -1.341814  0.334281 -0.162227  1.007824  2.826008  1.458383
28 -1.585746 -0.899734  0.921494 -0.211762 -0.059182  0.058308  0.915377 -0.696321  0.
↪150664 -3.060395  0.403620 -0.026602 -0.240481  0.577223 -1.088417  0.326687
29 -0.986248  0.169729 -1.158091  1.019673  0.646039  0.917399 -0.010435  0.366366  0.
↪922729  0.869610 -1.209247 -0.671466  0.332872 -2.013086 -1.602549  0.333109

[30 rows x 16 columns]
```

aggregate

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called `df` and splitting it into groups `by1` and `by2`:

```
df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)
```

The *groupby()* method is similar to base R `aggregate` function.

```
In [9]: df = pd.DataFrame(
   ...:     {'v1': [1, 3, 5, 7, 8, 3, 5, np.nan, 4, 5, 7, 9],
   ...:      'v2': [11, 33, 55, 77, 88, 33, 55, np.nan, 44, 55, 77, 99],
   ...:      'by1': ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],
   ...:      'by2': ["wet", "dry", 99, 95, np.nan, "damp", 95, 99, "red", 99, np.nan,
   ...:              np.nan]})
   ...:

In [10]: g = df.groupby(['by1', 'by2'])

In [11]: g[['v1', 'v2']].mean()
Out[11]:
            v1    v2
by1  by2
1    95    5.0  55.0
     99    5.0  55.0
2    95    7.0  77.0
     99    NaN   NaN
big  damp  3.0  33.0
blue dry   3.0  33.0
red  red   4.0  44.0
     wet   1.0  11.0
```

For more details and examples see *the groupby documentation*.

match / %in%

A common way to select data in R is using `%in%` which is defined using the function `match`. The operator `%in%` is used to return a logical vector indicating if there is a match or not:

```
s <- 0:4
s %in% c(2,4)
```

The *isin()* method is similar to R `%in%` operator:

```
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)

In [13]: s.isin([2, 4])
Out[13]:
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

The `match` function returns a vector of the positions of matches of its first argument in its second:

```
s <- 0:4
match(s, c(2,4))
```

For more details and examples see *the reshaping documentation*.

tapply

`tapply` is similar to `aggregate`, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called `baseball`, and retrieving information based on the array `team`:

```
baseball <-
  data.frame(team = gl(5, 5,
             labels = paste("Team", LETTERS[1:5])),
             player = sample(letters, 25),
             batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball.example$team,
       max)
```

In `pandas` we may use *pivot_table()* method to handle this:

```
In [14]: import random

In [15]: import string

In [16]: baseball = pd.DataFrame(
   ....:     {'team': ["team %d" % (x + 1) for x in range(5)] * 5,
   ....:      'player': random.sample(list(string.ascii_lowercase), 25),
   ....:      'batting avg': np.random.uniform(.200, .400, 25)})
   ....:

In [17]: baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)
Out[17]:
```

(continues on next page)

(continued from previous page)

```
team            team 1    team 2    team 3    team 4    team 5
batting avg  0.352134  0.295327  0.397191  0.394457  0.396194
```

For more details and examples see *the reshaping documentation*.

subset

The `query()` method is similar to the base R `subset` function. In R you might want to get the rows of a `data.frame` where one column's values are less than another column's values:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,]  # note the comma
```

In `pandas`, there are a few ways to perform subsetting. You can use `query()` or pass an expression as if it were an index/slice as well as standard boolean indexing:

```
In [18]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [19]: df.query('a <= b')
Out[19]: 
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [20]: df[df.a <= df.b]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [21]: df.loc[df.a <= df.b]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550
```

For more details and examples see *the query documentation*.

with

An expression using a data.frame called `df` in R with the columns `a` and `b` would be evaluated using `with` like so:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b  # same as the previous expression
```

In `pandas` the equivalent expression, using the *eval()* method, would be:

```
In [22]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [23]: df.eval('a + b')
Out[23]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64

In [24]: df.a + df.b  # same as the previous expression
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64
```

In certain cases *eval()* will be much faster than evaluation in pure Python. For more details and examples see *the eval documentation*.

plyr

`plyr` is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, `a` for `arrays`, `l` for `lists`, and `d` for `data.frame`. The table below shows how these data structures could be mapped in Python.

R	Python
array	list
lists	dictionary or list of objects
data.frame	dataframe

ddply

An expression using a data.frame called `df` in R where you want to summarize `x` by `month`:

```
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
      mean = round(mean(x), 2),
      sd = round(sd(x), 2))
```

In `pandas` the equivalent expression, using the *groupby()* method, would be:

```
In [25]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 120),
   ....:                    'y': np.random.uniform(7., 334., 120),
   ....:                    'z': np.random.uniform(1.7, 20.7, 120),
   ....:                    'month': [5, 6, 7, 8] * 30,
   ....:                    'week': np.random.randint(1, 4, 120)})
   ....:

In [26]: grouped = df.groupby(['month', 'week'])

In [27]: grouped['x'].agg([np.mean, np.std])
Out[27]:
                  mean        std
month week
5     1      63.653367  40.601965
      2      78.126605  53.342400
      3      92.091886  57.630110
6     1      81.747070  54.339218
      2      70.971205  54.687287
      3     100.968344  54.010081
7     1      61.576332  38.844274
      2      61.733510  48.209013
      3      71.688795  37.595638
8     1      62.741922  34.618153
      2      91.774627  49.790202
      3      73.936856  60.773900
```

For more details and examples see *the groupby documentation*.

reshape / reshape2

melt.array

An expression using a 3 dimensional array called `a` in R where you want to melt it into a data.frame:

```
a <- array(c(1:23, NA), c(2,3,4))
data.frame(melt(a))
```

In Python, since `a` is a list, you can simply use list comprehension.

```
In [28]: a = np.array(list(range(1, 24)) + [np.NAN]).reshape(2, 3, 4)

In [29]: pd.DataFrame([tuple(list(x) + [val]) for x, val in np.ndenumerate(a)])
Out[29]:
    0  1  2     3
0   0  0  0   1.0
1   0  0  1   2.0
2   0  0  2   3.0
3   0  0  3   4.0
4   0  1  0   5.0
5   0  1  1   6.0
6   0  1  2   7.0
.. .. .. ..   ...
17  1  1  1  18.0
18  1  1  2  19.0
19  1  1  3  20.0
20  1  2  0  21.0
21  1  2  1  22.0
22  1  2  2  23.0
23  1  2  3   NaN

[24 rows x 4 columns]
```

melt.list

An expression using a list called `a` in R where you want to melt it into a data.frame:

```
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so *`DataFrame()`* method would convert it to a dataframe as required.

```
In [30]: a = list(enumerate(list(range(1, 5)) + [np.NAN]))

In [31]: pd.DataFrame(a)
Out[31]:
   0    1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  NaN
```

For more details and examples see *the Into to Data Structures documentation*.

melt.data.frame

An expression using a data.frame called `cheese` in R where you want to reshape the data.frame:

```
cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
```

(continues on next page)

(continued from previous page)

```
)
melt(cheese, id=c("first", "last"))
```

In Python, the *melt()* method is the R equivalent:

```
In [32]: cheese = pd.DataFrame({'first': ['John', 'Mary'],
   ....:                        'last': ['Doe', 'Bo'],
   ....:                        'height': [5.5, 6.0],
   ....:                        'weight': [130, 150]})
   ....:

In [33]: pd.melt(cheese, id_vars=['first', 'last'])
Out[33]:
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0

In [34]: cheese.set_index(['first', 'last']).stack()  # alternative way
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
first  last
John   Doe   height      5.5
             weight    130.0
Mary   Bo    height      6.0
             weight    150.0
dtype: float64
```

For more details and examples see *the reshaping documentation*.

cast

In R `acast` is an expression using a data.frame called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7),4),
  week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python the best way is to make use of *pivot_table()*:

```
In [35]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 12),
   ....:                    'y': np.random.uniform(7., 334., 12),
   ....:                    'z': np.random.uniform(1.7, 20.7, 12),
   ....:                    'month': [5, 6, 7] * 4,
   ....:                    'week': [1, 2] * 6})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [36]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [37]: pd.pivot_table(mdf, values='value', index=['variable', 'week'],
   ....:                columns=['month'], aggfunc=np.mean)
   ....:
Out[37]:
month                  5           6           7
variable week
x        1      93.888747   98.762034   55.219673
         2      94.391427   38.112932   83.942781
y        1      94.306912  279.454811  227.840449
         2      87.392662  193.028166  173.899260
z        1      11.016009   10.079307   16.170549
         2       8.476111   17.638509   19.003494
```

Similarly for `dcast` which uses a data.frame called `df` in R to aggregate information based on `Animal` and `FeedType`:

```
df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
             'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))
```

Python can approach this in two different ways. Firstly, similar to above using *pivot_table()*:

```
In [38]: df = pd.DataFrame({
   ....:     'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
   ....:                'Animal2', 'Animal3'],
   ....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
   ....:     'Amount': [10, 7, 4, 2, 5, 6, 2],
   ....: })
   ....:

In [39]: df.pivot_table(values='Amount', index='Animal', columns='FeedType',
   ....:                aggfunc='sum')
   ....:
Out[39]:
FeedType     A     B
Animal
Animal1   10.0   5.0
Animal2    2.0  13.0
Animal3    6.0   NaN
```

The second approach is to use the *groupby()* method:

```
In [40]: df.groupby(['Animal', 'FeedType'])['Amount'].sum()
Out[40]:
Animal   FeedType
Animal1  A           10
         B            5
```

(continues on next page)

(continued from previous page)

```
Animal2  A           2
         B          13
Animal3  A           6
Name: Amount, dtype: int64
```

For more details and examples see *the reshaping documentation* or *the groupby documentation*.

factor

pandas has a data type for categorical data.

```
cut(c(1,2,3,4,5,6), 3)
factor(c(1,2,3,2,2,3))
```

In pandas this is accomplished with `pd.cut` and `astype("category")`:

```
In [41]: pd.cut(pd.Series([1, 2, 3, 4, 5, 6]), 3)
Out[41]:
0    (0.995, 2.667]
1    (0.995, 2.667]
2    (2.667, 4.333]
3    (2.667, 4.333]
4      (4.333, 6.0]
5      (4.333, 6.0]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]

In [42]: pd.Series([1, 2, 3, 2, 2, 3]).astype("category")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    2
2    3
3    2
4    2
5    3
dtype: category
Categories (3, int64): [1, 2, 3]
```

For more details and examples see *categorical introduction* and the *API documentation*. There is also a documentation regarding the *differences to R's factor*.

3.5.2 Comparison with SQL

Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Most of the examples will utilize the `tips` dataset found within pandas tests. We'll read the data into a DataFrame called *tips* and assume we have a database table of the same name and structure.

```
In [3]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/data/tips.csv')
   ...:

In [4]: tips = pd.read_csv(url)

In [5]: tips.head()
Out[5]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

SELECT

In SQL, selection is done using a comma-separated list of columns you'd like to select (or a `*` to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
Out[6]:
   total_bill   tip smoker    time
0       16.99  1.01     No  Dinner
1       10.34  1.66     No  Dinner
2       21.01  3.50     No  Dinner
3       23.68  3.31     No  Dinner
4       24.59  3.61     No  Dinner
```

Calling the DataFrame without the list of column names would display all columns (akin to SQL's `*`).

WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```
In [7]: tips[tips['time'] == 'Dinner'].head(5)
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
```

(continues on next page)

(continued from previous page)

```
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The above statement is simply passing a `Series` of True/False objects to the DataFrame, returning all rows with True.

```
In [8]: is_dinner = tips['time'] == 'Dinner'

In [9]: is_dinner.value_counts()
Out[9]:
True     176
False     68
Name: time, dtype: int64

In [10]: tips[is_dinner].head(5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Just like SQL's OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

```
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```

```
# tips of more than $5.00 at Dinner meals
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
Out[11]:
     total_bill    tip     sex smoker  day    time  size
23        39.42   7.58    Male     No  Sat  Dinner     4
44        30.40   5.60    Male     No  Sun  Dinner     4
47        32.40   6.00    Male     No  Sun  Dinner     4
52        34.81   5.20  Female     No  Sun  Dinner     4
59        48.27   6.73    Male     No  Sat  Dinner     4
116       29.93   5.07    Male     No  Sun  Dinner     4
155       29.85   5.14  Female     No  Sun  Dinner     5
170       50.81  10.00    Male    Yes  Sat  Dinner     3
172        7.25   5.15    Male    Yes  Sun  Dinner     2
181       23.33   5.65    Male    Yes  Sun  Dinner     2
183       23.17   6.50    Male    Yes  Sun  Dinner     4
211       25.89   5.16    Male    Yes  Sat  Dinner     4
212       48.33   9.00    Male     No  Sat  Dinner     4
214       28.17   6.50  Female    Yes  Sat  Dinner     3
239       29.03   5.92    Male     No  Sat  Dinner     3
```

```
-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

```
# tips by parties of at least 5 diners OR bill total was more than $45
In [12]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
Out[12]:
     total_bill    tip     sex smoker   day    time  size
59        48.27   6.73    Male     No   Sat  Dinner     4
125       29.80   4.20  Female     No  Thur   Lunch     6
141       34.30   6.70    Male     No  Thur   Lunch     6
142       41.19   5.00    Male     No  Thur   Lunch     5
143       27.05   5.00  Female     No  Thur   Lunch     6
155       29.85   5.14  Female     No   Sun  Dinner     5
156       48.17   5.00    Male     No   Sun  Dinner     6
170       50.81  10.00    Male    Yes   Sat  Dinner     3
182       45.35   3.50    Male    Yes   Sun  Dinner     3
185       20.69   5.00    Male     No   Sun  Dinner     5
187       30.46   2.00    Male    Yes   Sun  Dinner     5
212       48.33   9.00    Male     No   Sat  Dinner     4
216       28.15   3.00    Male    Yes   Sat  Dinner     5
```

NULL checking is done using the *notna()* and *isna()* methods.

```
In [13]: frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],
   ....:                       'col2': ['F', np.NaN, 'G', 'H', 'I']})
   ....:

In [14]: frame
Out[14]:
  col1 col2
0    A    F
1    B  NaN
2  NaN    G
3    C    H
4    D    I
```

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```
In [15]: frame[frame['col2'].isna()]
Out[15]:
  col1 col2
1    B  NaN
```

Getting items where `col1` IS NOT NULL can be done with *notna()*.

```
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```
In [16]: frame[frame['col1'].notna()]
Out[16]:
  col1 col2
0    A    F
1    B  NaN
```

(continues on next page)

(continued from previous page)

```
3    C    H
4    D    I
```

GROUP BY

In pandas, SQL's GROUP BY operations are performed using the similarly named *groupby()* method. *groupby()* typically refers to a process where we'd like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female     87
Male      157
*/
```

The pandas equivalent would be:

```
In [17]: tips.groupby('sex').size()
Out[17]:
sex
Female     87
Male      157
dtype: int64
```

Notice that in the pandas code we used *size()* and not *count()*. This is because *count()* applies the function to each column, returning the number of `not null` records within each.

```
In [18]: tips.groupby('sex').count()
Out[18]:
        total_bill  tip  smoker  day  time  size
sex
Female          87   87      87   87    87    87
Male           157  157     157  157   157   157
```

Alternatively, we could have applied the *count()* method to an individual column:

```
In [19]: tips.groupby('sex')['total_bill'].count()
Out[19]:
sex
Female     87
Male      157
Name: total_bill, dtype: int64
```

Multiple functions can also be applied at once. For instance, say we'd like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

```
SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
```

(continues on next page)

(continued from previous page)

```
/*
Fri    2.734737   19
Sat    2.993103   87
Sun    3.255132   76
Thur   2.771452   62
*/
```

```
In [20]: tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
Out[20]:
           tip  day
day
Fri   2.734737   19
Sat   2.993103   87
Sun   3.255132   76
Thur  2.771452   62
```

Grouping by more than one column is done by passing a list of columns to the *groupby()* method.

```
SELECT smoker, day, COUNT(*), AVG(tip)
FROM tips
GROUP BY smoker, day;
/*
smoker day
No     Fri      4  2.812500
       Sat     45  3.102889
       Sun     57  3.167895
       Thur    45  2.673778
Yes    Fri     15  2.714000
       Sat     42  2.875476
       Sun     19  3.516842
       Thur    17  3.030000
*/
```

```
In [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})
Out[21]:
             tip
            size      mean
smoker day
No     Fri    4.0  2.812500
       Sat   45.0  3.102889
       Sun   57.0  3.167895
       Thur  45.0  2.673778
Yes    Fri   15.0  2.714000
       Sat   42.0  2.875476
       Sun   19.0  3.516842
       Thur  17.0  3.030000
```

JOIN

JOINs can be performed with *join()* or *merge()*. By default, *join()* will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

```
In [22]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [23]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....:
```

Assume we have two database tables of the same name and structure as our DataFrames.

Now let's go over the various types of JOINs.

INNER JOIN

```
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
```

```
# merge performs an INNER JOIN by default
In [24]: pd.merge(df1, df2, on='key')
Out[24]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
```

`merge()` also offers parameters for cases when you'd like to join one DataFrame's column with another DataFrame's index.

```
In [25]: indexed_df2 = df2.set_index('key')

In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)
Out[26]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
3   D -1.135632  0.119209
```

LEFT OUTER JOIN

```
-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df1
In [27]: pd.merge(df1, df2, on='key', how='left')
Out[27]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
```

(continues on next page)

(continued from previous page)

```
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

RIGHT JOIN

```
-- show all records from df2
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df2
In [28]: pd.merge(df1, df2, on='key', how='right')
Out[28]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236
```

FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

```
-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from both frames
In [29]: pd.merge(df1, df2, on='key', how='outer')
Out[29]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

UNION

UNION ALL can be performed using *concat()*.

```
In [30]: df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],
   ....:                     'rank': range(1, 4)})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [31]: df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],
   ....:                     'rank': [1, 4, 5]})
   ....:
```

```
SELECT city, rank
FROM df1
UNION ALL
SELECT city, rank
FROM df2;
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
      Chicago     1
       Boston     4
  Los Angeles     5
*/
```

```
In [32]: pd.concat([df1, df2])
Out[32]:
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
0        Chicago     1
1         Boston     4
2    Los Angeles     5
```

SQL's UNION is similar to UNION ALL, however UNION will remove duplicate rows.

```
SELECT city, rank
FROM df1
UNION
SELECT city, rank
FROM df2;
-- notice that there is only one Chicago record this time
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
       Boston     4
  Los Angeles     5
*/
```

In pandas, you can use `concat()` in conjunction with `drop_duplicates()`.

```
In [33]: pd.concat([df1, df2]).drop_duplicates()
Out[33]:
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
1         Boston     4
```

(continues on next page)

(continued from previous page)

```
2     Los Angeles      5
```

Pandas equivalents for some SQL analytic and aggregate functions

Top N rows with offset

```
-- MySQL
SELECT * FROM tips
ORDER BY tip DESC
LIMIT 10 OFFSET 5;
```

```
In [34]: tips.nlargest(10 + 5, columns='tip').tail(10)
Out[34]:
     total_bill   tip     sex smoker   day    time  size
183       23.17  6.50    Male    Yes   Sun  Dinner     4
214       28.17  6.50  Female    Yes   Sat  Dinner     3
47        32.40  6.00    Male     No   Sun  Dinner     4
239       29.03  5.92    Male     No   Sat  Dinner     3
88        24.71  5.85    Male     No  Thur   Lunch     2
181       23.33  5.65    Male    Yes   Sun  Dinner     2
44        30.40  5.60    Male     No   Sun  Dinner     4
52        34.81  5.20  Female     No   Sun  Dinner     4
85        34.83  5.17  Female     No  Thur   Lunch     4
211       25.89  5.16    Male    Yes   Sat  Dinner     4
```

Top N rows per group

```
-- Oracle's ROW_NUMBER() analytic function
SELECT * FROM (
  SELECT
    t.*,
    ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn
  FROM tips t
)
WHERE rn < 3
ORDER BY day, rn;
```

```
In [35]: (tips.assign(rn=tips.sort_values(['total_bill'], ascending=False)
   ....:                     .groupby(['day'])
   ....:                     .cumcount() + 1)
   ....:      .query('rn < 3')
   ....:      .sort_values(['day', 'rn']))
   ....:
Out[35]:
     total_bill    tip     sex smoker   day    time  size  rn
95        40.17   4.73    Male    Yes   Fri  Dinner     4   1
90        28.97   3.00    Male    Yes   Fri  Dinner     2   2
170       50.81  10.00    Male    Yes   Sat  Dinner     3   1
212       48.33   9.00    Male     No   Sat  Dinner     4   2
156       48.17   5.00    Male     No   Sun  Dinner     6   1
182       45.35   3.50    Male    Yes   Sun  Dinner     3   2
```

(continues on next page)

(continued from previous page)

```
197        43.11   5.00  Female     Yes  Thur   Lunch     4   1
142        41.19   5.00    Male      No  Thur   Lunch     5   2
```

the same using *rank(method='first')* function

```
In [36]: (tips.assign(rnk=tips.groupby(['day'])['total_bill']
   ....:                      .rank(method='first', ascending=False))
   ....:       .query('rnk < 3')
   ....:       .sort_values(['day', 'rnk']))
   ....:
Out[36]:
     total_bill    tip     sex smoker   day    time  size  rnk
95        40.17   4.73    Male    Yes   Fri  Dinner     4  1.0
90        28.97   3.00    Male    Yes   Fri  Dinner     2  2.0
170       50.81  10.00    Male    Yes   Sat  Dinner     3  1.0
212       48.33   9.00    Male     No   Sat  Dinner     4  2.0
156       48.17   5.00    Male     No   Sun  Dinner     6  1.0
182       45.35   3.50    Male    Yes   Sun  Dinner     3  2.0
197       43.11   5.00  Female    Yes  Thur   Lunch     4  1.0
142       41.19   5.00    Male     No  Thur   Lunch     5  2.0
```

```
-- Oracle's RANK() analytic function
SELECT * FROM (
  SELECT
    t.*,
    RANK() OVER(PARTITION BY sex ORDER BY tip) AS rnk
  FROM tips t
  WHERE tip < 2
)
WHERE rnk < 3
ORDER BY sex, rnk;
```

Let's find tips with (rank < 3) per gender group for (tips < 2). Notice that when using `rank(method='min')` function *rnk_min* remains the same for the same *tip* (as Oracle's RANK() function)

```
In [37]: (tips[tips['tip'] < 2]
   ....:     .assign(rnk_min=tips.groupby(['sex'])['tip']
   ....:                         .rank(method='min'))
   ....:     .query('rnk_min < 3')
   ....:     .sort_values(['sex', 'rnk_min']))
   ....:
Out[37]:
     total_bill   tip     sex smoker  day    time  size  rnk_min
67         3.07  1.00  Female    Yes  Sat  Dinner     1      1.0
92         5.75  1.00  Female    Yes  Fri  Dinner     2      1.0
111        7.25  1.00  Female     No  Sat  Dinner     1      1.0
236       12.60  1.00    Male    Yes  Sat  Dinner     2      1.0
237       32.83  1.17    Male    Yes  Sat  Dinner     2      2.0
```

UPDATE

```
UPDATE tips
SET tip = tip*2
WHERE tip < 2;
```

```
In [38]: tips.loc[tips['tip'] < 2, 'tip'] *= 2
```

DELETE

```
DELETE FROM tips
WHERE tip > 9;
```

In pandas we select the rows that should remain, instead of deleting them

```
In [39]: tips = tips.loc[tips['tip'] <= 9]
```

3.5.3 Comparison with SAS

For potential users coming from SAS this page is meant to demonstrate how different SAS operations would be performed in pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Note: Throughout this tutorial, the pandas `DataFrame` will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the `DataFrame`. This is often used in interactive work (e.g. Jupyter notebook or terminal) - the equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

Data Structures

General Terminology Translation

pandas	SAS
`DataFrame`	data set
column	variable
row	observation
groupby	BY-group
`NaN`	.

`DataFrame` / `Series`

A `DataFrame` in pandas is analogous to a SAS data set - a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set using SAS's `DATA` step, can also be accomplished in pandas.

A `Series` is the data structure that represents one column of a `DataFrame`. SAS doesn't have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column in the `DATA` step.

`Index`

Every `DataFrame` and `Series` has an `Index` - which are labels on the *rows* of the data. SAS does not have an exactly analogous concept. A data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed during the `DATA` step (_N_).

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled `Index` or `MultiIndex` can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the *indexing documentation* for much more on how to use an `Index` effectively.

Data Input / Output

Constructing a DataFrame from Values

A SAS data set can be built from specified values by placing the data after a `datalines` statement and specifying the column names.

```
data df;
    input x y;
    datalines;
    1 2
    3 4
    5 6
    ;
run;
```

A pandas `DataFrame` can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({'x': [1, 3, 5], 'y': [2, 4, 6]})

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading External Data

Like SAS, pandas provides utilities for reading in data from many formats. The `tips` dataset, found within the pandas tests (csv) will be used in many of the following examples.

SAS provides `PROC IMPORT` to read csv data into a data set.

```
proc import datafile='tips.csv' dbms=csv out=tips replace;
    getnames=yes;
run;
```

The pandas method is *read_csv()*, which works similarly.

```
In [5]: url = ('https://raw.github.com/pandas-dev/'
   ...:        'pandas/master/pandas/tests/data/tips.csv')
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Like `PROC IMPORT`, `read_csv` can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

In addition to text/csv, pandas supports a variety of other data formats such as Excel, HDF5, and SQL databases. These are all read via a `pd.read_*` function. See the *IO documentation* for more details.

Exporting Data

The inverse of `PROC IMPORT` in SAS is `PROC EXPORT`

```
proc export data=tips outfile='tips2.csv' dbms=csv;
run;
```

Similarly in pandas, the opposite of `read_csv` is *to_csv()*, and other data formats follow a similar api.

```
tips.to_csv('tips2.csv')
```

Data Operations

Operations on Columns

In the `DATA` step, arbitrary math expressions can be used on new or existing columns.

```
data tips;
    set tips;
    total_bill = total_bill - 2;
    new_bill = total_bill / 2;
run;
```

pandas provides similar vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2.0

In [10]: tips.head()
Out[10]:
   total_bill   tip     sex smoker  day    time  size  new_bill
0       14.99  1.01  Female     No  Sun  Dinner     2     7.495
1        8.34  1.66    Male     No  Sun  Dinner     3     4.170
2       19.01  3.50    Male     No  Sun  Dinner     3     9.505
3       21.68  3.31    Male     No  Sun  Dinner     2    10.840
4       22.59  3.61  Female     No  Sun  Dinner     4    11.295
```

Filtering

Filtering in SAS is done with an `if` or `where` statement, on one or more columns.

```
data tips;
    set tips;
    if total_bill > 10;
run;

data tips;
    set tips;
    where total_bill > 10;
    /* equivalent in this case - where happens before the
       DATA step begins and can also be used in PROC statements */
run;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*

```
In [11]: tips[tips['total_bill'] > 10].head()
Out[11]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/Then Logic

In SAS, if/then logic can be used to create new columns.

```
data tips;
    set tips;
    format bucket $4.;

    if total_bill < 10 then bucket = 'low';
    else bucket = 'high';
run;
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [12]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [13]: tips.head()
Out[13]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date Functionality

SAS provides a variety of functions to do operations on date/datetime columns.

```
data tips;
    set tips;
    format date1 date2 date1_plusmonth mmddyy10.;
    date1 = mdy(1, 15, 2013);
    date2 = mdy(2, 15, 2015);
    date1_year = year(date1);
    date2_month = month(date2);
    * shift date to beginning of next interval;
    date1_next = intnx('MONTH', date1, 1);
    * count intervals between dates;
    months_between = intck('MONTH', date1, date2);
run;
```

The equivalent pandas operations are shown below. In addition to these functions pandas supports other Time Series features not available in Base SAS (such as resampling and custom offsets) - see the *timeseries documentation* for more details.

```
In [14]: tips['date1'] = pd.Timestamp('2013-01-15')

In [15]: tips['date2'] = pd.Timestamp('2015-02-15')

In [16]: tips['date1_year'] = tips['date1'].dt.year

In [17]: tips['date2_month'] = tips['date2'].dt.month

In [18]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [19]: tips['months_between'] = (
   ....:     tips['date2'].dt.to_period('M') - tips['date1'].dt.to_period('M'))
   ....:

In [20]: tips[['date1', 'date2', 'date1_year', 'date2_month',
   ....:       'date1_next', 'months_between']].head()
   ....:
Out[20]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of Columns

SAS provides keywords in the `DATA` step to select, drop, and rename columns.

```
data tips;
    set tips;
    keep sex total_bill tip;
run;

data tips;
    set tips;
    drop sex;
run;

data tips;
    set tips;
    rename total_bill=total_bill_2;
run;
```

The same operations are expressed in pandas below.

```
# keep
In [21]: tips[['sex', 'total_bill', 'tip']].head()
Out[21]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61

# drop
In [22]: tips.drop('sex', axis=1).head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [23]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by Values

Sorting in SAS is accomplished via `PROC SORT`

```
proc sort data=tips;
    by sex total_bill;
run;
```

pandas objects have a `sort_values()` method, which takes a list of columns to sort by.

```
In [24]: tips = tips.sort_values(['sex', 'total_bill'])

In [25]: tips.head()
Out[25]: 
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String Processing

Length

SAS determines the length of a character string with the LENGTHN and LENGTHC functions. `LENGTHN` excludes trailing blanks and `LENGTHC` includes trailing blanks.

```
data _null_;
set tips;
put(LENGTHN(time));
put(LENGTHC(time));
run;
```

Python determines the length of a character string with the `len` function. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [26]: tips['time'].str.len().head()
Out[26]: 
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64

In [27]: tips['time'].str.rstrip().str.len().head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[27]: 
↪
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64
```

Find

SAS determines the position of a character in a string with the FINDW function. `FINDW` takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
data _null_;
set tips;
put(FINDW(sex,'ale'));
run;
```

Python determines the position of a character in a string with the `find` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [28]: tips['sex'].str.find("ale").head()
Out[28]: 
67     3
92     3
111    3
145    3
135    3
Name: sex, dtype: int64
```

Substring

SAS extracts a substring from a string based on its position with the SUBSTR function.

```
data _null_;
set tips;
put(substr(sex,1,1));
run;
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [29]: tips['sex'].str[0:1].head()
Out[29]: 
67     F
92     F
111    F
145    F
135    F
Name: sex, dtype: object
```

Scan

The SAS SCAN function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
data firstlast;
input String $60.;
First_Name = scan(string, 1);
Last_Name = scan(string, -1);
```

(continues on next page)

(continued from previous page)

```
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [30]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [31]: firstlast['First_Name'] = firstlast['String'].str.split(" ", expand=True)[0]

In [32]: firstlast['Last_Name'] = firstlast['String'].str.rsplit(" ", expand=True)[0]

In [33]: firstlast
Out[33]:
       String First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Upcase, Lowcase, and Propcase

The SAS UPCASE LOWCASE and PROPCASE functions change the case of the argument.

```
data firstlast;
input String $60.;
string_up = UPCASE(string);
string_low = LOWCASE(string);
string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [34]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [35]: firstlast['string_up'] = firstlast['String'].str.upper()

In [36]: firstlast['string_low'] = firstlast['String'].str.lower()

In [37]: firstlast['string_prop'] = firstlast['String'].str.title()

In [38]: firstlast
Out[38]:
       String   string_up  string_low string_prop
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [39]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [40]: df1
Out[40]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [41]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [42]: df2
Out[42]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In SAS, data must be explicitly sorted before merging. Different types of joins are accomplished using the `in=` dummy variables to track whether a match was found in one or both input frames.

```
proc sort data=df1;
    by key;
run;

proc sort data=df2;
    by key;
run;

data left_join inner_join right_join outer_join;
    merge df1(in=a) df2(in=b);

    if a and b then output inner_join;
    if a then output left_join;
    if b then output right_join;
    if a or b then output outer_join;
run;
```

pandas DataFrames have a *merge()* method, which provides similar functionality. Note that the data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

```
In [43]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [44]: inner_join
Out[44]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
```

(continues on next page)

(continued from previous page)

```
2   D -1.135632  0.119209

In [45]: left_join = df1.merge(df2, on=['key'], how='left')

In [46]: left_join
Out[46]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [47]: right_join = df1.merge(df2, on=['key'], how='right')

In [48]: right_join
Out[48]: 
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [49]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [50]: outer_join
Out[50]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing Data

Like SAS, pandas has a representation for missing data - which is the special float value `NaN` (not a number). Many of the semantics are the same, for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [51]: outer_join
Out[51]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [52]: outer_join['value_x'] + outer_join['value_y']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0         NaN
```

(continues on next page)

(continued from previous page)

```
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [53]: outer_join['value_x'].sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪-3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in SAS you could do this to filter missing values.

```
data outer_join_nulls;
    set outer_join;
    if value_x = .;
run;

data outer_join_no_nulls;
    set outer_join;
    if value_x ^= .;
run;
```

Which doesn't work in pandas. Instead, the `pd.isna` or `pd.notna` functions should be used for comparisons.

```
In [54]: outer_join[pd.isna(outer_join['value_x'])]
Out[54]:
  key  value_x   value_y
5   E      NaN -1.044236

In [55]: outer_join[pd.notna(outer_join['value_x'])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[55]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

pandas also provides a variety of methods to work with missing data - some of which would be challenging to express in SAS. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the *missing data documentation* for more.

```
In [56]: outer_join.dropna()
Out[56]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [57]: outer_join.fillna(method='ffill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  key   value_x   value_y
0   A  0.469112       NaN
```

(continues on next page)

(continued from previous page)

```
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236

In [58]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

SAS's PROC SUMMARY can be used to group by one or more key variables and compute aggregations on numeric columns.

```
proc summary data=tips nway;
    class sex smoker;
    var total_bill tip;
    output out=tips_summed sum=;
run;
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [59]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()

In [60]: tips_summed.head()
Out[60]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In SAS, if the group aggregations need to be used with the original frame, it must be merged back together. For example, to subtract the mean for each observation by smoker group.

```
proc summary data=tips missing nway;
    class smoker;
    var total_bill;
```

(continues on next page)

(continued from previous page)

```
    output out=smoker_means mean(total_bill)=group_bill;
run;

proc sort data=tips;
    by smoker;
run;

data tips;
    merge tips(in=a) smoker_means(in=b);
    by smoker;
    adj_total_bill = total_bill - group_bill;
    if a and b;
run;
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [61]: gb = tips.groupby('smoker')['total_bill']

In [62]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [63]: tips.head()
Out[63]:
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By Group Processing

In addition to aggregation, pandas `groupby` can be used to replicate most other by group processing from SAS. For example, this `DATA` step reads the data by sex/smoker group and filters to the first entry for each.

```
proc sort data=tips;
   by sex smoker;
run;

data tips_first;
    set tips;
    by sex smoker;
    if FIRST.sex or FIRST.smoker then output;
run;
```

In pandas this would be written as:

```
In [64]: tips.groupby(['sex', 'smoker']).first()
Out[64]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other Considerations

Disk vs Memory

pandas operates exclusively in memory, where a SAS data set exists on disk. This means that the size of data able to be loaded in pandas is limited by your machine's memory, but also that the operations on that data may be faster.

If out of core processing is needed, one possibility is the dask.dataframe library (currently in development) which provides a subset of pandas functionality for an on-disk DataFrame

Data Interop

pandas provides a *read_sas()* method that can read SAS data saved in the XPORT or SAS7BDAT binary format.

```
libname xportout xport 'transport-file.xpt';
data xportout.tips;
    set tips(rename=(total_bill=tbill));
    * xport variable names limited to 6 characters;
run;
```

```
df = pd.read_sas('transport-file.xpt')
df = pd.read_sas('binary-file.sas7bdat')
```

You can also specify the file format directly. By default, pandas will try to infer the file format based on its extension.

```
df = pd.read_sas('transport-file.xpt', format='xport')
df = pd.read_sas('binary-file.sas7bdat', format='sas7bdat')
```

XPORT is a relatively limited format and the parsing of it is not as optimized as some of the other pandas readers. An alternative way to interop data between SAS and pandas is to serialize to csv.

```
# version 0.17, 10M rows

In [8]: %time df = pd.read_sas('big.xpt')
Wall time: 14.6 s

In [9]: %time df = pd.read_csv('big.csv')
Wall time: 4.86 s
```

3.5.4 Comparison with Stata

For potential users coming from Stata this page is meant to demonstrate how different Stata operations would be performed in pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows. This means that we can refer to the libraries as pd and np, respectively, for the rest of the document.

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Note: Throughout this tutorial, the pandas `DataFrame` will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the `DataFrame`. This is often used in interactive work (e.g. Jupyter notebook or terminal) – the equivalent in Stata would be:

```
list in 1/5
```

Data Structures

General Terminology Translation

pandas	Stata
`DataFrame`	data set
column	variable
row	observation
groupby	bysort
`NaN`	`.`

`DataFrame` / `Series`

A `DataFrame` in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

A `Series` is the data structure that represents one column of a `DataFrame`. Stata doesn't have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column of a data set in Stata.

`Index`

Every `DataFrame` and `Series` has an `Index` – labels on the *rows* of the data. Stata does not have an exactly analogous concept. In Stata, a data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed with `_n`.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled `Index` or `MultiIndex` can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the *indexing documentation* for much more on how to use an `Index` effectively.

Data Input / Output

Constructing a DataFrame from Values

A Stata data set can be built from specified values by placing the data after an `input` statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({'x': [1, 3, 5], 'y': [2, 4, 6]})

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading External Data

Like Stata, pandas provides utilities for reading in data from many formats. The tips data set, found within the pandas tests (csv) will be used in many of the following examples.

Stata provides import delimited to read csv data into a data set in memory. If the tips.csv file is in the current working directory, we can import it as follows.

```
import delimited tips.csv
```

The pandas method is *read_csv()*, which works similarly. Additionally, it will automatically download the data set if presented with a url.

```
In [5]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/data/tips.csv')
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Like import delimited, *read_csv()* can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

Pandas can also read Stata data sets in .dta format with the *read_stata()* function.

```
df = pd.read_stata('data.dta')
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a `pd.read_*` function. See the *IO documentation* for more details.

Exporting Data

The inverse of `import delimited` in Stata is `export delimited`

```
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is `DataFrame.to_csv()`.

```
tips.to_csv('tips2.csv')
```

Pandas can also export to Stata file format with the `DataFrame.to_stata()` method.

```
tips.to_stata('tips2.dta')
```

Data Operations

Operations on Columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```
replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill
```

pandas provides similar vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2

In [10]: tips.head()
Out[10]:
   total_bill   tip     sex smoker  day    time  size  new_bill
0       14.99  1.01  Female     No  Sun  Dinner     2     7.495
1        8.34  1.66    Male     No  Sun  Dinner     3     4.170
2       19.01  3.50    Male     No  Sun  Dinner     3     9.505
3       21.68  3.31    Male     No  Sun  Dinner     2    10.840
4       22.59  3.61  Female     No  Sun  Dinner     4    11.295

In [11]: tips = tips.drop('new_bill', axis=1)
```

Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [12]: tips[tips['total_bill'] > 10].head()
Out[12]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/Then Logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [13]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [14]: tips.head()
Out[14]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date Functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the *timeseries documentation* for more details.

```
In [15]: tips['date1'] = pd.Timestamp('2013-01-15')

In [16]: tips['date2'] = pd.Timestamp('2015-02-15')

In [17]: tips['date1_year'] = tips['date1'].dt.year

In [18]: tips['date2_month'] = tips['date2'].dt.month

In [19]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [20]: tips['months_between'] = (tips['date2'].dt.to_period('M')
   ....:                            - tips['date1'].dt.to_period('M'))
   ....:

In [21]: tips[['date1', 'date2', 'date1_year', 'date2_month', 'date1_next',
   ....:       'months_between']].head()
   ....:
Out[21]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of Columns

Stata provides keywords to select, drop, and rename columns.

```
keep sex total_bill tip

drop sex

rename total_bill total_bill_2
```

The same operations are expressed in pandas below. Note that in contrast to Stata, these operations do not happen in place. To make these changes persist, assign the operation back to a variable.

```
# keep
In [22]: tips[['sex', 'total_bill', 'tip']].head()
Out[22]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61

# drop
In [23]: tips.drop('sex', axis=1).head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
```

(continues on next page)

(continued from previous page)

```
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [24]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by Values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas objects have a *`DataFrame.sort_values()`* method, which takes a list of columns to sort by.

```
In [25]: tips = tips.sort_values(['sex', 'total_bill'])

In [26]: tips.head()
Out[26]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String Processing

Finding Length of String

Stata determines the length of a character string with the `strlen()` and `ustrlen()` functions for ASCII and Unicode strings, respectively.

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

Python determines the length of a character string with the `len` function. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [27]: tips['time'].str.len().head()
Out[27]:
67     6
92     6
111    6
```

(continues on next page)

(continued from previous page)

```
145     5
135     5
Name: time, dtype: int64

In [28]: tips['time'].str.rstrip().str.len().head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]:
↪
67      6
92      6
111     6
145     5
135     5
Name: time, dtype: int64
```

Finding Position of Substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
generate str_position = strpos(sex, "ale")
```

Python determines the position of a character in a string with the `find()` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [29]: tips['sex'].str.find("ale").head()
Out[29]:
67      3
92      3
111     3
145     3
135     3
Name: sex, dtype: int64
```

Extracting Substring by Position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [30]: tips['sex'].str[0:1].head()
Out[30]:
67      F
92      F
111     F
145     F
135     F
Name: sex, dtype: object
```

Extracting nth Word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [31]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [32]: firstlast['First_Name'] = firstlast['string'].str.split(" ", expand=True)[0]

In [33]: firstlast['Last_Name'] = firstlast['string'].str.rsplit(" ", expand=True)[0]

In [34]: firstlast
Out[34]:
       string First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Changing Case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [35]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [36]: firstlast['upper'] = firstlast['string'].str.upper()

In [37]: firstlast['lower'] = firstlast['string'].str.lower()

In [38]: firstlast['title'] = firstlast['string'].str.title()
```

(continues on next page)

(continued from previous page)

```
In [39]: firstlast
Out[39]:
       string        upper        lower        title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [40]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [41]: df1
Out[41]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [42]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [43]: df2
Out[43]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
end
generate value = rnormal()
save df2.dta

* Now create df1 in memory
clear
input str1 key
```

(continues on next page)

(continued from previous page)

```
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas DataFrames have a *DataFrame.merge()* method, which provides similar functionality. Note that different join types are accomplished via the `how` keyword.

```
In [44]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [45]: inner_join
Out[45]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209

In [46]: left_join = df1.merge(df2, on=['key'], how='left')

In [47]: left_join
Out[47]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [48]: right_join = df1.merge(df2, on=['key'], how='right')

In [49]: right_join
Out[49]:
  key   value_x   value_y
0   B -0.282863  1.212112
```

(continues on next page)

(continued from previous page)

```
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [50]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [51]: outer_join
Out[51]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing Data

Like Stata, pandas has a representation for missing data – the special float value NaN (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [52]: outer_join
Out[52]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [53]: outer_join['value_x'] + outer_join['value_y']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0         NaN
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [54]: outer_join['value_x'].sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪-3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

This doesn't work in pandas. Instead, the `pd.isna()` or `pd.notna()` functions should be used for comparisons.

```
In [55]: outer_join[pd.isna(outer_join['value_x'])]
Out[55]:
  key  value_x   value_y
5   E      NaN -1.044236

In [56]: outer_join[pd.notna(outer_join['value_x'])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

Pandas also provides a variety of methods to work with missing data – some of which would be challenging to express in Stata. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the *missing data documentation* for more.

```
# Drop rows with any missing value
In [57]: outer_join.dropna()
Out[57]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

# Fill forwards
In [58]: outer_join.fillna(method='ffill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236

# Impute missing values with the mean
In [59]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

Stata's `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [60]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()

In [61]: tips_summed.head()
Out[61]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [62]: gb = tips.groupby('smoker')['total_bill']

In [63]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [64]: tips.head()
Out[64]:
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By Group Processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [65]: tips.groupby(['sex', 'smoker']).first()
Out[65]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
```

(continues on next page)

(continued from previous page)

```
Female No              5.25  1.00   Sat  Dinner     1      -11.938278
       Yes             1.07  1.00   Sat  Dinner     1      -17.686344
Male   No              5.51  2.00  Thur   Lunch     2      -11.678278
       Yes             5.25  5.15   Sun  Dinner     2      -13.506344
```

Other Considerations

Disk vs Memory

Pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine's memory. If out of core processing is needed, one possibility is the dask.dataframe library, which provides a subset of pandas functionality for an on-disk `DataFrame`.

3.6 Tutorials

This is a guide to many pandas tutorials, geared mainly for new users.

3.6.1 Internal Guides

pandas' own *10 Minutes to pandas*.

More complex recipes are in the *Cookbook*.

A handy pandas cheat sheet.

3.6.2 Community Guides

pandas Cookbook by Julia Evans

The goal of this 2015 cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails. For the table of contents, see the pandas-cookbook GitHub repository.

Learn Pandas by Hernan Rojas

A set of lesson for new pandas users: https://bitbucket.org/hrojas/learn-pandas

Practical data analysis with Python

This guide is an introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as munging data, aggregating data, visualizing data and time series.

Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

Modern pandas

Tutorial series written in 2016 by Tom Augspurger. The source may be found in the GitHub repository TomAugspurger/effective-pandas.

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization
- Timeseries

Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

Video Tutorials

- Pandas From The Ground Up (2015) (2:24) GitHub repo
- Introduction Into Pandas (2016) (1:28) GitHub repo
- Pandas: .head() to .tail() (2016) (1:26) GitHub repo
- Data analysis in Python with pandas (2016-2018) GitHub repo and Jupyter Notebook
- Best practices with pandas (2018) GitHub repo and Jupyter Notebook

Various Tutorials

- Wes McKinney's (pandas BDFL) blog
- Statistical analysis made easy in Python with SciPy and pandas DataFrames, by Randal Olson
- Statistical Data Analysis in Python, tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- Financial analysis in Python, by Thomas Wiecki
- Intro to pandas data structures, by Greg Reda
- Pandas and Python: Top 10, by Manish Amde
- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples

CHAPTER

FOUR

USER GUIDE

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as "working with missing data"), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

Further information on any specific method can be obtained in the *API Reference*.

4.1 IO Tools (Text, CSV, HDF5, . . .)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	*read_csv*	*to_csv*
text	JSON	*read_json*	*to_json*
text	HTML	*read_html*	*to_html*
text	Local clipboard	*read_clipboard*	*to_clipboard*
binary	MS Excel	*read_excel*	*to_excel*
binary	HDF5 Format	*read_hdf*	*to_hdf*
binary	Feather Format	*read_feather*	*to_feather*
binary	Parquet Format	*read_parquet*	*to_parquet*
binary	Msgpack	*read_msgpack*	*to_msgpack*
binary	Stata	*read_stata*	*to_stata*
binary	SAS	*read_sas*	
binary	Python Pickle Format	*read_pickle*	*to_pickle*
SQL	SQL	*read_sql*	*to_sql*
SQL	Google Big Query	*read_gbq*	*to_gbq*

Here is an informal performance comparison for some of these IO methods.

Note: For examples that use the `StringIO` class, make sure you import it according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

4.1.1 CSV & Text files

The workhorse function for reading text files (a.k.a. flat files) is `read_csv()`. See the *cookbook* for some advanced strategies.

Parsing options

`read_csv()` accepts the following common arguments:

Basic

filepath_or_buffer [various] Either a path to a file (a `str`, `pathlib.Path`, or `py._path.local.LocalPath`), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or `StringIO`).

sep [str, defaults to `','` for `read_csv()`, `\t` for `read_table()`] Delimiter to use. If sep is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

delimiter [str, default `None`] Alternative argument name for sep.

delim_whitespace [boolean, default False] Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

Column and Index Locations and Names

header [int or list of ints, default `'infer'`] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names.

The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so header=0 denotes the first line of data rather than the first line of the file.

names [array-like, default `None`] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

index_col [int or sequence or `False`, default `None`] Column to use as the row labels of the `DataFrame`. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to *not* use the first column as the index (row names).

usecols [list-like or callable, default `None`] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo',` 'bar'])[['foo', 'bar']] for columns in ['foo', 'bar'] order or pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```
In [1]: from pandas.compat import StringIO, BytesIO

In [2]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

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

In [4]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['COL1', 'COL3
↪'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:
↪
  col1  col3
0    a     1
1    a     2
2    c     3
```

Using this parameter results in much faster parsing time and lower memory usage.

squeeze [boolean, default False] If the parsed data only contains one column then return a Series.

prefix [str, default None] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...

mangle_dupe_cols [boolean, default True] Duplicate columns will be specified as 'X', 'X.1'...'X.N', rather than 'X'...'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

General Parsing Configuration

dtype [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (unsupported with engine='python'). Use *str* or *object* together with suitable na_values settings to preserve and not interpret dtype.

New in version 0.20.0: support for the Python parser.

engine [{'c', 'python'}] Parser engine to use. The C engine is faster while the Python engine is currently more feature-complete.

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, default None] Values to consider as True.

false_values [list, default None] Values to consider as False.

skipinitialspace [boolean, default False] Skip spaces after delimiter.

skiprows [list-like or integer, default None] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```
In [5]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

In [6]: pd.read_csv(StringIO(data))
Out[6]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [7]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]:
↪
  col1 col2  col3
0    a    b     2
```

skipfooter [int, default 0] Number of lines at bottom of file to skip (unsupported with engine='c').

nrows [int, default None] Number of rows of file to read. Useful for reading pieces of large files.

low_memory [boolean, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser)

memory_map [boolean, default False] If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

NA and Missing Data Handling

na_values [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See *na values const* below for a list of the values interpreted as NaN by default.

keep_default_na [boolean, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is True, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is True, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is False, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is False, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as False, the *keep_default_na* and *na_values* parameters will be ignored.

na_filter [boolean, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbose [boolean, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [boolean, default `True`] If `True`, skip over blank lines rather than interpreting as NaN values.

Datetime Handling

parse_dates [boolean or list of ints or names or list of lists or dict, default `False`.]

- If `True` -> try parsing the index.
- If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- If `{'foo': [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. A fast-path exists for iso8601-formatted dates.

infer_datetime_format [boolean, default `False`] If `True` and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

keep_date_col [boolean, default `False`] If `True` and parse_dates specifies combining multiple columns then keep the original columns.

date_parser [function, default `None`] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirst [boolean, default `False`] DD/MM format dates, international and European format.

Iteration

iterator [boolean, default `False`] Return *TextFileReader* object for iteration or getting chunks with `get_chunk()`.

chunksize [int, default `None`] Return *TextFileReader* object for iteration. See *iterating and chunking* below.

Quoting, Compression, and File Format

compression [{`'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `None`}, default `'infer'`] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip, or xz if filepath_or_buffer is a string ending in '.gz', '.bz2', '.zip', or '.xz', respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to `None` for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

Changed in version 0.24.0: 'infer' option added and set to default.

thousands [str, default `None`] Thousands separator.

decimal [str, default `'.'`] Character to recognize as decimal point. E.g. use `','` for European data.

float_precision [string, default None] Specifies which converter the C engine should use for floating-point values. The options are `None` for the ordinary converter, `high` for the high-precision converter, and `round_trip` for the round-trip converter.

lineterminator [str (length 1), default `None`] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1)] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int or `csv.QUOTE_*` instance, default `0`] Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

doublequote [boolean, default `True`] When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements **inside** a field as a single `quotechar` element.

escapechar [str (length 1), default `None`] One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

comment [str, default `None`] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if `comment='#'`, parsing '#empty\na,b,c\n1,2,3' with *header=0* will result in 'a,b,c' being treated as the header.

encoding [str, default `None`] Encoding to use for UTF when reading/writing (e.g. `'utf-8'`). List of Python standard encodings.

dialect [str or `csv.Dialect` instance, default `None`] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a ParserWarning will be issued. See `csv.Dialect` documentation for more details.

tupleize_cols [boolean, default `False`]

Deprecated since version 0.21.0.

This argument will be removed and will always convert to MultiIndex

Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns).

Error Handling

error_bad_lines [boolean, default `True`] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If `False`, then these "bad lines" will dropped from the `DataFrame` that is returned. See *bad lines* below.

warn_bad_lines [boolean, default `True`] If error_bad_lines is `False`, and warn_bad_lines is `True`, a warning for each "bad line" will be output.

Specifying column data types

You can indicate the data type for the whole `DataFrame` or individual columns:

```
In [8]: data = ('a,b,c,d\n'
   ...:         '1,2,3,4\n'
   ...:         '5,6,7,8\n'
   ...:         '9,10,11')
   ...:

In [9]: print(data)
a,b,c,d
1,2,3,4
5,6,7,8
9,10,11

In [10]: df = pd.read_csv(StringIO(data), dtype=object)
```

(continues on next page)

(continued from previous page)

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

In [12]: df['a'][0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:
↪'1'

In [13]: df = pd.read_csv(StringIO(data),
   ....:                  dtype={'b': object, 'c': np.float64, 'd': 'Int64'})
   ....:

In [14]: df.dtypes
Out[14]:
a      int64
b     object
c    float64
d      Int64
dtype: object
```

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one `dtype`. If you're unfamiliar with these concepts, you can see *here* to learn more about dtypes, and *here* to learn more about `object` conversion in pandas.

For instance, you can use the `converters` argument of `read_csv()`:

```
In [15]: data = ("col_1\n"
   ....:         "1\n"
   ....:         "2\n"
   ....:         "'A'\n"
   ....:         "4.22")
   ....:

In [16]: df = pd.read_csv(StringIO(data), converters={'col_1': str})

In [17]: df
Out[17]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [18]: df['col_1'].apply(type).value_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]:
<class 'str'>    4
Name: col_1, dtype: int64
```

Or you can use the `to_numeric()` function to coerce the dtypes after reading in the data,

```
In [19]: df2 = pd.read_csv(StringIO(data))

In [20]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')
```

(continues on next page)

(continued from previous page)

```
In [21]: df2
Out[21]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22

In [22]: df2['col_1'].apply(type).value_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
<class 'float'>    4
Name: col_1, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as `NaN`.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to `NaN` out the data anomalies, then *`to_numeric()`* is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the `converters` argument of *`read_csv()`* would certainly be worth trying.

> New in version 0.20.0: support for the Python parser.
>
> The `dtype` option is supported by the 'python' engine.

Note: In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [23]: col_1 = list(range(500000)) + ['a', 'b'] + list(range(500000))

In [24]: df = pd.DataFrame({'col_1': col_1})

In [25]: df.to_csv('foo.csv')

In [26]: mixed_df = pd.read_csv('foo.csv')

In [27]: mixed_df['col_1'].apply(type).value_counts()
Out[27]:
<class 'int'>    737858
<class 'str'>    262144
Name: col_1, dtype: int64

In [28]: mixed_df['col_1'].dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]:␣
↪dtype('O')
```

will result with *mixed_df* containing an `int` dtype for certain chunks of the column, and `str` for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a `dtype` of `object`, which is used for columns with mixed dtypes.

Specifying Categorical dtype

New in version 0.19.0.

Categorical columns can be parsed directly by specifying dtype='category' or dtype=CategoricalDtype(categories, ordered).

```
In [29]: data = ('col1,col2,col3\n'
   ....:         'a,b,1\n'
   ....:         'a,b,2\n'
   ....:         'c,d,3')
   ....:

In [30]: pd.read_csv(StringIO(data))
Out[30]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [31]: pd.read_csv(StringIO(data)).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[31]:␣
↪
col1    object
col2    object
col3     int64
dtype: object

In [32]: pd.read_csv(StringIO(data), dtype='category').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
col1    category
col2    category
col3    category
dtype: object
```

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

```
In [33]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[33]:
col1    category
col2      object
col3       int64
dtype: object
```

New in version 0.21.0.

Specifying dtype='cateogry' will result in an unordered Categorical whose categories are the unique values observed in the data. For more control on the categories and order, create a CategoricalDtype ahead of time, and pass that for that column's dtype.

```
In [34]: from pandas.api.types import CategoricalDtype

In [35]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[36]:
col1    category
col2      object
col3       int64
dtype: object
```

When using dtype=CategoricalDtype, "unexpected" values outside of dtype.categories are treated as missing values.

```
In [37]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [38]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[38]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

This matches the behavior of `Categorical.set_categories()`.

Note: With `dtype='category'`, 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()*.

When `dtype` is a `CategoricalDtype` with homogeneous `categories` (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [39]: df = pd.read_csv(StringIO(data), dtype='category')

In [40]: df.dtypes
Out[40]:
col1    category
col2    category
col3    category
dtype: object

In [41]: df['col3']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[41]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [42]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [43]: df['col3']
Out[43]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and Using Columns

Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

```
In [44]: data = ('a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [45]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [46]: pd.read_csv(StringIO(data))
\\\\\\\\\\\\\\\\\\\\\\\Out[46]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

By specifying the `names` argument in conjunction with `header` you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [47]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [48]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
\\\\\\\\\\\\\\\\\\\\\\\Out[48]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [49]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  foo bar baz
0   a   b   c
1   1   2   3
2   4   5   6
3   7   8   9
```

If the header is in a row other than the first, pass the row number to `header`. This will skip the preceding rows:

```
In [50]: data = ('skip this skip it\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [51]: pd.read_csv(StringIO(data), header=1)
Out[51]:
   a  b  c
0  1  2  3
```

(continues on next page)

(continued from previous page)

```
1  4  5  6
2  7  8  9
```

Note: Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to `header=None`.

Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```
In [52]: data = ('a,b,a\n'
   ....:          '0,1,2\n'
   ....:          '3,4,5')
   ....:

In [53]: pd.read_csv(StringIO(data))
Out[53]:
   a  b  a.1
0  0  1    2
1  3  4    5
```

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns 'X', ..., 'X' to become 'X', 'X.1', ..., 'X.N'. If `mangle_dupe_cols=False`, duplicate data can arise:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
   a  b  a
0  2  1  2
1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

Filtering columns (`usecols`)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

New in version 0.20.0: support for callable *usecols* arguments

```
In [54]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'
```

(continues on next page)

(continued from previous page)

```
In [55]: pd.read_csv(StringIO(data))
Out[55]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [56]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [57]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [58]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  c
0  1  3
1  4  6
2  7  9
```

The `usecols` argument can also be used to specify which columns not to use in the final result:

```
In [59]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[59]:
   b    d
0  2  foo
1  5  bar
2  8  baz
```

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.

Comments and Empty Lines

Ignoring line comments and empty lines

If the `comment` parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

```
In [60]: data = ('\n'
   ....:         'a,b,c\n'
   ....:         '  \n'
   ....:         '# commented line\n'
   ....:         '1,2,3\n'
   ....:         '\n'
   ....:         '4,5,6')
```

(continues on next page)

(continued from previous page)

```
   ....:

In [61]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [62]: pd.read_csv(StringIO(data), comment='#')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[62]:
   a  b  c
0  1  2  3
1  4  5  6
```

If `skip_blank_lines=False`, then `read_csv` will not ignore blank lines:

```
In [63]: data = ('a,b,c\n'
   ....:         '\n'
   ....:         '1,2,3\n'
   ....:         '\n'
   ....:         '\n'
   ....:         '4,5,6')
   ....:

In [64]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[64]:
     a    b    c
0  NaN  NaN  NaN
1  1.0  2.0  3.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  4.0  5.0  6.0
```

Warning: The presence of ignored lines might create ambiguities involving line numbers; the parameter `header` uses row numbers (ignoring commented/empty lines), while `skiprows` uses line numbers (including commented/empty lines):

```
In [65]: data = ('#comment\n'
   ....:         'a,b,c\n'
   ....:         'A,B,C\n'
   ....:         '1,2,3')
   ....:

In [66]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[66]:
   A  B  C
0  1  2  3

In [67]: data = ('A,B,C\n'
   ....:         '#comment\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3')
   ....:

In [68]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[68]:
   a  b  c
0  1  2  3
```

If both header and skiprows are specified, header will be relative to the end of skiprows. For example:

```
In [69]: data = ('# empty\n'
   ....:         '# second empty line\n'
   ....:         '# third emptyline\n'
   ....:         'X,Y,Z\n'
   ....:         '1,2,3\n'
   ....:         'A,B,C\n'
   ....:         '1,2.,4.\n'
   ....:         '5.,NaN,10.0\n')
   ....:

In [70]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0


In [71]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[71]: 
↪
     A    B     C
0  1.0  2.0   4.0
1  5.0  NaN  10.0
```

Comments

Sometimes comments or meta data may be included in a file:

```
In [72]: print(open('tmp.csv').read())
ID,level,category
Patient1,123000,x # really unpleasant
Patient2,23000,y # wouldn't take his medicine
Patient3,1234018,z # awesome
```

By default, the parser includes the comments in the output:

```
In [73]: df = pd.read_csv('tmp.csv')

In [74]: df
Out[74]: 
         ID    level                        category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

We can suppress the comments using the comment keyword:

```
In [75]: df = pd.read_csv('tmp.csv', comment='#')

In [76]: df
Out[76]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z
```

Dealing with Unicode Data

The `encoding` argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

```
In [77]: data = (b'word,length\n'
   ....:         b'Tr\xc3\xa4umen,7\n'
   ....:         b'Gr\xc3\xbc\xc3\x9fe,5')
   ....:

In [78]: data = data.decode('utf8').encode('latin-1')

In [79]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [80]: df
Out[80]:
      word  length
0  Träumen       7
1    Grüße       5

In [81]: df['word'][1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[81]: 'Grüße'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won't parse correctly at all without specifying the encoding. Full list of Python standard encodings.

Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the `DataFrame`'s row names:

```
In [82]: data = ('a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....:

In [83]: pd.read_csv(StringIO(data))
Out[83]:
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

```
In [84]: data = ('index,a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
```

(continues on next page)

(continued from previous page)

```
   ....:

In [85]: pd.read_csv(StringIO(data), index_col=0)
Out[85]:
            a    b     c
index
4       apple  bat   5.7
8      orange  cow  10.0
```

Ordinarily, you can achieve this behavior using the `index_col` option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass `index_col=False`:

```
In [86]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [87]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [88]: pd.read_csv(StringIO(data))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[88]:
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [89]: pd.read_csv(StringIO(data), index_col=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\O
↪
   a       b    c
0  4   apple  bat
1  8  orange  cow
```

If a subset of data is being parsed using the `usecols` option, the `index_col` specification is based on that subset, not the original data.

```
In [90]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [91]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [92]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[92]:
     b   c
4  bat NaN
8  cow NaN

In [93]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[93]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying Date Columns

To better facilitate working with datetime data, `read_csv()` uses the keyword arguments `parse_dates` and `date_parser` to allow users to specify a variety of columns and date/time formats to turn the input text data into `datetime` objects.

The simplest case is to just pass in `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [94]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [95]: df
Out[95]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5

# These are Python datetime objects
In [96]: df.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[ns]',␣
↪name='date', freq=None)
```

It is often the case that we may want to store date and time data separately, or store various date fields separately. the `parse_dates` keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to `parse_dates`, the resulting date columns will be prepended to the output (so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

```
In [97]: print(open('tmp.csv').read())
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

In [98]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [99]: df
Out[99]:
                  1_2                 1_3     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
```

(continues on next page)

(continued from previous page)

```
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [100]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   .....:                  keep_date_col=True)
   .....:

In [101]: df
Out[101]:
                  1_2                 1_3     0         1         2         3     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127  19:00:00  18:56:00  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127  20:00:00  19:56:00  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127  21:00:00  20:56:00 -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127  21:00:00  21:18:00 -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127  22:00:00  21:56:00 -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127  23:00:00  22:56:00 -0.59
```

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, `parse_dates=[1, 2]` indicates that the second and third columns should each be parsed as separate date columns while `parse_dates=[[1, 2]]` means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [102]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [104]: df
Out[104]:
              nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The *index_col* specification is based off of this new set of columns rather than the original data columns:

```
In [105]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  index_col=0)  # index is the nominal column
   .....:

In [107]: df
Out[107]:
                                 actual     0     4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
```

(continues on next page)

(continued from previous page)

```
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Note: If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `to_datetime()` after `pd.read_csv`.

Note: read_csv has a fast_path for parsing datetime strings in iso8601 format, e.g "2000-01-01T00:01:02+00:00" and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

Note: When passing a dict as the *parse_dates* argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use *collections.OrderedDict* instead of a regular *dict* if this matters to you. Because of this, when using a dict for 'parse_dates' in conjunction with the *index_col* argument, it's best to specify *index_col* as a column label rather then as an index on the resulting frame.

Date Parsing Functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```
In [108]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  date_parser=pd.io.date_converters.parse_date_time)
   .....:

In [109]: df
Out[109]:
              nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Pandas will try to call the `date_parser` function in three different ways. If an exception is raised, the next one is tried:

1. `date_parser` is first called with one or more arrays as arguments, as defined using *parse_dates* (e.g., `date_parser(['2013', '2013'], ['1', '2'])`).
2. If #1 fails, `date_parser` is called with all the columns concatenated row-wise into a single array (e.g., `date_parser(['2013 1', '2013 2'])`).
3. If #2 fails, `date_parser` is called once for every row with one or more string arguments from the columns indicated with *parse_dates* (e.g., `date_parser('2013', '1')` for the first row, `date_parser('2013', '2')` for the second, etc.).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using `infer_datetime_format=True` (see section below).
2. If you know the format, use `pd.to_datetime()`: `date_parser=lambda x: pd.to_datetime(x, format=...)`.
3. If you have a really non-standard format, use a custom `date_parser` function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

You can explore the date parsing functionality in date_converters.py and add your own. We would love to turn this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contains functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single date rather than the entire array.

Inferring Datetime Format

If you have `parse_dates` enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting `infer_datetime_format=True`. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, `infer_datetime_format` should not have any negative consequences if enabled.

Here are some examples of datetime strings that can be guessed (All representing December 30th, 2011 at 00:00:00):

- "20111230"
- "2011/12/30"
- "20111230 00:00:00"
- "12/30/2011 00:00:00"
- "30/Dec/2011 00:00:00"
- "30/December/2011 00:00:00"

Note that `infer_datetime_format` is sensitive to `dayfirst`. With `dayfirst=True`, it will guess "01/12/2011" to be December 1st. With `dayfirst=False` (default) it will guess "01/12/2011" to be January 12th.

```
# Try to infer the format for the index column
In [110]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
   .....:                  infer_datetime_format=True)
   .....:

In [111]: df
Out[111]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

International Date Formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a `dayfirst` keyword is provided:

```
In [112]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [113]: pd.read_csv('tmp.csv', parse_dates=[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[113]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c

In [114]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        date  value cat
0 2000-06-01      5   a
1 2000-06-02     10   b
2 2000-06-03     15   c
```

Specifying method for floating-point conversion

The parameter `float_precision` can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```
In [115]: val = '0.3066101993807095471566981359501369297504425048828125'

In [116]: data = 'a,b,c\n1,2,{0}'.format(val)

In [117]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision=None)['c'][0] - float(val))
   .....:
Out[117]: 1.1102230246251565e-16

In [118]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='high')['c'][0] - float(val))
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[118]: 5.551115123125783e-17

In [119]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='round_trip')['c'][0] - float(val))
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[119]: 0.0
```

Thousand Separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```
In [120]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [121]: df = pd.read_csv('tmp.csv', sep='|')

In [122]: df
Out[122]:
         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [123]: df.level.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly:

```
In [124]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [125]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [126]: df
Out[126]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [127]: df.level.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪dtype('int64')
```

NA Values

To control which values are parsed as missing values (which are signified by `NaN`), specify a string in `na_values`. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a `float`, like `5.0` or an `integer` like `5`), the corresponding equivalent values will also imply a missing value (in this case effectively `[5.0, 5]` are recognized as `NaN`).

To completely override the default values that are recognized as missing, specify `keep_default_na=False`.

The default `NaN` recognized values are `['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', '']`.

Let us consider some examples:

```
pd.read_csv('path_to_file.csv', na_values=[5])
```

In the example above 5 and 5.0 will be recognized as NaN, in addition to the defaults. A string will first be interpreted as a numerical 5, then as a NaN.

```
pd.read_csv('path_to_file.csv', keep_default_na=False, na_values=[""])
```

Above, only an empty field will be recognized as NaN.

```
pd.read_csv('path_to_file.csv', keep_default_na=False, na_values=["NA", "0"])
```

Above, both NA and 0 as strings are NaN.

```
pd.read_csv('path_to_file.csv', na_values=["Nope"])
```

The default values, in addition to the string "Nope" are recognized as NaN.

Infinity

inf like values will be parsed as np.inf (positive infinity), and -inf as -np.inf (negative infinity). These will ignore the case of the value, meaning Inf, will also be parsed as np.inf.

Returning Series

Using the squeeze keyword, the parser will return output with a single column as a Series:

```
In [128]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [129]: output = pd.read_csv('tmp.csv', squeeze=True)

In [130]: output
Out[130]:
Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [131]: type(output)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪pandas.core.series.Series
```

Boolean values

The common values True, False, TRUE, and FALSE are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the true_values and false_values options as follows:

```
In [132]: data = ('a,b,c\n'
   .....:         '1,Yes,2\n'
   .....:         '3,No,4')
```

(continues on next page)

(continued from previous page)

```
   .....:

In [133]: print(data)
a,b,c
1,Yes,2
3,No,4

In [134]: pd.read_csv(StringIO(data))
\\\\\\\\\\\\\\\\\\\\\Out[134]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [135]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[135]:
   a      b  c
0  1   True  2
1  3  False  4
```

Handling "bad" lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many fields will raise an error by default:

```
In [136]: data = ('a,b,c\n'
   .....:         '1,2,3\n'
   .....:         '4,5,6,7\n'
   .....:         '8,9,10')
   .....:

In [137]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
<ipython-input-137-6388c394e6b8> in <module>
----> 1 pd.read_csv(StringIO(data))

/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, iterator,␣
↪chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting,␣
↪doublequote, escapechar, comment, encoding, dialect, tupleize_cols, error_bad_lines,
↪ warn_bad_lines, delim_whitespace, low_memory, memory_map, float_precision)
    695                     skip_blank_lines=skip_blank_lines)
    696
--> 697         return _read(filepath_or_buffer, kwds)
    698
    699     parser_f.__name__ = name

/pandas/pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
    428
    429     try:
--> 430         data = parser.read(nrows)
    431     finally:
```

(continues on next page)

(continued from previous page)

```
    432         parser.close()

/pandas/pandas/io/parsers.py in read(self, nrows)
   1132     def read(self, nrows=None):
   1133         nrows = _validate_integer('nrows', nrows)
-> 1134         ret = self._engine.read(nrows)
   1135
   1136         # May alter columns / col_dict

/pandas/pandas/io/parsers.py in read(self, nrows)
   1988     def read(self, nrows=None):
   1989         try:
-> 1990             data = self._reader.read(nrows)
   1991         except StopIteration:
   1992             if self._first_chunk:

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.read()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_low_memory()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_rows()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._tokenize_rows()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.raise_parser_error()

ParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4
```

You can elect to skip bad lines:

```
In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
Skipping line 3: expected 3 fields, saw 4

Out[29]:
   a  b   c
0  1  2   3
1  8  9  10
```

You can also use the `usecols` parameter to eliminate extraneous column data that appear in some lines but not others:

```
In [30]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

 Out[30]:
    a  b   c
 0  1  2   3
 1  4  5   6
 2  8  9  10
```

Dialect

The `dialect` keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a `csv.Dialect` instance.

Suppose you had data with unenclosed quotes:

```
In [138]: print(data)
label1,label2,label3
index1,"a,c,e
index2,b,d,f
```

By default, read_csv uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using dialect:

```
In [139]: import csv

In [140]: dia = csv.excel()

In [141]: dia.quoting = csv.QUOTE_NONE

In [142]: pd.read_csv(StringIO(data), dialect=dia)
Out[142]:
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [143]: data = 'a,b,c~1,2,3~4,5,6'

In [144]: pd.read_csv(StringIO(data), lineterminator='~')
Out[144]:
   a  b  c
0  1  2  3
1  4  5  6
```

Another common dialect option is skipinitialspace, to skip any whitespace after a delimiter:

```
In [145]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [146]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [147]: pd.read_csv(StringIO(data), skipinitialspace=True)
\\\\\\\\\\\\\\\\\\\\\\\\Out[147]:
   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to "do the right thing" and not be fragile. Type inference is a pretty big deal. If a column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the escapechar option:

```
In [148]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [149]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [150]: pd.read_csv(StringIO(data), escapechar='\\')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[150]:
                               a  b
0  hello, "Bob", nice to see you  5
```

Files with Fixed Width Columns

While *read_csv()* reads delimited data, the *read_fwf()* function works with data files that have known and fixed column widths. The function parameters to `read_fwf` are largely the same as *read_csv* with two extra parameters, and a different usage of the `delimiter` parameter:

- `colspecs`: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.
- `widths`: A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.
- `delimiter`: Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., '~').

Consider a typical fixed-width data file:

```
In [151]: print(open('bar.csv').read())
id8141    360.242940   149.910199   11950.7
id1594    444.953632   166.985655   11788.4
id1849    364.136849   183.628767   11806.2
id1230    413.836124   184.375703   11916.8
id1948    502.953953   173.237159   12468.3
```

In order to parse this file into a `DataFrame`, we simply need to supply the column specifications to the *read_fwf* function along with the file name:

```
# Column specifications are a list of half-intervals
In [152]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [153]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [154]: df
Out[154]:
                 1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
# Widths are a list of integers
In [155]: widths = [6, 14, 13, 10]

In [156]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [157]: df
Out[157]:
        0           1           2        3
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

The parser will take care of extra white spaces around the columns so it's ok to have extra separation between the columns in the file.

By default, `read_fwf` will try to infer the file's `colspecs` by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided `delimiter` (default delimiter is whitespace).

```
In [158]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [159]: df
Out[159]:
                 1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

New in version 0.20.0.

`read_fwf` supports the `dtype` parameter for specifying the types of parsed columns to be different from the inferred type.

```
In [160]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[160]:
1    float64
2    float64
3    float64
dtype: object

In [161]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[161]:
0     object
1    float64
2     object
3    float64
dtype: object
```

Indexes

Files with an "implicit" index column

Consider a file with one less entry in the header than the number of data column:

```
In [162]: print(open('foo.csv').read())
A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

In this special case, `read_csv` assumes that the first column is to be used as the index of the `DataFrame`:

```
In [163]: pd.read_csv('foo.csv')
Out[163]: 
          A  B  C
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Note that the dates weren't automatically parsed. In that case you would need to do as before:

```
In [164]: df = pd.read_csv('foo.csv', parse_dates=True)

In [165]: df.index
Out[165]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype=
↪'datetime64[ns]', freq=None)
```

Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [166]: print(open('data/mindex_ex.csv').read())
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

The `index_col` argument to `read_csv` can take a list of column numbers to turn multiple columns into a `MultiIndex` for the index of the returned object:

```
In [167]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [168]: df
```

(continues on next page)

(continued from previous page)

```
Out[168]:
                 zit   xit
year indiv
1977 A          1.20  0.60
     B          1.50  0.50
     C          1.70  0.80
1978 A          0.20  0.06
     B          0.70  0.20
     C          0.80  0.30
     D          0.90  0.50
     E          1.40  0.90
1979 C          0.20  0.15
     D          0.14  0.05
     E          0.50  0.15
     F          1.20  0.50
     G          3.40  1.90
     H          5.40  2.70
     I          6.40  1.20

In [169]: df.loc[1978]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       zit   xit
indiv
A      0.2  0.06
B      0.7  0.20
C      0.8  0.30
D      0.9  0.50
E      1.4  0.90
```

Reading columns with a `MultiIndex`

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the intervening rows.

```
In [170]: from pandas.util.testing import makeCustomDataframe as mkdf

In [171]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [172]: df.to_csv('mi.csv')

In [173]: print(open('mi.csv').read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2


In [174]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
C0              C_l0_g0 C_l0_g1 C_l0_g2
C1              C_l1_g0 C_l1_g1 C_l1_g2
C2              C_l2_g0 C_l2_g1 C_l2_g2
C3              C_l3_g0 C_l3_g1 C_l3_g2
R0       R1
R_l0_g0 R_l1_g0    R0C0    R0C1    R0C2
R_l0_g1 R_l1_g1    R1C0    R1C1    R1C2
R_l0_g2 R_l1_g2    R2C0    R2C1    R2C2
R_l0_g3 R_l1_g3    R3C0    R3C1    R3C2
R_l0_g4 R_l1_g4    R4C0    R4C1    R4C2
```

`read_csv` is also able to interpret a more common format of multi-columns indices.

```
In [175]: print(open('mi2.csv').read())
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12

In [176]: pd.read_csv('mi2.csv', header=[0, 1], index_col=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[176]:
     a         b   c
     q  r  s   t   u   v
one  1  2  3   4   5   6
two  7  8  9  10  11  12
```

Note: If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False)`, then any `names` on the columns index will be *lost*.

Automatically "sniffing" the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files, as pandas uses the `csv.Sniffer` class of the csv module. For this, you have to specify `sep=None`.

```
In [177]: print(open('tmp2.sv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498


In [178]: pd.read_csv('tmp2.sv', sep=None, engine='python')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
```

(continues on next page)

(continued from previous page)

```
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

Reading multiple files to create a single DataFrame

It's best to use `concat()` to combine multiple files. See the *cookbook* for an example.

Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

```
In [179]: print(open('tmp.sv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.4249723297888375|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498


In [180]: table = pd.read_csv('tmp.sv', sep='|')

In [181]: table
Out[181]: 
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

By specifying a `chunksize` to `read_csv`, the return value will be an iterable object of type `TextFileReader`:

```
In [182]: reader = pd.read_csv('tmp.sv', sep='|', chunksize=4)

In [183]: reader
```

(continues on next page)

(continued from previous page)

```
Out[183]: <pandas.io.parsers.TextFileReader at 0x7f380a233358>

In [184]: for chunk in reader:
   .....:     print(chunk)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\   Unnamed: 0          ␣
↪0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
   Unnamed: 0         0         1         2         3
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
   Unnamed: 0         0        1         2         3
8           8  1.075770 -0.10905  1.643563 -1.469388
9           9  0.357021 -0.67460 -1.776904 -0.968914
```

Specifying `iterator=True` will also return the `TextFileReader` object:

```
In [185]: reader = pd.read_csv('tmp.sv', sep='|', iterator=True)

In [186]: reader.get_chunk(5)
Out[186]:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
```

Specifying the parser engine

Under the hood pandas uses a fast and efficient parser implemented in C as well as a Python implementation which is currently more feature-complete. Where possible pandas uses the C parser (specified as `engine='c'`), but may fall back to Python if C-unsupported options are specified. Currently, C-unsupported options include:

- `sep` other than a single character (e.g. regex separators)
- `skipfooter`
- `sep=None` with `delim_whitespace=False`

Specifying any of the above options will produce a `ParserWarning` unless the python engine is selected explicitly using `engine='python'`.

Reading remote files

You can pass in a URL to a CSV file:

```
df = pd.read_csv('https://download.bls.gov/pub/time.series/cu/cu.item',
                 sep='\t')
```

S3 URLs are handled as well but require installing the S3Fs library:

```
df = pd.read_csv('s3://pandas-test/tips.csv')
```

If your S3 bucket requires cedentials you will need to set them as environment variables or in the `~/.aws/credentials` config file, refer to the S3Fs documentation on credentials.

Writing out Data

Writing to CSV format

The `Series` and `DataFrame` objects have an instance method `to_csv` which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- `path_or_buf`: A string path to the file to write or a StringIO
- `sep` : Field delimiter for the output file (default ",")
- `na_rep`: A string representation of a missing value (default '')
- `float_format`: Format string for floating point numbers
- `columns`: Columns to write (default None)
- `header`: Whether to write out the column names (default True)
- `index`: whether to write row (index) names (default True)
- `index_label`: Column label(s) for index column(s) if desired. If None (default), and *header* and *index* are True, then the index names are used. (A sequence should be given if the `DataFrame` uses MultiIndex).
- `mode` : Python write mode, default 'w'
- `encoding`: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- `line_terminator`: Character sequence denoting line end (default '\n')
- `quoting`: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a *float_format* then floats are converted to strings and csv.QUOTE_NONNUMERIC will treat them as non-numeric
- `quotechar`: Character used to quote fields (default '"')
- `doublequote`: Control quoting of `quotechar` in fields (default True)
- `escapechar`: Character used to escape `sep` and `quotechar` when appropriate (default None)
- `chunksize`: Number of rows to write at a time
- `tupleize_cols`: If False (default), write as a list of tuples, otherwise write in an expanded line format suitable for `read_csv`
- `date_format`: Format string for datetime objects

Writing a formatted string

The `DataFrame` object has an instance method `to_string` which allows control over the string representation of the object. All arguments are optional:

- `buf` default None, for example a StringIO object
- `columns` default None, which columns to write

- `col_space` default None, minimum width of each column.
- `na_rep` default `NaN`, representation of NA value
- `formatters` default None, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- `float_format` default None, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the `DataFrame`.
- `sparsify` default True, set to False for a `DataFrame` with a hierarchical index to print every MultiIndex key at each row.
- `index_names` default True, will print the names of the indices
- `index` default True, will print the index (ie, row labels)
- `header` default True, will print the column labels
- `justify` default `left`, will print column headers left- or right-justified

The `Series` object also has a `to_string` method, but with only the `buf`, `na_rep`, `float_format` arguments. There is also a `length` argument which, if set to `True`, will additionally output the length of the Series.

4.1.2 JSON

Read and write `JSON` format files and strings.

Writing JSON

A `Series` or `DataFrame` can be converted to a valid JSON string. Use `to_json` with optional parameters:

- `path_or_buf` : the pathname or buffer to write the output This can be `None` in which case a JSON string is returned
- `orient` :

  `Series:`

  - default is `index`
  - allowed values are {`split`, `records`, `index`}

  `DataFrame:`

  - default is `columns`
  - allowed values are {`split`, `records`, `index`, `columns`, `values`, `table`}

  The format of the JSON string

`split`	dict like {index -> [index], columns -> [columns], data -> [values]}
`records`	list like [{column -> value}, ... , {column -> value}]
`index`	dict like {index -> {column -> value}}
`columns`	dict like {column -> {index -> value}}
`values`	just the values array

- `date_format` : string, type of date conversion, 'epoch' for timestamp, 'iso' for ISO8601.
- `double_precision` : The number of decimal places to use when encoding floating point values, default 10.
- `force_ascii` : force encoded string to be ASCII, default True.

- `date_unit` : The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us' or 'ns' for seconds, milliseconds, microseconds and nanoseconds respectively. Default 'ms'.
- `default_handler` : The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.
- `lines` : If `records` orient, then will write each record per line as json.

Note `NaN`'s, `NaT`'s and `None` will be converted to `null` and `datetime` objects will be converted based on the `date_format` and `date_unit` parameters.

```
In [187]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [188]: json = dfj.to_json()

In [189]: json
Out[189]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.
↪0061535699,"4":0.8957173022},"B":{"0":0.4137381054,"1":-0.472034511,"2":-0.
↪3625429925,"3":-0.923060654,"4":0.8052440254}}'
```

Orient Options

There are a number of different options for the format of the resulting JSON file / string. Consider the following `DataFrame` and `Series`:

```
In [190]: dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
   .....:                     columns=list('ABC'), index=list('xyz'))
   .....:

In [191]: dfjo
Out[191]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [192]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [193]: sjo
Out[193]:
x    15
y    16
z    17
Name: D, dtype: int64
```

Column oriented (the default for `DataFrame`) serializes the data as nested JSON objects with column labels acting as the primary index:

```
In [194]: dfjo.to_json(orient="columns")
Out[194]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

# Not available for Series
```

Index oriented (the default for `Series`) similar to column oriented but the index labels are now primary:

```
In [195]: dfjo.to_json(orient="index")
Out[195]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'
```

(continues on next page)

(continued from previous page)

```
In [196]: sjo.to_json(orient="index")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[196]:␣
↪'{"x":15,"y":16,"z":17}'
```

Record oriented serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing `DataFrame` data to plotting libraries, for example the JavaScript library `d3.js`:

```
In [197]: dfjo.to_json(orient="records")
Out[197]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [198]: sjo.to_json(orient="records")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[198]:
↪'[15,16,17]'
```

Value oriented is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [199]: dfjo.to_json(orient="values")
Out[199]: '[[1,4,7],[2,5,8],[3,6,9]]'

# Not available for Series
```

Split oriented serializes to a JSON object containing separate entries for values, index and columns. Name is also included for `Series`:

```
In [200]: dfjo.to_json(orient="split")
Out[200]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,
↪6,9]]}'

In [201]: sjo.to_json(orient="split")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[201]
↪'{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

Table oriented serializes to the JSON Table Schema, allowing for the preservation of metadata including but not limited to dtypes and index names.

Note: Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the *split* option as it uses ordered containers.

Date Handling

Writing in ISO date format:

```
In [202]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [203]: dfd['date'] = pd.Timestamp('20130101')

In [204]: dfd = dfd.sort_index(1, ascending=False)

In [205]: json = dfd.to_json(date_format='iso')
```

(continues on next page)

(continued from previous page)

```
In [206]: json
Out[206]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":
↪"2013-01-01T00:00:00.000Z","3":"2013-01-01T00:00:00.000Z","4":"2013-01-01T00:00:00.
↪000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4
↪":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.
↪4108345112,"4":0.1320031703}}'
```

Writing in ISO date format, with microseconds:

```
In [207]: json = dfd.to_json(date_format='iso', date_unit='us')

In [208]: json
Out[208]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z
↪","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-
↪01T00:00:00.000000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3
↪":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Epoch timestamps, in seconds:

```
In [209]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [210]: json
Out[210]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4
↪":1356998400},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.
↪8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Writing to a file, with a date index and a date column:

```
In [211]: dfj2 = dfj.copy()

In [212]: dfj2['date'] = pd.Timestamp('20130101')

In [213]: dfj2['ints'] = list(range(5))

In [214]: dfj2['bools'] = True

In [215]: dfj2.index = pd.date_range('20130101', periods=5)

In [216]: dfj2.to_json('test.json')

In [217]: with open('test.json') as fh:
   .....:     print(fh.read())
   .....:
{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"1357171200000":-0.
↪0139597524,"1357257600000":-0.0061535699,"1357344000000":0.8957173022},"B":{
↪"1356998400000":0.4137381054,"1357084800000":-0.472034511,"1357171200000":-0.
↪3625429925,"1357257600000":-0.923060654,"1357344000000":0.8052440254},"date":{
↪"1356998400000":1356998400000,"1357084800000":1356998400000,"1357171200000
↪":1356998400000,"1357257600000":1356998400000,"1357344000000":1356998400000},"ints":
↪{"1356998400000":0,"1357084800000":1,"1357171200000":2,"1357257600000":3,
↪"1357344000000":4},"bools":{"1356998400000":true,"1357084800000":true,"1357171200000
↪":true,"1357257600000":true,"1357344000000":true}}
```

Fallback Behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. `np.complex`) then the `default_handler`, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a `toDict` method and call it. A `toDict` method should return a `dict` which will then be JSON serialized.
  - invoke the `default_handler` if one was provided.
  - convert the object to a `dict` by traversing its contents. However this will often fail with an `OverflowError` or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a `default_handler`. For example:

```
>>> DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json()  # raises
RuntimeError: Unhandled numpy dtype 15
```

can be dealt with by specifying a simple `default_handler`:

```
In [218]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[218]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a `DataFrame` if `typ` is not supplied or is `None`. To explicitly force `Series` parsing, pass `typ=series`

- `filepath_or_buffer` : a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, S3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json
- `typ` : type of object to recover (series or frame), default 'frame'
- `orient` :

  Series :

    - default is `index`
    - allowed values are {`split`, `records`, `index`}

  DataFrame

    - default is `columns`
    - allowed values are {`split`, `records`, `index`, `columns`, `values`, `table`}

  The format of the JSON string

`split`	dict like {index -> [index], columns -> [columns], data -> [values]}
`records`	list like [{column -> value}, … , {column -> value}]
`index`	dict like {index -> {column -> value}}
`columns`	dict like {column -> {index -> value}}
`values`	just the values array
`table`	adhering to the JSON Table Schema

- `dtype` : if True, infer dtypes, if a dict of column to dtype, then use those, if `False`, then don't infer dtypes at all, default is True, apply only to the data.
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is `True`
- `convert_dates` : a list of columns to parse for dates; If `True`, then try to parse date-like columns, default is `True`.
- `keep_default_dates` : boolean, default `True`. If parsing dates, then parse the default date-like columns.
- `numpy` : direct decoding to NumPy arrays. default is `False`; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.
- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality.
- `date_unit` : string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- `lines` : reads file as one json object per line.
- `encoding` : The encoding to use to decode py3 bytes.
- `chunksize` : when used in combination with `lines=True`, return a JsonReader which reads in `chunksize` lines per iteration.

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parseable.

If a non-default `orient` was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see *Orient Options* for an overview.

Data Conversion

The default of `convert_axes=True`, `dtype=True`, and `convert_dates=True` will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to `dtype`. `convert_axes` should only be set to `False` if you need to preserve string-like numbers (e.g. '1', '2') in an axes.

Note: Large integer values may be converted to dates if `convert_dates=True` and the data and / or column labels appear 'date-like'. The exact threshold depends on the `date_unit` specified. 'date-like' means that the column label meets one of the following criteria:

- it ends with `'_at'`
- it ends with `'_time'`
- it begins with `'timestamp'`
- it is `'modified'`
- it is `'date'`

Warning: When reading JSON data, automatic coercing into dtypes has some quirks:

- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was `float` data will be converted to `integer` if it can be done safely, e.g. a column of `1.`
- bool columns will be converted to `integer` on reconstruction

Thus there are times where you may want to specify specific dtypes via the `dtype` keyword argument.

Reading from a JSON string:

```
In [219]: pd.read_json(json)
Out[219]:
         date         B         A
0 2013-01-01  2.565646 -1.206412
1 2013-01-01  1.340309  1.431256
2 2013-01-01 -0.226169 -1.170299
3 2013-01-01  0.813850  0.410835
4 2013-01-01 -0.827317  0.132003
```

Reading from a file:

```
In [220]: pd.read_json('test.json')
Out[220]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

Don’t convert any data (but still convert axes and dates):

```
In [221]: pd.read_json('test.json', dtype=object).dtypes
Out[221]:
A        object
B        object
date     object
ints     object
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [222]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[222]:
A               float32
B               float64
date     datetime64[ns]
ints              int64
bools              int8
dtype: object
```

Preserve string indices:

```
In [223]: si = pd.DataFrame(np.zeros((4, 4)), columns=list(range(4)),
   .....:                   index=[str(i) for i in range(4)])
   .....:

In [224]: si
Out[224]:
     0    1    2    3
0  0.0  0.0  0.0  0.0
```

(continues on next page)

(continued from previous page)

```
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0

In [225]: si.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['0', '1', '2', '3'], dtype='object')

In [226]: si.columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Int64Index([0, 1, 2, 3], dtype='int64')

In [227]: json = si.to_json()

In [228]: sij = pd.read_json(json, convert_axes=False)

In [229]: sij
Out[229]:
   0  1  2  3
0  0  0  0  0
1  0  0  0  0
2  0  0  0  0
3  0  0  0  0

In [230]: sij.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[230]:␣
↪Index(['0', '1', '2', '3'], dtype='object')

In [231]: sij.columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['0', '1', '2', '3'], dtype='object')
```

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [232]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as millseconds -> Won't Work
In [233]: dfju = pd.read_json(json, date_unit='ms')

In [234]: dfju
Out[234]:
                            A         B                 date  ints  bools
1356998400000000000 -1.294524  0.413738  1356998400000000000     0   True
1357084800000000000  0.276662 -0.472035  1356998400000000000     1   True
1357171200000000000 -0.013960 -0.362543  1356998400000000000     2   True
1357257600000000000 -0.006154 -0.923061  1356998400000000000     3   True
1357344000000000000  0.895717  0.805244  1356998400000000000     4   True

# Let pandas detect the correct precision
In [235]: dfju = pd.read_json(json)

In [236]: dfju
Out[236]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
```

(continues on next page)

(continued from previous page)

```
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True

# Or specify that all timestamps are in nanoseconds
In [237]: dfju = pd.read_json(json, date_unit='ns')

In [238]: dfju
Out[238]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

The Numpy Parameter

Note: This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

If `numpy=True` is passed to `read_json` an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```
In [239]: randfloats = np.random.uniform(-100, 1000, 10000)

In [240]: randfloats.shape = (1000, 10)

In [241]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [242]: jsonfloats = dffloats.to_json()
```

```
In [243]: %timeit pd.read_json(jsonfloats)
11.6 ms +- 236 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [244]: %timeit pd.read_json(jsonfloats, numpy=True)
8.7 ms +- 227 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [245]: jsonfloats = dffloats.head(100).to_json()
```

```
In [246]: %timeit pd.read_json(jsonfloats)
8.28 ms +- 188 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [247]: %timeit pd.read_json(jsonfloats, numpy=True)
7.22 ms +- 279 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Warning: Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:

- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A `ValueError` may be raised, or incorrect output may be produced if this condition is not satisfied.
- labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using `to_json` but may not be the case if the JSON is from another source.

Normalization

pandas provides a utility function to take a dict or list of dicts and *normalize* this semi-structured data into a flat table.

```
In [248]: from pandas.io.json import json_normalize

In [249]: data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
   .....:         {'name': {'given': 'Mose', 'family': 'Regner'}},
   .....:         {'id': 2, 'name': 'Faye Raker'}]
   .....:

In [250]: json_normalize(data)
Out[250]:
    id        name name.family name.first name.given name.last
0  1.0         NaN         NaN     Coleen        NaN      Volk
1  NaN         NaN      Regner        NaN       Mose       NaN
2  2.0  Faye Raker         NaN        NaN        NaN       NaN
```

```
In [251]: data = [{'state': 'Florida',
   .....:          'shortname': 'FL',
   .....:          'info': {'governor': 'Rick Scott'},
   .....:          'counties': [{'name': 'Dade', 'population': 12345},
   .....:                       {'name': 'Broward', 'population': 40000},
   .....:                       {'name': 'Palm Beach', 'population': 60000}]},
   .....:         {'state': 'Ohio',
   .....:          'shortname': 'OH',
   .....:          'info': {'governor': 'John Kasich'},
   .....:          'counties': [{'name': 'Summit', 'population': 1234},
   .....:                       {'name': 'Cuyahoga', 'population': 1337}]}]
   .....:

In [252]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor
↪']])
Out[252]:
         name  population    state shortname info.governor
0        Dade       12345  Florida        FL    Rick Scott
1     Broward       40000  Florida        FL    Rick Scott
2  Palm Beach       60000  Florida        FL    Rick Scott
3      Summit        1234     Ohio        OH   John Kasich
4    Cuyahoga        1337     Ohio        OH   John Kasich
```

Line delimited json

New in version 0.19.0.

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

New in version 0.21.0.

For line-delimited json files, pandas can also return an iterator which reads in `chunksize` lines at a time. This can be useful for large files or to read from a stream.

```
In [253]: jsonl = '''
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: '''
   .....:

In [254]: df = pd.read_json(jsonl, lines=True)

In [255]: df
Out[255]:
   a  b
0  1  2
1  3  4

In [256]: df.to_json(orient='records', lines=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[256]: '{"a":1,"b":2}\n{"a":3,"b":4}'

# reader is an iterator that returns `chunksize` lines each iteration
In [257]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [258]: reader
Out[258]: <pandas.io.json.json.JsonReader at 0x7f380e7fe160>

In [259]: for chunk in reader:
   .....:     print(chunk)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Empty DataFrame
Columns: []
Index: []
   a  b
0  1  2
   a  b
1  3  4
```

Table Schema

New in version 0.20.0.

Table Schema is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the orient `table` to build a JSON string with two fields, `schema` and `data`.

```
In [260]: df = pd.DataFrame({'A': [1, 2, 3],
   .....:                    'B': ['a', 'b', 'c'],
   .....:                    'C': pd.date_range('2016-01-01', freq='d', periods=3)},
   .....:                   index=pd.Index(range(3), name='idx'))
   .....:

In [261]: df
```

(continues on next page)

(continued from previous page)

```
Out[261]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [262]: df.to_json(orient='table', date_format="iso")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'{"schema": {"fields":[{"name":"idx","type":"integer"},{"name":"A","type":"integer"}
↪,{"name":"B","type":"string"},{"name":"C","type":"datetime"}],"primaryKey":["idx"],
↪"pandas_version":"0.20.0"}, "data": [{"idx":0,"A":1,"B":"a","C":"2016-01-
↪01T00:00:00.000Z"},{"idx":1,"A":2,"B":"b","C":"2016-01-02T00:00:00.000Z"},{"idx":2,
↪"A":3,"B":"c","C":"2016-01-03T00:00:00.000Z"}]}'
```

The schema field contains the fields key, which itself contains a list of column name to type pairs, including the Index or MultiIndex (see below for a list of types). The schema field also contains a primaryKey field if the (Multi)index is unique.

The second field, data, contains the serialized data with the records orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:

Pandas type	Table Schema type
int64	integer
float64	number
bool	boolean
datetime64[ns]	datetime
timedelta64[ns]	duration
categorical	any
object	str

A few notes on the generated table schema:

- The schema object contains a pandas_version field. This contains the version of pandas' dialect of the schema, and will be incremented with each revision.
- All dates are converted to UTC when serializing. Even timezone naive values, which are treated as UTC with an offset of 0.

```
In [263]: from pandas.io.json import build_table_schema

In [264]: s = pd.Series(pd.date_range('2016', periods=4))

In [265]: build_table_schema(s)
Out[265]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- datetimes with a timezone (before serializing), include an additional field tz with the time zone name (e.g. 'US/Central').

```
In [266]: s_tz = pd.Series(pd.date_range('2016', periods=12,
   .....:                                tz='US/Central'))
   .....:

In [267]: build_table_schema(s_tz)
Out[267]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain and additional field `freq` with the period's frequency, e.g. `'A-DEC'`.

```
In [268]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC',
   .....:                                            periods=4))
   .....:

In [269]: build_table_schema(s_per)
Out[269]:
{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Categoricals use the `any` type and an `enum` constraint listing the set of possible values. Additionally, an `ordered` field is included:

```
In [270]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [271]: build_table_schema(s_cat)
Out[271]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values',
   'type': 'any',
   'constraints': {'enum': ['a', 'b']},
   'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- A `primaryKey` field, containing an array of labels, is included *if the index is unique*:

```
In [272]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [273]: build_table_schema(s_dupe)
Out[273]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0'}
```

- The `primaryKey` behavior is the same with MultiIndexes, but in this case the `primaryKey` is an array:

```
In [274]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
   .....:                                                           (0, 1)]))
   .....:

In [275]: build_table_schema(s_multi)
```

(continues on next page)

(continued from previous page)

```
Out[275]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '0.20.0'}
```

- The default naming roughly follows these rules:
  - For series, the `object.name` is used. If that's none, then the name is `values`
  - For `DataFrames`, the stringified version of the column name is used
  - For `Index` (not `MultiIndex`), `index.name` is used, with a fallback to `index` if that is None.
  - For `MultiIndex`, `mi.names` is used. If any level has no name, then `level_<i>` is used.

New in version 0.23.0.

`read_json` also accepts `orient='table'` as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner.

```
In [276]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
   .....:                    'bar': ['a', 'b', 'c', 'd'],
   .....:                    'baz': pd.date_range('2018-01-01', freq='d',␣
↪periods=4),
   .....:                    'qux': pd.Categorical(['a', 'b', 'c', 'c'])
   .....:                    }, index=pd.Index(range(4), name='idx'))
   .....:

In [277]: df
Out[277]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

In [278]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object

In [279]: df.to_json('test.json', orient='table')

In [280]: new_df = pd.read_json('test.json', orient='table')

In [281]: new_df
Out[281]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
```

(continues on next page)

(continued from previous page)

```
3      4   d 2018-01-04   c

In [282]: new_df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo              int64
bar             object
baz     datetime64[ns]
qux           category
dtype: object
```

Please note that the literal string 'index' as the name of an *Index* is not round-trippable, nor are any names beginning with 'level_' within a *MultiIndex*. These are used by default in *DataFrame.to_json()* to indicate missing values and the subsequent read cannot distinguish the intent.

```
In [283]: df.index.name = 'index'

In [284]: df.to_json('test.json', orient='table')

In [285]: new_df = pd.read_json('test.json', orient='table')

In [286]: print(new_df.index.name)
None
```

4.1.3 HTML

Reading HTML Content

Warning: We **highly encourage** you to read the *HTML Table Parsing gotchas* below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level `read_html()` function can accept an HTML string/file/URL and will parse HTML tables into list of pandas `DataFrames`. Let's look at a few examples.

Note: `read_html` returns a `list` of `DataFrame` objects, even if there is only a single table contained in the HTML content.

Read a URL with no options:

```
In [287]: url = 'https://www.fdic.gov/bank/individual/failed/banklist.html'

In [288]: dfs = pd.read_html(url)

In [289]: dfs
Out[289]:
[                                             Bank Name                City  ST  ␣
↪CERT                Acquiring Institution       Closing Date         Updated Date
 0                  Washington Federal Bank for Savings              Chicago   IL ␣
↪30570                   Royal Savings Bank  December 15, 2017   February 21, 2018
 1        The Farmers and Merchants State Bank of Argonia              Argonia   KS ␣
↪17719                          Conway Bank   October 13, 2017   February 21, 2018
```

(continues on next page)

(continued from previous page)

```
2                                  Fayette County Bank         Saint Elmo  IL ␣
↪1802             United Fidelity Bank, fsb       May 26, 2017       July 26, 2017
3    Guaranty Bank, (d/b/a BestBank in Georgia & Mi...          Milwaukee  WI ␣
↪30003  First-Citizens Bank & Trust Company        May 5, 2017      March 22, 2018
4                                       First NBC Bank        New Orleans  LA ␣
↪58302                        Whitney Bank     April 28, 2017    December 5, 2017
5                                        Proficio Bank  Cottonwood Heights  UT ␣
↪35495                   Cache Valley Bank      March 3, 2017       March 7, 2018
6                           Seaway Bank and Trust Company           Chicago  IL ␣
↪19328                  State Bank of Texas   January 27, 2017        May 18, 2017
..                                                ...                 ...  ..    ...
↪                                 ...               ...                 ...
548                         Hamilton Bank, NA  En Espanol               Miami  FL ␣
↪24382      Israel Discount Bank of New York   January 11, 2002  September 21, 2015
549                                 Sinclair National Bank           Gravette  AR ␣
↪34248                      Delta Trust & Bank  September 7, 2001     October 6, 2017
550                                     Superior Bank, FSB           Hinsdale  IL ␣
↪32646                    Superior Federal, FSB      July 27, 2001     August 19, 2014
551                                   Malta National Bank              Malta  OH  ␣
↪6629                       North Valley Bank        May 3, 2001   November 18, 2002
552                       First Alliance Bank & Trust Co.          Manchester  NH ␣
↪34264  Southern New Hampshire Bank & Trust   February 2, 2001   February 18, 2003
553                     National State Bank of Metropolis          Metropolis  IL  ␣
↪3815                 Banterra Bank of Marion  December 14, 2000      March 17, 2005
554                                     Bank of Honolulu             Honolulu  HI ␣
↪21029                      Bank of the Orient   October 13, 2000      March 17, 2005

[555 rows x 7 columns]]
```

Note: The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string:

```
In [290]: with open(file_path, 'r') as f:
   .....:     dfs = pd.read_html(f.read())
   .....:

In [291]: dfs
Out[291]:
[                                    Bank Name          City  ST   CERT                ␣
↪ Acquiring Institution       Closing Date       Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  WI  35386                ␣
↪ North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                        Central Arizona Bank    Scottsdale  AZ  34527                ␣
↪    Western State Bank       May 14, 2013       May 20, 2013
 2                                Sunrise Bank      Valdosta  GA  58185                ␣
↪          Synovus Bank       May 10, 2013       May 21, 2013
 3                       Pisgah Community Bank     Asheville  NC  58701                ␣
↪    Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                         Douglas County Bank  Douglasville  GA  21649                ␣
↪   Hamilton State Bank     April 26, 2013       May 16, 2013
 5                                Parkway Bank        Lenoir  NC  57158          ␣
↪CertusBank, National Association     April 26, 2013       May 17, 2013
 6                       Chipola Community Bank      Marianna  FL  58034        First␣
↪Federal Bank of Florida     April 19, 2013       May 16, 2013
```

(continues on next page)

(continued from previous page)

```
..                                            ...           ...  ..    ...         ␣
↪                    ...                  ...                 ...
498                  Hamilton Bank, NAEn Espanol         Miami   FL  24382     Israel␣
↪Discount Bank of New York   January 11, 2002       June 5, 2012
499                       Sinclair National Bank      Gravette  AR  34248          ␣
↪    Delta Trust & Bank  September 7, 2001  February 10, 2004
500                           Superior Bank, FSB      Hinsdale  IL  32646          ␣
↪ Superior Federal, FSB       July 27, 2001       June 5, 2012
501                          Malta National Bank         Malta  OH   6629          ␣
↪     North Valley Bank         May 3, 2001  November 18, 2002
502              First Alliance Bank & Trust Co.    Manchester  NH  34264  Southern New␣
↪Hampshire Bank & Trust   February 2, 2001  February 18, 2003
503            National State Bank of Metropolis    Metropolis  IL   3815         ␣
↪Banterra Bank of Marion  December 14, 2000     March 17, 2005
504                             Bank of Honolulu      Honolulu  HI  21029          ␣
↪    Bank of the Orient   October 13, 2000     March 17, 2005

[505 rows x 7 columns]]
```

You can even pass in an instance of `StringIO` if you so desire:

```
In [292]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:

In [293]: dfs = pd.read_html(sio)

In [294]: dfs
Out[294]:
[                                    Bank Name          City  ST   CERT          ␣
↪ Acquiring Institution       Closing Date       Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha        Kenosha  WI  35386          ␣
↪ North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                        Central Arizona Bank    Scottsdale  AZ  34527          ␣
↪    Western State Bank       May 14, 2013       May 20, 2013
 2                                Sunrise Bank      Valdosta  GA  58185          ␣
↪          Synovus Bank       May 10, 2013       May 21, 2013
 3                       Pisgah Community Bank     Asheville  NC  58701          ␣
↪    Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                         Douglas County Bank  Douglasville  GA  21649          ␣
↪   Hamilton State Bank     April 26, 2013       May 16, 2013
 5                                Parkway Bank        Lenoir  NC  57158   ␣
↪CertusBank, National Association     April 26, 2013       May 17, 2013
 6                       Chipola Community Bank      Marianna  FL  58034        First␣
↪Federal Bank of Florida     April 19, 2013       May 16, 2013
 ..                                        ...           ...  ..    ...          ␣
↪                   ...                  ...                 ...
 498                 Hamilton Bank, NAEn Espanol         Miami   FL  24382     Israel␣
↪Discount Bank of New York   January 11, 2002       June 5, 2012
 499                      Sinclair National Bank      Gravette  AR  34248          ␣
↪    Delta Trust & Bank  September 7, 2001  February 10, 2004
 500                          Superior Bank, FSB      Hinsdale  IL  32646          ␣
↪ Superior Federal, FSB       July 27, 2001       June 5, 2012
 501                         Malta National Bank         Malta  OH   6629          ␣
↪     North Valley Bank         May 3, 2001  November 18, 2002
 502             First Alliance Bank & Trust Co.    Manchester  NH  34264  Southern New␣
↪Hampshire Bank & Trust   February 2, 2001  February 18, 2003
```

(continues on next page)

(continued from previous page)

```
503         National State Bank of Metropolis    Metropolis   IL   3815          ␣
↪Banterra Bank of Marion  December 14, 2000    March 17, 2005
504                           Bank of Honolulu      Honolulu   HI  21029          ␣
↪    Bank of the Orient   October 13, 2000     March 17, 2005

[505 rows x 7 columns]]
```

Note: The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn't run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text:

```
match = 'Metcalf Bank'
df_list = pd.read_html(url, match=match)
```

Specify a header row (by default `<th>` or `<td>` elements located within a `<thead>` are used to form the column index, if multiple rows are contained within `<thead>` then a MultiIndex is created); if specified, the header row is taken from the data minus the parsed header elements (`<th>` elements).

```
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```
dfs = pd.read_html(url, index_col=0)
```

Specify a number of rows to skip:

```
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (`xrange` (Python 2 only) works as well):

```
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```
dfs1 = pd.read_html(url, attrs={'id': 'table'})
dfs2 = pd.read_html(url, attrs={'class': 'sortable'})
print(np.array_equal(dfs1[0], dfs2[0]))  # Should be True
```

Specify values that should be converted to NaN:

```
dfs = pd.read_html(url, na_values=['No Acquirer'])
```

New in version 0.19.

Specify whether to keep the default set of NaN values:

```
dfs = pd.read_html(url, keep_default_na=False)
```

New in version 0.19.

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```
url_mcc = 'https://en.wikipedia.org/wiki/Mobile_country_code'
dfs = pd.read_html(url_mcc, match='Telekom Albania', header=0,
                   converters={'MNC': str})
```

New in version 0.19.

Use some combination of the above:

```
dfs = pd.read_html(url, match='Metcalf Bank', index_col=0)
```

Read in pandas `to_html` output (with some loss of floating point precision):

```
df = pd.DataFrame(np.random.randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = pd.read_html(s, index_col=0)
```

The `lxml` backend will raise an error on a failed parse if that is the only parser you provide. If you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings. You may use:

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])
```

Or you could pass `flavor='lxml'` without a list:

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor='lxml')
```

However, if you have bs4 and html5lib installed and pass `None` or `['lxml', 'bs4']` then the parse will most likely succeed. Note that *as soon as a parse succeeds, the function will return*.

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])
```

Writing to HTML files

`DataFrame` objects have an instance method `to_html` which renders the contents of the `DataFrame` as an HTML table. The function arguments are as in the method `to_string` described above.

Note: Not all of the possible options for `DataFrame.to_html` are shown here for brevity's sake. See `to_html()` for the full set of options.

```
In [295]: df = pd.DataFrame(np.random.randn(2, 2))

In [296]: df
Out[296]: 
          0         1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [297]: print(df.to_html())  # raw html
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\<table␣
↪border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
```

(continues on next page)

(continued from previous page)

```
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

HTML:

The `columns` argument will limit the columns shown:

```
In [298]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
    </tr>
  </tbody>
</table>
```

HTML:

`float_format` takes a Python callable to control the precision of floating point values:

```
In [299]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
```

(continues on next page)

(continued from previous page)

```
      <td>-0.1847438576</td>
      <td>0.4969711327</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off:

```
In [300]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <td>1</td>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing `'dataframe'` class.

```
In [301]: print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class
↪']))
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
```

(continues on next page)

(continued from previous page)

```
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

The `render_links` argument provides the ability to add hyperlinks to cells that contain URLs.

New in version 0.24.

```
In [302]: url_df = pd.DataFrame({
   .....:     'name': ['Python', 'Pandas'],
   .....:     'url': ['https://www.python.org/', 'http://pandas.pydata.org']})
   .....:

In [303]: print(url_df.to_html(render_links=True))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>url</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Python</td>
      <td><a href="https://www.python.org/" target="_blank">https://www.python.org/</
↪a></td>
    </tr>
    <tr>
      <th>1</th>
      <td>Pandas</td>
      <td><a href="http://pandas.pydata.org" target="_blank">http://pandas.pydata.org
↪</a></td>
    </tr>
  </tbody>
</table>
```

HTML:

Finally, the `escape` argument allows you to control whether the "<", ">" and "&" characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

```
In [304]: df = pd.DataFrame({'a': list('&<>'), 'b': np.random.randn(3)})
```

Escaped:

```
In [305]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
```

(continues on next page)

(continued from previous page)

```
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&amp;</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&lt;</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>&gt;</td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```
In [306]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td><</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Note: Some browsers may not show a difference in the rendering of the previous two HTML tables.

HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function `read_html`.

Issues with lxml

- Benefits
  - **lxml** is very fast.
  - **lxml** requires Cython to install correctly.
- Drawbacks
  - **lxml** does *not* make any guarantees about the results of its parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the **lxml** backend, but **this backend will use html5lib** if **lxml** fails to parse
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if **lxml** fails.

Issues with BeautifulSoup4 using lxml as a backend

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

Issues with BeautifulSoup4 using html5lib as a backend

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is "correct", since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.
- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

4.1.4 Excel files

The `read_excel()` method can read Excel 2003 (`.xls`) and Excel 2007+ (`.xlsx`) files using the `xlrd` Python module. The `to_excel()` instance method is used for saving a `DataFrame` to Excel. Generally the semantics are similar to working with *csv* data. See the *cookbook* for some advanced strategies.

Reading Excel Files

In the most basic use-case, `read_excel` takes a path to an Excel file, and the `sheet_name` indicating which sheet to parse.

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', sheet_name='Sheet1')
```

`ExcelFile` class

To facilitate working with multiple sheets from the same file, the `ExcelFile` class can be used to wrap the file and can be passed into `read_excel` There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```
xlsx = pd.ExcelFile('path_to_file.xls')
df = pd.read_excel(xlsx, 'Sheet1')
```

The `ExcelFile` class can also be used as a context manager.

```
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

The `sheet_names` property will generate a list of the sheet names in the file.

The primary use-case for an `ExcelFile` is parsing multiple sheets with different parameters:

```
data = {}
# For when Sheet1's format differs from Sheet2
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to `read_excel` with no loss in performance.

```
# using the ExcelFile class
data = {}
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=None,
                                   na_values=['NA'])

# equivalent using the read_excel function
data = pd.read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'],
                     index_col=None, na_values=['NA'])
```

Specifying Sheets

Note: The second argument is `sheet_name`, not to be confused with `ExcelFile.sheet_names`.

Note: An ExcelFile's attribute `sheet_names` provides access to a list of sheets.

- The arguments `sheet_name` allows specifying the sheet or sheets to read.
- The default value for `sheet_name` is 0, indicating to read the first sheet
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.

- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Pass a `None` to return a dictionary of all available sheets.

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

Using all default values:

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls')
```

Using None to get all sheets:

```
# Returns a dictionary of DataFrames
pd.read_excel('path_to_file.xls', sheet_name=None)
```

Using a list to get multiple sheets:

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel('path_to_file.xls', sheet_name=['Sheet1', 3])
```

`read_excel` can read more than one sheet, by setting `sheet_name` to either a list of sheet names, a list of sheet positions, or `None` to read all sheets. Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

Reading a `MultiIndex`

`read_excel` can read a `MultiIndex` index, by passing a list of columns to `index_col` and a `MultiIndex` column by passing a list of rows to `header`. If either the `index` or `columns` have serialized level names those will be read in as well by specifying the rows/columns that make up the levels.

For example, to read in a `MultiIndex` index without names:

```
In [307]: df = pd.DataFrame({'a': [1, 2, 3, 4], 'b': [5, 6, 7, 8]},
   .....:                    index=pd.MultiIndex.from_product([['a', 'b'], ['c', 'd
↪']]))
   .....:

In [308]: df.to_excel('path_to_file.xlsx')

In [309]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [310]: df
Out[310]:
     a  b
a c  1  5
  d  2  6
b c  3  7
  d  4  8
```

If the index has level names, they will parsed as well, using the same parameters.

```
In [311]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [312]: df.to_excel('path_to_file.xlsx')

In [313]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [314]: df
Out[314]:
           a  b
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

If the source file has both `MultiIndex` index and columns, lists specifying each should be passed to `index_col` and `header`:

```
In [315]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']],
   .....:                                          names=['c1', 'c2'])
   .....:

In [316]: df.to_excel('path_to_file.xlsx')

In [317]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1], header=[0, 1])

In [318]: df
Out[318]:
c1         a
c2         b  d
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

Parsing Specific Columns

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. `read_excel` takes a `usecols` keyword to allow you to specify a subset of columns to parse.

Deprecated since version 0.24.0.

Passing in an integer for `usecols` has been deprecated. Please pass in a list of ints from 0 to `usecols` inclusive instead.

If `usecols` is an integer, then it is assumed to indicate the last column to be parsed.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=2)
```

You can also specify a comma-delimited set of Excel columns and ranges as a string:

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols='A,C:E')
```

If `usecols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=[0, 2, 3])
```

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`.

New in version 0.24.

If `usecols` is a list of strings, it is assumed that each string corresponds to a column name provided either by the user in `names` or inferred from the document header row(s). Those strings define which columns will be parsed:

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=['foo', 'bar'])
```

Element order is ignored, so `usecols=['baz', 'joe']` is the same as `['joe', 'baz']`.

New in version 0.24.

If `usecols` is callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=lambda x: x.isalpha())
```

Parsing Dates

Datetime-like values are normally automatically converted to the appropriate dtype when reading the excel file. But if you have a column of strings that *look* like dates (but are not actually formatted as dates in excel), you can use the `parse_dates` keyword to parse those strings to datetimes:

```
pd.read_excel('path_to_file.xls', 'Sheet1', parse_dates=['date_strings'])
```

Cell Converters

It is possible to transform the contents of Excel cells via the `converters` option. For instance, to convert a column to boolean:

```
pd.read_excel('path_to_file.xls', 'Sheet1', converters={'MyBools': bool})
```

This options handles missing values and treats exceptions in the converters as missing data. Transformations are applied cell by cell rather than to the column as a whole, so the array dtype is not guaranteed. For instance, a column of integers with missing values cannot be transformed to an array with integer dtype, because NaN is strictly a float. You can manually mask missing data to recover integer dtype:

```
def cfun(x):
    return int(x) if x else -1


pd.read_excel('path_to_file.xls', 'Sheet1', converters={'MyInts': cfun})
```

dtype Specifications

New in version 0.20.

As an alternative to converters, the type for an entire column can be specified using the *dtype* keyword, which takes a dictionary mapping column names to types. To interpret data with no type inference, use the type `str` or `object`.

```
pd.read_excel('path_to_file.xls', dtype={'MyInts': 'int64', 'MyText': str})
```

Writing Excel Files

Writing Excel Files to Disk

To write a DataFrame object to a sheet of an Excel file, you can use the to_excel instance method. The arguments are largely the same as to_csv described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the DataFrame should be written. For example:

```
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Files with a .xls extension will be written using xlwt and those with a .xlsx extension will be written using xlsxwriter (if available) or openpyxl.

The DataFrame will be written in a way that tries to mimic the REPL output. The index_label will be placed in the second row instead of the first. You can place it in the first row by setting the merge_cells option in to_excel() to False:

```
df.to_excel('path_to_file.xlsx', index_label='label', merge_cells=False)
```

In order to write separate DataFrames to separate sheets in a single Excel file, one can pass an ExcelWriter.

```
with pd.ExcelWriter('path_to_file.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

Note: Wringing a little more performance out of read_excel Internally, Excel stores all numeric data as floats. Because this can produce unexpected behavior when reading in data, pandas defaults to trying to convert integers to floats if it doesn't lose information (1.0 --> 1). You can pass convert_float=False to disable this behavior, which may give a slight performance improvement.

Writing Excel Files to Memory

Pandas supports writing Excel files to buffer-like objects such as StringIO or BytesIO using ExcelWriter.

```
# Safe import for either Python 2.x or 3.x
try:
    from io import BytesIO
except ImportError:
    from cStringIO import StringIO as BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter(bio, engine='xlsxwriter')
df.to_excel(writer, sheet_name='Sheet1')

# Save the workbook
writer.save()
```

(continues on next page)

(continued from previous page)

```
# Seek to the beginning and read to copy the workbook to a variable in memory
bio.seek(0)
workbook = bio.read()
```

Note: engine is optional but recommended. Setting the engine determines the version of workbook produced. Setting engine='xlrd' will produce an Excel 2003-format workbook (xls). Using either 'openpyxl' or 'xlsxwriter' will produce an Excel 2007-format workbook (xlsx). If omitted, an Excel 2007-formatted workbook is produced.

Excel writer engines

Pandas chooses an Excel writer via two methods:

1. the engine keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the XlsxWriter for .xlsx, openpyxl for .xlsm, and xlwt for .xls files. If you have multiple engines installed, you can set the default engine through *setting the config options* io.excel.xlsx.writer and io.excel.xls.writer. pandas will fall back on openpyxl for .xlsx files if Xlsxwriter is not available.

To specify which writer you want to use, you can pass an engine keyword argument to to_excel and to ExcelWriter. The built-in engines are:

- openpyxl: version 2.4 or higher is required
- xlsxwriter
- xlwt

```
# By setting the 'engine' in the DataFrame and Panel 'to_excel()' methods.
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='xlsxwriter')

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter('path_to_file.xlsx', engine='xlsxwriter')

# Or via pandas configuration.
from pandas import options                                     # noqa: E402
options.io.excel.xlsx.writer = 'xlsxwriter'

df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Style and Formatting

The look and feel of Excel worksheets created from pandas can be modified using the following parameters on the DataFrame's to_excel method.

- float_format : Format string for floating point numbers (default None).
- freeze_panes : A tuple of two integers representing the bottommost row and rightmost column to freeze. Each of these parameters is one-based, so (1, 1) will freeze the first row and first column (default None).

4.1.5 Clipboard

A handy way to grab data is to use the `read_clipboard()` method, which takes the contents of the clipboard buffer and passes them to the `read_csv` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a `DataFrame` by calling:

```
clipdf = pd.read_clipboard()
```

```
In [319]: clipdf
Out[319]:
   A  B  C
x  1  4  p
y  2  5  q
z  3  6  r
```

The `to_clipboard` method can be used to write the contents of a `DataFrame` to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a `DataFrame` into clipboard and reading it back.

```
In [320]: df = pd.DataFrame(np.random.randn(5, 3))

In [321]: df
Out[321]:
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

In [322]: df.to_clipboard()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
PyperclipException                        Traceback (most recent call last)
<ipython-input-322-d9e762ebf7d3> in <module>
----> 1 df.to_clipboard()

/pandas/pandas/core/generic.py in to_clipboard(self, excel, sep, **kwargs)
   2639         """
   2640         from pandas.io import clipboards
-> 2641         clipboards.to_clipboard(self, excel=excel, sep=sep, **kwargs)
   2642
   2643     def to_xarray(self):

/pandas/pandas/io/clipboards.py in to_clipboard(obj, excel, sep, **kwargs)
    129             if PY2:
    130                 text = text.decode('utf-8')
--> 131             clipboard_set(text)
    132             return
    133         except TypeError:
```

(continues on next page)

(continued from previous page)

```
/pandas/pandas/io/clipboard/clipboards.py in __call__(self, *args, **kwargs)
    134
    135         def __call__(self, *args, **kwargs):
--> 136             raise PyperclipException(EXCEPT_MSG)
    137
    138         if PY2:

PyperclipException:
    Pyperclip could not find a copy/paste mechanism for your system.
    For more information, please visit https://pyperclip.readthedocs.org

In [323]: pd.read_clipboard()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
PyperclipException                        Traceback (most recent call last)
<ipython-input-323-8cbad928c47b> in <module>
----> 1 pd.read_clipboard()

/pandas/pandas/io/clipboards.py in read_clipboard(sep, **kwargs)
     35     from pandas.io.clipboard import clipboard_get
     36     from pandas.io.parsers import read_csv
---> 37     text = clipboard_get()
     38
     39     # try to decode (if needed on PY3)

/pandas/pandas/io/clipboard/clipboards.py in __call__(self, *args, **kwargs)
    134
    135         def __call__(self, *args, **kwargs):
--> 136             raise PyperclipException(EXCEPT_MSG)
    137
    138         if PY2:

PyperclipException:
    Pyperclip could not find a copy/paste mechanism for your system.
    For more information, please visit https://pyperclip.readthedocs.org
```

We can see that we got the same content back, which we had earlier written to the clipboard.

Note: You may need to install xclip or xsel (with gtk, PyQt5, PyQt4 or qtpy) on Linux to use these methods.

4.1.6 Pickling

All pandas objects are equipped with `to_pickle` methods which use Python's `cPickle` module to save data structures to disk using the pickle format.

```
In [324]: df
Out[324]:
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
```

(continues on next page)

(continued from previous page)

```
4 -1.175743 -0.172372 -0.734129

In [325]: df.to_pickle('foo.pkl')
```

The `read_pickle` function in the `pandas` namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [326]: pd.read_pickle('foo.pkl')
Out[326]:
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

Warning: Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3/library/pickle.html

Warning: Several internal refactoring have been done while still preserving compatibility with pickles created with older versions of pandas. However, for such cases, pickled `DataFrames`, `Series` etc, must be read with `pd.read_pickle`, rather than `pickle.load`.

See here and here for some examples of compatibility-breaking changes. See this question for a detailed explanation.

Compressed pickle files

New in version 0.20.0.

read_pickle(), *DataFrame.to_pickle()* and *Series.to_pickle()* can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz` are supported for reading and writing. The `zip` file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If 'infer', then use `gzip`, `bz2`, `zip`, or `xz` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, or `'.xz'`, respectively.

```
In [327]: df = pd.DataFrame({
   .....:     'A': np.random.randn(1000),
   .....:     'B': 'foo',
   .....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
   .....:

In [328]: df
Out[328]:
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
```

(continues on next page)

(continued from previous page)

```
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type:

```
In [329]: df.to_pickle("data.pkl.compress", compression="gzip")

In [330]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [331]: rt
Out[331]:
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension:

```
In [332]: df.to_pickle("data.pkl.xz", compression="infer")

In [333]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [334]: rt
Out[334]:
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
```

(continues on next page)

(continued from previous page)

```
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

The default is to 'infer':

```
In [335]: df.to_pickle("data.pkl.gz")

In [336]: rt = pd.read_pickle("data.pkl.gz")

In [337]: rt
Out[337]:
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [338]: df["A"].to_pickle("s1.pkl.bz2")

In [339]: rt = pd.read_pickle("s1.pkl.bz2")

In [340]: rt
Out[340]:
0      0.478412
1     -0.783748
2      1.403558
3     -0.539282
4     -1.651012
5      0.692072
6      1.022171
         ...
993   -1.613932
994    1.088104
995   -0.632963
996   -0.585314
997   -0.275038
998   -0.937512
```

(continues on next page)

(continued from previous page)

```
999    0.632369
Name: A, Length: 1000, dtype: float64
```

4.1.7 msgpack

pandas supports the `msgpack` format for object serialization. This is a lightweight portable binary format, similar to binary JSON, that is highly space efficient, and provides good performance both on the writing (serialization), and reading (deserialization).

Warning: This is a very new feature of pandas. We intend to provide certain optimizations in the io of the `msgpack` data. Since this is marked as an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

```
In [341]: df = pd.DataFrame(np.random.rand(5, 2), columns=list('AB'))

In [342]: df.to_msgpack('foo.msg')

In [343]: pd.read_msgpack('foo.msg')
Out[343]:
          A         B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359

In [344]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [345]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1, 2, 3]), s)

In [346]: pd.read_msgpack('foo.msg')
Out[346]:
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 Freq: D, dtype: float64]
```

You can pass `iterator=True` to iterate over the unpacked results:

```
In [347]: for o in pd.read_msgpack('foo.msg', iterator=True):
   .....:     print(o)
   .....:
          A         B
```

(continues on next page)

(continued from previous page)

```
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359
foo
[1 2 3]
2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64
```

You can pass `append=True` to the writer to append to an existing pack:

```
In [348]: df.to_msgpack('foo.msg', append=True)

In [349]: pd.read_msgpack('foo.msg')
Out[349]:
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 Freq: D, dtype: float64,           A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359]
```

Unlike other io methods, `to_msgpack` is available on both a per-object basis, `df.to_msgpack()` and using the top-level `pd.to_msgpack(...)` where you can pack arbitrary collections of Python lists, dicts, scalars, while intermixing pandas objects.

```
In [350]: pd.to_msgpack('foo2.msg', {'dict': [{'df': df}, {'string': 'foo'},
   .....:                                     {'scalar': 1.}, {'s': s}]})
   .....:

In [351]: pd.read_msgpack('foo2.msg')
Out[351]:
{'dict': ({'df':           A         B
   0  0.170801  0.895366
   1  0.838238  0.052592
   2  0.664140  0.289750
   3  0.449593  0.872087
   4  0.983618  0.744359},
  {'string': 'foo'},
  {'scalar': 1.0},
  {'s': 2013-01-01    0.548134
```

(continues on next page)

(continued from previous page)

```
   2013-01-02    0.503447
   2013-01-03    0.348438
   2013-01-04    0.707267
   2013-01-05    0.261656
   Freq: D, dtype: float64})}
```

Read/Write API

Msgpacks can also be read from and written to strings.

```
In [352]: df.to_msgpack()
Out[352]: b'\x84\xa3typ\xadblock_
↪manager\xa5klass\xa9DataFrame\xa4axes\x92\x86\xa3typ\xa5index\xa5klass\xa5Index\xa4name\xc0\xa5dtyp
↪index\xa5klass\xaaRangeIndex\xa4name\xc0\xa5start\x00\xa4stop\x05\xa4step\x01\xa6blocks\x91\x86\xa4
↪<\xfd\xd2f\xcf\xdc\xc5?0\x15\xebN\xd9\xd2\xea?,\x9c\x16A\xa2@\xe5?\xd8/\xdd\xf4
↪"\xc6\xdc?\x11\x1e\x97\x1b\xcdy\xef?&\x1e<\xee\xd6\xa6\xec?p\xd3;\xb2N\xed\xaa?
↪h\xcb\xb1\xbdB\x8b\xd2?\xaf4\x01r"\xe8\xeb?)G6\xd9\xc9\xd1\xe7?
↪\xa5shape\x92\x02\x05\xa5dtype\xa7float64\xa5klass\xaaFloatBlock\xa8compress\xc0'
```

Furthermore you can concatenate the strings to produce a list of the original objects.

```
In [353]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[353]:
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 Freq: D, dtype: float64]
```

4.1.8 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the *cookbook* for some advanced strategies

> **Warning:** pandas requires `PyTables` >= 3.0.0. There is a indexing bug in `PyTables` < 3.2 which may appear when querying stores using an index. If you see a subset of results being returned, upgrade to `PyTables` >= 3.2. Stores created previously will need to be rewritten using the updated version.

```
In [354]: store = pd.HDFStore('store.h5')

In [355]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Objects can be written to the file just like adding key-value pairs to a dict:

```
In [356]: index = pd.date_range('1/1/2000', periods=8)

In [357]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [358]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....:

In [359]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:               major_axis=pd.date_range('1/1/2000', periods=5),
   .....:               minor_axis=['A', 'B', 'C', 'D'])
   .....:

# store.put('s', s) is an equivalent method
In [360]: store['s'] = s

In [361]: store['df'] = df

In [362]: store['wp'] = wp

# the type of stored data
In [363]: store.root.wp._v_attrs.pandas_type
Out[363]: 'wide'

In [364]: store
\\\\\\\\\\\\\\\\\Out[364]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

In a current or later Python session, you can retrieve stored objects:

```
# store.get('df') is an equivalent method
In [365]: store['df']
Out[365]:
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616

# dotted (attribute) access provides get as well
In [366]: store.df
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616
```

Deletion of the object specified by the key:

```
# store.remove('wp') is an equivalent method
In [367]: del store['wp']

In [368]: store
Out[368]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [369]: store.close()

In [370]: store
Out[370]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [371]: store.is_open
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[371]: False

# Working with, and automatically closing the store using a context manager
In [372]: with pd.HDFStore('store.h5') as store:
   .....:     store.keys()
   .....:
```

Read/Write API

`HDFStore` supports an top-level API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work.

```
In [373]: df_tl = pd.DataFrame({'A': list(range(5)), 'B': list(range(5))})

In [374]: df_tl.to_hdf('store_tl.h5', 'table', append=True)

In [375]: pd.read_hdf('store_tl.h5', 'table', where=['index>2'])
Out[375]:
   A  B
3  3  3
4  4  4
```

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting `dropna=True`.

```
In [376]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
   .....:                                 'col2': [1, np.nan, np.nan]})
   .....:

In [377]: df_with_missing
Out[377]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [378]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
```

(continues on next page)

(continued from previous page)

```
   .....:                        format='table', mode='w')
   .....:

In [379]: pd.read_hdf('file.h5', 'df_with_missing')
Out[379]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [380]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w', dropna=True)
   .....:

In [381]: pd.read_hdf('file.h5', 'df_with_missing')
Out[381]:
   col1  col2
0   0.0   1.0
2   2.0   NaN
```

This is also true for the major axis of a `Panel`:

```
In [382]: matrix = [[[np.nan, np.nan, np.nan], [1, np.nan, np.nan]],
   .....:           [[np.nan, np.nan, np.nan], [np.nan, 5, 6]],
   .....:           [[np.nan, np.nan, np.nan], [np.nan, 3, np.nan]]]
   .....:

In [383]: panel_with_major_axis_all_missing = pd.Panel(matrix,
   .....:                                               items=['Item1', 'Item2', 'Item3
↪'],
   .....:                                               major_axis=[1, 2],
   .....:                                               minor_axis=['A', 'B', 'C'])
   .....:

In [384]: panel_with_major_axis_all_missing
Out[384]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 1 to 2
Minor_axis axis: A to C

In [385]: panel_with_major_axis_all_missing.to_hdf('file.h5', 'panel',
   .....:                                         dropna=True,
   .....:                                         format='table',
   .....:                                         mode='w')
   .....:

In [386]: reloaded = pd.read_hdf('file.h5', 'panel')

In [387]: reloaded
Out[387]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 1 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 2 to 2
Minor_axis axis: A to C
```

Fixed Format

The examples above show storing using `put`, which write the HDF5 to `PyTables` in a fixed array format, called the `fixed` format. These types of stores are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The `fixed` format stores offer very fast writing and slightly faster reading than `table` stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

Warning: A `fixed` format will raise a `TypeError` if you try to retrieve using a `where`:

```
>>> pd.DataFrame(np.random.randn(10, 2)).to_hdf('test_fixed.h5', 'df')
>>> pd.read_hdf('test_fixed.h5', 'df', where='index>5')
TypeError: cannot pass a where specification when reading a fixed format.
           this store must be selected in its entirety
```

Table Format

`HDFStore` supports another `PyTables` format on disk, the `table` format. Conceptually a `table` is shaped very much like a DataFrame, with rows and columns. A `table` may be appended to in the same or other sessions. In addition, delete and query type operations are supported. This format is specified by `format='table'` or `format='t'` to `append` or `put` or `to_hdf`.

This format can be set as an option as well `pd.set_option('io.hdf.default_format','table')` to enable `put/append/to_hdf` to by default store in the `table` format.

```
In [388]: store = pd.HDFStore('store.h5')

In [389]: df1 = df[0:4]

In [390]: df2 = df[4:]

# append data (creates a table automatically)
In [391]: store.append('df', df1)

In [392]: store.append('df', df2)

In [393]: store
Out[393]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [394]: store.select('df')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[394]:
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616

# the type of stored data
```

(continues on next page)

(continued from previous page)

```
In [395]: store.root.df._v_attrs.pandas_type
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'frame_table'
```

Note: You can also create a `table` by passing `format='table'` or `format='t'` to a `put` operation.

Hierarchical Keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. `foo/bar/bah`), which will generate a hierarchy of sub-stores (or `Groups` in PyTables parlance). Keys can be specified with out the leading '/' and are **always** absolute (e.g. 'foo' refers to '/foo'). Removal operations can remove everything in the sub-store and **below**, so be *careful*.

```
In [396]: store.put('foo/bar/bah', df)

In [397]: store.append('food/orange', df)

In [398]: store.append('food/apple', df)

In [399]: store
Out[399]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [400]: store.keys()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[400]: ['/df',
↪ '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [401]: store.remove('food')

In [402]: store
Out[402]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

You can walk through the group hierarchy using the `walk` method which will yield a tuple for each group key along with the relative keys of its contents.

New in version 0.24.0.

```
In [403]: for (path, subgroups, subkeys) in store.walk():
   .....:     for subgroup in subgroups:
   .....:         print('GROUP: {}/{}'.format(path, subgroup))
   .....:     for subkey in subkeys:
   .....:         key = '/'.join([path, subkey])
   .....:         print('KEY: {}'.format(key))
   .....:         print(store.get(key))
   .....:
GROUP: /foo
KEY: /df
                   A         B         C
```

(continues on next page)

(continued from previous page)

```
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616
GROUP: /foo/bar
KEY: /foo/bar/bah
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616
```

Warning: Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

```
In [8]: store.foo.bar.bah
AttributeError: 'HDFStore' object has no attribute 'foo'

# you can directly access the actual PyTables node but using the root node
In [9]: store.root.foo.bar.bah
Out[9]:
/foo/bar/bah (Group) ''
  children := ['block0_items' (Array), 'block0_values' (Array), 'axis0' (Array),
↪'axis1' (Array)]
```

Instead, use explicit string based keys:

```
In [404]: store['foo/bar/bah']
Out[404]:
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616
```

Storing Types

Storing Mixed Types in a Table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent attempts at appending longer strings will raise a `ValueError`.

Passing `min_itemsize={`values`: size}` as a parameter to append will set a larger minimum for the string columns. Storing `floats, strings, ints, bools, datetime64` are currently supported. For string columns, passing `nan_rep = 'nan'` to append will change the default nan representation on disk (which converts to/from *np.nan*), this defaults to *nan*.

```
In [405]: df_mixed = pd.DataFrame({'A': np.random.randn(8),
   .....:                          'B': np.random.randn(8),
   .....:                          'C': np.array(np.random.randn(8), dtype='float32'),
   .....:                          'string': 'string',
   .....:                          'int': 1,
   .....:                          'bool': True,
   .....:                          'datetime64': pd.Timestamp('20010102')},
   .....:                         index=list(range(8)))
   .....:

In [406]: df_mixed.loc[df_mixed.index[3:5],
   .....:              ['A', 'B', 'string', 'datetime64']] = np.nan
   .....:

In [407]: store.append('df_mixed', df_mixed, min_itemsize={'values': 50})

In [408]: df_mixed1 = store.select('df_mixed')

In [409]: df_mixed1
Out[409]:
          A         B         C  string  int  bool datetime64
0  0.376816 -1.507533  0.255584  string    1  True 2001-01-02
1 -0.161614  0.335303  0.450263  string    1  True 2001-01-02
2 -1.636805 -1.340566  0.755221  string    1  True 2001-01-02
3       NaN       NaN -1.506656     NaN    1  True        NaT
4       NaN       NaN  0.808794     NaN    1  True        NaT
5  0.843452 -0.585357  0.019915  string    1  True 2001-01-02
6 -0.122918 -1.273599  0.300003  string    1  True 2001-01-02
7 -0.026122 -0.487935 -0.727093  string    1  True 2001-01-02

In [410]: df_mixed1.get_dtype_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
float64           2
float32           1
object            1
int64             1
bool              1
datetime64[ns]    1
dtype: int64

# we have provided a minimum string column size
In [411]: store.root.df_mixed.table
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
/df_mixed/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
  "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
  "values_block_2": Int64Col(shape=(1,), dflt=0, pos=3),
  "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
  "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
```

(continues on next page)

(continued from previous page)

```
  "values_block_5": StringCol(itemsize=50, shape=(1,), dflt=b'', pos=6)}
  byteorder := 'little'
  chunkshape := (689,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

Storing MultiIndex DataFrames

Storing MultiIndex `DataFrames` as tables is very similar to storing/selecting from homogeneous index `DataFrames`.

```
In [412]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   .....:                               ['one', 'two', 'three']],
   .....:                       codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   .....:                              [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:                       names=['foo', 'bar'])
   .....:

In [413]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
   .....:                      columns=['A', 'B', 'C'])
   .....:

In [414]: df_mi
Out[414]:
                  A         B         C
foo bar
foo one   -1.119363  1.878479 -0.026513
    two    0.573793  0.154237  3.272320
    three  0.077044  0.397034 -0.613983
bar one   -0.398346  0.507286 -0.368864
    two    1.096917  0.516017 -0.501550
baz two    0.138212  0.218366  0.848216
    three -0.948325  0.278775 -0.764608
qux one    1.145069  1.033972 -0.130405
    two   -1.561954 -0.872400  0.038366
    three -0.359613 -0.256250  0.754720

In [415]: store.append('df_mi', df_mi)

In [416]: store.select('df_mi')
Out[416]:
                  A         B         C
foo bar
foo one   -1.119363  1.878479 -0.026513
    two    0.573793  0.154237  3.272320
    three  0.077044  0.397034 -0.613983
bar one   -0.398346  0.507286 -0.368864
    two    1.096917  0.516017 -0.501550
baz two    0.138212  0.218366  0.848216
    three -0.948325  0.278775 -0.764608
qux one    1.145069  1.033972 -0.130405
    two   -1.561954 -0.872400  0.038366
    three -0.359613 -0.256250  0.754720
```

(continues on next page)

(continued from previous page)

```
# the levels are automatically included as data columns
In [417]: store.select('df_mi', 'foo=bar')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                A         B         C
foo bar
bar one -0.398346  0.507286 -0.368864
    two  1.096917  0.516017 -0.501550
```

Querying

Querying a Table

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

A query is specified using the `Term` class under the hood, as a boolean expression.

- `index` and `columns` are supported indexers of a `DataFrames`.
- `major_axis`, `minor_axis`, and `items` are supported indexers of the Panel.
- if `data_columns` are specified, these can be used as additional indexers.

Valid comparison operators are:

`=, ==, !=, >, >=, <, <=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `(` and `)` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

Note:

- = will be automatically expanded to the comparison operator ==
- ~ is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`
- `"columns in ['A', 'D']"`
- `'columns = A'`
- `'columns == A'`
- `"~(columns = ['A', 'B'])"`
- `'index > df.index[3] & string = "bar"'`

- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The `indexers` are on the left-hand side of the sub-expression:

`columns`, `major_axis`, `ts`

The right-hand side of the sub-expression (after a comparison operator) can be:

- functions that will be evaluated, e.g. `Timestamp('2012-02-01')`
- strings, e.g. `"bar"`
- date-like, e.g. `20130101`, or `"20130101"`
- lists, e.g. `"['A', 'B']"`
- variables that are defined in the local names space, e.g. `date`

Note: Passing a string to a query by interpolating it into the query expression is not recommended. Simply assign the string of interest to a variable and use that variable in an expression. For example, do this

```
string = "HolyMoly'"
store.select('df', 'index == string')
```

instead of this

```
string = "HolyMoly'"
store.select('df', 'index == %s' % string)
```

The latter will **not** work and will raise a `SyntaxError`.Note that there's a single quote followed by a double quote in the `string` variable.

If you *must* interpolate, use the `'%r'` format specifier

```
store.select('df', 'index == %r' % string)
```

which will quote `string`.

Here are some examples:

```
In [418]: dfq = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'),
   .....:                    index=pd.date_range('20130101', periods=10))
   .....:

In [419]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [420]: store.select('dfq', "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[420]:
                   A         B
2013-01-05 -0.039404  0.601192
2013-01-06 -0.492289 -0.189464
2013-01-07  0.956560  0.406396
2013-01-08 -1.736617  0.036195
2013-01-09  0.461206 -0.335209
2013-01-10  0.617881  0.396581
```

Use and inline column reference

```
In [421]: store.select('dfq', where="A>0 or C>0")
Out[421]:
                   A         B         C         D
2013-01-01  0.435520  0.837254  1.034678 -0.396519
2013-01-02  0.199649 -1.207961 -1.398402 -0.041796
2013-01-04  2.258150  0.559061  0.637610 -0.127414
2013-01-06 -0.492289 -0.189464  0.545629  0.826707
2013-01-07  0.956560  0.406396  0.881640 -0.810675
2013-01-08 -1.736617  0.036195  1.505626  1.051324
2013-01-09  0.461206 -0.335209  0.861521  0.107618
2013-01-10  0.617881  0.396581  1.235199  0.111300
```

Works with a Panel as well.

```
In [422]: store.append('wp', wp)

In [423]: store
Out[423]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [424]: store.select('wp',
   .....:              "major_axis>pd.Timestamp('20000102') & minor_axis=['A', 'B']")
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[424]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B
```

The `columns` keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a `'columns=list_of_columns_to_filter'`:

```
In [425]: store.select('df', "columns=['A', 'B']")
Out[425]:
                   A         B
2000-01-01  0.954551  0.859671
2000-01-02 -1.211386 -0.852728
2000-01-03  1.064650  1.014927
2000-01-04  0.254343 -0.875526
2000-01-05 -0.906920  0.988185
2000-01-06  0.205007 -0.772889
2000-01-07  0.768606  0.298656
2000-01-08 -0.618845  1.122438
```

`start` and `stop` parameters can be specified to limit the total search space. These are in terms of the total number of rows in a table.

```
# this is effectively what the storage of a Panel looks like
In [426]: wp.to_frame()
Out[426]:
                     Item1     Item2
major      minor
2000-01-01 A     -0.340872  0.758552
           B     -0.303900  0.384775
```

(continues on next page)

(continued from previous page)

```
            C     -0.871948 -1.133177
            D      0.151508 -1.002973
2000-01-02 A      0.587939 -1.644393
            B      0.043096 -0.311536
            C      0.362451 -0.615506
...                     ...       ...
2000-01-04 B      0.479460  0.542245
            C     -0.630652 -0.384429
            D     -0.182400  0.170697
2000-01-05 A     -0.789979 -0.200289
            B     -0.282700  1.220322
            C     -0.813404 -1.001273
            D      1.252998  0.162172

[20 rows x 2 columns]

# limiting the search
In [427]: store.select('wp', "major_axis>20000102 & minor_axis=['A', 'B']",
   .....:              start=0, stop=10)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 1 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to B
```

Note: `select` will raise a `ValueError` if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is **not** a data_column.

`select` will raise a `SyntaxError` if the query expression is not valid.

Using timedelta64[ns]

You can store and query using the `timedelta64[ns]` type. Terms can be specified in the format: `<float>(<unit>)`, where float may be signed (and fractional), and unit can be `D,s,ms,us,ns` for the timedelta. Here's an example:

```
In [428]: from datetime import timedelta

In [429]: dftd = pd.DataFrame({'A': pd.Timestamp('20130101'),
   .....:                      'B': [pd.Timestamp('20130101') + timedelta(days=i,
   .....:                                                                   seconds=10)
   .....:                            for i in range(10)]})
   .....:

In [430]: dftd['C'] = dftd['A'] - dftd['B']

In [431]: dftd
Out[431]:
           A                   B                  C
0 2013-01-01 2013-01-01 00:00:10  -1 days +23:59:50
```

(continues on next page)

(continued from previous page)

```
1 2013-01-01 2013-01-02 00:00:10  -2 days +23:59:50
2 2013-01-01 2013-01-03 00:00:10  -3 days +23:59:50
3 2013-01-01 2013-01-04 00:00:10  -4 days +23:59:50
4 2013-01-01 2013-01-05 00:00:10  -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10  -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10  -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10  -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10  -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50

In [432]: store.append('dftd', dftd, data_columns=True)

In [433]: store.select('dftd', "C<'-3.5D'")
Out[433]:
           A                   B                  C
4 2013-01-01 2013-01-05 00:00:10  -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10  -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10  -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10  -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10  -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50
```

Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`.

Note: Indexes are automagically created on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to `append`.

```
# we have automagically already created an index (in the first section)
In [434]: i = store.root.df.table.cols.index.index

In [435]: i.optlevel, i.kind
Out[435]: (6, 'medium')

# change an index by passing new parameters
In [436]: store.create_table_index('df', optlevel=9, kind='full')

In [437]: i = store.root.df.table.cols.index.index

In [438]: i.optlevel, i.kind
Out[438]: (9, 'full')
```

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```
In [439]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [440]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [441]: st = pd.HDFStore('appends.h5', mode='w')
```

(continues on next page)

(continued from previous page)

```
In [442]: st.append('df', df_1, data_columns=['B'], index=False)

In [443]: st.append('df', df_2, data_columns=['B'], index=False)

In [444]: st.get_storer('df').table
Out[444]:
/df/table (Table(20,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
```

Then create the index when finished appending.

```
In [445]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [446]: st.get_storer('df').table
Out[446]:
/df/table (Table(20,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
  autoindex := True
  colindexes := {
    "B": Index(9, full, shuffle, zlib(1)).is_csi=True}

In [447]: st.close()
```

See here for how to create a completely-sorted-index (CSI) on an existing store.

Query via Data Columns

You can designate (and index) certain columns that you want to be able to perform queries (other than the *indexable* columns, which you can always query). For instance say you want to perform this common operation, on-disk, and return just the frame that matches this query. You can specify `data_columns = True` to force all columns to be `data_columns`.

```
In [448]: df_dc = df.copy()

In [449]: df_dc['string'] = 'foo'

In [450]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan

In [451]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'

In [452]: df_dc['string2'] = 'cool'

In [453]: df_dc.loc[df_dc.index[1:3], ['B', 'C']] = 1.0
```

(continues on next page)

(continued from previous page)

```
In [454]: df_dc
Out[454]:
                   A         B         C string string2
2000-01-01  0.954551  0.859671 -1.170458    foo    cool
2000-01-02 -1.211386  1.000000  1.000000    foo    cool
2000-01-03  1.064650  1.000000  1.000000    foo    cool
2000-01-04  0.254343 -0.875526 -0.980856    foo    cool
2000-01-05 -0.906920  0.988185 -1.596540    NaN    cool
2000-01-06  0.205007 -0.772889 -0.043509    NaN    cool
2000-01-07  0.768606  0.298656 -1.294022    foo    cool
2000-01-08 -0.618845  1.122438 -0.914616    bar    cool

# on-disk operations
In [455]: store.append('df_dc', df_dc, data_columns=['B', 'C', 'string', 'string2'])

In [456]: store.select('df_dc', where='B > 0')
Out[456]:
                   A         B         C string string2
2000-01-01  0.954551  0.859671 -1.170458    foo    cool
2000-01-02 -1.211386  1.000000  1.000000    foo    cool
2000-01-03  1.064650  1.000000  1.000000    foo    cool
2000-01-05 -0.906920  0.988185 -1.596540    NaN    cool
2000-01-07  0.768606  0.298656 -1.294022    foo    cool
2000-01-08 -0.618845  1.122438 -0.914616    bar    cool

# getting creative
In [457]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A    B    C string string2
2000-01-02 -1.211386  1.0  1.0    foo    cool
2000-01-03  1.064650  1.0  1.0    foo    cool

# this is in-memory version of this type of selection
In [458]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A    B    C string string2
2000-01-02 -1.211386  1.0  1.0    foo    cool
2000-01-03  1.064650  1.0  1.0    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [459]: store.root.df_dc.table
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
/df_dc/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2),
  "C": Float64Col(shape=(), dflt=0.0, pos=3),
  "string": StringCol(itemsize=3, shape=(), dflt=b'', pos=4),
  "string2": StringCol(itemsize=4, shape=(), dflt=b'', pos=5)}
  byteorder := 'little'
  chunkshape := (1680,)
  autoindex := True
```

(continues on next page)

(continued from previous page)

```
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

There is some performance degradation by making lots of columns into *data columns*, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!).

Iterator

You can pass `iterator=True` or `chunksize=number_in_a_chunk` to `select` and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```
In [460]: for df in store.select('df', chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01  0.954551  0.859671 -1.170458
2000-01-02 -1.211386 -0.852728 -0.450781
2000-01-03  1.064650  1.014927 -0.810399
                   A         B         C
2000-01-04  0.254343 -0.875526 -0.980856
2000-01-05 -0.906920  0.988185 -1.596540
2000-01-06  0.205007 -0.772889 -0.043509
                   A         B         C
2000-01-07  0.768606  0.298656 -1.294022
2000-01-08 -0.618845  1.122438 -0.914616
```

Note: You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in pd.read_hdf('store.h5', 'df', chunksize=3):
    print(df)
```

Note, that the chunksize keyword applies to the **source** rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```
In [461]: dfeq = pd.DataFrame({'number': np.arange(1, 11)})

In [462]: dfeq
Out[462]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
```

(continues on next page)

(continued from previous page)

```
6       7
7       8
8       9
9      10

In [463]: store.append('dfeq', dfeq, data_columns=['number'])

In [464]: def chunks(l, n):
   .....:     return [l[i:i + n] for i in range(0, len(l), n)]
   .....:

In [465]: evens = [2, 4, 6, 8, 10]

In [466]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [467]: for c in chunks(coordinates, 2):
   .....:     print(store.select('dfeq', where=c))
   .....:
   number
1       2
3       4
   number
5       6
7       8
   number
9      10
```

Advanced Queries

Select a Single Column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

```
In [468]: store.select_column('df_dc', 'index')
Out[468]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
3   2000-01-04
4   2000-01-05
5   2000-01-06
6   2000-01-07
7   2000-01-08
Name: index, dtype: datetime64[ns]

In [469]: store.select_column('df_dc', 'string')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    foo
1    foo
2    foo
3    foo
```

(continues on next page)

(continued from previous page)

```
4     NaN
5     NaN
6     foo
7     bar
Name: string, dtype: object
```

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [470]: df_coord = pd.DataFrame(np.random.randn(1000, 2),
   .....:                         index=pd.date_range('20000101', periods=1000))
   .....:

In [471]: store.append('df_coord', df_coord)

In [472]: c = store.select_as_coordinates('df_coord', 'index > 20020101')

In [473]: c
Out[473]:
Int64Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
            ...
            990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
           dtype='int64', length=268)

In [474]: store.select('df_coord', where=c)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   0         1
2002-01-02 -0.084214 -0.013194
2002-01-03 -0.992486 -1.686663
2002-01-04  0.535116  0.231741
2002-01-05 -1.044620 -0.915880
2002-01-06 -1.854848  0.594045
2002-01-07  0.571847 -0.214103
2002-01-08 -0.291002 -0.727758
...              ...       ...
2002-09-20  0.496286  1.089641
2002-09-21 -1.464885 -1.138942
2002-09-22 -0.639688  0.996575
2002-09-23 -0.287881  2.368693
2002-09-24  0.234548 -0.358702
2002-09-25  0.883037  0.838329
2002-09-26 -0.255707  1.926149

[268 rows x 2 columns]
```

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this `mask` would be a resulting `index` from an indexing operation. This example selects the months of a datetimeindex which are 5.

```
In [475]: df_mask = pd.DataFrame(np.random.randn(1000, 2),
   .....:                        index=pd.date_range('20000101', periods=1000))
   .....:

In [476]: store.append('df_mask', df_mask)

In [477]: c = store.select_column('df_mask', 'index')

In [478]: where = c[pd.DatetimeIndex(c).month == 5].index

In [479]: store.select('df_mask', where=where)
Out[479]:
                   0         1
2000-05-01  0.498698 -0.643722
2000-05-02 -0.028228  0.070209
2000-05-03 -0.791176  0.393495
2000-05-04  2.410230 -0.368339
2000-05-05 -1.934392  2.398912
2000-05-06  0.521658 -2.389278
2000-05-07  0.395639 -0.003721
...              ...       ...
2002-05-25 -0.193055 -1.233890
2002-05-26  1.205218 -0.450279
2002-05-27  1.245164 -0.295348
2002-05-28  0.179561  0.648173
2002-05-29  0.524473 -0.528987
2002-05-30  2.782922  2.358034
2002-05-31 -1.565489  0.005781

[93 rows x 2 columns]
```

Storer Object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [480]: store.get_storer('df_dc').nrows
Out[480]: 8
```

Multiple Table Queries

The methods `append_to_multiple` and `select_as_multiple` can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table's index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

The `append_to_multiple` method splits a given single DataFrame into multiple tables according to `d`, a dictionary that maps the table names to a list of 'columns' you want in that table. If *None* is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument `selector` defines which table is the selector table (which you can make queries from). The argument `dropna` will drop rows from the input `DataFrame` to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely `np.NaN`, that row will be dropped from all tables.

If `dropna` is False, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. Remember that entirely `np.Nan` rows are not written to the HDFStore, so if you choose to call `dropna=False`, some tables may have more rows than others, and therefore `select_as_multiple` may not work or it may return unexpected results.

```
In [481]: df_mt = pd.DataFrame(np.random.randn(8, 6),
   .....:                      index=pd.date_range('1/1/2000', periods=8),
   .....:                      columns=['A', 'B', 'C', 'D', 'E', 'F'])
   .....:

In [482]: df_mt['foo'] = 'bar'

In [483]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [484]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None},
   .....:                          df_mt, selector='df1_mt')
   .....:

In [485]: store
Out[485]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [486]: store.select('df1_mt')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[486]:
                   A         B
2000-01-01 -1.517439 -0.453150
2000-01-02       NaN       NaN
2000-01-03 -1.309888  0.253324
2000-01-04  1.345157  0.811016
2000-01-05  0.221351 -0.097178
2000-01-06 -0.820806  0.332738
2000-01-07  0.965282  0.744376
2000-01-08 -1.173242 -1.454623

In [487]: store.select('df2_mt')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   C         D         E         F  foo
2000-01-01 -0.827739 -1.421726 -0.929968 -0.363586  bar
2000-01-02  0.577961 -0.200132 -2.117306 -0.061709  bar
2000-01-03  1.686667 -1.818496 -1.552377 -0.152178  bar
2000-01-04  0.914579  0.338951  1.062033  0.203995  bar
2000-01-05 -0.043690 -1.394338 -0.593160 -0.564757  bar
2000-01-06  0.384450 -2.073892 -0.118535 -1.575343  bar
2000-01-07  0.532858 -0.495289  1.888109  1.644645  bar
2000-01-08  2.034578 -1.532878 -0.414398 -2.048101  bar

# as a multiple
In [488]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                          selector='df1_mt')
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D         E         F  foo
2000-01-04  1.345157  0.811016  0.914579  0.338951  1.062033  0.203995  bar
```

(continues on next page)

(continued from previous page)

```
2000-01-07  0.965282  0.744376  0.532858 -0.495289  1.888109  1.644645  bar
```

Delete from a Table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the `PyTables` deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very expensive operation depending on the orientation of your data. This is especially true in higher dimensional objects (`Panel` and `Panel4D`). To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the `indexables`.

Data is ordered (on the disk) in terms of the `indexables`. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- **date_1**
  - id_1
  - id_2
  - .
  - id_n
- **date_2**
  - id_1
  - .
  - id_n

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is removed, then the following data moved. On the other hand a delete operation on the `minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

```
# returns the number of rows deleted
In [489]: store.remove('wp', 'major_axis > 20000102')
Out[489]: 12

In [490]: store.select('wp')
\\\\\\\\\\\\\Out[490]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

Warning: Please note that HDF5 **DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again, **WILL TEND TO INCREASE THE FILE SIZE**.

To *repack and clean* the file, use *ptrepack*.

Notes & Caveats

Compression

`PyTables` allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

`complevel` specifies if and how hard data is to be compressed. `complevel=0` and `complevel=None` disables compression and `0<complevel<10` enables compression.

`complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:

- zlib: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
- lzo: Fast compression and decompression.
- bzip2: Good compression rates.
- blosc: Fast compression and decompression.

New in version 0.20.2: Support for alternative blosc compressors:

- blosc:blosclz This is the default compressor for `blosc`
- blosc:lz4: A compact, very popular and fast compressor.
- blosc:lz4hc: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
- blosc:snappy: A popular compressor used in many places.
- blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
- blosc:zstd: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

Note: If the library specified with the `complib` option is missing on your platform, compression defaults to `zlib` without further ado.

Enable compression for all objects within the file:

```
store_compressed = pd.HDFStore('store_compressed.h5', complevel=9,
                               complib='blosc:blosclz')
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```
store.append('df', df, complib='zlib', complevel=5)
```

ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

Caveats

> **Warning:** `HDFStore` is **not-threadsafe for writing**. The underlying `PyTables` only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the (GH2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- Once a `table` is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

> **Warning:** `PyTables` will show a `NaturalNameWarning` if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a `where` clause and are generally a bad idea.

DataTypes

`HDFStore` will map an object dtype to the `PyTables` underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating : `float64, float32, float16`	`np.nan`
integer : `int64, int32, int8, uint64,uint32, uint8`	
boolean	
`datetime64[ns]`	`NaT`
`timedelta64[ns]`	`NaT`
categorical : see the section below	
object : `strings`	`np.nan`

`unicode` columns are not supported, and **WILL FAIL**.

Categorical Data

You can write data that contains `category` dtypes to a `HDFStore`. Queries work the same as if it was an object array. However, the `category` dtyped data is stored in a more efficient manner.

```
In [491]: dfcat = pd.DataFrame({'A': pd.Series(list('aabbcdba')).astype('category'),
   .....:                       'B': np.random.randn(8)})
   .....:

In [492]: dfcat
Out[492]:
   A         B
0  a -2.367211
1  a -0.058314
2  b -0.432893
3  b -0.056124
4  c  1.692911
5  d  0.771411
6  b -0.132217
7  a  1.575762

In [493]: dfcat.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    category
B     float64
dtype: object

In [494]: cstore = pd.HDFStore('cats.h5', mode='w')

In [495]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [496]: result = cstore.select('dfcat', where="A in ['b', 'c']")

In [497]: result
Out[497]:
   A         B
2  b -0.432893
3  b -0.056124
4  c  1.692911
6  b -0.132217

In [498]: result.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[498]:␣
↪
A    category
B     float64
dtype: object
```

String Columns

min_itemsize

The underlying implementation of `HDFStore` uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specify the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data_columns* to have this min_itemsize.

Passing a `min_itemsize` dict will cause all passed columns to be created as *data_columns* automatically.

Note: If you are not passing any `data_columns`, then the `min_itemsize` will be the maximum of the length of any string passed

```
In [499]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'}, index=list(range(5)))

In [500]: dfs
Out[500]:
     A    B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
4  foo  bar

# A and B have a size of 30
In [501]: store.append('dfs', dfs, min_itemsize=30)

In [502]: store.get_storer('dfs').table
Out[502]:
/dfs/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [503]: store.append('dfs2', dfs, min_itemsize={'A': 30})

In [504]: store.get_storer('dfs2').table
Out[504]:
/dfs2/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
  "A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

nan_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [505]: dfss = pd.DataFrame({'A': ['foo', 'bar', 'nan']})

In [506]: dfss
Out[506]:
     A
0  foo
1  bar
2  nan

In [507]: store.append('dfss', dfss)

In [508]: store.select('dfss')
Out[508]:
     A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [509]: store.append('dfss2', dfss, nan_rep='_nan_')

In [510]: store.select('dfss2')
Out[510]:
     A
0  foo
1  bar
2  nan
```

External Compatibility

`HDFStore` writes `table` format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, `HDFStore` can read native `PyTables` format tables.

It is possible to write an `HDFStore` object that can easily be imported into `R` using the `rhdf5` library (Package website). Create a table format store like this:

```
In [511]: df_for_r = pd.DataFrame({"first": np.random.rand(100),
   .....:                          "second": np.random.rand(100),
   .....:                          "class": np.random.randint(0, 2, (100, ))},
   .....:                         index=range(100))
   .....:

In [512]: df_for_r.head()
Out[512]:
      first    second  class
0  0.365127  0.012583      0
1  0.004826  0.607160      1
2  0.373122  0.092975      1
3  0.207335  0.828299      0
4  0.515948  0.832586      1

In [513]: store_export = pd.HDFStore('export.h5')

In [514]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [515]: store_export
```

(continues on next page)

(continued from previous page)

```
Out[515]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
```

In R this file can be read into a `data.frame` object using the `rhdf5` library. The following example function reads the corresponding column names and data values from the values and assembles them into a `data.frame`:

```
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.

library(rhdf5)

loadhdf5data <- function(h5File) {

listing <- h5ls(h5File)
# Find all data nodes, values are stored in *_values and corresponding column
# titles in *_items
data_nodes <- grep("_values", listing$name)
name_nodes <- grep("_items", listing$name)
data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
columns = list()
for (idx in seq(data_paths)) {
  # NOTE: matrices returned by h5read have to be transposed to obtain
  # required Fortran order!
  data <- data.frame(t(h5read(h5File, data_paths[idx])))
  names <- t(h5read(h5File, name_paths[idx]))
  entry <- data.frame(data)
  colnames(entry) <- names
  columns <- append(columns, entry)
}

data <- data.frame(columns)

return(data)
}
```

Now you can import the `DataFrame` into R:

```
> data = loadhdf5data("transfer.hdf5")
> head(data)
         first    second class
1 0.4170220047 0.3266449     0
2 0.7203244934 0.5270581     0
3 0.0001143748 0.8859421     1
4 0.3023325726 0.3572698     1
5 0.1467558908 0.9085352     1
6 0.0923385948 0.6233601     1
```

Note: The R function lists the entire HDF5 file's contents and assembles the `data.frame` object from all matching nodes, so use this only as a starting point if you have stored multiple `DataFrame` objects to a single HDF5 file.

Performance

- `tables` format come with a writing performance penalty as compared to `fixed` stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=<int>` to the first `append`, to set the TOTAL number of expected rows that `PyTables` will expected. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See Here for more information and some solutions.

4.1.9 Feather

New in version 0.20.0.

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize DataFrames, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.
- The format will NOT write an `Index`, or `MultiIndex` for the `DataFrame` and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation.

```
In [516]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.Categorical(list('abc')),
   .....:                    'g': pd.date_range('20130101', periods=3),
   .....:                    'h': pd.date_range('20130101', periods=3, tz='US/Eastern
↪'),
   .....:                    'i': pd.date_range('20130101', periods=3, freq='ns')})
   .....:

In [517]: df
Out[517]:
   a  b  c    d      e  f          g                         h                   ␣
↪     i
```

(continues on next page)

(continued from previous page)

```
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
↪000000000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
↪000000001
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
↪000000002

In [518]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a                         object
b                          int64
c                          uint8
d                        float64
e                           bool
f                       category
g                 datetime64[ns]
h     datetime64[ns, US/Eastern]
i                 datetime64[ns]
dtype: object
```

Write to a feather file.

```
In [519]: df.to_feather('example.feather')
```

Read from a feather file.

```
In [520]: result = pd.read_feather('example.feather')

In [521]: result
Out[521]:
   a  b  c    d      e  f          g                         h                    ␣
↪     i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.
↪000000000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.
↪000000001
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.
↪000000002

# we preserve dtypes
In [522]: result.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a                         object
b                          int64
c                          uint8
d                        float64
e                           bool
f                       category
g                 datetime64[ns]
h     datetime64[ns, US/Eastern]
i                 datetime64[ns]
dtype: object
```

4.1.10 Parquet

New in version 0.21.0.

Apache Parquet provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize `DataFrame` s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- The `pyarrow` engine always writes the index to the output, but `fastparquet` only writes non-default indexes. This extra column can cause problems for non-Pandas consumers that are not expecting it. You can force including or omitting indexes with the `index` argument, regardless of the underlying engine.
- Index level names, if specified, must be strings.
- Categorical dtypes can be serialized to parquet, but will de-serialize as `object` dtype.
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

You can specify an `engine` to direct the serialization. This can be one of `pyarrow`, or `fastparquet`, or `auto`. If the engine is NOT specified, then the `pd.options.io.parquet.engine` option is checked; if this is also `auto`, then `pyarrow` is tried, and falling back to `fastparquet`.

See the documentation for pyarrow and fastparquet.

Note: These engines are very similar and should read/write nearly identical parquet format files. Currently `pyarrow` does not support timedelta data, `fastparquet>=0.1.4` supports timezone aware datetimes. These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [523]: df = pd.DataFrame({'a': list('abc'),
   .....:                    'b': list(range(1, 4)),
   .....:                    'c': np.arange(3, 6).astype('u1'),
   .....:                    'd': np.arange(4.0, 7.0, dtype='float64'),
   .....:                    'e': [True, False, True],
   .....:                    'f': pd.date_range('20130101', periods=3),
   .....:                    'g': pd.date_range('20130101', periods=3, tz='US/Eastern
↪')})
   .....:

In [524]: df
Out[524]: 
   a  b  c    d      e          f                         g
0  a  1  3  4.0   True 2013-01-01 2013-01-01 00:00:00-05:00
1  b  2  4  5.0  False 2013-01-02 2013-01-02 00:00:00-05:00
2  c  3  5  6.0   True 2013-01-03 2013-01-03 00:00:00-05:00

In [525]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a                        object
b                         int64
c                         uint8
```

(continues on next page)

(continued from previous page)

```
d                        float64
e                           bool
f                 datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Write to a parquet file.

```
In [526]: df.to_parquet('example_pa.parquet', engine='pyarrow')

In [527]: df.to_parquet('example_fp.parquet', engine='fastparquet')
```

Read from a parquet file.

```
In [528]: result = pd.read_parquet('example_fp.parquet', engine='fastparquet')

In [529]: result = pd.read_parquet('example_pa.parquet', engine='pyarrow')

In [530]: result.dtypes
Out[530]: 
a                         object
b                          int64
c                          uint8
d                        float64
e                           bool
f                 datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Read only certain columns of a parquet file.

```
In [531]: result = pd.read_parquet('example_fp.parquet',
   .....:                          engine='fastparquet', columns=['a', 'b'])
   .....: 

In [532]: result.dtypes
Out[532]: 
a    object
b     int64
dtype: object
```

Handling Indexes

Serializing a `DataFrame` to parquet may include the implicit index as one or more columns in the output file. Thus, this code:

```
In [533]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})

In [534]: df.to_parquet('test.parquet', engine='pyarrow')
```

creates a parquet file with *three* columns if you use `pyarrow` for serialization: `a`, `b`, and `__index_level_0__`. If you're using `fastparquet`, the index may or may not be written to the file.

This unexpected extra column causes some databases like Amazon Redshift to reject the file, because that column doesn't exist in the target table.

If you want to omit a dataframe's indexes when writing, pass `index=False` to *to_parquet()*:

```
In [535]: df.to_parquet('test.parquet', index=False)
```

This creates a parquet file with just the two expected columns, `a` and `b`. If your `DataFrame` has a custom index, you won't get it back when you load this file into a `DataFrame`.

Passing `index=True` will *always* write the index, even if that's not the underlying engine's default behavior.

Partitioning Parquet files

New in version 0.24.0.

Parquet supports partitioning of data based on the values of one or more columns.

```
In [536]: df = pd.DataFrame({'a': [0, 0, 1, 1], 'b': [0, 1, 0, 1]})

In [537]: df.to_parquet(fname='test', engine='pyarrow',
   .....:               partition_cols=['a'], compression=None)
   .....:
```

The *fname* specifies the parent directory to which data will be saved. The *partition_cols* are the column names by which the dataset will be partitioned. Columns are partitioned in the order they are given. The partition splits are determined by the unique values in the partition columns. The above example creates a partitioned dataset that may look like:

```
test
├── a=0
│   ├── 0bac803e32dc42ae83fddfd029cbdebc.parquet
│   └──  ...
└── a=1
    ├── e6ab24a4f45147b49b54a662f0c412a3.parquet
    └── ...
```

4.1.11 SQL Queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python's standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some *cookbook examples* for some advanced strategies.

The key functions are:

read_sql_table(table_name, con[, schema, . . .])	Read SQL database table into a DataFrame.
read_sql_query(sql, con[, index_col, . . .])	Read SQL query into a DataFrame.
read_sql(sql, con[, index_col, . . .])	Read SQL query or database table into a DataFrame.
DataFrame.to_sql(name, con[, schema, . . .])	Write records stored in a DataFrame to a SQL database.

pandas.read_sql_table

`pandas.`**`read_sql_table`**(*table_name*, *con*, *schema=None*, *index_col=None*, *coerce_float=True*, *parse_dates=None*, *columns=None*, *chunksize=None*)

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters

table_name [string] Name of SQL table in database.

con [SQLAlchemy connectable (or database string URI)] SQLite DBAPI connection mode not supported.

schema [string, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [boolean, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *`pandas.to_datetime()`* Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default: None] List of column names to select from SQL table

chunksize [int, default None] If specified, returns an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_query` Read SQL query into a DataFrame.

`read_sql`

Notes

Any datetime values with time zone information will be converted to UTC.

pandas.read_sql_query

`pandas.`**`read_sql_query`**(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *chunksize=None*)

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an *index_col* parameter to use one of the columns as the index, otherwise default integer index will be used.

Parameters

sql [string SQL query or SQLAlchemy Selectable (select or text object)] SQL query to be executed.

con [SQLAlchemy connectable(engine/connection), database string URI,] or sqlite3 DBAPI2 connection Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [boolean, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_table` Read SQL database table into a DataFrame.

`read_sql`

Notes

Any datetime values with time zone information parsed via the *parse_dates* parameter will be converted to UTC.

pandas.read_sql

`pandas.`**`read_sql`**(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *columns=None*, *chunksize=None*)

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around `read_sql_table` and `read_sql_query` (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to `read_sql_query`, while a database table name will be routed to `read_sql_table`. Note that the delegated function might have more specific notes about their functionality not listed here.

Parameters

sql [string or SQLAlchemy Selectable (select or text object)] SQL query to be executed or a table name.

con [SQLAlchemy connectable (engine/connection) or database string URI] or DBAPI2 connection (fallback mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [boolean, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *`pandas.to_datetime()`* Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default: None] List of column names to select from SQL table (only used when reading a table).

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_table` Read SQL database table into a DataFrame.

`read_sql_query` Read SQL query into a DataFrame.

pandas.DataFrame.to_sql

`DataFrame.to_sql`(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [?] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

method [{None, 'multi', callable}, default None] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section *insert method*.

New in version 0.24.0.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

New in version 0.24.0.

References

[?], [?]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

Note: The function *read_sql()* is a convenience wrapper around *read_sql_table()* and *read_sql_query()* (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the SQlite SQL database engine. You can use a temporary SQLite database where data are stored in "memory".

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on `create_engine()` and the URI formatting, see the examples below and the SQLAlchemy documentation

```
In [538]: from sqlalchemy import create_engine

# Create your engine.
In [539]: engine = create_engine('sqlite:///:memory:')
```

If you want to manage your own connections you can pass one of those instead:

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

Writing DataFrames

Assuming the following data is in a `DataFrame` `data`, we can insert it into the database using *to_sql()*.

id	Date	Col_1	Col_2	Col_3
26	2012-10-18	X	25.7	True
42	2012-10-19	Y	-12.4	False
63	2012-10-20	Z	5.73	True

```
In [540]: data
Out[540]:
   id        Date Col_1  Col_2  Col_3
0  26 2010-10-18     X  27.50   True
1  42 2010-10-19     Y -12.50  False
2  63 2010-10-20     Z   5.73   True

In [541]: data.to_sql('data', engine)
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the `chunksize` parameter when calling `to_sql`. For example, the following writes `data` to the database in batches of 1000 rows at a time:

```
In [542]: data.to_sql('data_chunked', engine, chunksize=1000)
```

SQL data types

to_sql() will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype `object`, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the `dtype` argument. This argument needs a dictionary mapping column names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
In [543]: from sqlalchemy.types import String

In [544]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

Note: Due to the limited support for timedelta's in the different database flavors, columns with type `timedelta64` will be written as integer values as nanoseconds to the database and a warning will be raised.

Note: Columns of `category` dtype will be converted to the dense representation as you would get with `np.asarray(categorical)` (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does **not** generate a categorical.

Datetime data types

Using SQLAlchemy, *to_sql()* is capable of writing datetime data that is timezone naive or timezone aware. However, the resulting data stored in the database ultimately depends on the supported data type for datetime data of the database system being used.

The following table lists supported data types for datetime data for some common databases. Other database dialects may have different data types for datetime data.

Database	SQL Datetime Types	Timezone Support
SQLite	`TEXT`	No
MySQL	`TIMESTAMP` or `DATETIME`	No
PostgreSQL	`TIMESTAMP` or `TIMESTAMP WITH TIME ZONE`	Yes

When writing timezone aware data to databases that do not support timezones, the data will be written as timezone naive timestamps that are in local time with respect to the timezone.

read_sql_table() is also capable of reading datetime data that is timezone aware or naive. When reading `TIMESTAMP WITH TIME ZONE` types, pandas will convert the data to UTC.

Insertion Method

New in version 0.24.0.

The parameter `method` controls the SQL insertion clause used. Possible values are:

- `None`: Uses standard SQL `INSERT` clause (one per row).
- `'multi'`: Pass multiple values in a single `INSERT` clause. It uses a *special* SQL syntax not supported by all backends. This usually provides better performance for analytic databases like *Presto* and *Redshift*, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documention.
- callable with signature `(pd_table, conn, keys, data_iter)`: This can be used to implement a more performant insertion method based on specific backend dialect features.

Example of a callable using PostgreSQL COPY clause:

```
# Alternative to_sql() *method* for DBs that support COPY FROM
import csv
from io import StringIO

def psql_insert_copy(table, conn, keys, data_iter):
    # gets a DBAPI connection that can provide a cursor
    dbapi_conn = conn.connection
    with dbapi_conn.cursor() as cur:
        s_buf = StringIO()
        writer = csv.writer(s_buf)
        writer.writerows(data_iter)
        s_buf.seek(0)

        columns = ', '.join('"{}"'.format(k) for k in keys)
        if table.schema:
            table_name = '{}.{}'.format(table.schema, table.name)
        else:
            table_name = table.name

        sql = 'COPY {} ({}) FROM STDIN WITH CSV'.format(
            table_name, columns)
        cur.copy_expert(sql=sql, file=s_buf)
```

Reading Tables

`read_sql_table()` will read a database table given the table name and optionally a subset of columns to read.

Note: In order to use `read_sql_table()`, you **must** have the SQLAlchemy optional dependency installed.

```
In [545]: pd.read_sql_table('data', engine)
Out[545]:
   index  id        Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

You can also specify the name of the column as the `DataFrame` index, and specify a subset of columns to be read.

```
In [546]: pd.read_sql_table('data', engine, index_col='id')
Out[546]:
    index       Date Col_1  Col_2  Col_3
id
26      0 2010-10-18     X  27.50   True
42      1 2010-10-19     Y -12.50  False
63      2 2010-10-20     Z   5.73   True

In [547]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

And you can explicitly force columns to be parsed as dates:

```
In [548]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[548]:
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to *pandas.to_datetime()*:

```
pd.read_sql_table('data', engine, parse_dates={'Date': '%Y-%m-%d'})
pd.read_sql_table('data', engine,
                  parse_dates={'Date': {'format': '%Y-%m-%d %H:%M:%S'}})
```

You can check if a table exists using `has_table()`

Schema support

Reading from and writing to different schema's is supported through the `schema` keyword in the *read_sql_table()* and *to_sql()* functions. Note however that this depends on the database flavor (sqlite does not have schema's). For example:

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

Querying

You can query using raw SQL in the *read_sql_query()* function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```
In [549]: pd.read_sql_query('SELECT * FROM data', engine)
Out[549]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X  27.50      1
1      1  42  2010-10-19 00:00:00.000000     Y -12.50      0
2      2  63  2010-10-20 00:00:00.000000     Z   5.73      1
```

Of course, you can specify a more "complex" query.

```
In [550]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;",␣
↪engine)
Out[550]:
   id Col_1  Col_2
0  42     Y  -12.5
```

The *read_sql_query()* function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [551]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [552]: df.to_sql('data_chunks', engine, index=False)
```

```
In [553]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks",
   .....:                                engine, chunksize=5):
   .....:     print(chunk)
   .....:
          a         b         c
0  1.892774 -0.253477  0.714395
1  0.664179  0.897140  0.455791
2  1.549590  1.031231  0.196447
3  0.190188  0.619078  0.036658
4 -0.100501  0.201772  1.763002
          a         b         c
0  0.454977 -1.958922 -0.628529
1  0.133171 -1.274374  2.518925
2 -0.517547 -0.360773  0.877961
3 -1.881598 -0.699067 -1.566913
4  0.824581 -0.674292 -1.136157
          a         b         c
0  0.897844  1.430243  0.356573
1  0.425873 -0.183641 -0.264369
2 -1.136008  1.249329 -0.940339
3  0.495185 -0.458671  2.425292
4 -1.502778  0.510224  0.700035
          a         b         c
0  0.864174  0.839935 -1.670980
1 -0.579623  2.291643  0.616315
2 -0.003224  1.075287 -0.657822
3  1.471841 -0.284241  0.675387
4 -1.676314  0.126986  0.031905
```

You can also run a plain query without creating a `DataFrame` with `execute()`. This is useful for queries that don't return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')
```

(continues on next page)

(continued from previous page)

```
# or absolute, starting with a slash:
engine = create_engine('sqlite:////absolute/path/to/foo.db')
```

For more information see the examples the SQLAlchemy documentation

Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [554]: import sqlalchemy as sa

In [555]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'),
   .....:             engine, params={'col1': 'X'})
   .....:
Out[555]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X   27.5      1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [556]: metadata = sa.MetaData()

In [557]: data_table = sa.Table('data', metadata,
   .....:                       sa.Column('index', sa.Integer),
   .....:                       sa.Column('Date', sa.DateTime),
   .....:                       sa.Column('Col_1', sa.String),
   .....:                       sa.Column('Col_2', sa.Float),
   .....:                       sa.Column('Col_3', sa.Boolean),
   .....:                       )
   .....:

In [558]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 is True),␣
↪engine)
Out[558]:
Empty DataFrame
Columns: [index, Date, Col_1, Col_2, Col_3]
Index: []
```

You can combine SQLAlchemy expressions with parameters passed to *read_sql()* using `sqlalchemy.bindparam()`

```
In [559]: import datetime as dt

In [560]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date
↪'))

In [561]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[561]:
   index       Date Col_1  Col_2  Col_3
0      1 2010-10-19     Y -12.50  False
1      2 2010-10-20     Z   5.73   True
```

Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```
import sqlite3
con = sqlite3.connect(':memory:')
```

And then issue the following queries:

```
data.to_sql('data', con)
pd.read_sql_query("SELECT * FROM data", con)
```

4.1.12 Google BigQuery

Warning: Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package `pandas-gbq`. You can `pip install pandas-gbq` to get it.

The `pandas-gbq` package provides functionality to read/write from Google BigQuery.

pandas integrates with this external package. if `pandas-gbq` is installed, you can use the pandas methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from `pandas-gbq`.

Full documentation can be found here.

4.1.13 Stata Format

Writing to Stata format

The method `to_stata()` will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

```
In [562]: df = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [563]: df.to_stata('stata.dta')
```

Stata data files have limited data type support; only strings with 244 or fewer characters, `int8`, `int16`, `int32`, `float32` and `float64` can be stored in `.dta` files. Additionally, *Stata* reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in Stata for a particular data type will retype the variable to the next larger size. For example, `int8` values are restricted to lie between -127 and 100 in Stata, and so variables with values above 100 will trigger a conversion to `int16`. `nan` values in floating points data types are stored as the basic missing data type (`.` in *Stata*).

Note: It is not possible to export missing data values for integer data types.

The *Stata* writer gracefully handles other data types including `int64`, `bool`, `uint8`, `uint16`, `uint32` by casting to the smallest supported type that can represent the data. For example, data with a type of `uint8` will be cast to `int8` if all values are less than 100 (the upper bound for non-missing `int8` data in *Stata*), or, if values are outside of this range, the variable is cast to `int16`.

Warning: Conversion from `int64` to `float64` may result in a loss of precision if `int64` values are larger than 2**53.

Warning: `StataWriter` and `to_stata()` only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write *Stata* dta files with strings longer than 244 characters raises a `ValueError`.

Reading from Stata format

The top-level function `read_stata` will read a dta file and return either a `DataFrame` or a `StataReader` that can be used to read the file incrementally.

```
In [564]: pd.read_stata('stata.dta')
Out[564]: 
   index         A         B
0      0  1.789587 -1.454551
1      1  0.420343 -1.046421
2      2  0.104704 -1.113877
3      3 -0.837801 -0.257054
4      4 -2.735632  0.702059
5      5 -0.015027  0.123337
6      6 -0.163379 -0.702725
7      7  1.885344  1.187743
8      8 -0.061320 -0.413435
9      9  0.037276  0.006548
```

Specifying a `chunksize` yields a `StataReader` instance that can be used to read `chunksize` lines from the file at a time. The `StataReader` object can be used as an iterator.

```
In [565]: reader = pd.read_stata('stata.dta', chunksize=3)

In [566]: for df in reader:
   .....:     print(df.shape)
   .....: 
(3, 3)
(3, 3)
(3, 3)
(1, 3)
```

For more fine-grained control, use `iterator=True` and specify `chunksize` with each call to `read()`.

```
In [567]: reader = pd.read_stata('stata.dta', iterator=True)

In [568]: chunk1 = reader.read(5)

In [569]: chunk2 = reader.read(5)
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have `object` data type.

Note: `read_stata()` and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

Note: Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

Categorical Data

`Categorical` data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a `Categorical` and information about *whether* the variable is ordered is lost when exporting.

Warning: *Stata* only supports string value labels, and so `str` is called on the categories when exporting data. Exporting `Categorical` variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as `Categorical` variables using the keyword argument `convert_categoricals` (`True` by default). The keyword argument `order_categoricals` (`True` by default) determines whether imported `Categorical` variables are ordered.

Note: When importing categorical data, the values of the variables in the *Stata* data file are not preserved since `Categorical` variables always use integer data types between `-1` and `n-1` where `n` is the number of categories. If the original values in the *Stata* data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original *Stata* data values and the category codes of imported Categorical variables: missing values are assigned code `-1`, and the smallest original value is assigned `0`, the second smallest is assigned `1` and so on until the largest original value is assigned the code `n-1`.

Note: *Stata* supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a `Categorical` with string categories for the values that are labeled and numeric categories for values with no label.

4.1.14 SAS Formats

The top-level function `read_sas()` can read (but not write) SAS *xport* (.XPT) and (since *v0.18.0*) *SAS7BDAT* (.sas7bdat) format files.

SAS files only contain two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates. By default the whole file is read and returned as a `DataFrame`.

Specify a `chunksize` or use `iterator=True` to obtain reader objects (`XportReader` or `SAS7BDATReader`) for incrementally reading the file. The reader objects also have attributes that contain additional information about the file and its variables.

Read a SAS7BDAT file:

```
df = pd.read_sas('sas_data.sas7bdat')
```

Obtain an iterator and read an XPORT file 100,000 lines at a time:

```
def do_something(chunk):
    pass

rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
for chunk in rdr:
    do_something(chunk)
```

The specification for the xport file format is available from the SAS web site.

No official documentation is available for the SAS7BDAT format.

4.1.15 Other file formats

pandas itself only supports IO with a limited set of file formats that map cleanly to its tabular data model. For reading and writing other file formats into and from pandas, we recommend these packages from the broader community.

netCDF

xarray provides data structures inspired by the pandas `DataFrame` for working with multi-dimensional datasets, with a focus on the netCDF file format and easy conversion to and from pandas.

4.1.16 Performance Considerations

This is an informal comparison of various IO methods, using pandas 0.20.3. Timings are machine dependent and small differences should be ignored.

```
In [1]: sz = 1000000
In [2]: df = pd.DataFrame({'A': np.random.randn(sz), 'B': [1] * sz})

In [3]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A    1000000 non-null float64
B    1000000 non-null int64
dtypes: float64(1), int64(1)
memory usage: 15.3 MB
```

Given the next test set:

```
from numpy.random import randn

sz = 1000000
df = pd.DataFrame({'A': randn(sz), 'B': [1] * sz})


def test_sql_write(df):
    if os.path.exists('test.sql'):
        os.remove('test.sql')
    sql_db = sqlite3.connect('test.sql')
    df.to_sql(name='test_table', con=sql_db)
    sql_db.close()


def test_sql_read():
    sql_db = sqlite3.connect('test.sql')
    pd.read_sql_query("select * from test_table", sql_db)
    sql_db.close()


def test_hdf_fixed_write(df):
    df.to_hdf('test_fixed.hdf', 'test', mode='w')


def test_hdf_fixed_read():
    pd.read_hdf('test_fixed.hdf', 'test')


def test_hdf_fixed_write_compress(df):
    df.to_hdf('test_fixed_compress.hdf', 'test', mode='w', complib='blosc')


def test_hdf_fixed_read_compress():
    pd.read_hdf('test_fixed_compress.hdf', 'test')


def test_hdf_table_write(df):
    df.to_hdf('test_table.hdf', 'test', mode='w', format='table')


def test_hdf_table_read():
    pd.read_hdf('test_table.hdf', 'test')


def test_hdf_table_write_compress(df):
    df.to_hdf('test_table_compress.hdf', 'test', mode='w',
              complib='blosc', format='table')


def test_hdf_table_read_compress():
    pd.read_hdf('test_table_compress.hdf', 'test')


def test_csv_write(df):
    df.to_csv('test.csv', mode='w')
```

(continues on next page)

(continued from previous page)

```
def test_csv_read():
    pd.read_csv('test.csv', index_col=0)


def test_feather_write(df):
    df.to_feather('test.feather')


def test_feather_read():
    pd.read_feather('test.feather')


def test_pickle_write(df):
    df.to_pickle('test.pkl')


def test_pickle_read():
    pd.read_pickle('test.pkl')


def test_pickle_write_compress(df):
    df.to_pickle('test.pkl.compress', compression='xz')


def test_pickle_read_compress():
    pd.read_pickle('test.pkl.compress', compression='xz')
```

When writing, the top-three functions in terms of speed are are `test_pickle_write`, `test_feather_write` and `test_hdf_fixed_write_compress`.

```
In [14]: %timeit test_sql_write(df)
2.37 s ± 36.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [15]: %timeit test_hdf_fixed_write(df)
194 ms ± 65.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [26]: %timeit test_hdf_fixed_write_compress(df)
119 ms ± 2.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [16]: %timeit test_hdf_table_write(df)
623 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [27]: %timeit test_hdf_table_write_compress(df)
563 ms ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [17]: %timeit test_csv_write(df)
3.13 s ± 49.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [30]: %timeit test_feather_write(df)
103 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [31]: %timeit test_pickle_write(df)
109 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [32]: %timeit test_pickle_write_compress(df)
3.33 s ± 55.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

When reading, the top three are `test_feather_read`, `test_pickle_read` and `test_hdf_fixed_read`.

```
In [18]: %timeit test_sql_read()
1.35 s ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %timeit test_hdf_fixed_read()
14.3 ms ± 438 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [28]: %timeit test_hdf_fixed_read_compress()
23.5 ms ± 672 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [20]: %timeit test_hdf_table_read()
35.4 ms ± 314 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [29]: %timeit test_hdf_table_read_compress()
42.6 ms ± 2.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [22]: %timeit test_csv_read()
516 ms ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [33]: %timeit test_feather_read()
4.06 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [34]: %timeit test_pickle_read()
6.5 ms ± 172 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [35]: %timeit test_pickle_read_compress()
588 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Space on disk (in bytes)

```
34816000 Aug 21 18:00 test.sql
24009240 Aug 21 18:00 test_fixed.hdf
 7919610 Aug 21 18:00 test_fixed_compress.hdf
24458892 Aug 21 18:00 test_table.hdf
 8657116 Aug 21 18:00 test_table_compress.hdf
28520770 Aug 21 18:00 test.csv
16000248 Aug 21 18:00 test.feather
16000848 Aug 21 18:00 test.pkl
 7554108 Aug 21 18:00 test.pkl.compress
```

4.2 Indexing and Selecting Data

The axis labeling information in pandas objects serves many purposes:

- Identifies data (i.e. provides *metadata*) using known indicators, important for analysis, visualization, and interactive console display.
- Enables automatic and explicit data alignment.
- Allows intuitive getting and setting of subsets of the data set.

In this section, we will focus on the final point: namely, how to slice, dice, and generally get and set subsets of pandas objects. The primary focus will be on Series and DataFrame as they have received more development attention in this area.

Note: The Python and NumPy indexing operators `[]` and attribute operator `.` provide quick and easy access to pandas data structures across a wide range of use cases. This makes interactive work intuitive, as there's little new to learn if you already know how to deal with Python dictionaries and NumPy arrays. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommended that you take advantage of the optimized pandas data access methods exposed in this chapter.

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Warning: Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see *here*.

See the *MultiIndex / Advanced Indexing* for `MultiIndex` and more advanced indexing documentation.

See the *cookbook* for some advanced strategies.

4.2.1 Different Choices for Indexing

Object selection has had a number of user-requested additions in order to support more explicit location based indexing. Pandas now supports three types of multi-axis indexing.

- `.loc` is primarily label based, but may also be used with a boolean array. `.loc` will raise `KeyError` when the items are not found. Allowed inputs are:
  - A single label, e.g. `5` or `'a'` (Note that `5` is interpreted as a *label* of the index. This use is **not** an integer position along the index.).
  - A list or array of labels `['a', 'b', 'c']`.
  - A slice object with labels `'a':'f'` (Note that contrary to usual python slices, **both** the start and the stop are included, when present in the index! See *Slicing with labels*.).
  - A boolean array
  - A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above).

    New in version 0.18.1.

  See more at *Selection by Label*.
- `.iloc` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array. `.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy *slice* semantics). Allowed inputs are:
  - An integer e.g. `5`.
  - A list or array of integers `[4, 3, 0]`.
  - A slice object with ints `1:7`.
  - A boolean array.
  - A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above).

    New in version 0.18.1.

See more at *Selection by Position*, *Advanced Indexing* and *Advanced Hierarchical*.

- `.loc`, `.iloc`, and also `[]` indexing can accept a `callable` as indexer. See more at *Selection By Callable*.

Getting values from an object with multi-axes selection uses the following notation (using `.loc` as an example, but the following applies to `.iloc` as well). Any of the axes accessors may be the null slice `:`. Axes left out of the specification are assumed to be `:`, e.g. `p.loc['a']` is equivalent to `p.loc['a', :, :]`.

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`
Panel	`p.loc[item_indexer,major_indexer, minor_indexer]`

4.2.2 Basics

As mentioned when introducing the data structures in the *last section*, the primary function of indexing with `[]` (a.k.a. `__getitem__` for those familiar with implementing class behavior in Python) is selecting out lower-dimensional slices. The following table shows return type values when indexing pandas objects with `[]`:

Object Type	Selection	Return Value Type
Series	`series[label]`	scalar value
DataFrame	`frame[colname]`	`Series` corresponding to colname
Panel	`panel[itemname]`	`DataFrame` corresponding to the itemname

Here we construct a simple time series data set to use for illustrating the indexing functionality:

```
In [1]: dates = pd.date_range('1/1/2000', periods=8)

In [2]: df = pd.DataFrame(np.random.randn(8, 4),
   ...:                   index=dates, columns=['A', 'B', 'C', 'D'])
   ...:

In [3]: df
Out[3]:
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

In [4]: panel = pd.Panel({'one': df, 'two': df - df.mean()})

In [5]: panel
Out[5]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D
```

Note: None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using `[]`:

```
In [6]: s = df['A']

In [7]: s[dates[5]]
Out[7]: -0.6736897080883706

In [8]: panel['two']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]:
                   A         B         C         D
2000-01-01  0.409571  0.113086 -0.610826 -0.936507
2000-01-02  1.152571  0.222735  1.017442 -0.845111
2000-01-03 -0.921390 -1.708620  0.403304  1.270929
2000-01-04  0.662014 -0.310822 -0.141342  0.470985
2000-01-05 -0.484513  0.962970  1.174465 -0.888276
2000-01-06 -0.733231  0.509598 -0.580194  0.724113
2000-01-07  0.345164  0.972995 -0.816769 -0.840143
2000-01-08 -0.430188 -0.761943 -0.446079  1.044010
```

You can pass a list of columns to `[]` to select columns in that order. If a column is not contained in the DataFrame, an exception will be raised. Multiple columns can also be set in this manner:

```
In [9]: df
Out[9]:
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

In [10]: df[['B', 'A']] = df[['A', 'B']]

In [11]: df
Out[11]:
                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04 -0.706771  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885
```

You may find this useful for applying a transform (in-place) to a subset of the columns.

Warning: pandas aligns all AXES when setting `Series` and `DataFrame` from `.loc`, and `.iloc`.

This will **not** modify `df` because the column alignment is before value assignment.

```
In [12]: df[['A', 'B']]
Out[12]:
                   A         B
2000-01-01 -0.282863  0.469112
2000-01-02 -0.173215  1.212112
2000-01-03 -2.104569 -0.861849
2000-01-04 -0.706771  0.721555
2000-01-05  0.567020 -0.424972
2000-01-06  0.113648 -0.673690
2000-01-07  0.577046  0.404705
2000-01-08 -1.157892 -0.370647

In [13]: df.loc[:, ['B', 'A']] = df[['A', 'B']]

In [14]: df[['A', 'B']]
Out[14]:
                   A         B
2000-01-01 -0.282863  0.469112
2000-01-02 -0.173215  1.212112
2000-01-03 -2.104569 -0.861849
2000-01-04 -0.706771  0.721555
2000-01-05  0.567020 -0.424972
2000-01-06  0.113648 -0.673690
2000-01-07  0.577046  0.404705
2000-01-08 -1.157892 -0.370647
```

The correct way to swap column values is by using raw values:

```
In [15]: df.loc[:, ['B', 'A']] = df[['A', 'B']].to_numpy()

In [16]: df[['A', 'B']]
Out[16]:
                   A         B
2000-01-01  0.469112 -0.282863
2000-01-02  1.212112 -0.173215
2000-01-03 -0.861849 -2.104569
2000-01-04  0.721555 -0.706771
2000-01-05 -0.424972  0.567020
2000-01-06 -0.673690  0.113648
2000-01-07  0.404705  0.577046
2000-01-08 -0.370647 -1.157892
```

4.2.3 Attribute Access

You may access an index on a `Series`, column on a `DataFrame`, and an item on a `Panel` directly as an attribute:

```
In [17]: sa = pd.Series([1, 2, 3], index=list('abc'))

In [18]: dfa = df.copy()
```

```
In [19]: sa.b
Out[19]: 2

In [20]: dfa.A
\\\\\\\\\\\Out[20]:
```

(continues on next page)

(continued from previous page)

```
2000-01-01    0.469112
2000-01-02    1.212112
2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
2000-01-06   -0.673690
2000-01-07    0.404705
2000-01-08   -0.370647
Freq: D, Name: A, dtype: float64

In [21]: panel.one
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
```

```
In [22]: sa.a = 5

In [23]: sa
Out[23]:
a    5
b    2
c    3
dtype: int64

In [24]: dfa.A = list(range(len(dfa.index)))  # ok if A already exists

In [25]: dfa
Out[25]:
            A         B         C         D
2000-01-01  0 -0.282863 -1.509059 -1.135632
2000-01-02  1 -0.173215  0.119209 -1.044236
2000-01-03  2 -2.104569 -0.494929  1.071804
2000-01-04  3 -0.706771 -1.039575  0.271860
2000-01-05  4  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.577046 -1.715002 -1.039268
2000-01-08  7 -1.157892 -1.344312  0.844885

In [26]: dfa['A'] = list(range(len(dfa.index)))  # use this form to create a new␣
↪column

In [27]: dfa
Out[27]:
            A         B         C         D
2000-01-01  0 -0.282863 -1.509059 -1.135632
2000-01-02  1 -0.173215  0.119209 -1.044236
2000-01-03  2 -2.104569 -0.494929  1.071804
2000-01-04  3 -0.706771 -1.039575  0.271860
```

(continues on next page)

(continued from previous page)

```
2000-01-05  4  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.577046 -1.715002 -1.039268
2000-01-08  7 -1.157892 -1.344312  0.844885
```

Warning:

- You can use this access only if the index element is a valid Python identifier, e.g. `s.1` is not allowed. See here for an explanation of valid identifiers.
- The attribute will not be available if it conflicts with an existing method name, e.g. `s.min` is not allowed.
- Similarly, the attribute will not be available if it conflicts with any of the following list: `index`, `major_axis`, `minor_axis`, `items`.
- In any of these cases, standard indexing will still work, e.g. `s['1']`, `s['min']`, and `s['index']` will access the corresponding element or column.

If you are using the IPython environment, you may also use tab-completion to see these accessible attributes.

You can also assign a `dict` to a row of a `DataFrame`:

```
In [28]: x = pd.DataFrame({'x': [1, 2, 3], 'y': [3, 4, 5]})

In [29]: x.iloc[1] = {'x': 9, 'y': 99}

In [30]: x
Out[30]:
   x   y
0  1   3
1  9  99
2  3   5
```

You can use attribute access to modify an existing element of a Series or column of a DataFrame, but be careful; if you try to use attribute access to create a new column, it creates a new attribute rather than a new column. In 0.21.0 and later, this will raise a `UserWarning`:

```
In [1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In [2]: df.two = [4, 5, 6]
UserWarning: Pandas doesn't allow Series to be assigned into nonexistent columns -␣
↪see https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute_access
In [3]: df
Out[3]:
   one
0  1.0
1  2.0
2  3.0
```

4.2.4 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the *Selection by Position* section detailing the `.iloc` method. For now, we explain the semantics of slicing using the `[]` operator.

With Series, the syntax works exactly as with an ndarray, returning a slice of the values and the corresponding labels:

```
In [31]: s[:5]
Out[31]:
2000-01-01    0.469112
2000-01-02    1.212112
2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
Freq: D, Name: A, dtype: float64

In [32]: s[::2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-01    0.469112
2000-01-03   -0.861849
2000-01-05   -0.424972
2000-01-07    0.404705
Freq: 2D, Name: A, dtype: float64

In [33]: s[::-1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-08   -0.370647
2000-01-07    0.404705
2000-01-06   -0.673690
2000-01-05   -0.424972
2000-01-04    0.721555
2000-01-03   -0.861849
2000-01-02    1.212112
2000-01-01    0.469112
Freq: -1D, Name: A, dtype: float64
```

Note that setting works as well:

```
In [34]: s2 = s.copy()

In [35]: s2[:5] = 0

In [36]: s2
Out[36]:
2000-01-01    0.000000
2000-01-02    0.000000
2000-01-03    0.000000
2000-01-04    0.000000
2000-01-05    0.000000
2000-01-06   -0.673690
2000-01-07    0.404705
2000-01-08   -0.370647
Freq: D, Name: A, dtype: float64
```

With DataFrame, slicing inside of `[]` **slices the rows**. This is provided largely as a convenience since it is such a common operation.

```
In [37]: df[:3]
Out[37]:
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
```

(continues on next page)

(continued from previous page)

```
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804

In [38]: df[::-1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
```

4.2.5 Selection By Label

Warning: Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Warning:

`.loc` is strict when you present slicers that are not compatible (or convertible) with the index type. For example using integers in a `DatetimeIndex`. These will raise a `TypeError`.

```
In [39]: dfl = pd.DataFrame(np.random.randn(5, 4),
   ....:                    columns=list('ABCD'),
   ....:                    index=pd.date_range('20130101', periods=5))
   ....:

In [40]: dfl
Out[40]:
                   A         B         C         D
2013-01-01  1.075770 -0.109050  1.643563 -1.469388
2013-01-02  0.357021 -0.674600 -1.776904 -0.968914
2013-01-03 -1.294524  0.413738  0.276662 -0.472035
2013-01-04 -0.013960 -0.362543 -0.006154 -0.923061
2013-01-05  0.895717  0.805244 -1.206412  2.565646
```

```
In [4]: dfl.loc[2:3]
TypeError: cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'>␣
↪with these indexers [2] of <type 'int'>
```

String likes in slicing *can* be convertible to the type of the index and lead to natural slicing.

```
In [41]: dfl.loc['20130102':'20130104']
Out[41]:
                   A         B         C         D
2013-01-02  0.357021 -0.674600 -1.776904 -0.968914
2013-01-03 -1.294524  0.413738  0.276662 -0.472035
2013-01-04 -0.013960 -0.362543 -0.006154 -0.923061
```

> **Warning:** Starting in 0.21.0, pandas will show a `FutureWarning` if indexing with a list with missing labels. In the future this will raise a `KeyError`. See *list-like Using loc with missing keys in a list is Deprecated*.

pandas provides a suite of methods in order to have **purely label based indexing**. This is a strict inclusion based protocol. Every label asked for must be in the index, or a `KeyError` will be raised. When slicing, both the start bound **AND** the stop bound are *included*, if present in the index. Integers are valid labels, but they refer to the label **and not the position**.

The `.loc` attribute is the primary access method. The following are valid inputs:

- A single label, e.g. `5` or `'a'` (Note that `5` is interpreted as a *label* of the index. This use is **not** an integer position along the index.).
- A list or array of labels `['a', 'b', 'c']`.
- A slice object with labels `'a':'f'` (Note that contrary to usual python slices, **both** the start and the stop are included, when present in the index! See *Slicing with labels*.).
- A boolean array.
- A `callable`, see *Selection By Callable*.

```
In [42]: s1 = pd.Series(np.random.randn(6), index=list('abcdef'))

In [43]: s1
Out[43]: 
a    1.431256
b    1.340309
c   -1.170299
d   -0.226169
e    0.410835
f    0.813850
dtype: float64

In [44]: s1.loc['c':]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
c   -1.170299
d   -0.226169
e    0.410835
f    0.813850
dtype: float64

In [45]: s1.loc['b']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1.3403088497993827
```

Note that setting works as well:

```
In [46]: s1.loc['c':] = 0

In [47]: s1
Out[47]: 
a    1.431256
b    1.340309
c    0.000000
d    0.000000
e    0.000000
```

(continues on next page)

(continued from previous page)

```
f    0.000000
dtype: float64
```

With a DataFrame:

```
In [48]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list('abcdef'),
   ....:                    columns=list('ABCD'))
   ....:

In [49]: df1
Out[49]: 
          A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
c  1.024180  0.569605  0.875906 -2.211372
d  0.974466 -2.006747 -0.410001 -0.078638
e  0.545952 -1.219217 -1.226825  0.769804
f -1.281247 -0.727707 -0.121306 -0.097883

In [50]: df1.loc[['a', 'b', 'd'], :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
d  0.974466 -2.006747 -0.410001 -0.078638
```

Accessing via label slices:

```
In [51]: df1.loc['d':, 'A':'C']
Out[51]: 
          A         B         C
d  0.974466 -2.006747 -0.410001
e  0.545952 -1.219217 -1.226825
f -1.281247 -0.727707 -0.121306
```

For getting a cross section using a label (equivalent to `df.xs('a')`):

```
In [52]: df1.loc['a']
Out[52]: 
A    0.132003
B   -0.827317
C   -0.076467
D   -1.187678
Name: a, dtype: float64
```

For getting values with a boolean array:

```
In [53]: df1.loc['a'] > 0
Out[53]: 
A     True
B    False
C    False
D    False
Name: a, dtype: bool
```

(continues on next page)

(continued from previous page)

```
In [54]: df1.loc[:, df1.loc['a'] > 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[54]:
          A
a  0.132003
b  1.130127
c  1.024180
d  0.974466
e  0.545952
f -1.281247
```

For getting a value explicitly (equivalent to deprecated `df.get_value('a','A')`):

```
# this is also equivalent to ``df1.at['a','A']``
In [55]: df1.loc['a', 'A']
Out[55]: 0.13200317033032932
```

Slicing with labels

When using `.loc` with slices, if both the start and the stop labels are present in the index, then elements *located* between the two (including them) are returned:

```
In [56]: s = pd.Series(list('abcde'), index=[0, 3, 2, 5, 4])

In [57]: s.loc[3:5]
Out[57]:
3    b
2    c
5    d
dtype: object
```

If at least one of the two is absent, but the index is sorted, and can be compared against start and stop labels, then slicing will still work as expected, by selecting labels which *rank* between the two:

```
In [58]: s.sort_index()
Out[58]:
0    a
2    c
3    b
4    e
5    d
dtype: object

In [59]: s.sort_index().loc[1:6]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[59]:
2    c
3    b
4    e
5    d
dtype: object
```

However, if at least one of the two is absent *and* the index is not sorted, an error will be raised (since doing otherwise would be computationally expensive, as well as potentially ambiguous for mixed type indexes). For instance, in the above example, `s.loc[1:6]` would raise `KeyError`.

4.2.6 Selection By Position

> **Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

Pandas provides a suite of methods in order to get **purely integer based indexing**. The semantics follow closely Python and NumPy slicing. These are `0-based` indexing. When slicing, the start bounds is *included*, while the upper bound is *excluded*. Trying to use a non-integer, even a **valid** label will raise an `IndexError`.

The `.iloc` attribute is the primary access method. The following are valid inputs:

- An integer e.g. `5`.
- A list or array of integers `[4, 3, 0]`.
- A slice object with ints `1:7`.
- A boolean array.
- A `callable`, see *Selection By Callable*.

```
In [60]: s1 = pd.Series(np.random.randn(5), index=list(range(0, 10, 2)))

In [61]: s1
Out[61]:
0    0.695775
2    0.341734
4    0.959726
6   -1.110336
8   -0.619976
dtype: float64

In [62]: s1.iloc[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[62
↪
0    0.695775
2    0.341734
4    0.959726
dtype: float64

In [63]: s1.iloc[3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪-1.1103361028911669
```

Note that setting works as well:

```
In [64]: s1.iloc[:3] = 0

In [65]: s1
Out[65]:
0    0.000000
2    0.000000
4    0.000000
6   -1.110336
8   -0.619976
dtype: float64
```

With a DataFrame:

```
In [66]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list(range(0, 12, 2)),
   ....:                    columns=list(range(0, 8, 2)))
   ....:

In [67]: df1
Out[67]:
           0         2         4         6
0   0.149748 -0.732339  0.687738  0.176444
2   0.403310 -0.154951  0.301624 -2.179861
4  -1.369849 -0.954208  1.462696 -1.743161
6  -0.826591 -0.345352  1.314232  0.690579
8   0.995761  2.396780  0.014871  3.357427
10 -0.317441 -1.236269  0.896171 -0.487602
```

Select via integer slicing:

```
In [68]: df1.iloc[:3]
Out[68]:
          0         2         4         6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161

In [69]: df1.iloc[1:5, 2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          4         6
2  0.301624 -2.179861
4  1.462696 -1.743161
6  1.314232  0.690579
8  0.014871  3.357427
```

Select via integer list:

```
In [70]: df1.iloc[[1, 3, 5], [1, 3]]
Out[70]:
           2         6
2  -0.154951 -2.179861
6  -0.345352  0.690579
10 -1.236269 -0.487602
```

```
In [71]: df1.iloc[1:3, :]
Out[71]:
          0         2         4         6
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
```

```
In [72]: df1.iloc[:, 1:3]
Out[72]:
           2         4
0  -0.732339  0.687738
2  -0.154951  0.301624
4  -0.954208  1.462696
6  -0.345352  1.314232
8   2.396780  0.014871
10 -1.236269  0.896171
```

```
# this is also equivalent to ``df1.iat[1,1]``
In [73]: df1.iloc[1, 1]
Out[73]: -0.15495077442490321
```

For getting a cross section using an integer position (equiv to `df.xs(1)`):

```
In [74]: df1.iloc[1]
Out[74]:
0    0.403310
2   -0.154951
4    0.301624
6   -2.179861
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/Numpy.

```
# these are allowed in python/numpy.
In [75]: x = list('abcdef')

In [76]: x
Out[76]: ['a', 'b', 'c', 'd', 'e', 'f']

In [77]: x[4:10]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[77]: ['e', 'f']

In [78]: x[8:10]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[78]: []

In [79]: s = pd.Series(x)

In [80]: s
Out[80]:
0    a
1    b
2    c
3    d
4    e
5    f
dtype: object

In [81]: s.iloc[4:10]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[81]:
4    e
5    f
dtype: object

In [82]: s.iloc[8:10]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Series([], dtype: object)
```

Note that using slices that go out of bounds can result in an empty axis (e.g. an empty DataFrame being returned).

```
In [83]: dfl = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [84]: dfl
Out[84]:
          A         B
```

(continues on next page)

(continued from previous page)

```
0 -0.082240 -2.182937
1  0.380396  0.084844
2  0.432390  1.519970
3 -0.493662  0.600178
4  0.274230  0.132885

In [85]: dfl.iloc[:, 2:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [86]: dfl.iloc[:, 1:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          B
0 -2.182937
1  0.084844
2  1.519970
3  0.600178
4  0.132885

In [87]: dfl.iloc[4:6]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
         A         B
4  0.27423  0.132885
```

A single indexer that is out of bounds will raise an `IndexError`. A list of indexers where any element is out of bounds will raise an `IndexError`.

```
>>> dfl.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds

>>> dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

4.2.7 Selection By Callable

New in version 0.18.1.

`.loc`, `.iloc`, and also `[]` indexing can accept a `callable` as indexer. The `callable` must be a function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing.

```
In [88]: df1 = pd.DataFrame(np.random.randn(6, 4),
   ....:                    index=list('abcdef'),
   ....:                    columns=list('ABCD'))
   ....:

In [89]: df1
Out[89]:
          A         B         C         D
a -0.023688  2.410179  1.450520  0.206053
b -0.251905 -2.213588  1.063327  1.266143
```

(continues on next page)

(continued from previous page)

```
c  0.299368 -0.863838  0.408204 -1.048089
d -0.025747 -0.988387  0.094055  1.262731
e  1.289997  0.082423 -0.055758  0.536580
f -0.489682  0.369374 -0.034571 -2.484478

In [90]: df1.loc[lambda df: df.A > 0, :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C         D
c  0.299368 -0.863838  0.408204 -1.048089
e  1.289997  0.082423 -0.055758  0.536580

In [91]: df1.loc[:, lambda df: ['A', 'B']]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B
a -0.023688  2.410179
b -0.251905 -2.213588
c  0.299368 -0.863838
d -0.025747 -0.988387
e  1.289997  0.082423
f -0.489682  0.369374

In [92]: df1.iloc[:, lambda df: [0, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B
a -0.023688  2.410179
b -0.251905 -2.213588
c  0.299368 -0.863838
d -0.025747 -0.988387
e  1.289997  0.082423
f -0.489682  0.369374

In [93]: df1[lambda df: df.columns[0]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a   -0.023688
b   -0.251905
c    0.299368
d   -0.025747
e    1.289997
f   -0.489682
Name: A, dtype: float64
```

You can use callable indexing in `Series`.

```
In [94]: df1.A.loc[lambda s: s > 0]
Out[94]:
c    0.299368
e    1.289997
Name: A, dtype: float64
```

Using these methods / indexers, you can chain data selection operations without using temporary variable.

```
In [95]: bb = pd.read_csv('data/baseball.csv', index_col='id')
```

(continues on next page)

(continued from previous page)

```
In [96]: (bb.groupby(['year', 'team']).sum()
   ....:     .loc[lambda df: df.r > 100])
   ....: 
Out[96]: 
           stint    g    ab    r    h  X2b  X3b  hr    rbi    sb   cs   bb     so  ␣
↪ibb   hbp    sh    sf  gidp
year team                                                                          ␣
↪
2007 CIN       6  379   745  101  203   35    2  36  125.0  10.0  1.0  105  127.0  14.
↪0   1.0   1.0  15.0  18.0
     DET       5  301  1062  162  283   54    4  37  144.0  24.0  7.0   97  176.0   3.
↪0  10.0   4.0   8.0  28.0
     HOU       4  311   926  109  218   47    6  14   77.0  10.0  4.0   60  212.0   3.
↪0   9.0  16.0   6.0  17.0
     LAN      11  413  1021  153  293   61    3  36  154.0   7.0  5.0  114  141.0   8.
↪0   9.0   3.0   8.0  29.0
     NYN      13  622  1854  240  509  101    3  61  243.0  22.0  4.0  174  310.0  24.
↪0  23.0  18.0  15.0  48.0
     SFN       5  482  1305  198  337   67    6  40  171.0  26.0  7.0  235  188.0  51.
↪0   8.0  16.0   6.0  41.0
     TEX       2  198   729  115  200   40    4  28  115.0  21.0  4.0   73  140.0   4.
↪0   5.0   2.0   8.0  16.0
     TOR       4  459  1408  187  378   96    2  58  223.0   4.0  2.0  190  265.0  16.
↪0  12.0   4.0  16.0  38.0
```

4.2.8 IX Indexer is Deprecated

Warning: Starting in 0.20.0, 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 recommended methods of indexing are:

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

```
In [97]: dfd = pd.DataFrame({'A': [1, 2, 3],
   ....:                     'B': [4, 5, 6]},
   ....:                    index=list('abc'))
   ....: 

In [98]: dfd
Out[98]: 
   A  B
a  1  4
b  2  5
c  3  6
```

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

```
In [3]: dfd.ix[[0, 2], 'A']
Out[3]:
```

(continues on next page)

(continued from previous page)

```
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 [99]: dfd.loc[dfd.index[[0, 2]], 'A']
Out[99]:
a    1
c    3
Name: A, dtype: int64
```

This can also be expressed using `.iloc`, by explicitly getting locations on the indexers, and using *positional* indexing to select things.

```
In [100]: dfd.iloc[[0, 2], dfd.columns.get_loc('A')]
Out[100]:
a    1
c    3
Name: A, dtype: int64
```

For getting *multiple* indexers, using `.get_indexer`:

```
In [101]: dfd.iloc[[0, 2], dfd.columns.get_indexer(['A', 'B'])]
Out[101]:
   A  B
a  1  4
c  3  6
```

4.2.9 Indexing with list with missing labels is Deprecated

Warning: Starting in 0.21.0, using `.loc` or `[]` with a list with one or more missing labels, is deprecated, in favor of `.reindex`.

In prior versions, using `.loc[list-of-labels]` would work as long as *at least 1* of the keys was found (otherwise it would raise a `KeyError`). This behavior is deprecated and will show a warning message pointing to this section. The recommended alternative is to use `.reindex()`.

For example.

```
In [102]: s = pd.Series([1, 2, 3])

In [103]: s
Out[103]:
0    1
1    2
2    3
dtype: int64
```

Selection with all keys found is unchanged.

```
In [104]: s.loc[[1, 2]]
Out[104]:
```

(continues on next page)

(continued from previous page)

```
1    2
2    3
dtype: int64
```

Previous Behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Current Behavior

```
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc with any non-matching elements will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-
↪listlike

Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Reindexing

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`. See also the section on *reindexing*.

```
In [105]: s.reindex([1, 2, 3])
Out[105]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Alternatively, if you want to select only *valid* keys, the following is idiomatic and efficient; it is guaranteed to preserve the dtype of the selection.

```
In [106]: labels = [1, 2, 3]

In [107]: s.loc[s.index.intersection(labels)]
Out[107]:
1    2
2    3
dtype: int64
```

Having a duplicated index will raise for a `.reindex()`:

```
In [108]: s = pd.Series(np.arange(4), index=['a', 'a', 'b', 'c'])

In [109]: labels = ['c', 'd']
```

```
In [17]: s.reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

Generally, you can intersect the desired labels with the current axis, and then reindex.

```
In [110]: s.loc[s.index.intersection(labels)].reindex(labels)
Out[110]:
c    3.0
d    NaN
dtype: float64
```

However, this would *still* raise if your resulting index is duplicated.

```
In [41]: labels = ['a', 'd']

In [42]: s.loc[s.index.intersection(labels)].reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

4.2.10 Selecting Random Samples

A random selection of rows or columns from a Series, DataFrame, or Panel with the `sample()` method. The method will sample rows by default, and accepts a specific number of rows/columns to return, or a fraction of rows.

```
In [111]: s = pd.Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1 row.
In [112]: s.sample()
Out[112]:
4    4
dtype: int64

# One may specify either a number of rows:
In [113]: s.sample(n=3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[113]:
0    0
4    4
1    1
dtype: int64

# Or a fraction of the rows:
In [114]: s.sample(frac=0.5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[114]:
5    5
3    3
1    1
dtype: int64
```

By default, `sample` will return each row at most once, but one can also sample with replacement using the `replace` option:

```
In [115]: s = pd.Series([0, 1, 2, 3, 4, 5])

# Without replacement (default):
In [116]: s.sample(n=6, replace=False)
Out[116]:
0    0
1    1
5    5
3    3
2    2
4    4
dtype: int64

# With replacement:
In [117]: s.sample(n=6, replace=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[117]:
0    0
4    4
3    3
2    2
4    4
4    4
dtype: int64
```

By default, each row has an equal probability of being selected, but if you want rows to have different probabilities, you can pass the `sample` function sampling weights as `weights`. These weights can be a list, a NumPy array, or a Series, but they must be of the same length as the object you are sampling. Missing values will be treated as a weight of zero, and inf values are not allowed. If weights do not sum to 1, they will be re-normalized by dividing all weights by the sum of the weights. For example:

```
In [118]: s = pd.Series([0, 1, 2, 3, 4, 5])

In [119]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [120]: s.sample(n=3, weights=example_weights)
Out[120]:
5    5
4    4
3    3
dtype: int64

# Weights will be re-normalized automatically
In [121]: example_weights2 = [0.5, 0, 0, 0, 0, 0]

In [122]: s.sample(n=1, weights=example_weights2)
Out[122]:
0    0
dtype: int64
```

When applied to a DataFrame, you can use a column of the DataFrame as sampling weights (provided you are sampling rows and not columns) by simply passing the name of the column as a string.

```
In [123]: df2 = pd.DataFrame({'col1': [9, 8, 7, 6],
   .....:                     'weight_column': [0.5, 0.4, 0.1, 0]})
   .....:

In [124]: df2.sample(n=3, weights='weight_column')
```

(continues on next page)

(continued from previous page)

```
Out[124]:
   col1  weight_column
1     8            0.4
0     9            0.5
2     7            0.1
```

sample also allows users to sample columns instead of rows using the axis argument.

```
In [125]: df3 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

In [126]: df3.sample(n=1, axis=1)
Out[126]:
   col1
0     1
1     2
2     3
```

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

```
In [127]: df4 = pd.DataFrame({'col1': [1, 2, 3], 'col2': [2, 3, 4]})

# With a given seed, the sample will always draw the same rows.
In [128]: df4.sample(n=2, random_state=2)
Out[128]:
   col1  col2
2     3     4
1     2     3

In [129]: df4.sample(n=2, random_state=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[129]:
   col1  col2
2     3     4
1     2     3
```

4.2.11 Setting With Enlargement

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

In the Series case this is effectively an appending operation.

```
In [130]: se = pd.Series([1, 2, 3])

In [131]: se
Out[131]:
0    1
1    2
2    3
dtype: int64

In [132]: se[5] = 5.

In [133]: se
Out[133]:
0    1.0
```

(continues on next page)

(continued from previous page)

```
1    2.0
2    3.0
5    5.0
dtype: float64
```

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

```
In [134]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
   .....:                   columns=['A', 'B'])
   .....:

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

In [136]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [137]: dfi
Out[137]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an append operation on the DataFrame.

```
In [138]: dfi.loc[3] = 5

In [139]: dfi
Out[139]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```

4.2.12 Fast scalar value getting and setting

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

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

```
In [140]: s.iat[5]
Out[140]: 5

In [141]: df.at[dates[5], 'A']
\\\\\\\\\\\\Out[141]: -0.6736897080883706
```

(continues on next page)

(continued from previous page)

```
In [142]: df.iat[3, 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[142]: 0.7215551622443669
```

You can also set using these same indexers.

```
In [143]: df.at[dates[5], 'E'] = 7

In [144]: df.iat[3, 0] = 7
```

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

```
In [145]: df.at[dates[-1] + 1, 0] = 7

In [146]: df
Out[146]:
                   A         B         C         D    E    0
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632  NaN  NaN
2000-01-02  1.212112 -0.173215  0.119209 -1.044236  NaN  NaN
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804  NaN  NaN
2000-01-04  7.000000 -0.706771 -1.039575  0.271860  NaN  NaN
2000-01-05 -0.424972  0.567020  0.276232 -1.087401  NaN  NaN
2000-01-06 -0.673690  0.113648 -1.478427  0.524988  7.0  NaN
2000-01-07  0.404705  0.577046 -1.715002 -1.039268  NaN  NaN
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885  NaN  NaN
2000-01-09       NaN       NaN       NaN       NaN  NaN  7.0
```

4.2.13 Boolean indexing

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

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

```
In [147]: s = pd.Series(range(-3, 4))

In [148]: s
Out[148]:
0   -3
1   -2
2   -1
3    0
4    1
5    2
6    3
dtype: int64

In [149]: s[s > 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[149]:
4    1
5    2
6    3
dtype: int64
```

(continues on next page)

(continued from previous page)

```
In [150]: s[(s < -1) | (s > 0.5)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -3
1   -2
4    1
5    2
6    3
dtype: int64

In [151]: s[~(s < 0)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
3    0
4    1
5    2
6    3
dtype: int64
```

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

```
In [152]: df[df['A'] > 0]
Out[152]:
                   A         B         C         D   E   0
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632 NaN NaN
2000-01-02  1.212112 -0.173215  0.119209 -1.044236 NaN NaN
2000-01-04  7.000000 -0.706771 -1.039575  0.271860 NaN NaN
2000-01-07  0.404705  0.577046 -1.715002 -1.039268 NaN NaN
```

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

```
In [153]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'three', 'two', 'one', 'six
↪'],
   .....:                    'b': ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
   .....:                    'c': np.random.randn(7)})
   .....:

# only want 'two' or 'three'
In [154]: criterion = df2['a'].map(lambda x: x.startswith('t'))

In [155]: df2[criterion]
Out[155]:
       a  b         c
2    two  y  0.041290
3  three  x  0.361719
4    two  y -0.238075

# equivalent but slower
In [156]: df2[[x.startswith('t') for x in df2['a']]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪
       a  b         c
2    two  y  0.041290
3  three  x  0.361719
4    two  y -0.238075
```

(continues on next page)

(continued from previous page)

```
# Multiple criteria
In [157]: df2[criterion & (df2['b'] == 'x')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a  b         c
3  three  x  0.361719
```

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

```
In [158]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
Out[158]:
   b         c
3  x  0.361719
```

4.2.14 Indexing with isin

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

```
In [159]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')

In [160]: s
Out[160]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [161]: s.isin([2, 4, 6])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[161]:
4    False
3    False
2     True
1    False
0     True
dtype: bool

In [162]: s[s.isin([2, 4, 6])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2    2
0    4
dtype: int64
```

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

```
In [163]: s[s.index.isin([2, 4, 6])]
Out[163]:
4    0
2    2
```

(continues on next page)

(continued from previous page)

```
dtype: int64

# compare it to the following
In [164]: s.reindex([2, 4, 6])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[164]:
2    2.0
4    0.0
6    NaN
dtype: float64
```

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

```
In [165]: s_mi = pd.Series(np.arange(6),
   .....:                  index=pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c
↪']]))
   .....:

In [166]: s_mi
Out[166]:
0  a    0
   b    1
   c    2
1  a    3
   b    4
   c    5
dtype: int64

In [167]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[167]:
↪
0  c    2
1  a    3
dtype: int64

In [168]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0  a    0
   c    2
1  a    3
   c    5
dtype: int64
```

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

```
In [169]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
   .....:                    'ids2': ['a', 'n', 'c', 'n']})
   .....:

In [170]: values = ['a', 'b', 1, 3]

In [171]: df.isin(values)
Out[171]:
    vals    ids   ids2
```

(continues on next page)

(continued from previous page)

```
0   True   True   True
1  False   True  False
2   True  False  False
3  False  False  False
```

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

```
In [172]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [173]: df.isin(values)
Out[173]:
    vals    ids   ids2
0   True   True  False
1  False   True  False
2   True  False  False
3  False  False  False
```

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

```
In [174]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}

In [175]: row_mask = df.isin(values).all(1)

In [176]: df[row_mask]
Out[176]:
   vals ids ids2
0     1   a    a
```

4.2.15 The `where()` Method and Masking

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

To return only the selected rows:

```
In [177]: s[s > 0]
Out[177]:
3    1
2    2
1    3
0    4
dtype: int64
```

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

```
In [178]: s.where(s > 0)
Out[178]:
4    NaN
3    1.0
2    2.0
1    3.0
0    4.0
dtype: float64
```

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

```
In [179]: df[df < 0]
Out[179]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525       NaN       NaN
2000-01-02 -0.352480       NaN -1.192319       NaN
2000-01-03 -0.864883       NaN -0.227870       NaN
2000-01-04       NaN -1.222082       NaN -1.233203
2000-01-05       NaN -0.605656 -1.169184       NaN
2000-01-06       NaN -0.948458       NaN -0.684718
2000-01-07 -2.670153 -0.114722       NaN -0.048048
2000-01-08       NaN       NaN -0.048788 -0.808838
```

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

```
In [180]: df.where(df < 0, -df)
Out[180]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525 -0.485855 -0.245166
2000-01-02 -0.352480 -0.390389 -1.192319 -1.655824
2000-01-03 -0.864883 -0.299674 -0.227870 -0.281059
2000-01-04 -0.846958 -1.222082 -0.600705 -1.233203
2000-01-05 -0.669692 -0.605656 -1.169184 -0.342416
2000-01-06 -0.868584 -0.948458 -2.297780 -0.684718
2000-01-07 -2.670153 -0.114722 -0.168904 -0.048048
2000-01-08 -0.801196 -1.392071 -0.048788 -0.808838
```

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

```
In [181]: s2 = s.copy()

In [182]: s2[s2 < 0] = 0

In [183]: s2
Out[183]:
4    0
3    1
2    2
1    3
0    4
dtype: int64

In [184]: df2 = df.copy()

In [185]: df2[df2 < 0] = 0

In [186]: df2
Out[186]:
                   A         B         C         D
2000-01-01  0.000000  0.000000  0.485855  0.245166
2000-01-02  0.000000  0.390389  0.000000  1.655824
2000-01-03  0.000000  0.299674  0.000000  0.281059
2000-01-04  0.846958  0.000000  0.600705  0.000000
2000-01-05  0.669692  0.000000  0.000000  0.342416
2000-01-06  0.868584  0.000000  2.297780  0.000000
```

(continues on next page)

(continued from previous page)

```
2000-01-07  0.000000  0.000000  0.168904  0.000000
2000-01-08  0.801196  1.392071  0.000000  0.000000
```

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

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

In [188]: df_orig.where(df > 0, -df, inplace=True)

In [189]: df_orig
Out[189]:
                   A         B         C         D
2000-01-01  2.104139  1.309525  0.485855  0.245166
2000-01-02  0.352480  0.390389  1.192319  1.655824
2000-01-03  0.864883  0.299674  0.227870  0.281059
2000-01-04  0.846958  1.222082  0.600705  1.233203
2000-01-05  0.669692  0.605656  1.169184  0.342416
2000-01-06  0.868584  0.948458  2.297780  0.684718
2000-01-07  2.670153  0.114722  0.168904  0.048048
2000-01-08  0.801196  1.392071  0.048788  0.808838
```

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

```
In [190]: df.where(df < 0, -df) == np.where(df < 0, df, -df)
Out[190]:
               A     B     C     D
2000-01-01  True  True  True  True
2000-01-02  True  True  True  True
2000-01-03  True  True  True  True
2000-01-04  True  True  True  True
2000-01-05  True  True  True  True
2000-01-06  True  True  True  True
2000-01-07  True  True  True  True
2000-01-08  True  True  True  True
```

alignment

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

```
In [191]: df2 = df.copy()

In [192]: df2[df2[1:4] > 0] = 3

In [193]: df2
Out[193]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525  0.485855  0.245166
2000-01-02 -0.352480  3.000000 -1.192319  3.000000
2000-01-03 -0.864883  3.000000 -0.227870  3.000000
2000-01-04  3.000000 -1.222082  3.000000 -1.233203
2000-01-05  0.669692 -0.605656 -1.169184  0.342416
```

(continues on next page)

(continued from previous page)

```
2000-01-06  0.868584 -0.948458  2.297780 -0.684718
2000-01-07 -2.670153 -0.114722  0.168904 -0.048048
2000-01-08  0.801196  1.392071 -0.048788 -0.808838
```

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

```
In [194]: df2 = df.copy()

In [195]: df2.where(df2 > 0, df2['A'], axis='index')
Out[195]:
                   A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02 -0.352480  0.390389 -0.352480  1.655824
2000-01-03 -0.864883  0.299674 -0.864883  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196
```

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

```
In [196]: df2 = df.copy()

In [197]: df.apply(lambda x, y: x.where(x > 0, y), y=df['A'])
Out[197]:
                   A         B         C         D
2000-01-01 -2.104139 -2.104139  0.485855  0.245166
2000-01-02 -0.352480  0.390389 -0.352480  1.655824
2000-01-03 -0.864883  0.299674 -0.864883  0.281059
2000-01-04  0.846958  0.846958  0.600705  0.846958
2000-01-05  0.669692  0.669692  0.669692  0.342416
2000-01-06  0.868584  0.868584  2.297780  0.868584
2000-01-07 -2.670153 -2.670153  0.168904 -2.670153
2000-01-08  0.801196  1.392071  0.801196  0.801196
```

New in version 0.18.1.

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

```
In [198]: df3 = pd.DataFrame({'A': [1, 2, 3],
   .....:                     'B': [4, 5, 6],
   .....:                     'C': [7, 8, 9]})
   .....:

In [199]: df3.where(lambda x: x > 4, lambda x: x + 10)
Out[199]:
    A   B  C
0  11  14  7
1  12   5  8
2  13   6  9
```

Mask

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

```
In [200]: s.mask(s >= 0)
Out[200]:
4   NaN
3   NaN
2   NaN
1   NaN
0   NaN
dtype: float64

In [201]: df.mask(df >= 0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[201]:
                   A         B         C         D
2000-01-01 -2.104139 -1.309525       NaN       NaN
2000-01-02 -0.352480       NaN -1.192319       NaN
2000-01-03 -0.864883       NaN -0.227870       NaN
2000-01-04       NaN -1.222082       NaN -1.233203
2000-01-05       NaN -0.605656 -1.169184       NaN
2000-01-06       NaN -0.948458       NaN -0.684718
2000-01-07 -2.670153 -0.114722       NaN -0.048048
2000-01-08       NaN       NaN -0.048788 -0.808838
```

4.2.16 The `query()` Method

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

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

```
In [202]: n = 10

In [203]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [204]: df
Out[204]:
          a         b         c
0  0.438921  0.118680  0.863670
1  0.138138  0.577363  0.686602
2  0.595307  0.564592  0.520630
3  0.913052  0.926075  0.616184
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
6  0.792342  0.216974  0.564056
7  0.397890  0.454131  0.915716
8  0.074315  0.437913  0.019794
9  0.559209  0.502065  0.026437

# pure python
In [205]: df[(df.a < df.b) & (df.b < df.c)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b         c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716

# query
```

(continues on next page)

(continued from previous page)

```
In [206]: df.query('(a < b) & (b < c)')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b         c
1  0.138138  0.577363  0.686602
4  0.078718  0.854477  0.898725
5  0.076404  0.523211  0.591538
7  0.397890  0.454131  0.915716
```

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

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

In [208]: df.index.name = 'a'

In [209]: df
Out[209]:
   b  c
a
0  0  4
1  0  1
2  3  4
3  4  3
4  1  4
5  0  3
6  0  1
7  3  4
8  2  3
9  1  1

In [210]: df.query('a < b and b < c')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   b  c
a
2  3  4
```

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

```
In [211]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)), columns=list('bc'))

In [212]: df
Out[212]:
   b  c
0  3  1
1  3  0
2  5  6
3  5  2
4  7  4
5  0  1
6  2  5
7  0  1
8  6  0
9  7  9

In [213]: df.query('index < b < c')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪
```

(continues on next page)

(continued from previous page)

```
   b  c
2  5  6
```

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

```
In [214]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})

In [215]: df.index.name = 'a'

In [216]: df.query('a > 2')  # uses the column 'a', not the index
Out[216]:
   a
a
1  3
3  3
```

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

```
In [217]: df.query('index > 2')
Out[217]:
   a
a
3  3
4  2
```

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

MultiIndex query() Syntax

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

```
In [218]: n = 10

In [219]: colors = np.random.choice(['red', 'green'], size=n)

In [220]: foods = np.random.choice(['eggs', 'ham'], size=n)

In [221]: colors
Out[221]:
array(['red', 'red', 'red', 'green', 'green', 'green', 'green', 'green',
       'green', 'green'],
      dtype='<U5')

In [222]: foods
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
array(['ham', 'ham', 'eggs', 'eggs', 'eggs', 'ham', 'ham', 'eggs', 'eggs',
       'eggs'],
      dtype='<U4')

In [223]: index = pd.MultiIndex.from_arrays([colors, foods], names=['color', 'food'])
```

(continues on next page)

(continued from previous page)

```
In [224]: df = pd.DataFrame(np.random.randn(n, 2), index=index)

In [225]: df
Out[225]:
                   0         1
color food
red   ham   0.194889 -0.381994
      ham   0.318587  2.089075
      eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
      eggs -2.029766  0.792652
      ham   0.461007 -0.542749
      ham  -0.305384 -0.479195
      eggs  0.095031 -0.270099
      eggs -0.707140 -0.773882
      eggs  0.229453  0.304418

In [226]: df.query('color == "red"')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   0         1
color food
red   ham   0.194889 -0.381994
      ham   0.318587  2.089075
      eggs -0.728293 -0.090255
```

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

```
In [227]: df.index.names = [None, None]

In [228]: df
Out[228]:
                   0         1
red   ham   0.194889 -0.381994
      ham   0.318587  2.089075
      eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
      eggs -2.029766  0.792652
      ham   0.461007 -0.542749
      ham  -0.305384 -0.479195
      eggs  0.095031 -0.270099
      eggs -0.707140 -0.773882
      eggs  0.229453  0.304418

In [229]: df.query('ilevel_0 == "red"')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                 0         1
red ham   0.194889 -0.381994
    ham   0.318587  2.089075
    eggs -0.728293 -0.090255
```

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

query() Use Cases

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

```
In [230]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [231]: df
Out[231]:
          a         b         c
0  0.224283  0.736107  0.139168
1  0.302827  0.657803  0.713897
2  0.611185  0.136624  0.984960
3  0.195246  0.123436  0.627712
4  0.618673  0.371660  0.047902
5  0.480088  0.062993  0.185760
6  0.568018  0.483467  0.445289
7  0.309040  0.274580  0.587101
8  0.258993  0.477769  0.370255
9  0.550459  0.840870  0.304611

In [232]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)

In [233]: df2
Out[233]:
           a         b         c
0   0.357579  0.229800  0.596001
1   0.309059  0.957923  0.965663
2   0.123102  0.336914  0.318616
3   0.526506  0.323321  0.860813
4   0.518736  0.486514  0.384724
5   0.190804  0.505723  0.614533
6   0.891939  0.623977  0.676639
7   0.480559  0.378528  0.460858
8   0.420223  0.136404  0.141295
9   0.732206  0.419540  0.604675
10  0.604466  0.848974  0.896165
11  0.589168  0.920046  0.732716

In [234]: expr = '0.0 <= a <= c <= 0.5'

In [235]: map(lambda frame: frame.query(expr), [df, df2])
Out[235]: <map at 0x7f380eebcc88>
```

query() Python versus pandas Syntax Comparison

Full numpy-like syntax:

```
In [236]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=list('abc'))

In [237]: df
Out[237]:
   a  b  c
0  7  8  9
1  1  0  7
```

(continues on next page)

(continued from previous page)

```
2  2  7  2
3  6  2  2
4  2  6  3
5  3  8  2
6  1  7  2
7  5  1  5
8  9  8  0
9  1  5  0

In [238]: df.query('(a < b) & (b < c)')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b  c
0  7  8  9

In [239]: df[(df.a < df.b) & (df.b < df.c)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b  c
0  7  8  9
```

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

```
In [240]: df.query('a < b & b < c')
Out[240]:
   a  b  c
0  7  8  9
```

Use English instead of symbols:

```
In [241]: df.query('a < b and b < c')
Out[241]:
   a  b  c
0  7  8  9
```

Pretty close to how you might write it on paper:

```
In [242]: df.query('a < b < c')
Out[242]:
   a  b  c
0  7  8  9
```

The `in` and `not in` operators

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

```
# get all rows where columns "a" and "b" have overlapping values
In [243]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbbcccc'),
   .....:                    'c': np.random.randint(5, size=12),
   .....:                    'd': np.random.randint(9, size=12)})
   .....:

In [244]: df
Out[244]:
```

(continues on next page)

(continued from previous page)

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

In [245]: df.query('a in b')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2

# How you'd do it in pure Python
In [246]: df[df.a.isin(df.b)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
3  b  a  2  1
4  c  b  3  6
5  c  b  0  2

In [247]: df.query('a not in b')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

# pure Python
In [248]: df[~df.a.isin(df.b)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
```

(continues on next page)

(continued from previous page)

```
10  f  c  0  6
11  f  c  1  2
```

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

```
# rows where cols a and b have overlapping values
# and col c's values are less than col d's
In [249]: df.query('a in b and c < d')
Out[249]:
   a  b  c  d
0  a  a  2  6
1  a  a  4  7
2  b  a  1  6
4  c  b  3  6
5  c  b  0  2

# pure Python
In [250]: df[df.b.isin(df.a) & (df.c < df.d)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25
↪
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
4   c  b  3  6
5   c  b  0  2
10  f  c  0  6
11  f  c  1  2
```

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

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

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

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

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

```
In [251]: df.query('b == ["a", "b", "c"]')
Out[251]:
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
3   b  a  2  1
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
```

(continues on next page)

(continued from previous page)

```
9    e  c  2  0
10   f  c  0  6
11   f  c  1  2

# pure Python
In [252]: df[df.b.isin(["a", "b", "c"])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
0   a  a  2  6
1   a  a  4  7
2   b  a  1  6
3   b  a  2  1
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
7   d  b  2  1
8   e  c  4  3
9   e  c  2  0
10  f  c  0  6
11  f  c  1  2

In [253]: df.query('c == [1, 2]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2

In [254]: df.query('c != [1, 2]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
1   a  a  4  7
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
8   e  c  4  3
10  f  c  0  6

# using in/not in
In [255]: df.query('[1, 2] in c')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2

In [256]: df.query('[1, 2] not in c')
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
1   a  a  4  7
4   c  b  3  6
5   c  b  0  2
6   d  b  3  3
8   e  c  4  3
10  f  c  0  6

# pure Python
In [257]: df[df.c.isin([1, 2])]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    a  b  c  d
0   a  a  2  6
2   b  a  1  6
3   b  a  2  1
7   d  b  2  1
9   e  c  2  0
11  f  c  1  2
```

Boolean Operators

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

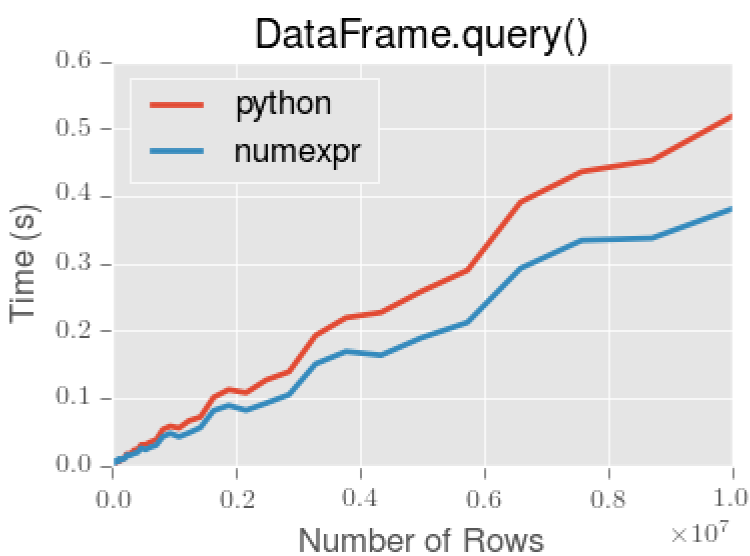
```
In [258]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [259]: df['bools'] = np.random.rand(len(df)) > 0.5

In [260]: df.query('~bools')
Out[260]:
          a         b         c  bools
2  0.697753  0.212799  0.329209  False
7  0.275396  0.691034  0.826619  False
8  0.190649  0.558748  0.262467  False

In [261]: df.query('not bools')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b         c  bools
2  0.697753  0.212799  0.329209  False
7  0.275396  0.691034  0.826619  False
8  0.190649  0.558748  0.262467  False

In [262]: df.query('not bools') == df[~df.bools]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      a     b     c  bools
2  True  True  True   True
7  True  True  True   True
8  True  True  True   True
```

Of course, expressions can be arbitrarily complex too:

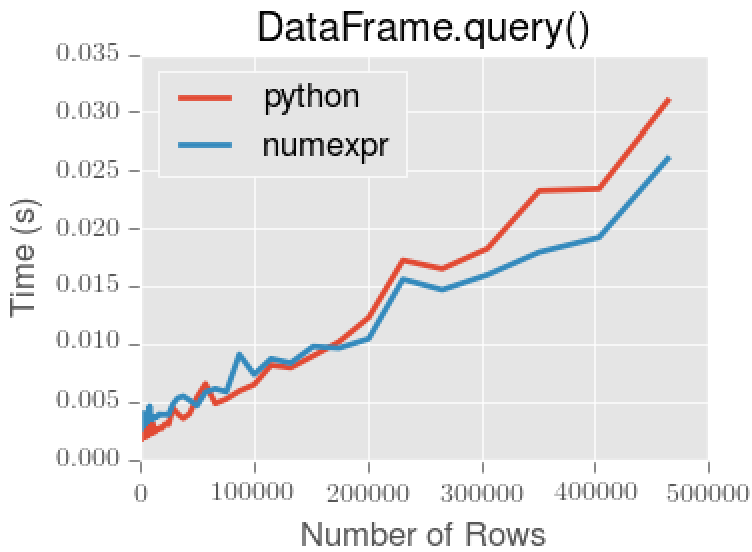
```
# short query syntax
In [263]: shorter = df.query('a < b < c and (not bools) or bools > 2')

# equivalent in pure Python
In [264]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]

In [265]: shorter
Out[265]:
          a         b         c  bools
7  0.275396  0.691034  0.826619  False

In [266]: longer
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[266]:␣
↪
          a         b         c  bools
7  0.275396  0.691034  0.826619  False

In [267]: shorter == longer
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      a     b     c  bools
7  True  True  True   True
```

Performance of `query()`

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



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



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

4.2.17 Duplicate Data

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

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

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

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

```
In [268]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two', 'three', 'four
↪'],
   .....:                     'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
   .....:                     'c': np.random.randn(7)})
   .....:

In [269]: df2
Out[269]:
       a  b         c
0    one  x -1.067137
1    one  y  0.309500
2    two  x -0.211056
3    two  y -1.842023
4    two  x -0.390820
```

(continues on next page)

(continued from previous page)

```
5  three  x -1.964475
6   four  x  1.298329

In [270]: df2.duplicated('a')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    False
1     True
2    False
3     True
4     True
5    False
6    False
dtype: bool

In [271]: df2.duplicated('a', keep='last')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     True
1    False
2     True
3     True
4    False
5    False
6    False
dtype: bool

In [272]: df2.duplicated('a', keep=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     True
1     True
2     True
3     True
4     True
5    False
6    False
dtype: bool

In [273]: df2.drop_duplicates('a')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a  b         c
0    one  x -1.067137
2    two  x -0.211056
5  three  x -1.964475
6   four  x  1.298329

In [274]: df2.drop_duplicates('a', keep='last')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a  b         c
1    one  y  0.309500
4    two  x -0.390820
5  three  x -1.964475
6   four  x  1.298329
```

(continues on next page)

(continued from previous page)

```
In [275]: df2.drop_duplicates('a', keep=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a  b         c
5  three  x -1.964475
6   four  x  1.298329
```

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

```
In [276]: df2.duplicated(['a', 'b'])
Out[276]:
0    False
1    False
2    False
3    False
4     True
5    False
6    False
dtype: bool

In [277]: df2.drop_duplicates(['a', 'b'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       a  b         c
0    one  x -1.067137
1    one  y  0.309500
2    two  x -0.211056
3    two  y -1.842023
5  three  x -1.964475
6   four  x  1.298329
```

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

```
In [278]: df3 = pd.DataFrame({'a': np.arange(6),
   .....:                     'b': np.random.randn(6)},
   .....:                    index=['a', 'a', 'b', 'c', 'b', 'a'])
   .....:

In [279]: df3
Out[279]:
   a         b
a  0  1.440455
a  1  2.456086
b  2  1.038402
c  3 -0.894409
b  4  0.683536
a  5  3.082764

In [280]: df3.index.duplicated()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array([False,  True, False, False,  True,  True], dtype=bool)

In [281]: df3[~df3.index.duplicated()]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
   a         b
a  0  1.440455
b  2  1.038402
c  3 -0.894409

In [282]: df3[~df3.index.duplicated(keep='last')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a         b
c  3 -0.894409
b  4  0.683536
a  5  3.082764

In [283]: df3[~df3.index.duplicated(keep=False)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a         b
c  3 -0.894409
```

4.2.18 Dictionary-like `get()` method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.

```
In [284]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [285]: s.get('a')  # equivalent to s['a']
Out[285]: 1

In [286]: s.get('x', default=-1)
\\\\\\\\\\\\Out[286]: -1
```

4.2.19 The `lookup()` Method

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

```
In [287]: dflookup = pd.DataFrame(np.random.rand(20, 4), columns = ['A', 'B', 'C', 'D
↪'])

In [288]: dflookup.lookup(list(range(0, 10, 2)), ['B', 'C', 'A', 'B', 'D'])
Out[288]: array([ 0.3506,  0.4779,  0.4825,  0.9197,  0.5019])
```

4.2.20 Index objects

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

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

```
In [289]: index = pd.Index(['e', 'd', 'a', 'b'])

In [290]: index
Out[290]: Index(['e', 'd', 'a', 'b'], dtype='object')

In [291]: 'd' in index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[291]: True
```

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

```
In [292]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')

In [293]: index.name
Out[293]: 'something'
```

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

```
In [294]: index = pd.Index(list(range(5)), name='rows')

In [295]: columns = pd.Index(['A', 'B', 'C'], name='cols')

In [296]: df = pd.DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [297]: df
Out[297]: 
cols          A         B         C
rows                               
0      1.295989  0.185778  0.436259
1      0.678101  0.311369 -0.528378
2     -0.674808 -1.103529 -0.656157
3      1.889957  2.076651 -1.102192
4     -1.211795 -0.791746  0.634724

In [298]: df['A']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
rows
0    1.295989
1    0.678101
2   -0.674808
3    1.889957
4   -1.211795
Name: A, dtype: float64
```

Setting metadata

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

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

See *Advanced Indexing* for usage of MultiIndexes.

```
In [299]: ind = pd.Index([1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
In [300]: ind.rename("apple")
Out[300]: Int64Index([1, 2, 3], dtype='int64', name='apple')

In [301]: ind
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[301]: Int64Index([1,␣
↪2, 3], dtype='int64')

In [302]: ind.set_names(["apple"], inplace=True)

In [303]: ind.name = "bob"

In [304]: ind
Out[304]: Int64Index([1, 2, 3], dtype='int64', name='bob')
```

`set_names`, `set_levels`, and `set_codes` also take an optional *level*‘ argument

```
In [305]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first
↪', 'second'])

In [306]: index
Out[306]:
MultiIndex(levels=[[0, 1, 2], ['one', 'two']],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [307]: index.levels[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['one', 'two'], dtype='object', name='second')

In [308]: index.set_levels(["a", "b"], level=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[[0, 1, 2], ['a', 'b']],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])
```

Set operations on Index objects

The two main operations are `union (|)` and `intersection (&)`. These can be directly called as instance methods or used via overloaded operators. Difference is provided via the `.difference()` method.

```
In [309]: a = pd.Index(['c', 'b', 'a'])

In [310]: b = pd.Index(['c', 'e', 'd'])

In [311]: a | b
Out[311]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [312]: a & b
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[312]: Index(['c'],␣
↪dtype='object')

In [313]: a.difference(b)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪Index(['a', 'b'], dtype='object')
```

Also available is the `symmetric_difference (^)` operation, which returns elements that appear in either `idx1` or `idx2`, but not in both. This is equivalent to the Index created by `idx1.difference(idx2).union(idx2.difference(idx1))`, with duplicates dropped.

```
In [314]: idx1 = pd.Index([1, 2, 3, 4])

In [315]: idx2 = pd.Index([2, 3, 4, 5])

In [316]: idx1.symmetric_difference(idx2)
Out[316]: Int64Index([1, 5], dtype='int64')

In [317]: idx1 ^ idx2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[317]: Int64Index([1, 5], dtype='int64
↪')
```

Note: The resulting index from a set operation will be sorted in ascending order.

Missing values

Important: Even though `Index` can hold missing values (`NaN`), it should be avoided if you do not want any unexpected results. For example, some operations exclude missing values implicitly.

`Index.fillna` fills missing values with specified scalar value.

```
In [318]: idx1 = pd.Index([1, np.nan, 3, 4])

In [319]: idx1
Out[319]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')

In [320]: idx1.fillna(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[320]:␣
↪Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')

In [321]: idx2 = pd.DatetimeIndex([pd.Timestamp('2011-01-01'),
   .....:                          pd.NaT,
   .....:                          pd.Timestamp('2011-01-03')])
   .....:

In [322]: idx2
Out[322]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'], dtype='datetime64[ns]',␣
↪freq=None)

In [323]: idx2.fillna(pd.Timestamp('2011-01-02'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3
↪DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], dtype='datetime64[ns]',␣
↪freq=None)
```

4.2.21 Set / Reset Index

Occasionally you will load or create a data set into a DataFrame and want to add an index after you've already done so. There are a couple of different ways.

Set an index

DataFrame has a `set_index()` method which takes a column name (for a regular `Index`) or a list of column names (for a `MultiIndex`). To create a new, re-indexed DataFrame:

```
In [324]: data
Out[324]:
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0

In [325]: indexed1 = data.set_index('c')

In [326]: indexed1
Out[326]:
     a    b    d
c
z  bar  one  1.0
y  bar  two  2.0
x  foo  one  3.0
w  foo  two  4.0

In [327]: indexed2 = data.set_index(['a', 'b'])

In [328]: indexed2
Out[328]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [329]: frame = data.set_index('c', drop=False)

In [330]: frame = frame.set_index(['a', 'b'], append=True)

In [331]: frame
Out[331]:
           c    d
c a   b
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0
```

Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):

```
In [332]: data.set_index('c', drop=False)
Out[332]:
     a    b  c    d
c
z  bar  one  z  1.0
```

(continues on next page)

(continued from previous page)

```
y  bar  two  y  2.0
x  foo  one  x  3.0
w  foo  two  w  4.0

In [333]: data.set_index(['a', 'b'], inplace=True)

In [334]: data
Out[334]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0
```

Reset the index

As a convenience, there is a new function on DataFrame called *reset_index()* which transfers the index values into the DataFrame's columns and sets a simple integer index. This is the inverse operation of *set_index()*.

```
In [335]: data
Out[335]:
         c    d
a   b
bar one  z  1.0
    two  y  2.0
foo one  x  3.0
    two  w  4.0

In [336]: data.reset_index()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     a    b  c    d
0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0
```

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the `names` attribute.

You can use the `level` keyword to remove only a portion of the index:

```
In [337]: frame
Out[337]:
           c    d
c a   b
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0

In [338]: frame.reset_index(level=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
         a  c    d
c b
z one  bar  z  1.0
y two  bar  y  2.0
x one  foo  x  3.0
w two  foo  w  4.0
```

`reset_index` takes an optional parameter `drop` which if true simply discards the index, instead of putting index values in the DataFrame's columns.

Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
data.index = index
```

4.2.22 Returning a view versus a copy

When setting values in a pandas object, care must be taken to avoid what is called `chained indexing`. Here is an example.

```
In [339]: dfmi = pd.DataFrame([list('abcd'),
   .....:                       list('efgh'),
   .....:                       list('ijkl'),
   .....:                       list('mnop')],
   .....:                      columns=pd.MultiIndex.from_product([['one', 'two'],
   .....:                                                          ['first', 'second
↪']]))
   .....:

In [340]: dfmi
Out[340]: 
    one          two       
  first second first second
0     a      b     c      d
1     e      f     g      h
2     i      j     k      l
3     m      n     o      p
```

Compare these two access methods:

```
In [341]: dfmi['one']['second']
Out[341]: 
0    b
1    f
2    j
3    n
Name: second, dtype: object
```

```
In [342]: dfmi.loc[:, ('one', 'second')]
Out[342]: 
0    b
1    f
```

(continues on next page)

(continued from previous page)

```
2    j
3    n
Name: (one, second), dtype: object
```

These both yield the same results, so which should you use? It is instructive to understand the order of operations on these and why method 2 (`.loc`) is much preferred over method 1 (chained `[]`).

`dfmi['one']` selects the first level of the columns and returns a DataFrame that is singly-indexed. Then another Python operation `dfmi_with_one['second']` selects the series indexed by `'second'`. This is indicated by the variable `dfmi_with_one` because pandas sees these operations as separate events. e.g. separate calls to `__getitem__`, so it has to treat them as linear operations, they happen one after another.

Contrast this to `df.loc[:,('one','second')]` which passes a nested tuple of `(slice(None),('one','second'))` to a single call to `__getitem__`. This allows pandas to deal with this as a single entity. Furthermore this order of operations *can* be significantly faster, and allows one to index *both* axes if so desired.

Why does assignment fail when using chained indexing?

The problem in the previous section is just a performance issue. What's up with the `SettingWithCopy` warning? We don't **usually** throw warnings around when you do something that might cost a few extra milliseconds!

But it turns out that assigning to the product of chained indexing has inherently unpredictable results. To see this, think about how the Python interpreter executes this code:

```
dfmi.loc[:, ('one', 'second')] = value
# becomes
dfmi.loc.__setitem__((slice(None), ('one', 'second')), value)
```

But this code is handled differently:

```
dfmi['one']['second'] = value
# becomes
dfmi.__getitem__('one').__setitem__('second', value)
```

See that `__getitem__` in there? Outside of simple cases, it's very hard to predict whether it will return a view or a copy (it depends on the memory layout of the array, about which pandas makes no guarantees), and therefore whether the `__setitem__` will modify `dfmi` or a temporary object that gets thrown out immediately afterward. **That's** what `SettingWithCopy` is warning you about!

Note: You may be wondering whether we should be concerned about the `loc` property in the first example. But `dfmi.loc` is guaranteed to be `dfmi` itself with modified indexing behavior, so `dfmi.loc.__getitem__` / `dfmi.loc.__setitem__` operate on `dfmi` directly. Of course, `dfmi.loc.__getitem__(idx)` may be a view or a copy of `dfmi`.

Sometimes a `SettingWithCopy` warning will arise at times when there's no obvious chained indexing going on. **These** are the bugs that `SettingWithCopy` is designed to catch! Pandas is probably trying to warn you that you've done this:

```
def do_something(df):
    foo = df[['bar', 'baz']]  # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...
    # We don't know whether this will modify df or not!
    foo['quux'] = value
    return foo
```

Yikes!

Evaluation order matters

When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.

Pandas has the `SettingWithCopyWarning` because assigning to a copy of a slice is frequently not intentional, but a mistake caused by chained indexing returning a copy where a slice was expected.

If you would like pandas to be more or less trusting about assignment to a chained indexing expression, you can set the *option* `mode.chained_assignment` to one of these values:

- `'warn'`, the default, means a `SettingWithCopyWarning` is printed.
- `'raise'` means pandas will raise a `SettingWithCopyException` you have to deal with.
- `None` will suppress the warnings entirely.

```
In [343]: dfb = pd.DataFrame({'a': ['one', 'one', 'two',
   .....:                           'three', 'two', 'one', 'six'],
   .....:                     'c': np.arange(7)})
   .....:

# This will show the SettingWithCopyWarning
# but the frame values will be set
In [344]: dfb['c'][dfb.a.str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```
>>> pd.set_option('mode.chained_assignment','warn')
>>> dfb[dfb.a.str.startswith('o')]['c'] = 42
Traceback (most recent call last)
     ...
SettingWithCopyWarning:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

A chained assignment can also crop up in setting in a mixed dtype frame.

Note: These setting rules apply to all of `.loc/.iloc`.

This is the correct access method:

```
In [345]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [346]: dfc.loc[0, 'A'] = 11

In [347]: dfc
Out[347]:
     A  B
0   11  1
1  bbb  2
2  ccc  3
```

This *can* work at times, but it is not guaranteed to, and therefore should be avoided:

```
In [348]: dfc = dfc.copy()

In [349]: dfc['A'][0] = 111

In [350]: dfc
Out[350]:
     A  B
0  111  1
1  bbb  2
2  ccc  3
```

This will **not** work at all, and so should be avoided:

```
>>> pd.set_option('mode.chained_assignment','raise')
>>> dfc.loc[0]['A'] = 1111
Traceback (most recent call last)
     ...
SettingWithCopyException:
     A value is trying to be set on a copy of a slice from a DataFrame.
     Try using .loc[row_index,col_indexer] = value instead
```

> **Warning:** The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.

4.3 MultiIndex / Advanced Indexing

This section covers *indexing with a MultiIndex* and *other advanced indexing features*.

See the *Indexing and Selecting Data* for general indexing documentation.

> **Warning:** Whether a copy or a reference is returned for a setting operation may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.

See the *cookbook* for some advanced strategies.

4.3.1 Hierarchical indexing (MultiIndex)

Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like `Series` (1d) and `DataFrame` (2d).

In this section, we will show what exactly we mean by "hierarchical" indexing and how it integrates with all of the pandas indexing functionality described above and in prior sections. Later, when discussing *group by* and *pivoting and reshaping data*, we'll show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the *cookbook* for some advanced strategies.

Changed in version 0.24.0: `MultiIndex.labels` has been renamed to *`MultiIndex.codes`* and `MultiIndex.set_labels` to *`MultiIndex.set_codes`*.

Creating a MultiIndex (hierarchical index) object

The *MultiIndex* object is the hierarchical analogue of the standard *Index* object which typically stores the axis labels in pandas objects. You can think of MultiIndex as an array of tuples where each tuple is unique. A MultiIndex can be created from a list of arrays (using *MultiIndex.from_arrays()*), an array of tuples (using *MultiIndex.from_tuples()*), a crossed set of iterables (using *MultiIndex.from_product()*), or a *DataFrame* (using *MultiIndex.from_frame()*). The Index constructor will attempt to return a MultiIndex when it is passed a list of tuples. The following examples demonstrate different ways to initialize MultiIndexes.

```
In [1]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ...:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ...:

In [2]: tuples = list(zip(*arrays))

In [3]: tuples
Out[3]:
[('bar', 'one'),
 ('bar', 'two'),
 ('baz', 'one'),
 ('baz', 'two'),
 ('foo', 'one'),
 ('foo', 'two'),
 ('qux', 'one'),
 ('qux', 'two')]

In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [5]: index
Out[5]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           codes=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [6]: s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]:
first  second
bar    one       0.469112
       two      -0.282863
baz    one      -1.509059
       two      -1.135632
foo    one       1.212112
       two      -0.173215
qux    one       0.119209
       two      -1.044236
dtype: float64
```

When you want every pairing of the elements in two iterables, it can be easier to use the *MultiIndex.from_product()* method:

```
In [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])
Out[9]:
```

(continues on next page)

(continued from previous page)

```
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           codes=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])
```

You can also construct a `MultiIndex` from a `DataFrame` directly, using the method *`MultiIndex.from_frame()`*. This is a complementary method to *`MultiIndex.to_frame()`*.

New in version 0.24.0.

```
In [10]: df = pd.DataFrame([['bar', 'one'], ['bar', 'two'],
   ....:                    ['foo', 'one'], ['foo', 'two']],
   ....:                   columns=['first', 'second'])
   ....:

In [11]: pd.MultiIndex.from_frame(df)
Out[11]:
MultiIndex(levels=[['bar', 'foo'], ['one', 'two']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['first', 'second'])
```

As a convenience, you can pass a list of arrays directly into `Series` or `DataFrame` to construct a `MultiIndex` automatically:

```
In [12]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
   ....:           np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
   ....:

In [13]: s = pd.Series(np.random.randn(8), index=arrays)

In [14]: s
Out[14]:
bar  one   -0.861849
     two   -2.104569
baz  one   -0.494929
     two    1.071804
foo  one    0.721555
     two   -0.706771
qux  one   -1.039575
     two    0.271860
dtype: float64

In [15]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)

In [16]: df
Out[16]:
                0         1         2         3
bar one -0.424972  0.567020  0.276232 -1.087401
    two -0.673690  0.113648 -1.478427  0.524988
baz one  0.404705  0.577046 -1.715002 -1.039268
    two -0.370647 -1.157892 -1.344312  0.844885
foo one  1.075770 -0.109050  1.643563 -1.469388
    two  0.357021 -0.674600 -1.776904 -0.968914
qux one -1.294524  0.413738  0.276662 -0.472035
    two -0.013960 -0.362543 -0.006154 -0.923061
```

All of the `MultiIndex` constructors accept a `names` argument which stores string names for the levels themselves. If no names are provided, `None` will be assigned:

```
In [17]: df.index.names
Out[17]: FrozenList([None, None])
```

This index can back any axis of a pandas object, and the number of **levels** of the index is up to you:

```
In [18]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'],␣
↪columns=index)

In [19]: df
Out[19]:
first        bar                 baz                 foo                 qux          
second       one       two       one       two       one       two       one       two
A       0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309 -1.170299 -0.226169
B       0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737
C      -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747

In [20]: pd.DataFrame(np.random.randn(6, 6), index=index[:6], columns=index[:6])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
first              bar                 baz                 foo          
second             one       two       one       two       one       two
first second                                                            
bar   one    -0.410001 -0.078638  0.545952 -1.219217 -1.226825  0.769804
      two    -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734
baz   one     0.959726 -1.110336 -0.619976  0.149748 -0.732339  0.687738
      two     0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849
foo   one    -0.954208  1.462696 -1.743161 -0.826591 -0.345352  1.314232
      two     0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441
```

We've "sparsified" the higher levels of the indexes to make the console output a bit easier on the eyes. Note that how the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_options()`:

```
In [21]: with pd.option_context('display.multi_sparse', False):
   ....:     df
   ....:
```

It's worth keeping in mind that there's nothing preventing you from using tuples as atomic labels on an axis:

```
In [22]: pd.Series(np.random.randn(8), index=tuples)
Out[22]:
(bar, one)   -1.236269
(bar, two)    0.896171
(baz, one)   -0.487602
(baz, two)   -0.082240
(foo, one)   -2.182937
(foo, two)    0.380396
(qux, one)    0.084844
(qux, two)    0.432390
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

Reconstructing the level labels

The method *get_level_values()* will return a vector of the labels for each location at a particular level:

```
In [23]: index.get_level_values(0)
Out[23]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], dtype='object
↪', name='first')

In [24]: index.get_level_values('second')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'], dtype='object',␣
↪name='second')
```

Basic indexing on axis with MultiIndex

One of the important features of hierarchical indexing is that you can select data by a "partial" label identifying a subgroup in the data. **Partial** selection "drops" levels of the hierarchical index in the result in a completely analogous way to selecting a column in a regular DataFrame:

```
In [25]: df['bar']
Out[25]:
second       one       two
A       0.895717  0.805244
B       0.410835  0.813850
C      -1.413681  1.607920

In [26]: df['bar', 'one']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64

In [27]: df['bar']['one']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    0.895717
B    0.410835
C   -1.413681
Name: one, dtype: float64

In [28]: s['qux']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
one   -1.039575
two    0.271860
dtype: float64
```

See *Cross-section with hierarchical index* for how to select on a deeper level.

Defined Levels

The repr of a `MultiIndex` shows all the defined levels of an index, even if they are not actually used. When slicing an index, you may notice this. For example:

```
In [29]: df.columns  # original MultiIndex
Out[29]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           codes=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [30]: df[['foo','qux']].columns  # sliced
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
           codes=[[2, 2, 3, 3], [0, 1, 0, 1]],
           names=['first', 'second'])
```

This is done to avoid a recomputation of the levels in order to make slicing highly performant. If you want to see only the used levels, you can use the *get_level_values()* method.

```
In [31]: df[['foo', 'qux']].columns.to_numpy()
Out[31]: array([('foo', 'one'), ('foo', 'two'), ('qux', 'one'), ('qux', 'two')],␣
↪dtype=object)

# for a specific level
In [32]: df[['foo', 'qux']].columns.get_level_values(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[32
↪Index(['foo', 'foo', 'qux', 'qux'], dtype='object', name='first')
```

To reconstruct the `MultiIndex` with only the used levels, the *remove_unused_levels()* method may be used.

New in version 0.20.0.

```
In [33]: df[['foo', 'qux']].columns.remove_unused_levels()
Out[33]:
MultiIndex(levels=[['foo', 'qux'], ['one', 'two']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['first', 'second'])
```

Data alignment and using `reindex`

Operations between differently-indexed objects having `MultiIndex` on the axes will work as you expect; data alignment will work the same as an Index of tuples:

```
In [34]: s + s[:-2]
Out[34]:
bar  one   -1.723698
     two   -4.209138
baz  one   -0.989859
     two    2.143608
foo  one    1.443110
     two   -1.413542
qux  one         NaN
     two         NaN
dtype: float64

In [35]: s + s[::2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
bar  one   -1.723698
```

(continues on next page)

(continued from previous page)

```
     two         NaN
baz  one   -0.989859
     two         NaN
foo  one    1.443110
     two         NaN
qux  one   -2.079150
     two         NaN
dtype: float64
```

The *reindex()* method of `Series`/`DataFrames` can be called with another `MultiIndex`, or even a list or array of tuples:

```
In [36]: s.reindex(index[:3])
Out[36]:
first  second
bar    one      -0.861849
       two      -2.104569
baz    one      -0.494929
dtype: float64

In [37]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo  two   -0.706771
bar  one   -0.861849
qux  one   -1.039575
baz  one   -0.494929
dtype: float64
```

4.3.2 Advanced indexing with hierarchical index

Syntactically integrating `MultiIndex` in advanced indexing with `.loc` is a bit challenging, but we've made every effort to do so. In general, MultiIndex keys take the form of tuples. For example, the following works as you would expect:

```
In [38]: df = df.T

In [39]: df
Out[39]:
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
      two     0.805244  0.813850  1.607920
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466
      two    -0.226169 -1.436737 -2.006747

In [40]: df.loc[('bar', 'two')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    0.805244
B    0.813850
```

(continues on next page)

(continued from previous page)

```
C    1.607920
Name: (bar, two), dtype: float64
```

Note that `df.loc['bar', 'two']` would also work in this example, but this shorthand notation can lead to ambiguity in general.

If you also want to index a specific column with `.loc`, you must use a tuple like this:

```
In [41]: df.loc[('bar', 'two'), 'A']
Out[41]: 0.8052440253863785
```

You don't have to specify all levels of the `MultiIndex` by passing only the first elements of the tuple. For example, you can use "partial" indexing to get all elements with `bar` in the first level as follows:

df.loc['bar']

This is a shortcut for the slightly more verbose notation `df.loc[('bar',),]` (equivalent to `df.loc['bar',]` in this example).

"Partial" slicing also works quite nicely.

```
In [42]: df.loc['baz':'foo']
Out[42]:
                     A         B         C
first second
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
```

You can slice with a 'range' of values, by providing a slice of tuples.

```
In [43]: df.loc[('baz', 'two'):('qux', 'one')]
Out[43]:
                     A         B         C
first second
baz   two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466

In [44]: df.loc[('baz', 'two'):'foo']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                     A         B         C
first second
baz   two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
```

Passing a list of labels or tuples works similar to reindexing:

```
In [45]: df.loc[[('bar', 'two'), ('qux', 'one')]]
Out[45]:
                     A         B         C
first second
bar   two     0.805244  0.813850  1.607920
qux   one    -1.170299  1.130127  0.974466
```

Note: It is important to note that tuples and lists are not treated identically in pandas when it comes to indexing. Whereas a tuple is interpreted as one multi-level key, a list is used to specify several keys. Or in other words, tuples go horizontally (traversing levels), lists go vertically (scanning levels).

Importantly, a list of tuples indexes several complete `MultiIndex` keys, whereas a tuple of lists refer to several values within a level:

```
In [46]: s = pd.Series([1, 2, 3, 4, 5, 6],
   ....:               index=pd.MultiIndex.from_product([["A", "B"], ["c", "d", "e
↪"]]))
   ....:

In [47]: s.loc[[("A", "c"), ("B", "d")]]  # list of tuples
Out[47]:
A  c    1
B  d    5
dtype: int64

In [48]: s.loc[(["A", "B"], ["c", "d"])]  # tuple of lists
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[48]:
A  c    1
   d    2
B  c    4
   d    5
dtype: int64
```

Using slicers

You can slice a `MultiIndex` by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see *Selection by Label*, including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of *that* level. You do not need to specify all the *deeper* levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

Warning: You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing *both* axes, rather than into say the `MultiIndex` for the rows.

You should do this:

```
df.loc[(slice('A1', 'A3'), ...), :]             # noqa: E999
```

You should **not** do this:

```
df.loc[(slice('A1', 'A3'), ...)]                # noqa: E999
```

```
In [49]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:
```

(continues on next page)

(continued from previous page)

```
In [50]: miindex = pd.MultiIndex.from_product([mklbl('A', 4),
   ....:                                       mklbl('B', 2),
   ....:                                       mklbl('C', 4),
   ....:                                       mklbl('D', 2)])
   ....:

In [51]: micolumns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'),
   ....:                                        ('b', 'foo'), ('b', 'bah')],
   ....:                                       names=['lvl0', 'lvl1'])
   ....:

In [52]: dfmi = pd.DataFrame(np.arange(len(miindex) * len(micolumns))
   ....:                     .reshape((len(miindex), len(micolumns))),
   ....:                     index=miindex,
   ....:                     columns=micolumns).sort_index().sort_index(axis=1)
   ....:

In [53]: dfmi
Out[53]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0    9    8   11   10
         D1   13   12   15   14
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0   25   24   27   26
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  233  232  235  234
         D1  237  236  239  238
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  249  248  251  250
         D1  253  252  255  254

[64 rows x 4 columns]
```

Basic MultiIndex slicing using slices, lists, and labels.

```
In [54]: dfmi.loc[(slice('A1', 'A3'), slice(None), ['C1', 'C3']), :]
Out[54]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A1 B0 C1 D0   73   72   75   74
         D1   77   76   79   78
      C3 D0   89   88   91   90
         D1   93   92   95   94
   B1 C1 D0  105  104  107  106
         D1  109  108  111  110
      C3 D0  121  120  123  122
...          ...  ...  ...  ...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
```

(continues on next page)

(continued from previous page)

```
        D1  237  236  239  238
     C3 D0  249  248  251  250
        D1  253  252  255  254

[24 rows x 4 columns]
```

You can use `pandas.IndexSlice` to facilitate a more natural syntax using `:`, rather than using `slice(None)`.

```
In [55]: idx = pd.IndexSlice

In [56]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[56]:
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
...          ...  ...
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[32 rows x 2 columns]
```

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```
In [57]: dfmi.loc['A1', (slice(None), 'foo')]
Out[57]:
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
      D1   76   78
   C2 D0   80   82
      D1   84   86
   C3 D0   88   90
...       ...  ...
B1 C0 D1  100  102
   C1 D0  104  106
      D1  108  110
   C2 D0  112  114
      D1  116  118
   C3 D0  120  122
      D1  124  126

[16 rows x 2 columns]

In [58]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
         D1   12   14
      C3 D0   24   26
         D1   28   30
   B1 C1 D0   40   42
         D1   44   46
      C3 D0   56   58
...          ...  ...
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[32 rows x 2 columns]
```

Using a boolean indexer you can provide selection related to the *values*.

```
In [59]: mask = dfmi[('a', 'foo')] > 200

In [60]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[60]:
lvl0           a    b
lvl1         foo  foo
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```
In [61]: dfmi.loc(axis=0)[:, :, ['C1', 'C3']]
Out[61]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
         D1   13   12   15   14
      C3 D0   25   24   27   26
         D1   29   28   31   30
   B1 C1 D0   41   40   43   42
         D1   45   44   47   46
      C3 D0   57   56   59   58
...          ...  ...  ...  ...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
```

(continues on next page)

(continued from previous page)

```
      C3 D0   249   248   251   250
         D1   253   252   255   254

[32 rows x 4 columns]
```

Furthermore, you can *set* the values using the following methods.

```
In [62]: df2 = dfmi.copy()

In [63]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

In [64]: df2
Out[64]: 
lvl0           a         b     
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0  -10  -10  -10  -10
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10

[64 rows x 4 columns]
```

You can use a right-hand-side of an alignable object as well.

```
In [65]: df2 = dfmi.copy()

In [66]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [67]: df2
Out[67]: 
lvl0              a               b        
lvl1            bar     foo     bah     foo
A0 B0 C0 D0       1       0       3       2
         D1       5       4       7       6
      C1 D0    9000    8000   11000   10000
         D1   13000   12000   15000   14000
      C2 D0      17      16      19      18
         D1      21      20      23      22
      C3 D0   25000   24000   27000   26000
...             ...     ...     ...     ...
A3 B1 C0 D1     229     228     231     230
      C1 D0  233000  232000  235000  234000
         D1  237000  236000  239000  238000
      C2 D0     241     240     243     242
         D1     245     244     247     246
      C3 D0  249000  248000  251000  250000
```

(continues on next page)

(continued from previous page)

```
         D1  253000  252000  255000  254000

[64 rows x 4 columns]
```

Cross-section

The `xs()` method of `DataFrame` additionally takes a level argument to make selecting data at a particular level of a `MultiIndex` easier.

```
In [68]: df
Out[68]:
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
      two     0.805244  0.813850  1.607920
baz   one    -1.206412  0.132003  1.024180
      two     2.565646 -0.827317  0.569605
foo   one     1.431256 -0.076467  0.875906
      two     1.340309 -1.187678 -2.211372
qux   one    -1.170299  1.130127  0.974466
      two    -0.226169 -1.436737 -2.006747

In [69]: df.xs('one', level='second')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
              A         B         C
first
bar    0.895717  0.410835 -1.413681
baz   -1.206412  0.132003  1.024180
foo    1.431256 -0.076467  0.875906
qux   -1.170299  1.130127  0.974466
```

```
# using the slicers
In [70]: df.loc[(slice(None), 'one'), :]
Out[70]:
                     A         B         C
first second
bar   one     0.895717  0.410835 -1.413681
baz   one    -1.206412  0.132003  1.024180
foo   one     1.431256 -0.076467  0.875906
qux   one    -1.170299  1.130127  0.974466
```

You can also select on the columns with `xs`, by providing the axis argument.

```
In [71]: df = df.T

In [72]: df.xs('one', level='second', axis=1)
Out[72]:
first        bar       baz       foo       qux
A       0.895717 -1.206412  1.431256 -1.170299
B       0.410835  0.132003 -0.076467  1.130127
C      -1.413681  1.024180  0.875906  0.974466
```

```
# using the slicers
In [73]: df.loc[:, (slice(None), 'one')]
Out[73]:
first         bar       baz       foo       qux
second        one       one       one       one
A        0.895717 -1.206412  1.431256 -1.170299
B        0.410835  0.132003 -0.076467  1.130127
C       -1.413681  1.024180  0.875906  0.974466
```

xs also allows selection with multiple keys.

```
In [74]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out[74]:
first         bar
second        one
A        0.895717
B        0.410835
C       -1.413681
```

```
# using the slicers
In [75]: df.loc[:, ('bar', 'one')]
Out[75]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64
```

You can pass drop_level=False to xs to retain the level that was selected.

```
In [76]: df.xs('one', level='second', axis=1, drop_level=False)
Out[76]:
first         bar       baz       foo       qux
second        one       one       one       one
A        0.895717 -1.206412  1.431256 -1.170299
B        0.410835  0.132003 -0.076467  1.130127
C       -1.413681  1.024180  0.875906  0.974466
```

Compare the above with the result using drop_level=True (the default value).

```
In [77]: df.xs('one', level='second', axis=1, drop_level=True)
Out[77]:
first       bar       baz       foo       qux
A      0.895717 -1.206412  1.431256 -1.170299
B      0.410835  0.132003 -0.076467  1.130127
C     -1.413681  1.024180  0.875906  0.974466
```

Advanced reindexing and alignment

Using the parameter level in the *reindex()* and *align()* methods of pandas objects is useful to broadcast values across a level. For instance:

```
In [78]: midx = pd.MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
   ....:                      codes=[[1, 1, 0, 0], [1, 0, 1, 0]])
   ....:
```

(continues on next page)

(continued from previous page)

```
In [79]: df = pd.DataFrame(np.random.randn(4, 2), index=midx)

In [80]: df
Out[80]:
               0         1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [81]: df2 = df.mean(level=0)

In [82]: df2
Out[82]:
             0         1
one   1.060074 -0.109716
zero  1.271532  0.713416

In [83]: df2.reindex(df.index, level=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[83]:
↪
               0         1
one  y  1.060074 -0.109716
     x  1.060074 -0.109716
zero y  1.271532  0.713416
     x  1.271532  0.713416

# aligning
In [84]: df_aligned, df2_aligned = df.align(df2, level=0)

In [85]: df_aligned
Out[85]:
               0         1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [86]: df2_aligned
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               0         1
one  y  1.060074 -0.109716
     x  1.060074 -0.109716
zero y  1.271532  0.713416
     x  1.271532  0.713416
```

Swapping levels with `swaplevel`

The *swaplevel()* method can switch the order of two levels:

```
In [87]: df[:5]
Out[87]:
               0         1
one  y  1.519970 -0.493662
```

(continues on next page)

(continued from previous page)

```
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520

In [88]: df[:5].swaplevel(0, 1, axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               0         1
y one   1.519970 -0.493662
x one   0.600178  0.274230
y zero  0.132885 -0.023688
x zero  2.410179  1.450520
```

Reordering levels with `reorder_levels`

The *reorder_levels()* method generalizes the `swaplevel` method, allowing you to permute the hierarchical index levels in one step:

```
In [89]: df[:5].reorder_levels([1, 0], axis=0)
Out[89]:
               0         1
y one   1.519970 -0.493662
x one   0.600178  0.274230
y zero  0.132885 -0.023688
x zero  2.410179  1.450520
```

Renaming names of an `Index` or `MultiIndex`

The *rename()* method is used to rename the labels of a `MultiIndex`, and is typically used to rename the columns of a `DataFrame`. The `columns` argument of `rename` allows a dictionary to be specified that includes only the columns you wish to rename.

```
In [90]: df.rename(columns={0: "col0", 1: "col1"})
Out[90]:
            col0      col1
one  y  1.519970 -0.493662
     x  0.600178  0.274230
zero y  0.132885 -0.023688
     x  2.410179  1.450520
```

This method can also be used to rename specific labels of the main index of the `DataFrame`.

```
In [91]: df.rename(index={"one": "two", "y": "z"})
Out[91]:
               0         1
two  z  1.519970 -0.493662
     x  0.600178  0.274230
zero z  0.132885 -0.023688
     x  2.410179  1.450520
```

The *rename_axis()* method is used to rename the name of a `Index` or `MultiIndex`. In particular, the names of the levels of a `MultiIndex` can be specified, which is useful if `reset_index()` is later used to move the values from the `MultiIndex` to a column.

```
In [92]: df.rename_axis(index=['abc', 'def'])
Out[92]:
                 0         1
abc  def
one  y    1.519970 -0.493662
     x    0.600178  0.274230
zero y    0.132885 -0.023688
     x    2.410179  1.450520
```

Note that the columns of a `DataFrame` are an index, so that using `rename_axis` with the `columns` argument will change the name of that index.

```
In [93]: df.rename_axis(columns="Cols").columns
Out[93]: RangeIndex(start=0, stop=2, step=1, name='Cols')
```

Both `rename` and `rename_axis` support specifying a dictionary, `Series` or a mapping function to map labels/names to new values.

4.3.3 Sorting a `MultiIndex`

For *MultiIndex*-ed objects to be indexed and sliced effectively, they need to be sorted. As with any index, you can use *sort_index()*.

```
In [94]: import random

In [95]: random.shuffle(tuples)

In [96]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))

In [97]: s
Out[97]:
foo  two    0.206053
qux  one   -0.251905
foo  one   -2.213588
bar  two    1.063327
baz  two    1.266143
     one    0.299368
bar  one   -0.863838
qux  two    0.408204
dtype: float64

In [98]: s.sort_index()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
bar  one   -0.863838
     two    1.063327
baz  one    0.299368
     two    1.266143
foo  one   -2.213588
     two    0.206053
qux  one   -0.251905
     two    0.408204
dtype: float64

In [99]: s.sort_index(level=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
bar  one   -0.863838
     two    1.063327
baz  one    0.299368
     two    1.266143
foo  one   -2.213588
     two    0.206053
qux  one   -0.251905
     two    0.408204
dtype: float64

In [100]: s.sort_index(level=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
bar  one   -0.863838
baz  one    0.299368
foo  one   -2.213588
qux  one   -0.251905
bar  two    1.063327
baz  two    1.266143
foo  two    0.206053
qux  two    0.408204
dtype: float64
```

You may also pass a level name to `sort_index` if the `MultiIndex` levels are named.

```
In [101]: s.index.set_names(['L1', 'L2'], inplace=True)

In [102]: s.sort_index(level='L1')
Out[102]:
L1   L2
bar  one   -0.863838
     two    1.063327
baz  one    0.299368
     two    1.266143
foo  one   -2.213588
     two    0.206053
qux  one   -0.251905
     two    0.408204
dtype: float64

In [103]: s.sort_index(level='L2')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
L1   L2
bar  one   -0.863838
baz  one    0.299368
foo  one   -2.213588
qux  one   -0.251905
bar  two    1.063327
baz  two    1.266143
foo  two    0.206053
qux  two    0.408204
dtype: float64
```

On higher dimensional objects, you can sort any of the other axes by level if they have a `MultiIndex`:

```
In [104]: df.T.sort_index(level=1, axis=1)
Out[104]:
        one      zero       one      zero
          x         x         y         y
0  0.600178  2.410179  1.519970  0.132885
1  0.274230  1.450520 -0.493662 -0.023688
```

Indexing will work even if the data are not sorted, but will be rather inefficient (and show a `PerformanceWarning`). It will also return a copy of the data rather than a view:

```
In [105]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
   .....:                     'joe': ['x', 'x', 'z', 'y'],
   .....:                     'jolie': np.random.rand(4)})
   .....:

In [106]: dfm = dfm.set_index(['jim', 'joe'])

In [107]: dfm
Out[107]:
            jolie
jim joe
0   x    0.490671
    x    0.120248
1   z    0.537020
    y    0.110968
```

```
In [4]: dfm.loc[(1, 'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.

Out[4]:
           jolie
jim joe
1   z    0.64094
```

Furthermore, if you try to index something that is not fully lexsorted, this can raise:

```
In [5]: dfm.loc[(0, 'y'):(1, 'z')]
UnsortedIndexError: 'Key length (2) was greater than MultiIndex lexsort depth (1)'
```

The `is_lexsorted()` method on a `MultiIndex` shows if the index is sorted, and the `lexsort_depth` property returns the sort depth:

```
In [108]: dfm.index.is_lexsorted()
Out[108]: False

In [109]: dfm.index.lexsort_depth
\\\\\\\\\\\\\\\\Out[109]: 1
```

```
In [110]: dfm = dfm.sort_index()

In [111]: dfm
Out[111]:
            jolie
jim joe
0   x    0.490671
    x    0.120248
```

(continues on next page)

(continued from previous page)

```
1    y    0.110968
     z    0.537020

In [112]: dfm.index.is_lexsorted()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True

In [113]: dfm.index.lexsort_depth
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪2
```

And now selection works as expected.

```
In [114]: dfm.loc[(0, 'y'):(1, 'z')]
Out[114]:
            jolie
jim joe
1   y    0.110968
    z    0.537020
```

4.3.4 Take Methods

Similar to NumPy ndarrays, pandas `Index`, `Series`, and `DataFrame` also provides the *take()* method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [115]: index = pd.Index(np.random.randint(0, 1000, 10))

In [116]: index
Out[116]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64
↪')

In [117]: positions = [0, 9, 3]

In [118]: index[positions]
Out[118]: Int64Index([214, 329, 567], dtype='int64')

In [119]: index.take(positions)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[119]: Int64Index([214, 329,␣
↪567], dtype='int64')

In [120]: ser = pd.Series(np.random.randn(10))

In [121]: ser.iloc[positions]
Out[121]:
0   -0.179666
9    1.824375
3    0.392149
dtype: float64

In [122]: ser.take(positions)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[122]:
0   -0.179666
9    1.824375
```

(continues on next page)

(continued from previous page)

```
3    0.392149
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [123]: frm = pd.DataFrame(np.random.randn(5, 3))

In [124]: frm.take([1, 4, 3])
Out[124]:
          0         1         2
1 -1.237881  0.106854 -1.276829
4  0.629675 -1.425966  1.857704
3  0.979542 -1.633678  0.615855

In [125]: frm.take([0, 2], axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          0         2
0  0.595974  0.601544
1 -1.237881 -1.276829
2 -0.767101  1.499591
3  0.979542  0.615855
4  0.629675  1.857704
```

It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```
In [126]: arr = np.random.randn(10)

In [127]: arr.take([False, False, True, True])
Out[127]: array([-1.1935, -1.1935,  0.6775,  0.6775])

In [128]: arr[[0, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[128]: array([-1.1935,  0.
↪6775])

In [129]: ser = pd.Series(np.random.randn(10))

In [130]: ser.take([False, False, True, True])
Out[130]:
0    0.233141
0    0.233141
1   -0.223540
1   -0.223540
dtype: float64

In [131]: ser.iloc[[0, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[131]:␣
↪
0    0.233141
1   -0.223540
dtype: float64
```

Finally, as a small note on performance, because the `take` method handles a narrower range of inputs, it can offer performance that is a good deal faster than fancy indexing.

```
In [132]: arr = np.random.randn(10000, 5)

In [133]: indexer = np.arange(10000)

In [134]: random.shuffle(indexer)

In [135]: %timeit arr[indexer]
   .....: %timeit arr.take(indexer, axis=0)
   .....:
186 us +- 7.36 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
50.5 us +- 644 ns per loop (mean +- std. dev. of 7 runs, 10000 loops each)
```

4.3.5 Index Types

We have discussed `MultiIndex` in the previous sections pretty extensively. Documentation about `DatetimeIndex` and `PeriodIndex` are shown *here*, and documentation about `TimedeltaIndex` is found *here*.

In the following sub-sections we will highlight some other index types.

CategoricalIndex

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a *`Categorical`* and allows efficient indexing and storage of an index with a large number of duplicated elements.

```
In [136]: from pandas.api.types import CategoricalDtype

In [137]: df = pd.DataFrame({'A': np.arange(6),
   .....:                    'B': list('aabbca')})
   .....:

In [138]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))

In [139]: df
Out[139]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [140]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[140]:
A       int64
B    category
dtype: object

In [141]: df.B.cat.categories
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index(['c', 'a', 'b'], dtype='object')
```

Setting the index will create a `CategoricalIndex`.

```
In [142]: df2 = df.set_index('B')

In [143]: df2.index
Out[143]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
↪ ordered=False, name='B', dtype='category')
```

Indexing with `__getitem__`/`.iloc`/`.loc` works similarly to an `Index` with duplicates. The indexers **must** be in the category or the operation will raise a `KeyError`.

```
In [144]: df2.loc['a']
Out[144]:
   A
B
a  0
a  1
a  5
```

The `CategoricalIndex` is **preserved** after indexing:

```
In [145]: df2.loc['a'].index
Out[145]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
↪ name='B', dtype='category')
```

Sorting the index will sort by the order of the categories (recall that we created the index with `CategoricalDtype(list('cab'))`, so the sorted order is `cab`).

```
In [146]: df2.sort_index()
Out[146]:
   A
B
c  4
a  0
a  1
a  5
b  2
b  3
```

Groupby operations on the index will preserve the index nature as well.

```
In [147]: df2.groupby(level=0).sum()
Out[147]:
   A
B
c  4
a  6
b  5

In [148]: df2.groupby(level=0).sum().index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[148]: CategoricalIndex(['c', 'a', 'b'],␣
↪categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Reindexing operations will return a resulting index based on the type of the passed indexer. Passing a list will return a plain-old `Index`; indexing with a `Categorical` will return a `CategoricalIndex`, indexed according to the categories of the **passed** `Categorical` dtype. This allows one to arbitrarily index these even with values **not** in the categories, similarly to how you can reindex **any** pandas index.

```
In [149]: df2.reindex(['a', 'e'])
Out[149]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [150]: df2.reindex(['a', 'e']).index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[150]: Index(['a', 'a', 'a',
↪'e'], dtype='object', name='B')

In [151]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde')))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [152]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde'))).index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c', 'd', 'e'],␣
↪ordered=False, name='B', dtype='category')
```

Warning: Reshaping and Comparison operations on a `CategoricalIndex` must have the same categories or a `TypeError` will be raised.

```
In [9]: df3 = pd.DataFrame({'A': np.arange(6), 'B': pd.Series(list('aabbca')).
↪astype('category')})

In [11]: df3 = df3.set_index('B')

In [11]: df3.index
Out[11]: CategoricalIndex([u'a', u'a', u'b', u'b', u'c', u'a'], categories=[u'a', u
↪'b', u'c'], ordered=False, name=u'B', dtype='category')

In [12]: pd.concat([df2, df3])
TypeError: categories must match existing categories when appending
```

Int64Index and RangeIndex

Warning: Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see *here*.

`Int64Index` is a fundamental basic index in pandas. This is an immutable array implementing an ordered, sliceable set. Prior to 0.18.0, the `Int64Index` would provide the default index for all `NDFrame` objects.

`RangeIndex` is a sub-class of `Int64Index` added in version 0.18.0, now providing the default index for all `NDFrame` objects. `RangeIndex` is an optimized version of `Int64Index` that can represent a monotonic ordered

set. These are analogous to Python range types.

Float64Index

By default a *Float64Index* will be automatically created when passing floating, or mixed-integer-floating values in index creation. This enables a pure label-based slicing paradigm that makes `[],ix,loc` for scalar indexing and slicing work exactly the same.

```
In [153]: indexf = pd.Index([1.5, 2, 3, 4.5, 5])

In [154]: indexf
Out[154]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [155]: sf = pd.Series(range(5), index=indexf)

In [156]: sf
Out[156]:
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
dtype: int64
```

Scalar selection for `[],.loc` will always be label based. An integer will match an equal float index (e.g. `3` is equivalent to `3.0`).

```
In [157]: sf[3]
Out[157]: 2

In [158]: sf[3.0]
\\\\\\\\\\\\Out[158]: 2

In [159]: sf.loc[3]
\\\\\\\\\\\\\\\\\\\\\\\\Out[159]: 2

In [160]: sf.loc[3.0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[160]: 2
```

The only positional indexing is via `iloc`.

```
In [161]: sf.iloc[3]
Out[161]: 3
```

A scalar index that is not found will raise a `KeyError`. Slicing is primarily on the values of the index when using `[],ix,loc`, and **always** positional when using `iloc`. The exception is when the slice is boolean, in which case it will always be positional.

```
In [162]: sf[2:4]
Out[162]:
2.0    1
3.0    2
dtype: int64

In [163]: sf.loc[2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[163]:
```

(continues on next page)

(continued from previous page)

```
2.0    1
3.0    2
dtype: int64

In [164]: sf.iloc[2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[164]:
↪
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```
In [165]: sf[2.1:4.6]
Out[165]:
3.0    2
4.5    3
dtype: int64

In [166]: sf.loc[2.1:4.6]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[166]:
3.0    2
4.5    3
dtype: int64
```

In non-float indexes, slicing using floats will raise a `TypeError`.

```
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type
↪(Int64Index)
```

Warning: Using a scalar float indexer for `.iloc` has been removed in 0.18.0, so the following will raise a `TypeError`:

```
In [3]: pd.Series(range(5)).iloc[3.0]
TypeError: cannot do positional indexing on <class 'pandas.indexes.range.RangeIndex
↪'> with these indexers [3.0] of <type 'float'>
```

Here is a typical use-case for using this type of indexing. Imagine that you have a somewhat irregular timedelta-like indexing scheme, but the data is recorded as floats. This could, for example, be millisecond offsets.

```
In [167]: dfir = pd.concat([pd.DataFrame(np.random.randn(5, 2),
   .....:                                index=np.arange(5) * 250.0,
   .....:                                columns=list('AB')),
   .....:                   pd.DataFrame(np.random.randn(6, 2),
   .....:                                index=np.arange(4, 10) * 250.1,
   .....:                                columns=list('AB'))])
   .....:

In [168]: dfir
Out[168]:
```

(continues on next page)

(continued from previous page)

```
                A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
1000.4 -0.179734  0.993962
1250.5 -0.212673  0.909872
1500.6 -0.733333 -0.349893
1750.7  0.456434 -0.306735
2000.8  0.553396  0.166221
2250.9 -0.101684 -0.734907
```

Selection operations then will always work on a value basis, for all selection operators.

```
In [169]: dfir[0:1000.4]
Out[169]:
                A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
1000.4 -0.179734  0.993962

In [170]: dfir.loc[0:1001, 'A']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0.0      -0.435772
250.0    -0.808286
500.0    -1.815703
750.0    -0.243487
1000.0    1.162969
1000.4   -0.179734
Name: A, dtype: float64

In [171]: dfir.loc[1000.4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A   -0.179734
B    0.993962
Name: 1000.4, dtype: float64
```

You could retrieve the first 1 second (1000 ms) of data as such:

```
In [172]: dfir[0:1000]
Out[172]:
                A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
```

If you need integer based selection, you should use `iloc`:

```
In [173]: dfir.iloc[0:5]
Out[173]:
               A         B
0.0    -0.435772 -1.188928
250.0  -0.808286 -0.284634
500.0  -1.815703  1.347213
750.0  -0.243487  0.514704
1000.0  1.162969 -0.287725
```

IntervalIndex

New in version 0.20.0.

IntervalIndex together with its own dtype, `IntervalDtype` as well as the *Interval* scalar type, allow first-class support in pandas for interval notation.

The `IntervalIndex` allows some unique indexing and is also used as a return type for the categories in *cut()* and *qcut()*.

> **Warning:** These indexing behaviors are provisional and may change in a future version of pandas.

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

```
In [174]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
   .....:                   index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
   .....:

In [175]: df
Out[175]:
        A
(0, 1]  1
(1, 2]  2
(2, 3]  3
(3, 4]  4
```

Label based indexing via `.loc` along the edges of an interval works as you would expect, selecting that particular interval.

```
In [176]: df.loc[2]
Out[176]:
A    2
Name: (1, 2], dtype: int64

In [177]: df.loc[[2, 3]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[177]:
        A
(1, 2]  2
(2, 3]  3
```

If you select a label *contained* within an interval, this will also select the interval.

```
In [178]: df.loc[2.5]
Out[178]:
A    3
```

(continues on next page)

(continued from previous page)

```
Name: (2, 3], dtype: int64

In [179]: df.loc[[2.5, 3.5]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[179]:
        A
(2, 3]  3
(3, 4]  4
```

`Interval` and `IntervalIndex` are used by `cut` and `qcut`:

```
In [180]: c = pd.cut(range(4), bins=2)

In [181]: c
Out[181]:
[(-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 [182]: c.categories
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]],
              closed='right',
              dtype='interval[float64]')
```

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

```
In [183]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[183]:
[(-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]]
```

Generating Ranges of Intervals

If we need intervals on a regular frequency, we can use the *interval_range()* function to create an `IntervalIndex` using various combinations of `start`, `end`, and `periods`. The default frequency for `interval_range` is a 1 for numeric intervals, and calendar day for datetime-like intervals:

```
In [184]: pd.interval_range(start=0, end=5)
Out[184]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right',
              dtype='interval[int64]')

In [185]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03], (2017-01-03, 2017-
↪01-04], (2017-01-04, 2017-01-05]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [186]: pd.interval_range(end=pd.Timedelta('3 days'), periods=3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:00:00, 2 days 00:00:00],
↪ (2 days 00:00:00, 3 days 00:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

The `freq` parameter can used to specify non-default frequencies, and can utilize a variety of *frequency aliases* with datetime-like intervals:

```
In [187]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[187]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]],
              closed='right',
              dtype='interval[float64]')

In [188]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4, freq='W')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-
↪01-22], (2017-01-22, 2017-01-29]],
              closed='right',
              dtype='interval[datetime64[ns]]')

In [189]: pd.interval_range(start=pd.Timedelta('0 days'), periods=3, freq='9H')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days 18:00:00],
↪ (0 days 18:00:00, 1 days 03:00:00]],
              closed='right',
              dtype='interval[timedelta64[ns]]')
```

Additionally, the `closed` parameter can be used to specify which side(s) the intervals are closed on. Intervals are closed on the right side by default.

```
In [190]: pd.interval_range(start=0, end=4, closed='both')
Out[190]:
IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]],
              closed='both',
              dtype='interval[int64]')

In [191]: pd.interval_range(start=0, end=4, closed='neither')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)],
              closed='neither',
              dtype='interval[int64]')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced intervals from `start` to `end` inclusively, with `periods` number of elements in the resulting `IntervalIndex`:

```
In [192]: pd.interval_range(start=0, end=6, periods=4)
Out[192]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
              closed='right',
              dtype='interval[float64]')
```

(continues on next page)

(continued from previous page)

```
In [193]: pd.interval_range(pd.Timestamp('2018-01-01'),
   .....:                   pd.Timestamp('2018-02-28'), periods=3)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-02-08␣
↪16:00:00], (2018-02-08 16:00:00, 2018-02-28]],
              closed='right',
              dtype='interval[datetime64[ns]]')
```

4.3.6 Miscellaneous indexing FAQ

Integer indexing

Label-based indexing with integer axis labels is a thorny topic. It has been discussed heavily on mailing lists and among various members of the scientific Python community. In pandas, our general viewpoint is that labels matter more than integer locations. Therefore, with an integer axis index *only* label-based indexing is possible with the standard tools like `.loc`. The following code will generate exceptions:

```
In [194]: s = pd.Series(range(5))

In [195]: s[-1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-195-76c3dce40054> in <module>
----> 1 s[-1]

/pandas/pandas/core/series.py in __getitem__(self, key)
    866         key = com.apply_if_callable(key, self)
    867         try:
--> 868             result = self.index.get_value(self, key)
    869 
    870             if not is_scalar(result):

/pandas/pandas/core/indexes/base.py in get_value(self, series, key)
   4318         try:
   4319             return self._engine.get_value(s, k,
-> 4320                                           tz=getattr(series.dtype, 'tz',␣
↪None))
   4321         except KeyError as e1:
   4322             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()

/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()

/pandas/pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

/pandas/pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.
↪Int64HashTable.get_item()

/pandas/pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.
↪Int64HashTable.get_item()
```

(continues on next page)

(continued from previous page)

```
KeyError: -1

In [196]: df = pd.DataFrame(np.random.randn(5, 4))

In [197]: df
Out[197]:
          0         1         2         3
0 -0.130121 -0.476046  0.759104  0.213379
1 -0.082641  0.448008  0.656420 -1.051443
2  0.594956 -0.151360 -0.069303  1.221431
3 -0.182832  0.791235  0.042745  2.069775
4  1.446552  0.019814 -1.389212 -0.702312

In [198]: df.loc[-2:]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          0         1         2         3
0 -0.130121 -0.476046  0.759104  0.213379
1 -0.082641  0.448008  0.656420 -1.051443
2  0.594956 -0.151360 -0.069303  1.221431
3 -0.182832  0.791235  0.042745  2.069775
4  1.446552  0.019814 -1.389212 -0.702312
```

This deliberate decision was made to prevent ambiguities and subtle bugs (many users reported finding bugs when the API change was made to stop "falling back" on position-based indexing).

Non-monotonic indexes require exact matches

If the index of a `Series` or `DataFrame` is monotonically increasing or decreasing, then the bounds of a label-based slice can be outside the range of the index, much like slice indexing a normal Python `list`. Monotonicity of an index can be tested with the `is_monotonic_increasing()` and `is_monotonic_decreasing()` attributes.

```
In [199]: df = pd.DataFrame(index=[2, 3, 3, 4, 5], columns=['data'],␣
↪data=list(range(5)))

In [200]: df.index.is_monotonic_increasing
Out[200]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [201]: df.loc[0:4, :]
\\\\\\\\\\\\\\Out[201]:
   data
2     0
3     1
3     2
4     3

# slice is are outside the index, so empty DataFrame is returned
In [202]: df.loc[13:15, :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[202]:
Empty DataFrame
Columns: [data]
Index: []
```

On the other hand, if the index is not monotonic, then both slice bounds must be *unique* members of the index.

```
In [203]: df = pd.DataFrame(index=[2, 3, 1, 4, 3, 5],
   .....:                   columns=['data'], data=list(range(6)))
   .....:

In [204]: df.index.is_monotonic_increasing
Out[204]: False

# OK because 2 and 4 are in the index
In [205]: df.loc[2:4, :]
\\\\\\\\\\\\\\\Out[205]:
   data
2     0
3     1
1     2
4     3
```

```
# 0 is not in the index
In [9]: df.loc[0:4, :]
KeyError: 0

# 3 is not a unique label
In [11]: df.loc[2:3, :]
KeyError: 'Cannot get right slice bound for non-unique label: 3'
```

`Index.is_monotonic_increasing` and `Index.is_monotonic_decreasing` only check that an index is weakly monotonic. To check for strict monotonicity, you can combine one of those with the *is_unique()* attribute.

```
In [206]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])

In [207]: weakly_monotonic
Out[207]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [208]: weakly_monotonic.is_monotonic_increasing
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[208]: True

In [209]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_unique
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[209]: False
```

Endpoints are inclusive

Compared with standard Python sequence slicing in which the slice endpoint is not inclusive, label-based slicing in pandas **is inclusive**. The primary reason for this is that it is often not possible to easily determine the "successor" or next element after a particular label in an index. For example, consider the following `Series`:

```
In [210]: s = pd.Series(np.random.randn(6), index=list('abcdef'))

In [211]: s
Out[211]:
a    0.301379
b    1.240445
c   -0.846068
d   -0.043312
e   -1.658747
```

(continues on next page)

(continued from previous page)

```
f   -0.819549
dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be accomplished as such:

```
In [212]: s[2:5]
Out[212]:
c   -0.846068
d   -0.043312
e   -1.658747
dtype: float64
```

However, if you only had `c` and `e`, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.loc['c':'e' + 1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design to make label-based slicing include both endpoints:

```
In [213]: s.loc['c':'e']
Out[213]:
c   -0.846068
d   -0.043312
e   -1.658747
dtype: float64
```

This is most definitely a "practicality beats purity" sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

Indexing potentially changes underlying Series dtype

The different indexing operation can potentially change the dtype of a `Series`.

```
In [214]: series1 = pd.Series([1, 2, 3])

In [215]: series1.dtype
Out[215]: dtype('int64')

In [216]: res = series1.reindex([0, 4])

In [217]: res.dtype
Out[217]: dtype('float64')

In [218]: res
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[218]:
0    1.0
4    NaN
dtype: float64
```

```
In [219]: series2 = pd.Series([True])

In [220]: series2.dtype
Out[220]: dtype('bool')
```

(continues on next page)

(continued from previous page)

```
In [221]: res = series2.reindex_like(series1)

In [222]: res.dtype
Out[222]: dtype('O')

In [223]: res
\\\\\\\\\\\\\\\\\\\\\Out[223]:
0    True
1     NaN
2     NaN
dtype: object
```

This is because the (re)indexing operations above silently inserts `NaNs` and the `dtype` changes accordingly. This can cause some issues when using `numpy ufuncs` such as `numpy.logical_and`.

See the this old issue for a more detailed discussion.

4.4 Merge, join, and concatenate

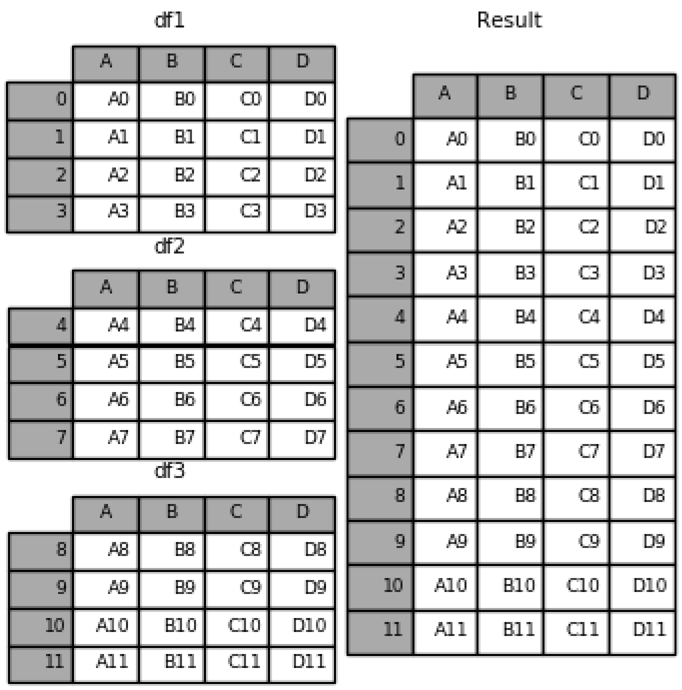
pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

4.4.1 Concatenating objects

The *concat()* function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say "if any" because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```
In [1]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ...:                     'B': ['B0', 'B1', 'B2', 'B3'],
   ...:                     'C': ['C0', 'C1', 'C2', 'C3'],
   ...:                     'D': ['D0', 'D1', 'D2', 'D3']},
   ...:                    index=[0, 1, 2, 3])
   ...:

In [2]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
   ...:                     'B': ['B4', 'B5', 'B6', 'B7'],
   ...:                     'C': ['C4', 'C5', 'C6', 'C7'],
   ...:                     'D': ['D4', 'D5', 'D6', 'D7']},
   ...:                    index=[4, 5, 6, 7])
   ...:

In [3]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
   ...:                     'B': ['B8', 'B9', 'B10', 'B11'],
   ...:                     'C': ['C8', 'C9', 'C10', 'C11'],
   ...:                     'D': ['D8', 'D9', 'D10', 'D11']},
   ...:                    index=[8, 9, 10, 11])
   ...:

In [4]: frames = [df1, df2, df3]

In [5]: result = pd.concat(frames)
```



Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of "what to do with the other axes":

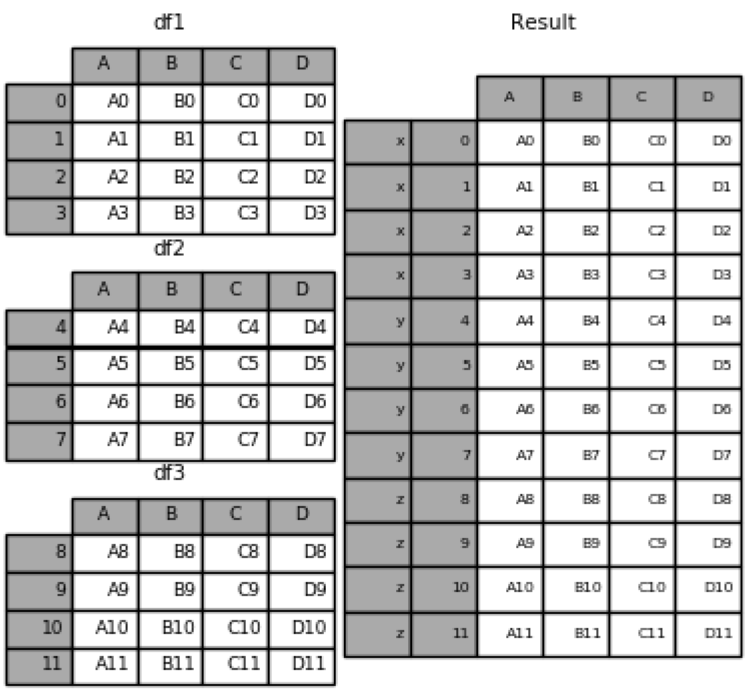
```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

- `objs` : a sequence or mapping of Series, DataFrame, or Panel objects. If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.
- `axis` : {0, 1, ...}, default 0. The axis to concatenate along.
- `join` : {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- `ignore_index` : boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.
- `join_axes` : list of Index objects. Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic.
- `keys` : sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.
- `levels` : list of sequences, default None. Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.

- `names` : list, default None. Names for the levels in the resulting hierarchical index.
- `verify_integrity` : boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.
- `copy` : boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments don't make much sense. Let's revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the `keys` argument:

```
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```



As you can see (if you've read the rest of the documentation), the resulting object's index has a *hierarchical index*. This means that we can now select out each chunk by key:

```
In [7]: result.loc['y']
Out[7]:
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

Note: It is worth noting that *concat()* (and therefore `append()`) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

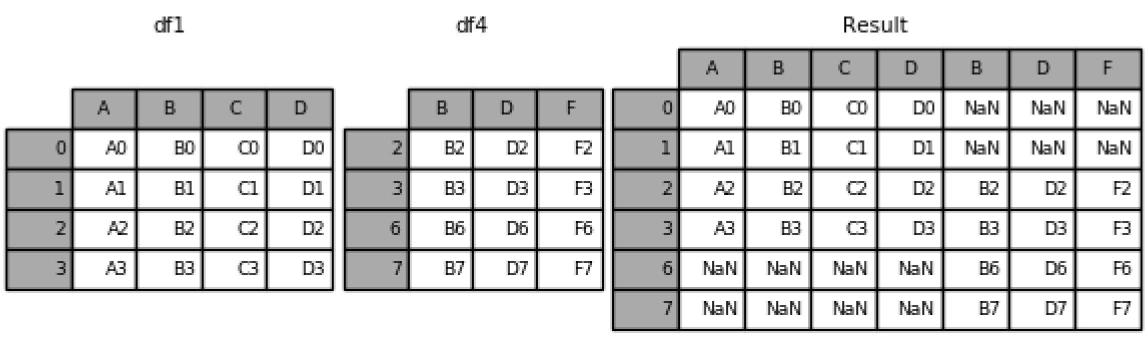
Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following three ways:

- Take the union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.
- Use a specific index, as passed to the `join_axes` argument.

Here is an example of each of these methods. First, the default `join='outer'` behavior:

```
In [8]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
   ...:                     'D': ['D2', 'D3', 'D6', 'D7'],
   ...:                     'F': ['F2', 'F3', 'F6', 'F7']},
   ...:                    index=[2, 3, 6, 7])
   ...:

In [9]: result = pd.concat([df1, df4], axis=1, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

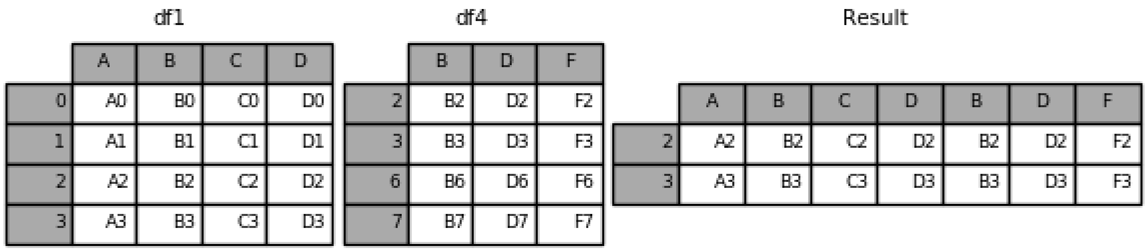
	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Warning: Changed in version 0.23.0.

The default behavior with `join='outer'` is to sort the other axis (columns in this case). In a future version of pandas, the default will be to not sort. We specified `sort=False` to opt in to the new behavior now.

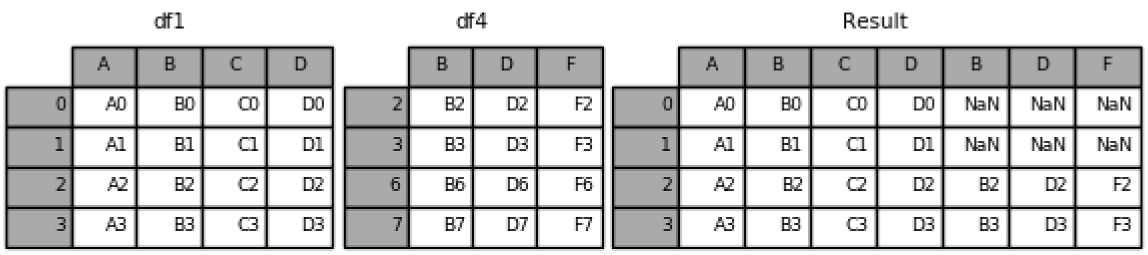
Here is the same thing with `join='inner'`:

```
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```



Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

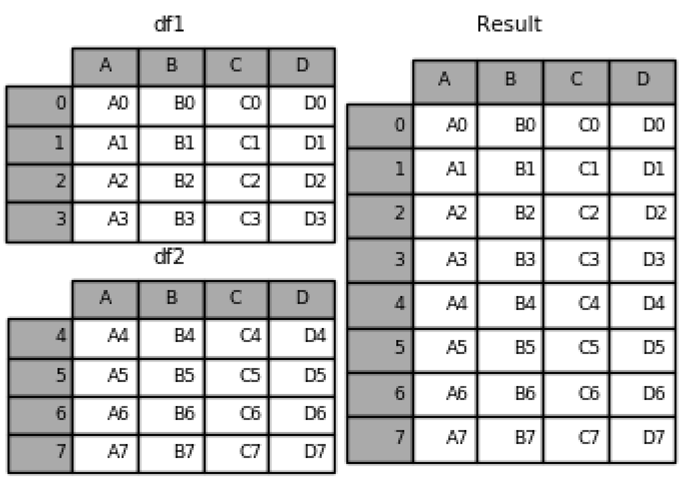
```
In [11]: result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
```



Concatenating using `append`

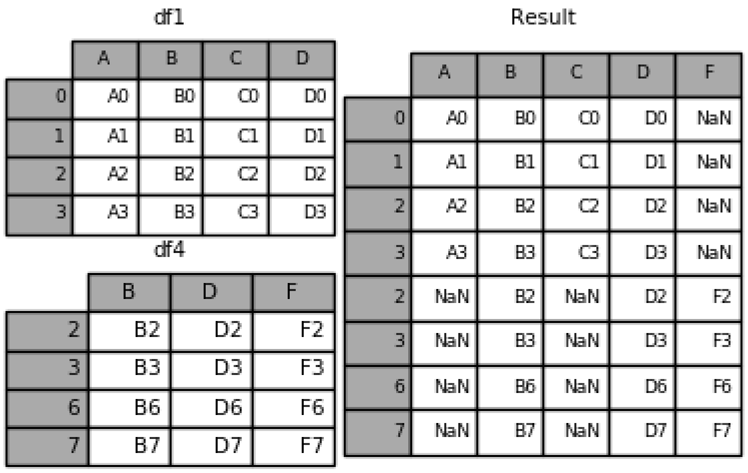
A useful shortcut to `concat()` are the `append()` instance methods on `Series` and `DataFrame`. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

```
In [12]: result = df1.append(df2)
```



In the case of `DataFrame`, the indexes must be disjoint but the columns do not need to be:

```
In [13]: result = df1.append(df4, sort=False)
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

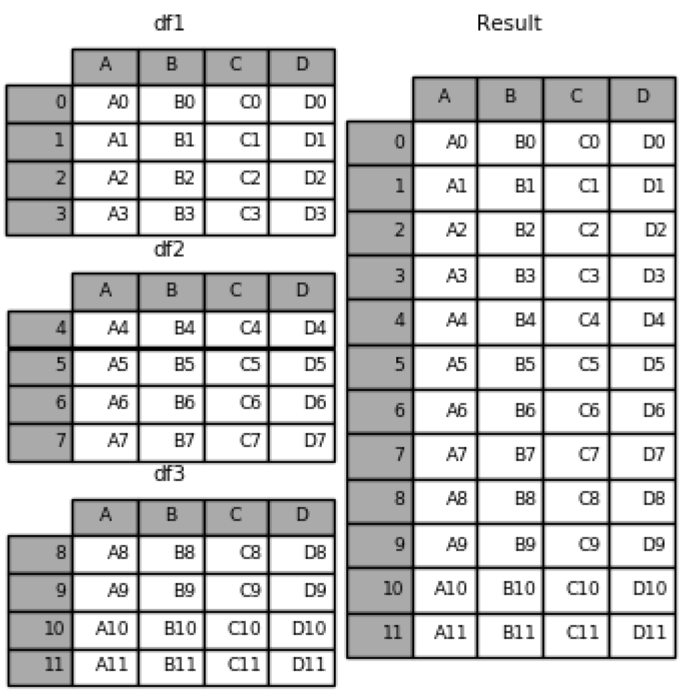
	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

`append` may take multiple objects to concatenate:

```
In [14]: result = df1.append([df2, df3])
```



df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

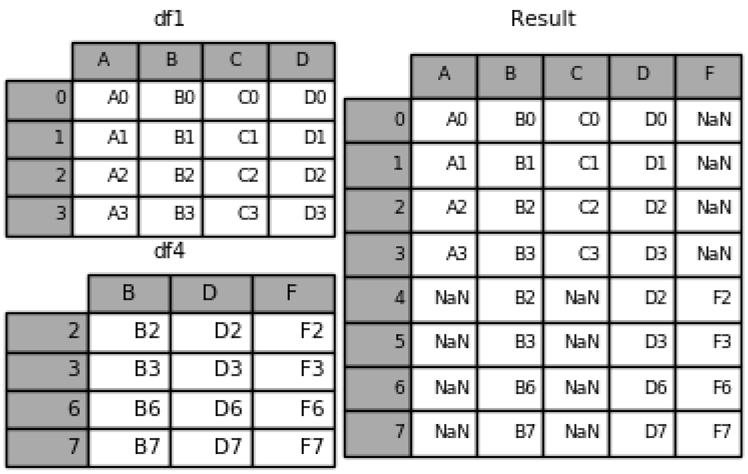
	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Note: Unlike the `append()` method, which appends to the original list and returns `None`, *`append()`* here **does not** modify `df1` and returns its copy with `df2` appended.

Ignoring indexes on the concatenation axis

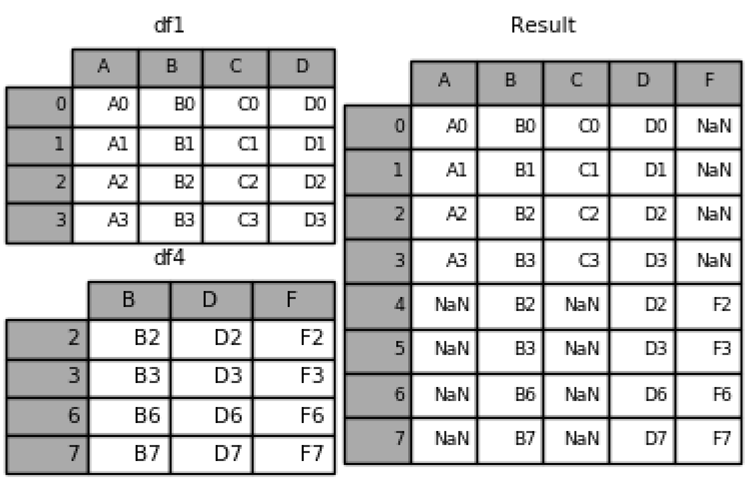
For `DataFrame` objects which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the `ignore_index` argument:

```
In [15]: result = pd.concat([df1, df4], ignore_index=True, sort=False)
```



This is also a valid argument to *`DataFrame.append()`*:

```
In [16]: result = df1.append(df4, ignore_index=True, sort=False)
```



Concatenating with mixed ndims

You can concatenate a mix of `Series` and `DataFrame` objects. The `Series` will be transformed to `DataFrame` with the column name as the name of the `Series`.

```
In [17]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')

In [18]: result = pd.concat([df1, s1], axis=1)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

Result

	A	B	C	D	X
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

Note: Since we're concatenating a `Series` to a `DataFrame`, we could have achieved the same result with *`DataFrame.assign()`*. To concatenate an arbitrary number of pandas objects (`DataFrame` or `Series`), use `concat`.

If unnamed `Series` are passed they will be numbered consecutively.

```
In [19]: s2 = pd.Series(['_0', '_1', '_2', '_3'])

In [20]: result = pd.concat([df1, s2, s2, s2], axis=1)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

0	_0
1	_1
2	_2
3	_3

Result

	A	B	C	D	0	1	2
0	A0	B0	C0	D0	_0	_0	_0
1	A1	B1	C1	D1	_1	_1	_1
2	A2	B2	C2	D2	_2	_2	_2
3	A3	B3	C3	D3	_3	_3	_3

Passing `ignore_index=True` will drop all name references.

```
In [21]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s1

	X
0	X0
1	X1
2	X2
3	X3

Result

	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

More concatenating with group keys

A fairly common use of the `keys` argument is to override the column names when creating a new `DataFrame` based on existing `Series`. Notice how the default behaviour consists on letting the resulting `DataFrame` inherit the parent `Series`' name, when these existed.

```
In [22]: s3 = pd.Series([0, 1, 2, 3], name='foo')

In [23]: s4 = pd.Series([0, 1, 2, 3])

In [24]: s5 = pd.Series([0, 1, 4, 5])

In [25]: pd.concat([s3, s4, s5], axis=1)
Out[25]:
   foo  0  1
0    0  0  0
1    1  1  1
2    2  2  4
3    3  3  5
```

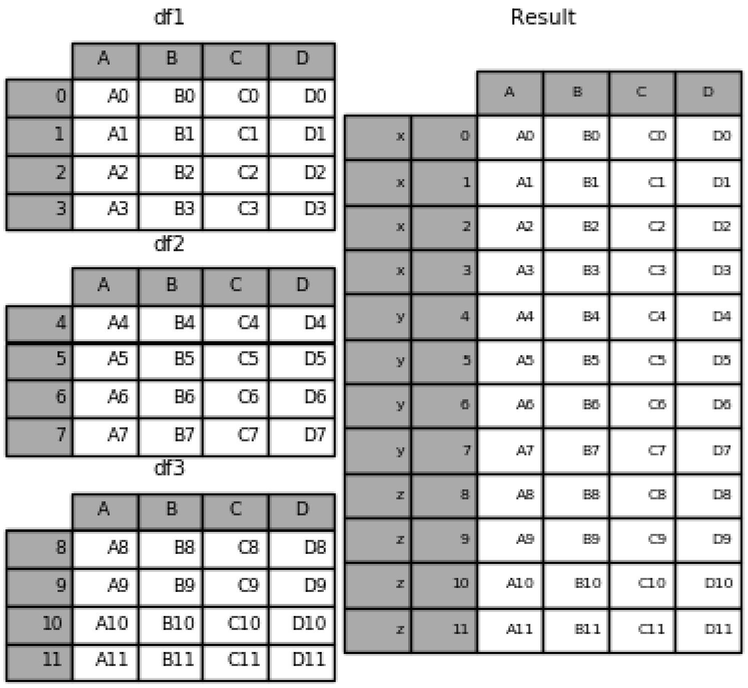
Through the `keys` argument we can override the existing column names.

```
In [26]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
Out[26]:
   red  blue  yellow
0    0     0       0
1    1     1       1
2    2     2       4
3    3     3       5
```

Let's consider a variation of the very first example presented:

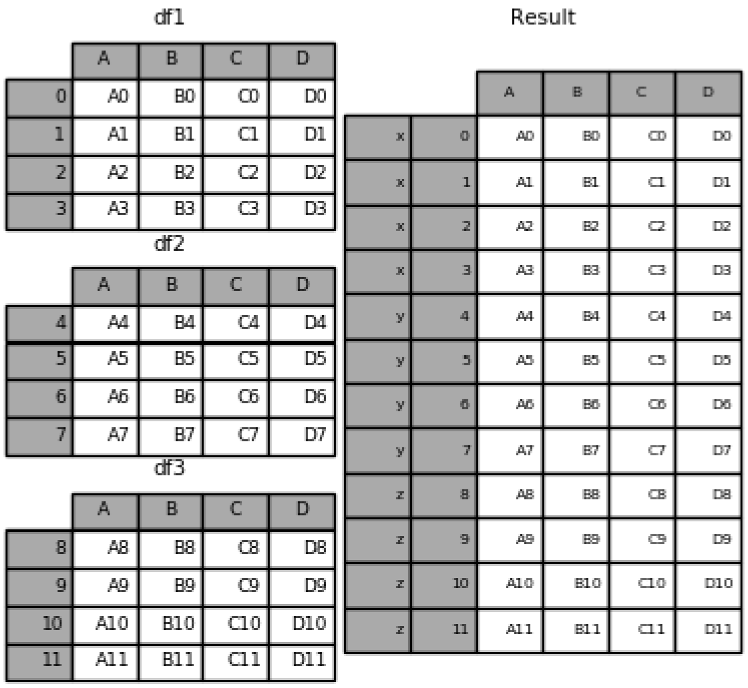
```
In [27]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

```

```

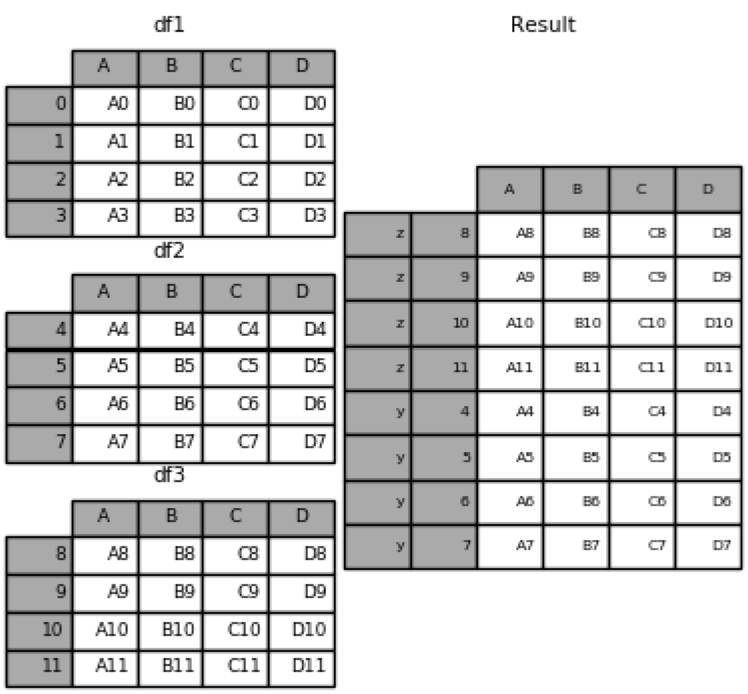


You can also pass a dict to `concat` in which case the dict keys will be used for the `keys` argument (unless other keys are specified):

```
In [28]: pieces = {'x': df1, 'y': df2, 'z': df3}

In [29]: result = pd.concat(pieces)
```




```
In [30]: result = pd.concat(pieces, keys=['z', 'y'])
```

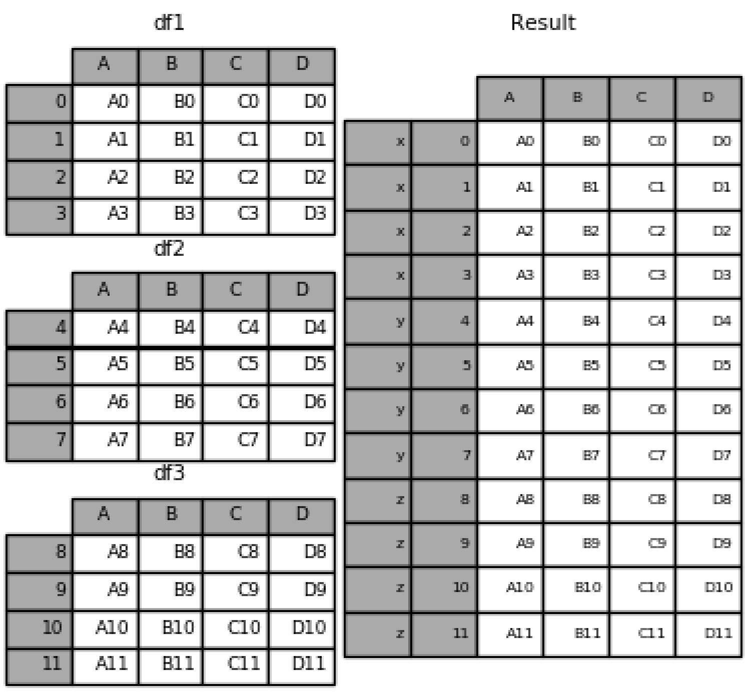


The MultiIndex created has levels that are constructed from the passed keys and the index of the `DataFrame` pieces:

```
In [31]: result.index.levels
Out[31]: FrozenList([['z', 'y'], [4, 5, 6, 7, 8, 9, 10, 11]])
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

```
In [32]: result = pd.concat(pieces, keys=['x', 'y', 'z'],
   ....:                    levels=[['z', 'y', 'x', 'w']],
   ....:                    names=['group_key'])
   ....:
```




```
In [33]: result.index.levels
Out[33]: FrozenList([['z', 'y', 'x', 'w'], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
```

This is fairly esoteric, but it is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

Appending rows to a DataFrame

While not especially efficient (since a new object must be created), you can append a single row to a `DataFrame` by passing a `Series` or dict to `append`, which returns a new `DataFrame` as above.

```
In [34]: s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])

In [35]: result = df1.append(s2, ignore_index=True)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

s2

A	X0
B	X1
C	X2
D	X3

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	X0	X1	X2	X3

You should use `ignore_index` with this method to instruct DataFrame to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed DataFrame and append or concatenate those objects.

You can also pass a list of dicts or Series:

```
In [36]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
   ....:          {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]
   ....:

In [37]: result = df1.append(dicts, ignore_index=True, sort=False)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

dicts

	A	B	C	X	Y
0	1	2	3	4.0	NaN
1	5	6	7	NaN	8.0

Result

	A	B	C	D	X	Y
0	A0	B0	C0	D0	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN
4	1	2	3	NaN	4.0	NaN
5	5	6	7	NaN	NaN	8.0

4.4.2 Database-style DataFrame or named Series joining/merging

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R). The reason for this is careful algorithmic design and the internal layout of the data in `DataFrame`.

See the *cookbook* for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a *comparison with SQL*.

pandas provides a single function, *merge()*, as the entry point for all standard database join operations between `DataFrame` or named `Series` objects:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
```

- `left`: A DataFrame or named Series object.
- `right`: Another DataFrame or named Series object.
- `on`: Column or index level names to join on. Must be found in both the left and right DataFrame and/or Series objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the DataFrames and/or Series will be inferred to be the join keys.
- `left_on`: Columns or index levels from the left DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.
- `right_on`: Columns or index levels from the right DataFrame or Series to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the DataFrame or Series.
- `left_index`: If `True`, use the index (row labels) from the left DataFrame or Series as its join key(s). In the case of a DataFrame or Series with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame or Series.
- `right_index`: Same usage as `left_index` for the right DataFrame or Series
- `how`: One of `'left'`, `'right'`, `'outer'`, `'inner'`. Defaults to `inner`. See below for more detailed description of each method.
- `sort`: Sort the result DataFrame by the join keys in lexicographical order. Defaults to `True`, setting to `False` will improve performance substantially in many cases.
- `suffixes`: A tuple of string suffixes to apply to overlapping columns. Defaults to `('_x', '_y')`.
- `copy`: Always copy data (default `True`) from the passed DataFrame or named Series objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.
- `indicator`: Add a column to the output DataFrame called `_merge` with information on the source of each row. `_merge` is Categorical-type and takes on a value of `left_only` for observations whose merge key only appears in `'left'` DataFrame or Series, `right_only` for observations whose merge key only appears in `'right'` DataFrame or Series, and `both` if the observation's merge key is found in both.
- `validate` : string, default None. If specified, checks if merge is of specified type.
  - "one_to_one" or "1:1": checks if merge keys are unique in both left and right datasets.
  - "one_to_many" or "1:m": checks if merge keys are unique in left dataset.
  - "many_to_one" or "m:1": checks if merge keys are unique in right dataset.
  - "many_to_many" or "m:m": allowed, but does not result in checks.

  New in version 0.21.0.

Note: Support for specifying index levels as the `on`, `left_on`, and `right_on` parameters was added in version 0.23.0. Support for merging named `Series` objects was added in version 0.24.0.

The return type will be the same as `left`. If `left` is a `DataFrame` or named `Series` and `right` is a subclass of `DataFrame`, the return type will still be `DataFrame`.

`merge` is a function in the pandas namespace, and it is also available as a `DataFrame` instance method *`merge()`*, with the calling `DataFrame` being implicitly considered the left object in the join.

The related `join()` method, uses `merge` internally for the index-on-index (by default) and column(s)-on-index join. If you are joining on index only, you may wish to use `DataFrame.join` to save yourself some typing.

Brief primer on merge methods (relational algebra)

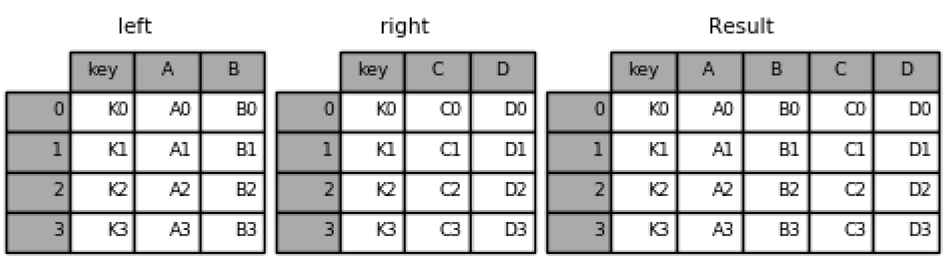
Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (`DataFrame` objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two `DataFrame` objects on their indexes (which must contain unique values).
- **many-to-one** joins: for example when joining an index (unique) to one or more columns in a different `DataFrame`.
- **many-to-many** joins: joining columns on columns.

Note: When joining columns on columns (potentially a many-to-many join), any indexes on the passed `DataFrame` objects **will be discarded**.

It is worth spending some time understanding the result of the **many-to-many** join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the **Cartesian product** of the associated data. Here is a very basic example with one unique key combination:

```
In [38]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:

In [39]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [40]: result = pd.merge(left, right, on='key')
```



left

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

right

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

Result

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

Here is a more complicated example with multiple join keys. Only the keys appearing in `left` and `right` are present (the intersection), since `how='inner'` by default.

```
In [41]: left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1'],
   ....:                      'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3']})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [42]: right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
   ....:                       'key2': ['K0', 'K0', 'K0', 'K0'],
   ....:                       'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']})
   ....:

In [43]: result = pd.merge(left, right, on=['key1', 'key2'])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

The `how` argument to `merge` specifies how to determine which keys are to be included in the resulting table. If a key combination **does not appear** in either the left or right tables, the values in the joined table will be `NA`. Here is a summary of the `how` options and their SQL equivalent names:

Merge method	SQL Join Name	Description
`left`	`LEFT OUTER JOIN`	Use keys from left frame only
`right`	`RIGHT OUTER JOIN`	Use keys from right frame only
`outer`	`FULL OUTER JOIN`	Use union of keys from both frames
`inner`	`INNER JOIN`	Use intersection of keys from both frames

```
In [44]: result = pd.merge(left, right, how='left', on=['key1', 'key2'])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

```
In [45]: result = pd.merge(left, right, how='right', on=['key1', 'key2'])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K2	K0	NaN	NaN	C3	D3

```
In [46]: result = pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

```
In [47]: result = pd.merge(left, right, how='inner', on=['key1', 'key2'])
```

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

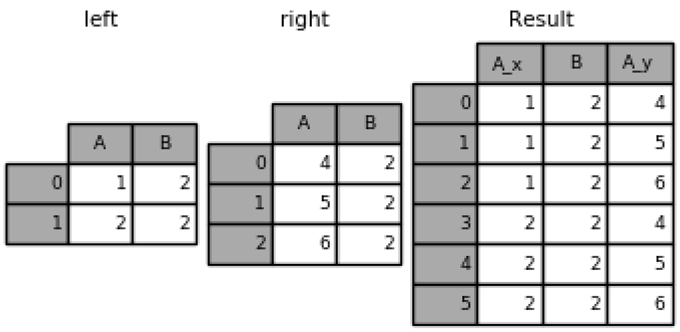
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

Here is another example with duplicate join keys in DataFrames:

```
In [48]: left = pd.DataFrame({'A': [1, 2], 'B': [2, 2]})

In [49]: right = pd.DataFrame({'A': [4, 5, 6], 'B': [2, 2, 2]})

In [50]: result = pd.merge(left, right, on='B', how='outer')
```



left

	A	B
0	1	2
1	2	2

right

	A	B
0	4	2
1	5	2
2	6	2

Result

	A_x	B	A_y
0	1	2	4
1	1	2	5
2	1	2	6
3	2	2	4
4	2	2	5
5	2	2	6

Warning: Joining / merging on duplicate keys can cause a returned frame that is the multiplication of the row dimensions, which may result in memory overflow. It is the user' s responsibility to manage duplicate values in keys before joining large DataFrames.

Checking for duplicate keys

New in version 0.21.0.

Users can use the `validate` argument to automatically check whether there are unexpected duplicates in their merge keys. Key uniqueness is checked before merge operations and so should protect against memory overflows. Checking key uniqueness is also a good way to ensure user data structures are as expected.

In the following example, there are duplicate values of `B` in the right `DataFrame`. As this is not a one-to-one merge – as specified in the `validate` argument – an exception will be raised.

```
In [51]: left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})

In [52]: right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2, 2]})
```

```
In [53]: result = pd.merge(left, right, on='B', how='outer', validate="one_to_one")
...
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right `DataFrame` but wants to ensure there are no duplicates in the left DataFrame, one can use the `validate='one_to_many'` argument instead, which will not raise an exception.

```
In [53]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[53]:
   A_x  B  A_y
0    1  1  NaN
1    2  2  4.0
2    2  2  5.0
3    2  2  6.0
```

The merge indicator

merge() accepts the argument `indicator`. If `True`, a Categorical-type column called `_merge` will be added to the output object that takes on values:

Observation Origin	`_merge` value
Merge key only in `'left'` frame	`left_only`
Merge key only in `'right'` frame	`right_only`
Merge key in both frames	`both`

```
In [54]: df1 = pd.DataFrame({'col1': [0, 1], 'col_left': ['a', 'b']})

In [55]: df2 = pd.DataFrame({'col1': [1, 2, 2], 'col_right': [2, 2, 2]})

In [56]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[56]:
   col1 col_left  col_right      _merge
0     0        a        NaN   left_only
1     1        b        2.0        both
2     2      NaN        2.0  right_only
3     2      NaN        2.0  right_only
```

The `indicator` argument will also accept string arguments, in which case the indicator function will use the value of the passed string as the name for the indicator column.

```
In [57]: pd.merge(df1, df2, on='col1', how='outer', indicator='indicator_column')
Out[57]:
   col1 col_left  col_right indicator_column
0     0        a        NaN        left_only
1     1        b        2.0             both
2     2      NaN        2.0       right_only
3     2      NaN        2.0       right_only
```

Merge Dtypes

New in version 0.19.0.

Merging will preserve the dtype of the join keys.

```
In [58]: left = pd.DataFrame({'key': [1], 'v1': [10]})

In [59]: left
Out[59]:
   key  v1
0    1  10

In [60]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [61]: right
Out[61]:
   key  v1
0    1  20
1    2  30
```

We are able to preserve the join keys:

```
In [62]: pd.merge(left, right, how='outer')
Out[62]:
   key  v1
0    1  10
```

(continues on next page)

(continued from previous page)

```
1    1  20
2    2  30

In [63]: pd.merge(left, right, how='outer').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[63]:
key     int64
v1      int64
dtype: object
```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.

```
In [64]: pd.merge(left, right, how='outer', on='key')
Out[64]:
   key  v1_x  v1_y
0    1  10.0    20
1    2   NaN    30

In [65]: pd.merge(left, right, how='outer', on='key').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[65]:
key       int64
v1_x    float64
v1_y      int64
dtype: object
```

New in version 0.20.0.

Merging will preserve `category` dtypes of the mergands. See also the section on *categoricals*.

The left frame.

```
In [66]: from pandas.api.types import CategoricalDtype

In [67]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [68]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))

In [69]: left = pd.DataFrame({'X': X,
   ....:                      'Y': np.random.choice(['one', 'two', 'three'],
   ....:                                            size=(10,))})
   ....:

In [70]: left
Out[70]:
     X      Y
0  bar    one
1  foo    one
2  foo  three
3  bar  three
4  foo    one
5  bar    one
6  bar  three
7  bar  three
8  bar  three
9  foo  three

In [71]: left.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
X    category
Y      object
dtype: object
```

The right frame.

```
In [72]: right = pd.DataFrame({'X': pd.Series(['foo', 'bar'],
   ....:                                      dtype=CategoricalDtype(['foo', 'bar'])),
   ....:                         'Z': [1, 2]})
   ....:

In [73]: right
Out[73]:
     X  Z
0  foo  1
1  bar  2

In [74]: right.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[74]:
X    category
Z       int64
dtype: object
```

The merged result:

```
In [75]: result = pd.merge(left, right, how='outer')

In [76]: result
Out[76]:
     X      Y  Z
0  bar    one  2
1  bar  three  2
2  bar    one  2
3  bar  three  2
4  bar  three  2
5  bar  three  2
6  foo    one  1
7  foo  three  1
8  foo    one  1
9  foo  three  1

In [77]: result.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
X    category
Y      object
Z       int64
dtype: object
```

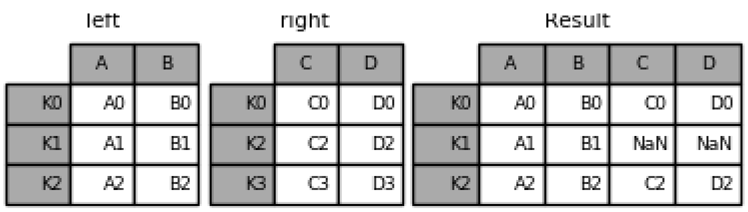
Note: The category dtypes must be *exactly* the same, meaning the same categories and the ordered attribute. Otherwise the result will coerce to `object` dtype.

Note: Merging on `category` dtypes that are the same can be quite performant compared to `object` dtype merging.

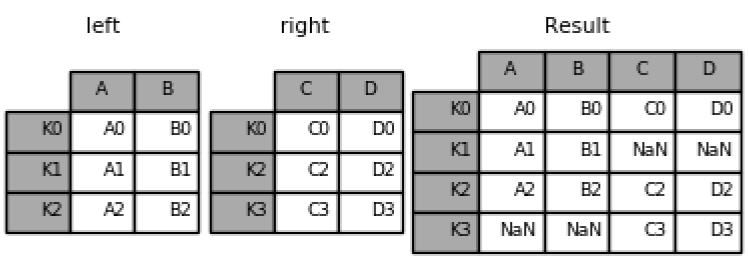
Joining on index

DataFrame.join() is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Here is a very basic example:

```
In [78]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                     index=['K0', 'K1', 'K2'])
   ....:

In [79]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D2', 'D3']},
   ....:                      index=['K0', 'K2', 'K3'])
   ....:

In [80]: result = left.join(right)
```

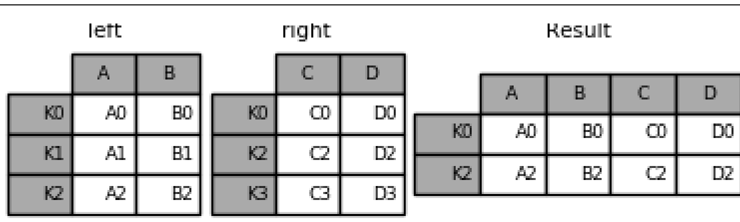



```
In [81]: result = left.join(right, how='outer')
```



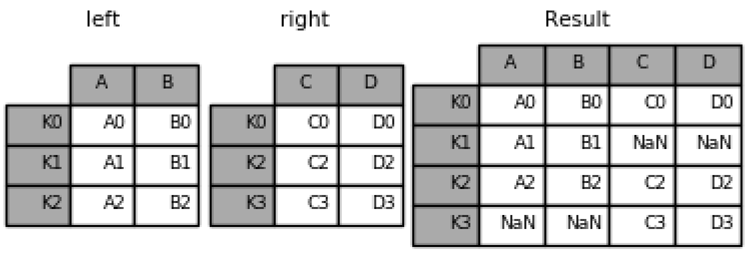
The same as above, but with how='inner'.

```
In [82]: result = left.join(right, how='inner')
```

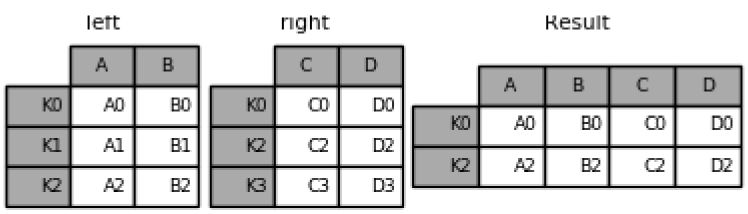


The data alignment here is on the indexes (row labels). This same behavior can be achieved using merge plus additional arguments instructing it to use the indexes:

```
In [83]: result = pd.merge(left, right, left_index=True, right_index=True, how='outer
↪')
```




```
In [84]: result = pd.merge(left, right, left_index=True, right_index=True, how='inner
↪');
```



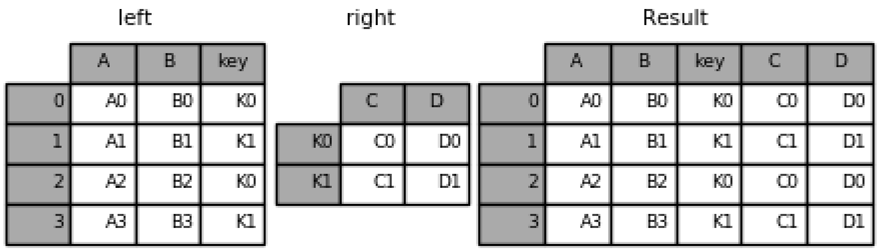
Joining key columns on an index

`join()` takes an optional `on` argument which may be a column or multiple column names, which specifies that the passed `DataFrame` is to be aligned on that column in the `DataFrame`. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
pd.merge(left, right, left_on=key_or_keys, right_index=True,
      how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the `DataFrame`'s is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [85]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [86]: right = pd.DataFrame({'C': ['C0', 'C1'],
   ....:                       'D': ['D0', 'D1']},
   ....:                      index=['K0', 'K1'])
   ....:

In [87]: result = left.join(right, on='key')
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

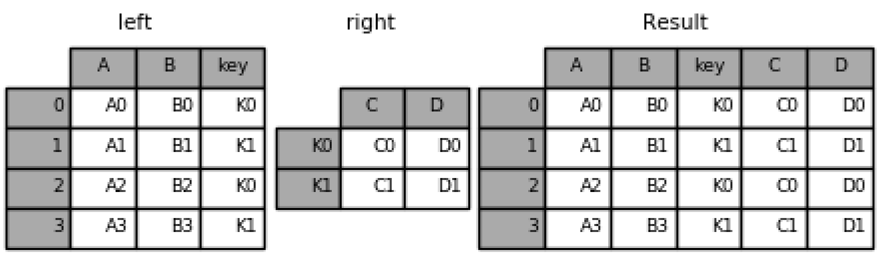
right

	C	D
K0	C0	D0
K1	C1	D1

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

```
In [88]: result = pd.merge(left, right, left_on='key', right_index=True,
   ....:                   how='left', sort=False);
   ....:
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

right

	C	D
K0	C0	D0
K1	C1	D1

Result

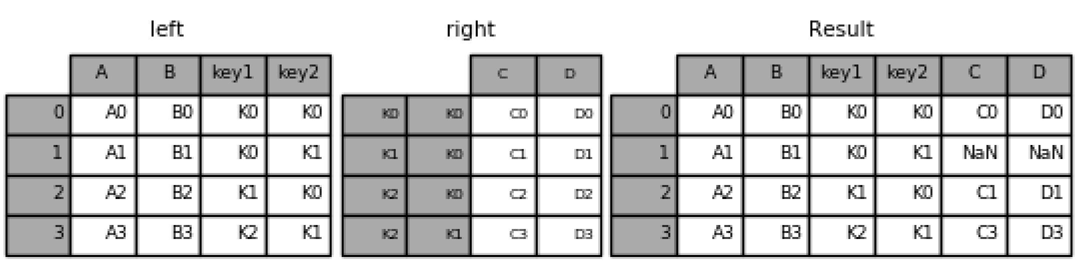
	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

To join on multiple keys, the passed DataFrame must have a MultiIndex:

```
In [89]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   ....:                      'key1': ['K0', 'K0', 'K1', 'K2'],
   ....:                      'key2': ['K0', 'K1', 'K0', 'K1']})
   ....:

In [90]: index = pd.MultiIndex.from_tuples([('K0', 'K0'), ('K1', 'K0'),
   ....:                                   ('K2', 'K0'), ('K2', 'K1')])
   ....:

In [91]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                      index=index)
   ....:
```

Now this can be joined by passing the two key column names:

```
In [92]: result = left.join(right, on=['key1', 'key2'])
```



left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

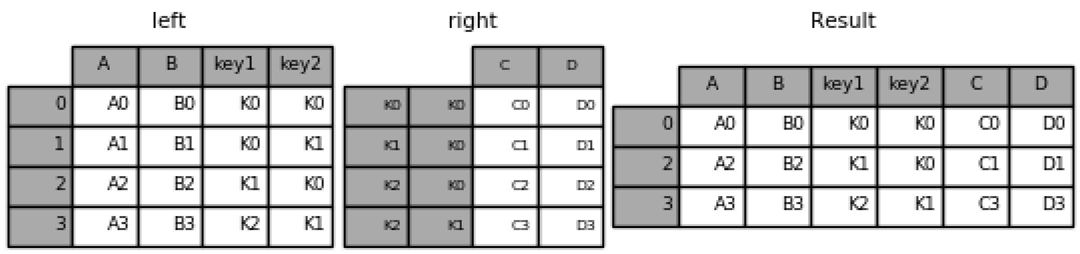
		C	D
K0	K0	C0	D0
K1	K0	C1	D1
K2	K0	C2	D2
K2	K1	C3	D3

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A3	B3	K2	K1	C3	D3

The default for `DataFrame.join` is to perform a left join (essentially a "VLOOKUP" operation, for Excel users), which uses only the keys found in the calling DataFrame. Other join types, for example inner join, can be just as easily performed:

```
In [93]: result = left.join(right, on=['key1', 'key2'], how='inner')
```

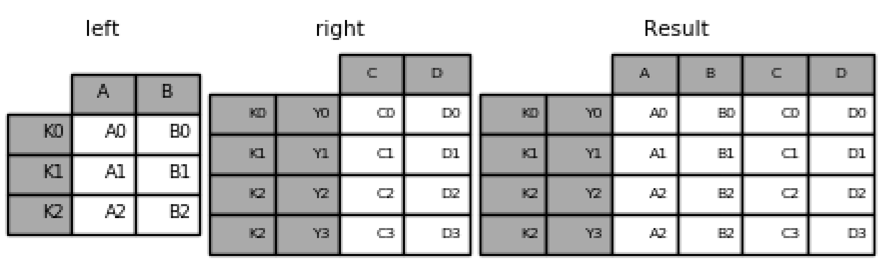


As you can see, this drops any rows where there was no match.

Joining a single Index to a MultiIndex

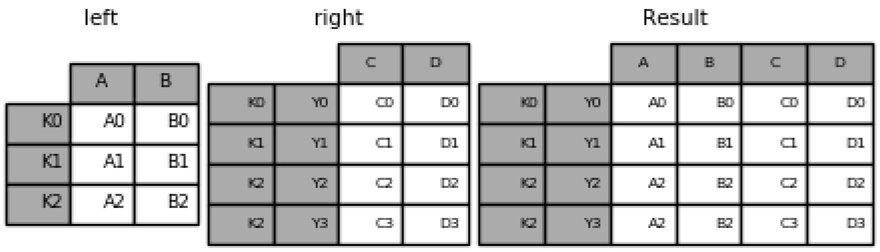
You can join a singly-indexed `DataFrame` with a level of a MultiIndexed `DataFrame`. The level will match on the name of the index of the singly-indexed frame against a level name of the MultiIndexed frame.

```
In [94]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   ....:                      'B': ['B0', 'B1', 'B2']},
   ....:                      index=pd.Index(['K0', 'K1', 'K2'], name='key'))
   ....:

In [95]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   ....:                                   ('K2', 'Y2'), ('K2', 'Y3')],
   ....:                                    names=['key', 'Y'])
   ....:

In [96]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   ....:                       index=index)
   ....:

In [97]: result = left.join(right, how='inner')
```



This is equivalent but less verbose and more memory efficient / faster than this.

```
In [98]: result = pd.merge(left.reset_index(), right.reset_index(),
   ....:        on=['key'], how='inner').set_index(['key','Y'])
   ....:
```



Joining with two MultiIndexes

This is supported in a limited way, provided that the index for the right argument is completely used in the join, and is a subset of the indices in the left argument, as in this example:

```
In [99]: leftindex = pd.MultiIndex.from_product([list('abc'), list('xy'), [1, 2]],
   ....:                                        names=['abc', 'xy', 'num'])
   ....:

In [100]: left = pd.DataFrame({'v1': range(12)}, index=leftindex)

In [101]: left
Out[101]:
            v1
abc xy num
a   x  1     0
       2     1
    y  1     2
       2     3
b   x  1     4
       2     5
    y  1     6
       2     7
c   x  1     8
       2     9
    y  1    10
       2    11

In [102]: rightindex = pd.MultiIndex.from_product([list('abc'), list('xy')],
   .....:                                         names=['abc', 'xy'])
   .....:

In [103]: right = pd.DataFrame({'v2': [100 * i for i in range(1, 7)]},␣
↪index=rightindex)

In [104]: right
Out[104]:
         v2
abc xy
a   x   100
    y   200
b   x   300
    y   400
c   x   500
    y   600
```

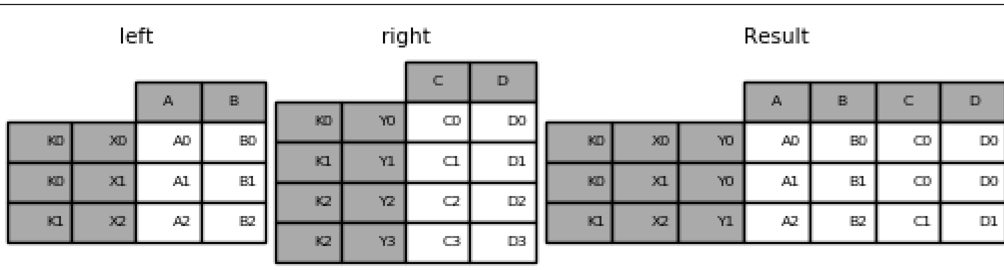
(continues on next page)

(continued from previous page)

```
In [105]: left.join(right, on=['abc', 'xy'], how='inner')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            v1   v2
abc xy num
a   x  1     0  100
       2     1  100
    y  1     2  200
       2     3  200
b   x  1     4  300
       2     5  300
    y  1     6  400
       2     7  400
c   x  1     8  500
       2     9  500
    y  1    10  600
       2    11  600
```

If that condition is not satisfied, a join with two multi-indexes can be done using the following code.

```
In [106]: leftindex = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
   .....:                                        ('K1', 'X2')],
   .....:                                       names=['key', 'X'])
   .....: 

In [107]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
   .....:                      'B': ['B0', 'B1', 'B2']},
   .....:                     index=leftindex)
   .....: 

In [108]: rightindex = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
   .....:                                         ('K2', 'Y2'), ('K2', 'Y3')],
   .....:                                        names=['key', 'Y'])
   .....: 

In [109]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   .....:                       'D': ['D0', 'D1', 'D2', 'D3']},
   .....:                      index=rightindex)
   .....: 

In [110]: result = pd.merge(left.reset_index(), right.reset_index(),
   .....:                   on=['key'], how='inner').set_index(['key', 'X', 'Y'])
   .....: 
```

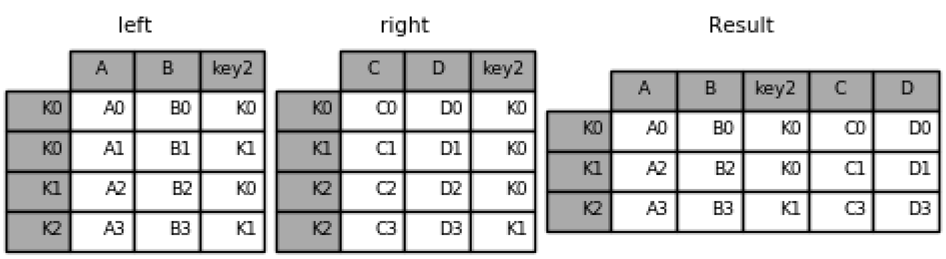


Merging on a combination of columns and index levels

New in version 0.23.

Strings passed as the `on`, `left_on`, and `right_on` parameters may refer to either column names or index level names. This enables merging `DataFrame` instances on a combination of index levels and columns without resetting indexes.

```
In [111]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [112]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   .....:                      'B': ['B0', 'B1', 'B2', 'B3'],
   .....:                      'key2': ['K0', 'K1', 'K0', 'K1']},
   .....:                     index=left_index)
   .....:

In [113]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [114]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   .....:                       'D': ['D0', 'D1', 'D2', 'D3'],
   .....:                       'key2': ['K0', 'K0', 'K0', 'K1']},
   .....:                      index=right_index)
   .....:

In [115]: result = left.merge(right, on=['key1', 'key2'])
```



left

	A	B	key2
K0	A0	B0	K0
K0	A1	B1	K1
K1	A2	B2	K0
K2	A3	B3	K1

right

	C	D	key2
K0	C0	D0	K0
K1	C1	D1	K0
K2	C2	D2	K0
K2	C3	D3	K1

Result

	A	B	key2	C	D
K0	A0	B0	K0	C0	D0
K1	A2	B2	K0	C1	D1
K2	A3	B3	K1	C3	D3

Note: When DataFrames are merged on a string that matches an index level in both frames, the index level is preserved as an index level in the resulting DataFrame.

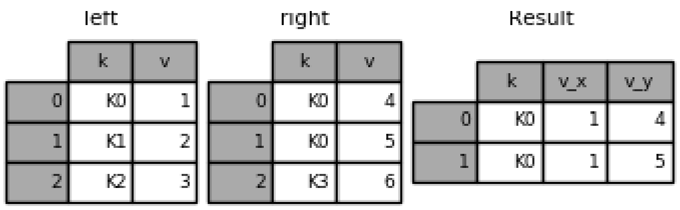
Note: When DataFrames are merged using only some of the levels of a *MultiIndex*, the extra levels will be dropped from the resulting merge. In order to preserve those levels, use `reset_index` on those level names to move those levels to columns prior to doing the merge.

Note: If a string matches both a column name and an index level name, then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

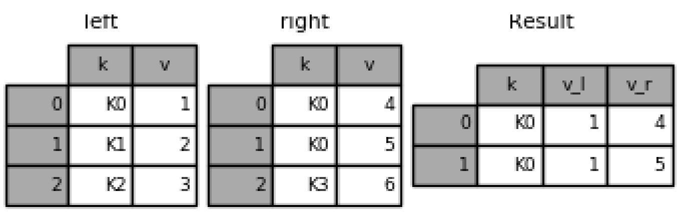
Overlapping value columns

The merge `suffixes` argument takes a tuple of list of strings to append to overlapping column names in the input `DataFrames` to disambiguate the result columns:

```
In [116]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})

In [117]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})

In [118]: result = pd.merge(left, right, on='k')
```

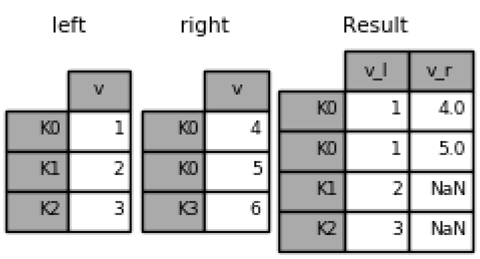



```
In [119]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```



`DataFrame.join()` has `lsuffix` and `rsuffix` arguments which behave similarly.

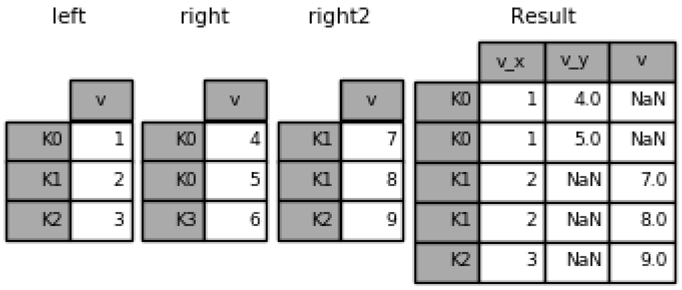
```
In [120]: left = left.set_index('k')

In [121]: right = right.set_index('k')

In [122]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```



Joining multiple DataFrame or Panel objects

A list or tuple of `DataFrames` can also be passed to `join()` to join them together on their indexes.

```
In [123]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])

In [124]: result = left.join([right, right2])
```



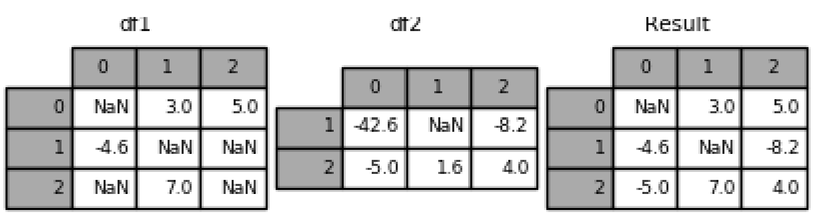
Merging together values within Series or DataFrame columns

Another fairly common situation is to have two like-indexed (or similarly indexed) `Series` or `DataFrame` objects and wanting to "patch" values in one object from values for matching indices in the other. Here is an example:

```
In [125]: df1 = pd.DataFrame([[np.nan, 3., 5.], [-4.6, np.nan, np.nan],
   .....:                     [np.nan, 7., np.nan]])
   .....:

In [126]: df2 = pd.DataFrame([[-42.6, np.nan, -8.2], [-5., 1.6, 4]],
   .....:                     index=[1, 2])
   .....:
```

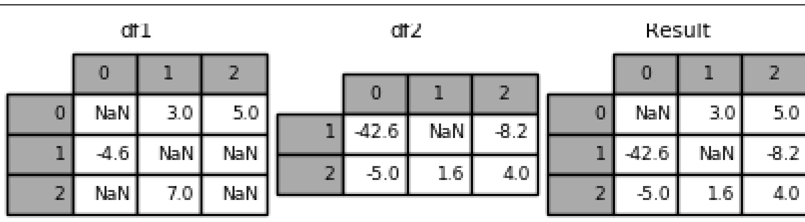
For this, use the *combine_first()* method:

```
In [127]: result = df1.combine_first(df2)
```



Note that this method only takes values from the right `DataFrame` if they are missing in the left `DataFrame`. A related method, *update()*, alters non-NA values in place:

```
In [128]: df1.update(df2)
```



4.4.3 Timeseries friendly merging

Merging Ordered Data

A *merge_ordered()* function allows combining time series and other ordered data. In particular it has an optional `fill_method` keyword to fill/interpolate missing data:

```
In [129]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
   .....:                      'lv': [1, 2, 3, 4],
   .....:                      's': ['a', 'b', 'c', 'd']})
   .....:

In [130]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
   .....:                       'rv': [1, 2, 3]})
   .....:

In [131]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')
Out[131]:
     k   lv  s   rv
0   K0  1.0  a  NaN
1   K1  1.0  a  1.0
2   K2  1.0  a  2.0
3   K4  1.0  a  3.0
4   K1  2.0  b  1.0
5   K2  2.0  b  2.0
6   K4  2.0  b  3.0
7   K1  3.0  c  1.0
8   K2  3.0  c  2.0
9   K4  3.0  c  3.0
10  K1  NaN  d  1.0
11  K2  4.0  d  2.0
12  K4  4.0  d  3.0
```

Merging AsOf

New in version 0.19.0.

A *merge_asof()* is similar to an ordered left-join except that we match on nearest key rather than equal keys. For each row in the `left DataFrame`, we select the last row in the `right DataFrame` whose `on` key is less than the left's key. Both DataFrames must be sorted by the key.

Optionally an asof merge can perform a group-wise merge. This matches the `by` key equally, in addition to the nearest match on the `on` key.

For example; we might have `trades` and `quotes` and we want to `asof` merge them.

```
In [132]: trades = pd.DataFrame({
   .....:     'time': pd.to_datetime(['20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.038',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.048']),
   .....:     'ticker': ['MSFT', 'MSFT',
   .....:                'GOOG', 'GOOG', 'AAPL'],
   .....:     'price': [51.95, 51.95,
   .....:               720.77, 720.92, 98.00],
   .....:     'quantity': [75, 155,
```

(continues on next page)

(continued from previous page)

```
   .....:                    100, 100, 100]},
   .....:     columns=['time', 'ticker', 'price', 'quantity'])
   .....:

In [133]: quotes = pd.DataFrame({
   .....:     'time': pd.to_datetime(['20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.023',
   .....:                             '20160525 13:30:00.030',
   .....:                             '20160525 13:30:00.041',
   .....:                             '20160525 13:30:00.048',
   .....:                             '20160525 13:30:00.049',
   .....:                             '20160525 13:30:00.072',
   .....:                             '20160525 13:30:00.075']),
   .....:     'ticker': ['GOOG', 'MSFT', 'MSFT',
   .....:                'MSFT', 'GOOG', 'AAPL', 'GOOG',
   .....:                'MSFT'],
   .....:     'bid': [720.50, 51.95, 51.97, 51.99,
   .....:             720.50, 97.99, 720.50, 52.01],
   .....:     'ask': [720.93, 51.96, 51.98, 52.00,
   .....:             720.93, 98.01, 720.88, 52.03]},
   .....:     columns=['time', 'ticker', 'bid', 'ask'])
   .....:
```

```
In [134]: trades
Out[134]:
                     time ticker   price  quantity
0 2016-05-25 13:30:00.023   MSFT   51.95        75
1 2016-05-25 13:30:00.038   MSFT   51.95       155
2 2016-05-25 13:30:00.048   GOOG  720.77       100
3 2016-05-25 13:30:00.048   GOOG  720.92       100
4 2016-05-25 13:30:00.048   AAPL   98.00       100

In [135]: quotes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                     time ticker     bid     ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT   51.95   51.96
2 2016-05-25 13:30:00.030   MSFT   51.97   51.98
3 2016-05-25 13:30:00.041   MSFT   51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL   97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT   52.01   52.03
```

By default we are taking the asof of the quotes.

```
In [136]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker')
   .....:
Out[136]:
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
```

(continues on next page)

(continued from previous page)

```
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within `2ms` between the quote time and the trade time.

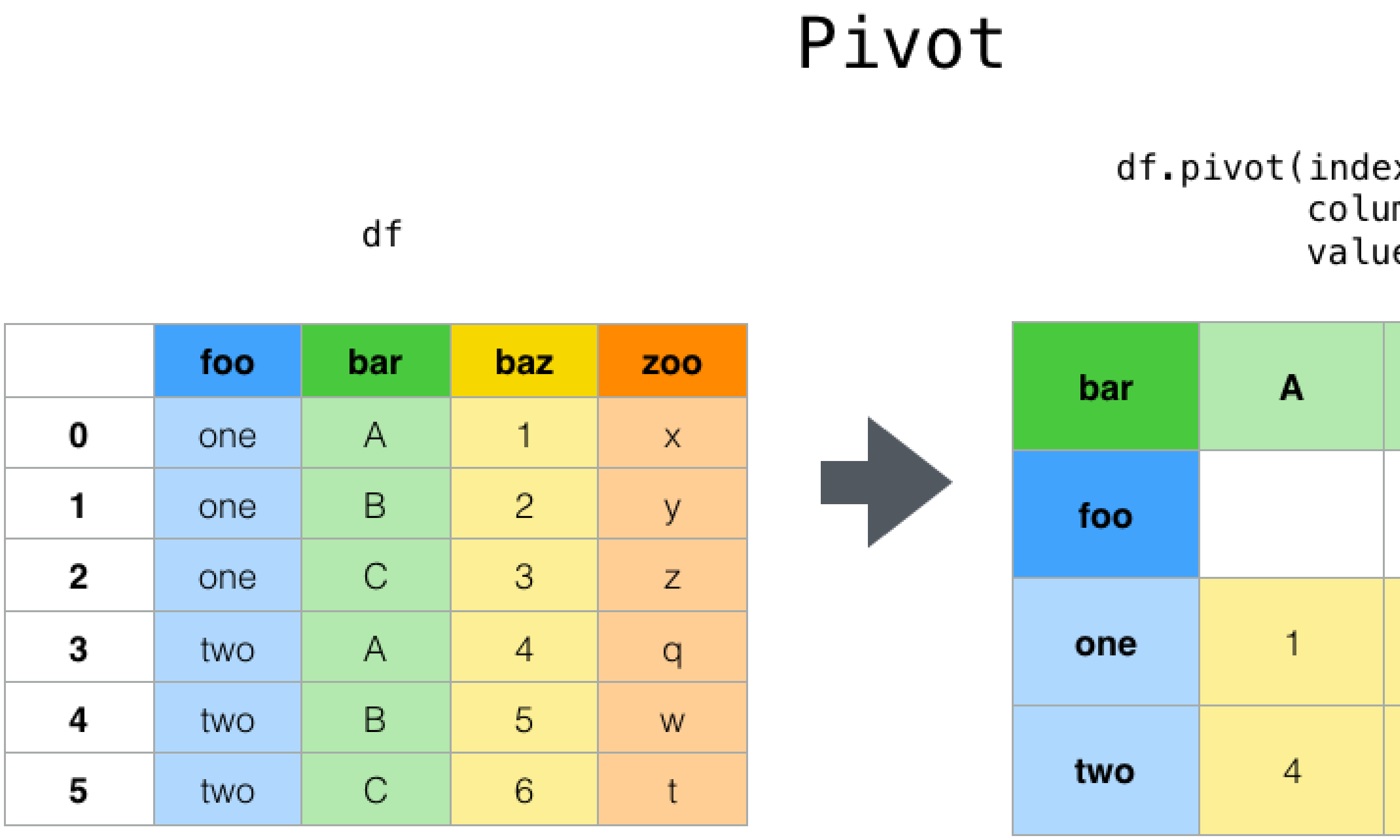
```
In [137]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('2ms'))
   .....:
Out[137]:
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155     NaN     NaN
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within `10ms` between the quote time and the trade time and we exclude exact matches on time. Note that though we exclude the exact matches (of the quotes), prior quotes **do** propagate to that point in time.

```
In [138]: pd.merge_asof(trades, quotes,
   .....:               on='time',
   .....:               by='ticker',
   .....:               tolerance=pd.Timedelta('10ms'),
   .....:               allow_exact_matches=False)
   .....:
Out[138]:
                     time ticker   price  quantity    bid    ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75    NaN    NaN
1 2016-05-25 13:30:00.038   MSFT   51.95       155  51.97  51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100    NaN    NaN
3 2016-05-25 13:30:00.048   GOOG  720.92       100    NaN    NaN
4 2016-05-25 13:30:00.048   AAPL   98.00       100    NaN    NaN
```

4.5 Reshaping and Pivot Tables

4.5.1 Reshaping by pivoting DataFrame objects



Data is often stored in so-called "stacked" or "record" format:

```
In [1]: df
Out[1]:
         date variable     value
0  2000-01-03        A  0.469112
1  2000-01-04        A -0.282863
2  2000-01-05        A -1.509059
3  2000-01-03        B -1.135632
4  2000-01-04        B  1.212112
5  2000-01-05        B -0.173215
6  2000-01-03        C  0.119209
7  2000-01-04        C -1.044236
8  2000-01-05        C -0.861849
9  2000-01-03        D -2.104569
10 2000-01-04        D -0.494929
11 2000-01-05        D  1.071804
```

For the curious here is how the above `DataFrame` was created:

```
import pandas.util.testing as tm

tm.N = 3


def unpivot(frame):
    N, K = frame.shape
    data = {'value': frame.to_numpy().ravel('F'),
            'variable': np.asarray(frame.columns).repeat(N),
            'date': np.tile(np.asarray(frame.index), K)}
    return pd.DataFrame(data, columns=['date', 'variable', 'value'])


df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable A we could do:

```
In [2]: df[df['variable'] == 'A']
Out[2]:
        date variable     value
0 2000-01-03        A  0.469112
1 2000-01-04        A -0.282863
2 2000-01-05        A -1.509059
```

But suppose we wish to do time series operations with the variables. A better representation would be where the `columns` are the unique variables and an `index` of dates identifies individual observations. To reshape the data into this form, we use the `DataFrame.pivot()` method (also implemented as a top level function `pivot()`):

```
In [3]: df.pivot(index='date', columns='variable', values='value')
Out[3]:
variable           A         B         C         D
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804
```

If the `values` argument is omitted, and the input `DataFrame` has more than one column of values which are not used as column or index inputs to `pivot`, then the resulting "pivoted" `DataFrame` will have *hierarchical columns* whose topmost level indicates the respective value column:

```
In [4]: df['value2'] = df['value'] * 2

In [5]: pivoted = df.pivot(index='date', columns='variable')

In [6]: pivoted
Out[6]:
               value                                       value2                     ␣
↪
variable           A         B         C         D         A         B         C    ␣
↪   D
date                                                                                  ␣
↪
2000-01-03  0.469112 -1.135632  0.119209 -2.104569  0.938225 -2.271265  0.238417 -4.
↪209138
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929 -0.565727  2.424224 -2.088472 -0.
↪989859
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804 -3.018117 -0.346429 -1.723698  2.
↪143608
```

(continues on next page)

(continued from previous page)

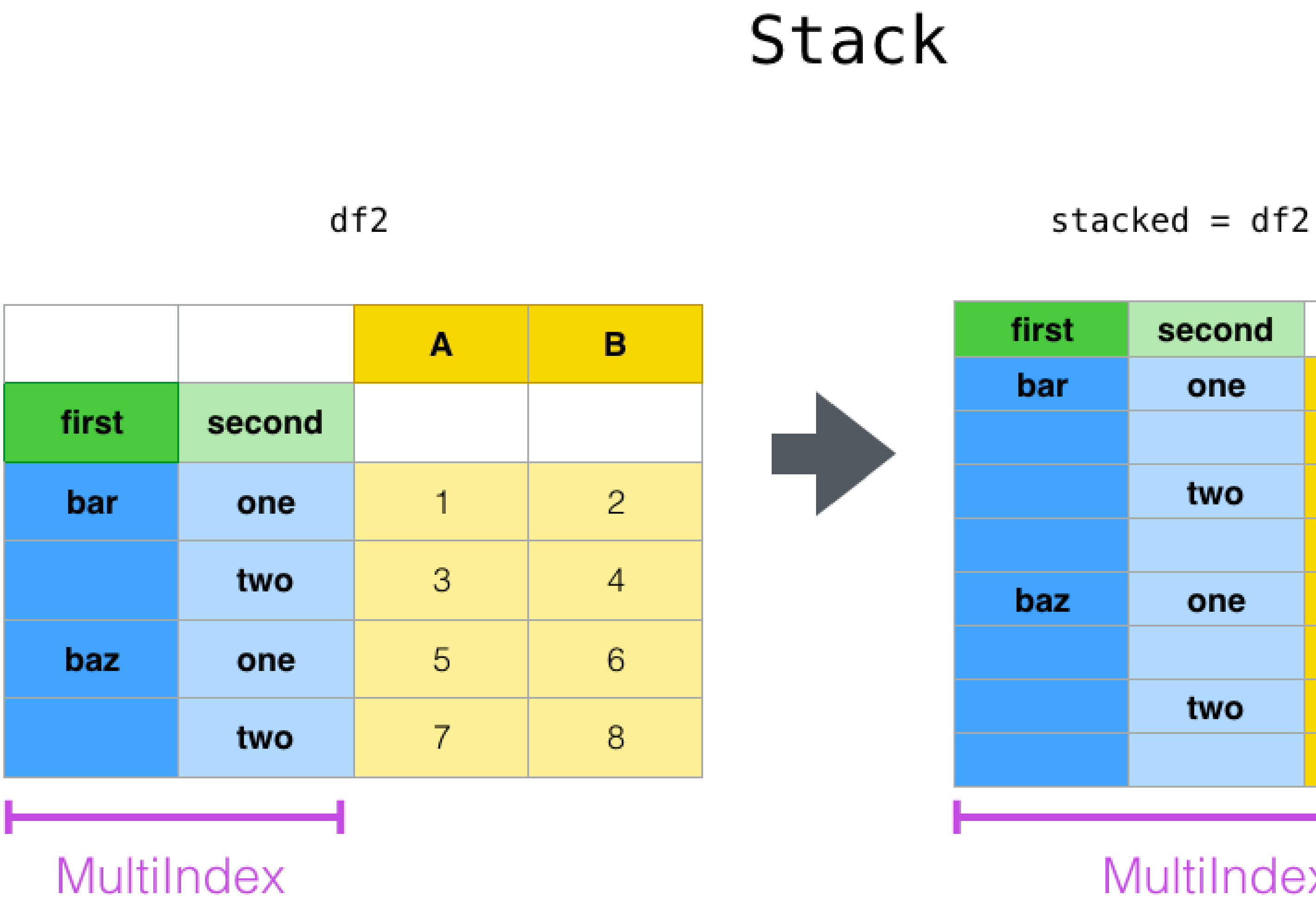
You can then select subsets from the pivoted DataFrame:

```
In [7]: pivoted['value2']
Out[7]: 
variable           A         B         C         D
date                                              
2000-01-03  0.938225 -2.271265  0.238417 -4.209138
2000-01-04 -0.565727  2.424224 -2.088472 -0.989859
2000-01-05 -3.018117 -0.346429 -1.723698  2.143608
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

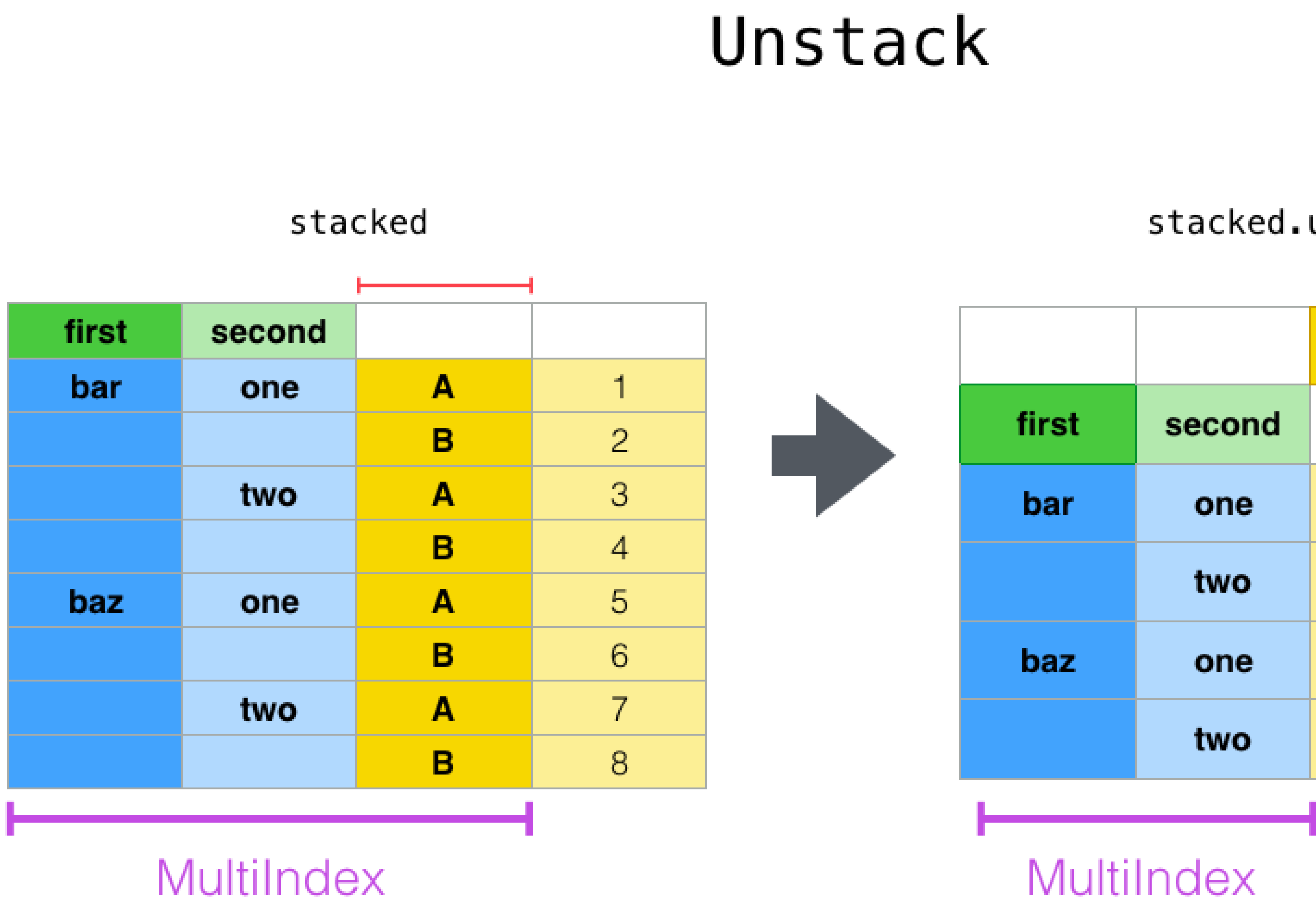
Note: *pivot()* will error with a `ValueError: Index contains duplicate entries, cannot reshape` if the index/column pair is not unique. In this case, consider using *pivot_table()* which is a generalization of pivot that can handle duplicate values for one index/column pair.

4.5.2 Reshaping by stacking and unstacking



Closely related to the *pivot()* method are the related *stack()* and *unstack()* methods available on `Series` and `DataFrame`. These methods are designed to work together with `MultiIndex` objects (see the section on *hierarchical indexing*). Here are essentially what these methods do:

- `stack`: "pivot" a level of the (possibly hierarchical) column labels, returning a `DataFrame` with an index with a new inner-most level of row labels.
- `unstack`: (inverse operation of `stack`) "pivot" a level of the (possibly hierarchical) row index to the column axis, producing a reshaped `DataFrame` with a new inner-most level of column labels.



The clearest way to explain is by example. Let's take a prior example data set from the hierarchical indexing section:

```
In [8]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
   ...:                      'foo', 'foo', 'qux', 'qux'],
   ...:                     ['one', 'two', 'one', 'two',
   ...:                      'one', 'two', 'one', 'two']]))
   ...:

In [9]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [10]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [11]: df2 = df[:4]

In [12]: df2
Out[12]:
                     A         B
first second
bar   one     0.721555 -0.706771
      two    -1.039575  0.271860
```

(continues on next page)

(continued from previous page)

```
baz   one    -0.424972  0.567020
      two     0.276232 -1.087401
```

The `stack` function "compresses" a level in the `DataFrame`'s columns to produce either:

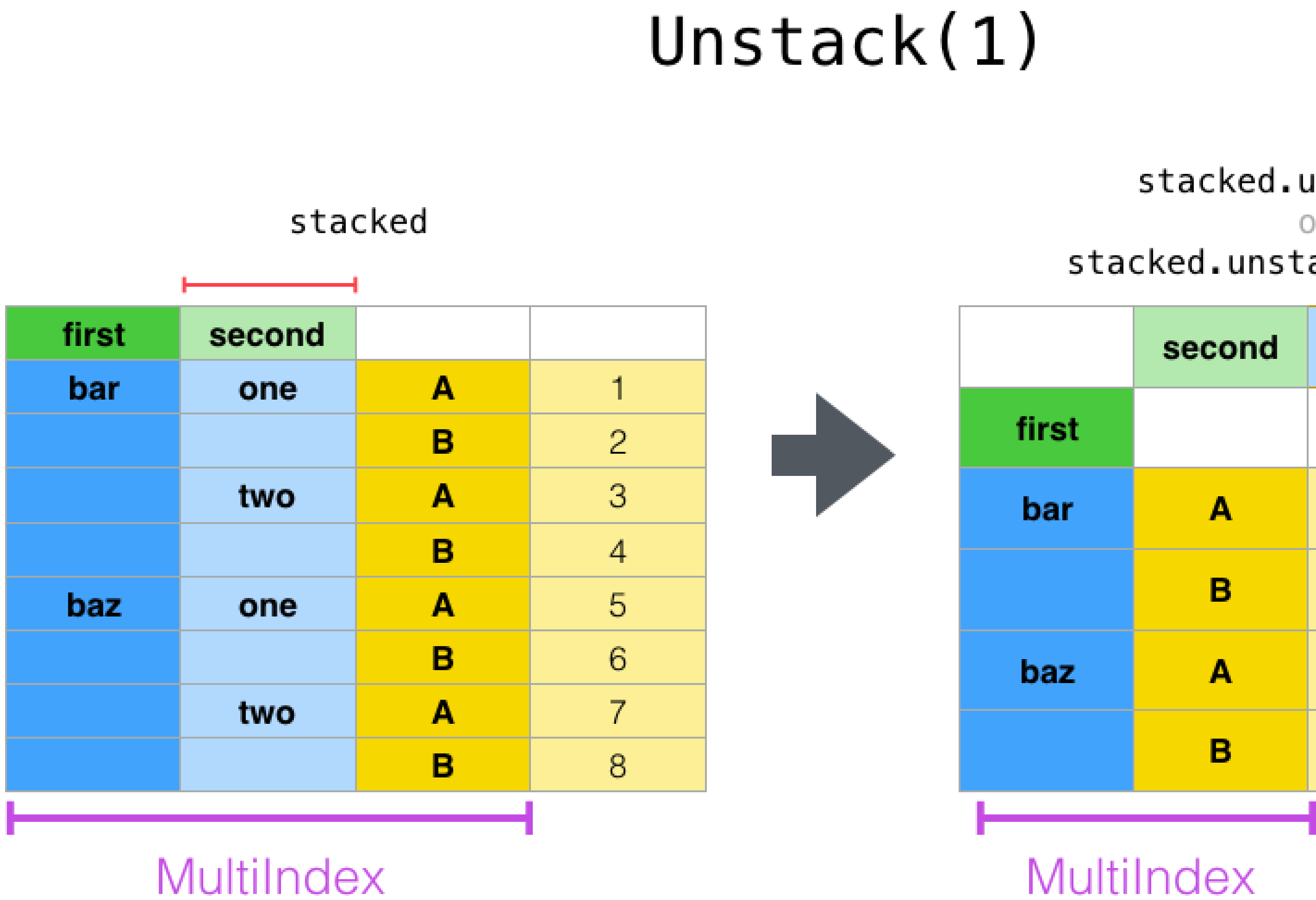
- A `Series`, in the case of a simple column Index.
- A `DataFrame`, in the case of a `MultiIndex` in the columns.

If the columns have a `MultiIndex`, you can choose which level to stack. The stacked level becomes the new lowest level in a `MultiIndex` on the columns:

```
In [13]: stacked = df2.stack()

In [14]: stacked
Out[14]:
first  second
bar    one     A    0.721555
               B   -0.706771
       two     A   -1.039575
               B    0.271860
baz    one     A   -0.424972
               B    0.567020
       two     A    0.276232
               B   -1.087401
dtype: float64
```

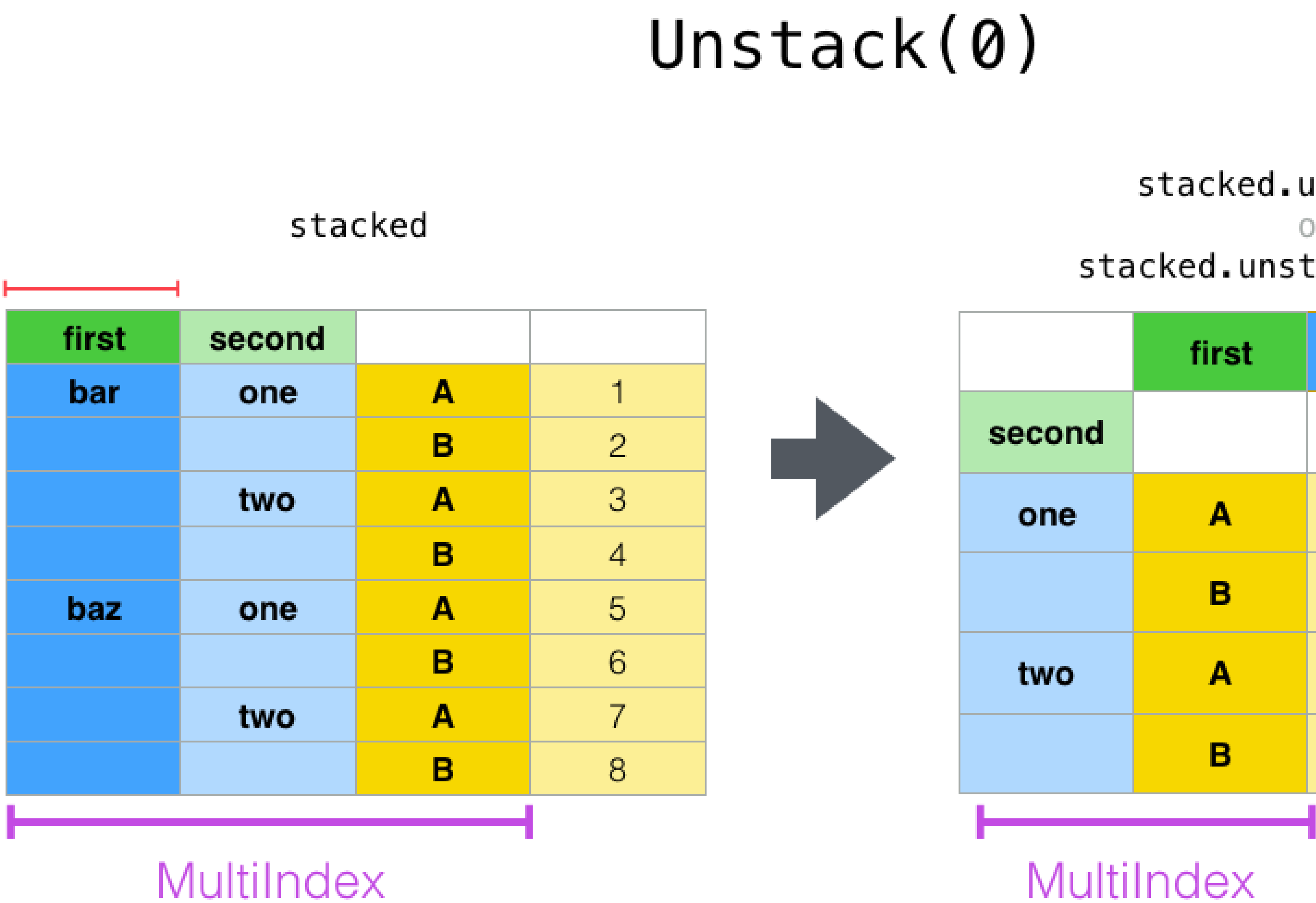
With a "stacked" `DataFrame` or `Series` (having a `MultiIndex` as the `index`), the inverse operation of `stack` is `unstack`, which by default unstacks the **last level**:

```
In [15]: stacked.unstack()
Out[15]:
                     A         B
first second
bar   one     0.721555 -0.706771
      two    -1.039575  0.271860
baz   one    -0.424972  0.567020
      two     0.276232 -1.087401

In [16]: stacked.unstack(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
second        one       two
first
bar   A  0.721555 -1.039575
      B -0.706771  0.271860
baz   A -0.424972  0.276232
      B  0.567020 -1.087401

In [17]: stacked.unstack(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
first          bar       baz
second
one    A  0.721555 -0.424972
       B -0.706771  0.567020
two    A -1.039575  0.276232
       B  0.271860 -1.087401
```



If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [18]: stacked.unstack('second')
Out[18]:
second          one       two
first
bar    A   0.721555 -1.039575
       B  -0.706771  0.271860
baz    A  -0.424972  0.276232
       B   0.567020 -1.087401
```



Notice that the stack and unstack methods implicitly sort the index levels involved. Hence a call to stack and then unstack, or vice versa, will result in a **sorted** copy of the original DataFrame or Series:

```
In [19]: index = pd.MultiIndex.from_product([[2, 1], ['a', 'b']])

In [20]: df = pd.DataFrame(np.random.randn(4), index=index, columns=['A'])

In [21]: df
Out[21]:
            A
2 a -0.370647
  b -1.157892
1 a -1.344312
  b  0.844885

In [22]: all(df.unstack().stack() == df.sort_index())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
↪True
```

The above code will raise a TypeError if the call to sort_index is removed.

Multiple Levels

You may also stack or unstack more than one level at a time by passing a list of levels, in which case the end result is as if each level in the list were processed individually.

```
In [23]: columns = pd.MultiIndex.from_tuples([
   ....:     ('A', 'cat', 'long'), ('B', 'cat', 'long'),
   ....:     ('A', 'dog', 'short'), ('B', 'dog', 'short')],
   ....:     names=['exp', 'animal', 'hair_length']
   ....: )
   ....:

In [24]: df = pd.DataFrame(np.random.randn(4, 4), columns=columns)

In [25]: df
Out[25]:
exp                 A         B         A         B
animal            cat       cat       dog       dog
hair_length      long      long     short     short
0            1.075770 -0.109050  1.643563 -1.469388
1            0.357021 -0.674600 -1.776904 -0.968914
2           -1.294524  0.413738  0.276662 -0.472035
3           -0.013960 -0.362543 -0.006154 -0.923061

In [26]: df.stack(level=['animal', 'hair_length'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp                          A         B
  animal hair_length
0 cat    long         1.075770 -0.109050
  dog    short        1.643563 -1.469388
1 cat    long         0.357021 -0.674600
  dog    short       -1.776904 -0.968914
2 cat    long        -1.294524  0.413738
  dog    short        0.276662 -0.472035
3 cat    long        -0.013960 -0.362543
  dog    short       -0.006154 -0.923061
```

The list of levels can contain either level names or level numbers (but not a mixture of the two).

```
# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [27]: df.stack(level=[1, 2])
Out[27]:
exp                          A         B
  animal hair_length
0 cat    long         1.075770 -0.109050
  dog    short        1.643563 -1.469388
1 cat    long         0.357021 -0.674600
  dog    short       -1.776904 -0.968914
2 cat    long        -1.294524  0.413738
  dog    short        0.276662 -0.472035
3 cat    long        -0.013960 -0.362543
  dog    short       -0.006154 -0.923061
```

Missing Data

These functions are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling `sort_index`, of course). Here is a more complex example:

```
In [28]: columns = pd.MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
   ....:                                      ('B', 'cat'), ('A', 'dog')],
   ....:                                     names=['exp', 'animal'])
   ....:

In [29]: index = pd.MultiIndex.from_product([('bar', 'baz', 'foo', 'qux'),
   ....:                                     ('one', 'two')],
   ....:                                    names=['first', 'second'])
   ....:

In [30]: df = pd.DataFrame(np.random.randn(8, 4), index=index, columns=columns)

In [31]: df2 = df.iloc[[0, 1, 2, 4, 5, 7]]

In [32]: df2
Out[32]:
exp                  A         B                   A
animal             cat       dog       cat       dog
first second
bar   one     0.895717  0.805244 -1.206412  2.565646
      two     1.431256  1.340309 -1.170299 -0.226169
baz   one     0.410835  0.813850  0.132003 -0.827317
foo   one    -1.413681  1.607920  1.024180  0.569605
      two     0.875906 -2.211372  0.974466 -2.006747
qux   two    -1.226825  0.769804 -1.281247 -0.727707
```

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [33]: df2.stack('exp')
Out[33]:
animal                 cat       dog
first second exp
bar   one    A    0.895717  2.565646
             B   -1.206412  0.805244
      two    A    1.431256 -0.226169
             B   -1.170299  1.340309
baz   one    A    0.410835 -0.827317
             B    0.132003  0.813850
foo   one    A   -1.413681  0.569605
             B    1.024180  1.607920
      two    A    0.875906 -2.006747
             B    0.974466 -2.211372
qux   two    A   -1.226825 -0.727707
             B   -1.281247  0.769804

In [34]: df2.stack('animal')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp                         A         B
first second animal
bar   one    cat     0.895717 -1.206412
             dog     2.565646  0.805244
```

(continues on next page)

(continued from previous page)

```
      two    cat     1.431256 -1.170299
             dog    -0.226169  1.340309
baz   one    cat     0.410835  0.132003
             dog    -0.827317  0.813850
foo   one    cat    -1.413681  1.024180
             dog     0.569605  1.607920
      two    cat     0.875906  0.974466
             dog    -2.006747 -2.211372
qux   two    cat    -1.226825 -1.281247
             dog    -0.727707  0.769804
```

Unstacking can result in missing values if subgroups do not have the same set of labels. By default, missing values will be replaced with the default fill value for that data type, `NaN` for float, `NaT` for datetimelike, etc. For integer types, by default data will converted to float and missing values will be set to `NaN`.

```
In [35]: df3 = df.iloc[[0, 1, 4, 7], [1, 2]]

In [36]: df3
Out[36]:
exp                    B
animal               dog       cat
first second
bar   one       0.805244 -1.206412
      two       1.340309 -1.170299
foo   one       1.607920  1.024180
qux   two       0.769804 -1.281247

In [37]: df3.unstack()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp            B
animal       dog                 cat
second       one       two       one       two
first
bar     0.805244  1.340309 -1.206412 -1.170299
foo     1.607920       NaN  1.024180       NaN
qux          NaN  0.769804       NaN -1.281247
```

New in version 0.18.0.

Alternatively, unstack takes an optional `fill_value` argument, for specifying the value of missing data.

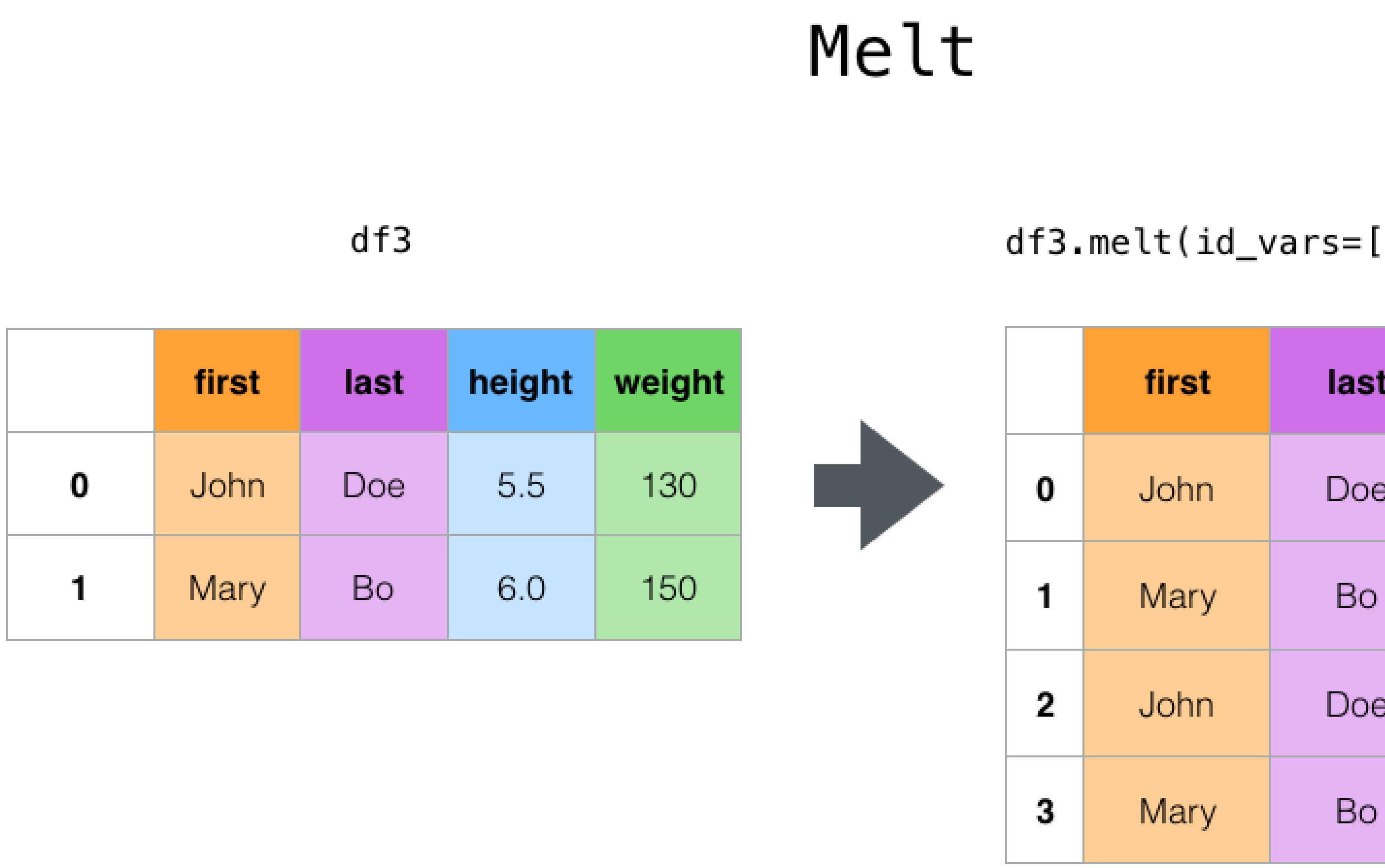
```
In [38]: df3.unstack(fill_value=-1e9)
Out[38]:
exp                  B
animal             dog                            cat
second             one           two              one           two
first
bar     8.052440e-01  1.340309e+00 -1.206412e+00 -1.170299e+00
foo     1.607920e+00 -1.000000e+09  1.024180e+00 -1.000000e+09
qux    -1.000000e+09  7.698036e-01 -1.000000e+09 -1.281247e+00
```

With a MultiIndex

Unstacking when the columns are a `MultiIndex` is also careful about doing the right thing:

```
In [39]: df[:3].unstack(0)
Out[39]:
exp              A                     B                                        A
animal         cat                   dog                 cat                  dog
first          bar       baz         bar      baz        bar       baz        bar       baz
second
one       0.895717  0.410835   0.805244   0.81385  -1.206412  0.132003   2.565646 -0.827317
two       1.431256       NaN   1.340309       NaN  -1.170299       NaN  -0.226169       NaN

In [40]: df2.unstack(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp              A                     B                                        A
animal         cat                   dog                   cat                  dog
second         one       two         one       two         one       two        one       two
first
bar       0.895717  1.431256   0.805244   1.340309 -1.206412 -1.170299   2.565646 -0.226169
baz       0.410835       NaN   0.813850        NaN  0.132003       NaN  -0.827317       NaN
foo      -1.413681  0.875906   1.607920  -2.211372  1.024180  0.974466   0.569605 -2.006747
qux            NaN -1.226825        NaN   0.769804       NaN -1.281247        NaN -0.727707
```

4.5.3 Reshaping by Melt



The top-level *melt()* function and the corresponding *DataFrame.melt()* are useful to massage a `DataFrame` into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are "unpivoted" to the row axis, leaving just two non-identifier columns, "variable" and "value". The names of those columns can be customized by supplying the `var_name` and `value_name` parameters.

For instance,

```
In [41]: cheese = pd.DataFrame({'first': ['John', 'Mary'],
   ....:                        'last': ['Doe', 'Bo'],
   ....:                        'height': [5.5, 6.0],
   ....:                        'weight': [130, 150]})
   ....:

In [42]: cheese
Out[42]:
  first last  height  weight
0  John  Doe     5.5     130
1  Mary   Bo     6.0     150

In [43]: cheese.melt(id_vars=['first', 'last'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0

In [44]: cheese.melt(id_vars=['first', 'last'], var_name='quantity')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  first last quantity  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0
```

Another way to transform is to use the *wide_to_long()* panel data convenience function. It is less flexible than *melt()*, but more user-friendly.

```
In [45]: dft = pd.DataFrame({"A1970": {0: "a", 1: "b", 2: "c"},
   ....:                     "A1980": {0: "d", 1: "e", 2: "f"},
   ....:                     "B1970": {0: 2.5, 1: 1.2, 2: .7},
   ....:                     "B1980": {0: 3.2, 1: 1.3, 2: .1},
   ....:                     "X": dict(zip(range(3), np.random.randn(3)))
   ....:                    })
   ....:

In [46]: dft["id"] = dft.index

In [47]: dft
Out[47]:
  A1970 A1980  B1970  B1980         X  id
0     a     d    2.5    3.2 -0.121306   0
1     b     e    1.2    1.3 -0.097883   1
2     c     f    0.7    0.1  0.695775   2

In [48]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                X  A    B
id year
0  1970 -0.121306  a  2.5
1  1970 -0.097883  b  1.2
2  1970  0.695775  c  0.7
0  1980 -0.121306  d  3.2
1  1980 -0.097883  e  1.3
2  1980  0.695775  f  0.1
```

4.5.4 Combining with stats and GroupBy

It should be no shock that combining `pivot` / `stack` / `unstack` with GroupBy and the basic Series and DataFrame statistical functions can produce some very expressive and fast data manipulations.

```
In [49]: df
Out[49]:
exp                    A         B                   A
animal               cat       dog       cat       dog
first second
bar   one     0.895717  0.805244 -1.206412  2.565646
      two     1.431256  1.340309 -1.170299 -0.226169
baz   one     0.410835  0.813850  0.132003 -0.827317
      two    -0.076467 -1.187678  1.130127 -1.436737
foo   one    -1.413681  1.607920  1.024180  0.569605
      two     0.875906 -2.211372  0.974466 -2.006747
qux   one    -0.410001 -0.078638  0.545952 -1.219217
      two    -1.226825  0.769804 -1.281247 -0.727707

In [50]: df.stack().mean(1).unstack()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
animal               cat       dog
first second
bar   one    -0.155347  1.685445
      two     0.130479  0.557070
baz   one     0.271419 -0.006733
      two     0.526830 -1.312207
foo   one    -0.194750  1.088763
      two     0.925186 -2.109060
qux   one     0.067976 -0.648927
      two    -1.254036  0.021048

# same result, another way
In [51]: df.groupby(level=1, axis=1).mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
animal               cat       dog
first second
bar   one    -0.155347  1.685445
      two     0.130479  0.557070
baz   one     0.271419 -0.006733
      two     0.526830 -1.312207
foo   one    -0.194750  1.088763
```

(continues on next page)

(continued from previous page)

```
      two     0.925186 -2.109060
qux   one     0.067976 -0.648927
      two    -1.254036  0.021048

In [52]: df.stack().groupby(level=1).mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp             A         B
second
one      0.071448  0.455513
two     -0.424186 -0.204486

In [53]: df.mean().unstack(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
exp             A         B
animal
cat      0.060843  0.018596
dog     -0.413580  0.232430
```

4.5.5 Pivot tables

While *pivot()* provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides *pivot_table()* for pivoting with aggregation of numeric data.

The function *pivot_table()* can be used to create spreadsheet-style pivot tables. See the *cookbook* for some advanced strategies.

It takes a number of arguments:

- `data`: a DataFrame object.
- `values`: a column or a list of columns to aggregate.
- `index`: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- `columns`: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- `aggfunc`: function to use for aggregation, defaulting to `numpy.mean`.

Consider a data set like this:

```
In [54]: import datetime

In [55]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 6,
   ....:                    'B': ['A', 'B', 'C'] * 8,
   ....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
   ....:                    'D': np.random.randn(24),
   ....:                    'E': np.random.randn(24),
   ....:                    'F': [datetime.datetime(2013, i, 1) for i in range(1, 13)]
   ....:                    + [datetime.datetime(2013, i, 15) for i in range(1, 13)]})
   ....:

In [56]: df
Out[56]:
```

(continues on next page)

(continued from previous page)

```
        A  B    C         D         E          F
0     one  A  foo  0.341734 -0.317441 2013-01-01
1     one  B  foo  0.959726 -1.236269 2013-02-01
2     two  C  foo -1.110336  0.896171 2013-03-01
3   three  A  bar -0.619976 -0.487602 2013-04-01
4     one  B  bar  0.149748 -0.082240 2013-05-01
5     one  C  bar -0.732339 -2.182937 2013-06-01
6     two  A  foo  0.687738  0.380396 2013-07-01
..    ... ..  ...       ...       ...        ...
17    one  C  bar -0.345352  0.206053 2013-06-15
18    two  A  foo  1.314232 -0.251905 2013-07-15
19  three  B  foo  0.690579 -2.213588 2013-08-15
20    one  C  foo  0.995761  1.063327 2013-09-15
21    one  A  bar  2.396780  1.266143 2013-10-15
22    two  B  bar  0.014871  0.299368 2013-11-15
23  three  C  bar  3.357427 -0.863838 2013-12-15

[24 rows x 6 columns]
```

We can produce pivot tables from this data very easily:

```
In [57]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
Out[57]:
C             bar       foo
A     B
one   A  1.120915 -0.514058
      B -0.338421  0.002759
      C -0.538846  0.699535
three A -1.181568       NaN
      B       NaN  0.433512
      C  0.588783       NaN
two   A       NaN  1.000985
      B  0.158248       NaN
      C       NaN  0.176180

In [58]: pd.pivot_table(df, values='D', index=['B'], columns=['A', 'C'], aggfunc=np.
↪sum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A       one                three                two
C       bar       foo       bar       foo       bar       foo
B
A  2.241830 -1.028115 -2.363137       NaN       NaN  2.001971
B -0.676843  0.005518       NaN  0.867024  0.316495       NaN
C -1.077692  1.399070  1.177566       NaN       NaN  0.352360

In [59]: pd.pivot_table(df, values=['D', 'E'], index=['B'], columns=['A', 'C'],
   ....:                aggfunc=np.sum)
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          D                                                               E                     ␣
↪
A       one                three                two                 one                     ␣
↪three                two
C       bar       foo       bar       foo       bar       foo       bar       foo           ␣
↪   bar       foo       bar       foo
```

(continues on next page)

(continued from previous page)

```
B                                                                                    ␣
↪
A  2.241830 -1.028115 -2.363137       NaN       NaN  2.001971  2.786113 -0.043211  1.
↪922577       NaN       NaN  0.128491
B -0.676843  0.005518       NaN  0.867024  0.316495       NaN  1.368280 -1.103384   ␣
↪  NaN -2.128743 -0.194294       NaN
C -1.077692  1.399070  1.177566       NaN       NaN  0.352360 -1.976883  1.495717 -0.
↪263660       NaN       NaN  0.872482
```

The result object is a `DataFrame` having potentially hierarchical indexes on the rows and columns. If the `values` column name is not given, the pivot table will include all of the data that can be aggregated in an additional level of hierarchy in the columns:

```
In [60]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])
Out[60]:
                D                   E
C             bar       foo       bar       foo
A     B
one   A  1.120915 -0.514058  1.393057 -0.021605
      B -0.338421  0.002759  0.684140 -0.551692
      C -0.538846  0.699535 -0.988442  0.747859
three A -1.181568       NaN  0.961289       NaN
      B       NaN  0.433512       NaN -1.064372
      C  0.588783       NaN -0.131830       NaN
two   A       NaN  1.000985       NaN  0.064245
      B  0.158248       NaN -0.097147       NaN
      C       NaN  0.176180       NaN  0.436241
```

Also, you can use `Grouper` for `index` and `columns` keywords. For detail of `Grouper`, see *Grouping with a Grouper specification*.

```
In [61]: pd.pivot_table(df, values='D', index=pd.Grouper(freq='M', key='F'),
   ....:                columns='C')
   ....:
Out[61]:
C                bar       foo
F
2013-01-31       NaN -0.514058
2013-02-28       NaN  0.002759
2013-03-31       NaN  0.176180
2013-04-30 -1.181568       NaN
2013-05-31 -0.338421       NaN
2013-06-30 -0.538846       NaN
2013-07-31       NaN  1.000985
2013-08-31       NaN  0.433512
2013-09-30       NaN  0.699535
2013-10-31  1.120915       NaN
2013-11-30  0.158248       NaN
2013-12-31  0.588783       NaN
```

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```
In [62]: table = pd.pivot_table(df, index=['A', 'B'], columns=['C'])

In [63]: print(table.to_string(na_rep=''))
                D                   E
```

(continues on next page)

(continued from previous page)

```
C             bar       foo       bar       foo
A     B
one   A  1.120915 -0.514058  1.393057 -0.021605
      B -0.338421  0.002759  0.684140 -0.551692
      C -0.538846  0.699535 -0.988442  0.747859
three A -1.181568            0.961289
      B            0.433512           -1.064372
      C  0.588783           -0.131830
two   A            1.000985            0.064245
      B  0.158248           -0.097147
      C            0.176180            0.436241
```

Note that `pivot_table` is also available as an instance method on DataFrame, i.e. *DataFrame.pivot_table()*.

Adding margins

If you pass `margins=True` to `pivot_table`, special `All` columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
In [64]: df.pivot_table(index=['A', 'B'], columns='C', margins=True, aggfunc=np.std)
Out[64]:
                D                             E
C             bar       foo       All       bar       foo       All
A     B
one   A  1.804346  1.210272  1.569879  0.179483  0.418374  0.858005
      B  0.690376  1.353355  0.898998  1.083825  0.968138  1.101401
      C  0.273641  0.418926  0.771139  1.689271  0.446140  1.422136
three A  0.794212       NaN  0.794212  2.049040       NaN  2.049040
      B       NaN  0.363548  0.363548       NaN  1.625237  1.625237
      C  3.915454       NaN  3.915454  1.035215       NaN  1.035215
two   A       NaN  0.442998  0.442998       NaN  0.447104  0.447104
      B  0.202765       NaN  0.202765  0.560757       NaN  0.560757
      C       NaN  1.819408  1.819408       NaN  0.650439  0.650439
All      1.556686  0.952552  1.246608  1.250924  0.899904  1.059389
```

4.5.6 Cross tabulations

Use *crosstab()* to compute a cross-tabulation of two (or more) factors. By default `crosstab` computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- `index`: array-like, values to group by in the rows.
- `columns`: array-like, values to group by in the columns.
- `values`: array-like, optional, array of values to aggregate according to the factors.
- `aggfunc`: function, optional, If no values array is passed, computes a frequency table.
- `rownames`: sequence, default `None`, must match number of row arrays passed.
- `colnames`: sequence, default `None`, if passed, must match number of column arrays passed.
- `margins`: boolean, default `False`, Add row/column margins (subtotals)

- `normalize`: boolean, {'all', 'index', 'columns'}, or {0,1}, default `False`. Normalize by dividing all values by the sum of values.

Any `Series` passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

```
In [65]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [66]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [67]: b = np.array([one, one, two, one, two, one], dtype=object)

In [68]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [69]: pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
Out[69]:
b     one        two
c    dull shiny dull shiny
a
bar     1     0    0     1
foo     2     1    1     0
```

If `crosstab` receives only two Series, it will provide a frequency table.

```
In [70]: df = pd.DataFrame({'A': [1, 2, 2, 2, 2], 'B': [3, 3, 4, 4, 4],
   ....:                    'C': [1, 1, np.nan, 1, 1]})
   ....:

In [71]: df
Out[71]:
   A  B    C
0  1  3  1.0
1  2  3  1.0
2  2  4  NaN
3  2  4  1.0
4  2  4  1.0

In [72]: pd.crosstab(df.A, df.B)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[72]:
↪
B  3  4
A
1  1  0
2  1  3
```

Any input passed containing `Categorical` data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

```
In [73]: foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])

In [74]: bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])

In [75]: pd.crosstab(foo, bar)
Out[75]:
col_0  d  e
row_0
```

(continues on next page)

(continued from previous page)

```
a       1   0
b       0   1
```

Normalization

New in version 0.18.1.

Frequency tables can also be normalized to show percentages rather than counts using the `normalize` argument:

```
In [76]: pd.crosstab(df.A, df.B, normalize=True)
Out[76]:
B    3    4
A
1  0.2  0.0
2  0.2  0.6
```

`normalize` can also normalize values within each row or within each column:

```
In [77]: pd.crosstab(df.A, df.B, normalize='columns')
Out[77]:
B    3    4
A
1  0.5  0.0
2  0.5  1.0
```

`crosstab` can also be passed a third `Series` and an aggregation function (`aggfunc`) that will be applied to the values of the third `Series` within each group defined by the first two `Series`:

```
In [78]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum)
Out[78]:
B    3    4
A
1  1.0  NaN
2  1.0  2.0
```

Adding Margins

Finally, one can also add margins or normalize this output.

```
In [79]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum, normalize=True,
   ....:             margins=True)
   ....:
Out[79]:
B       3    4   All
A
1    0.25  0.0  0.25
2    0.25  0.5  0.75
All  0.50  0.5  1.00
```

4.5.7 Tiling

The *cut()* function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

```
In [80]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [81]: pd.cut(ages, bins=3)
Out[81]:
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.
↪95, 26.667], (26.667, 43.333], (43.333, 60.0], (43.333, 60.0]]
Categories (3, interval[float64]): [(9.95, 26.667] < (26.667, 43.333] < (43.333, 60.
↪0]]
```

If the bins keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [82]: c = pd.cut(ages, bins=[0, 18, 35, 70])

In [83]: c
Out[83]:
[(0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70]]
Categories (3, interval[int64]): [(0, 18] < (18, 35] < (35, 70]]
```

New in version 0.20.0.

If the bins keyword is an IntervalIndex, then these will be used to bin the passed data.:

```
pd.cut([25, 20, 50], bins=c.categories)
```

4.5.8 Computing indicator / dummy variables

To convert a categorical variable into a "dummy" or "indicator" DataFrame, for example a column in a DataFrame (a Series) which has k distinct values, can derive a DataFrame containing k columns of 1s and 0s using *get_dummies()*:

```
In [84]: df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})

In [85]: pd.get_dummies(df['key'])
Out[85]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Sometimes it's useful to prefix the column names, for example when merging the result with the original DataFrame:

```
In [86]: dummies = pd.get_dummies(df['key'], prefix='key')

In [87]: dummies
Out[87]:
   key_a  key_b  key_c
0      0      1      0
1      0      1      0
2      1      0      0
3      0      0      1
4      1      0      0
5      0      1      0
```

(continues on next page)

(continued from previous page)

```
In [88]: df[['data1']].join(dummies)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

This function is often used along with discretization functions like `cut`:

```
In [89]: values = np.random.randn(10)

In [90]: values
Out[90]: 
array([ 0.4082, -1.0481, -0.0257, -0.9884,  0.0941,  1.2627,  1.29  ,
        0.0824, -0.0558,  0.5366])

In [91]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [92]: pd.get_dummies(pd.cut(values, bins))
Out[92]: 
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           1           0           0
1           0           0           0           0           0
2           0           0           0           0           0
3           0           0           0           0           0
4           1           0           0           0           0
5           0           0           0           0           0
6           0           0           0           0           0
7           1           0           0           0           0
8           0           0           0           0           0
9           0           0           1           0           0
```

See also *`Series.str.get_dummies`*.

`get_dummies()` also accepts a `DataFrame`. By default all categorical variables (categorical in the statistical sense, those with *object* or *categorical* dtype) are encoded as dummy variables.

```
In [93]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
   ....:                    'C': [1, 2, 3]})
   ....: 

In [94]: pd.get_dummies(df)
Out[94]: 
   C  A_a  A_b  B_b  B_c
0  1    1    0    0    1
1  2    0    1    0    1
2  3    1    0    1    0
```

All non-object columns are included untouched in the output. You can control the columns that are encoded with the `columns` keyword.

```
In [95]: pd.get_dummies(df, columns=['A'])
Out[95]: 
```

(continues on next page)

(continued from previous page)

```
   B  C  A_a  A_b
0  c  1    1    0
1  c  2    0    1
2  b  3    1    0
```

Notice that the B column is still included in the output, it just hasn't been encoded. You can drop B before calling get_dummies if you don't want to include it in the output.

As with the Series version, you can pass values for the prefix and prefix_sep. By default the column name is used as the prefix, and '_' as the prefix separator. You can specify prefix and prefix_sep in 3 ways:

- string: Use the same value for prefix or prefix_sep for each column to be encoded.
- list: Must be the same length as the number of columns being encoded.
- dict: Mapping column name to prefix.

```
In [96]: simple = pd.get_dummies(df, prefix='new_prefix')

In [97]: simple
Out[97]: 
   C  new_prefix_a  new_prefix_b  new_prefix_b  new_prefix_c
0  1             1             0             0             1
1  2             0             1             0             1
2  3             1             0             1             0

In [98]: from_list = pd.get_dummies(df, prefix=['from_A', 'from_B'])

In [99]: from_list
Out[99]: 
   C  from_A_a  from_A_b  from_B_b  from_B_c
0  1         1         0         0         1
1  2         0         1         0         1
2  3         1         0         1         0

In [100]: from_dict = pd.get_dummies(df, prefix={'B': 'from_B', 'A': 'from_A'})

In [101]: from_dict
Out[101]: 
   C  from_A_a  from_A_b  from_B_b  from_B_c
0  1         1         0         0         1
1  2         0         1         0         1
2  3         1         0         1         0
```

New in version 0.18.0.

Sometimes it will be useful to only keep k-1 levels of a categorical variable to avoid collinearity when feeding the result to statistical models. You can switch to this mode by turn on drop_first.

```
In [102]: s = pd.Series(list('abcaa'))

In [103]: pd.get_dummies(s)
Out[103]: 
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

(continues on next page)

(continued from previous page)

```
4  1  0  0

In [104]: pd.get_dummies(s, drop_first=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[104]:␣
↪
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

When a column contains only one level, it will be omitted in the result.

```
In [105]: df = pd.DataFrame({'A': list('aaaaa'), 'B': list('ababc')})

In [106]: pd.get_dummies(df)
Out[106]:
   A_a  B_a  B_b  B_c
0    1    1    0    0
1    1    0    1    0
2    1    1    0    0
3    1    0    1    0
4    1    0    0    1

In [107]: pd.get_dummies(df, drop_first=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   B_b  B_c
0    0    0
1    1    0
2    0    0
3    1    0
4    0    1
```

By default new columns will have `np.uint8` dtype. To choose another dtype, use the `dtype` argument:

```
In [108]: df = pd.DataFrame({'A': list('abc'), 'B': [1.1, 2.2, 3.3]})

In [109]: pd.get_dummies(df, dtype=bool).dtypes
Out[109]:
B      float64
A_a       bool
A_b       bool
A_c       bool
dtype: object
```

New in version 0.23.0.

4.5.9 Factorizing values

To encode 1-d values as an enumerated type use *factorize()*:

```
In [110]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])
```

(continues on next page)

(continued from previous page)

```
In [111]: x
Out[111]:
0       A
1       A
2     NaN
3       B
4    3.14
5     inf
dtype: object

In [112]: labels, uniques = pd.factorize(x)

In [113]: labels
Out[113]: array([ 0,  0, -1,  1,  2,  3])

In [114]: uniques
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[114]: Index(['A', 'B', 3.14, inf],␣
↪dtype='object')
```

Note that `factorize` is similar to `numpy.unique`, but differs in its handling of NaN:

Note: The following `numpy.unique` will fail under Python 3 with a `TypeError` because of an ordering bug. See also here.

```
In [1]: x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])
In [2]: pd.factorize(x, sort=True)
Out[2]:
(array([ 2,  2, -1,  3,  0,  1]),
 Index([3.14, inf, u'A', u'B'], dtype='object'))

In [3]: np.unique(x, return_inverse=True)[::-1]
Out[3]: (array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
```

Note: If you just want to handle one column as a categorical variable (like R's factor), you can use `df["cat_col"] = pd.Categorical(df["col"])` or `df["cat_col"] = df["col"].astype("category")`. For full docs on *`Categorical`*, see the *Categorical introduction* and the API documentation.

4.5.10 Examples

In this section, we will review frequently asked questions and examples. The column names and relevant column values are named to correspond with how this DataFrame will be pivoted in the answers below.

```
In [115]: np.random.seed([3, 1415])

In [116]: n = 20

In [117]: cols = np.array(['key', 'row', 'item', 'col'])

In [118]: df = cols + pd.DataFrame((np.random.randint(5, size=(n, 4))
   .....:                           // [2, 1, 2, 1]).astype(str))
   .....:
```

(continues on next page)

(continued from previous page)

```
In [119]: df.columns = cols

In [120]: df = df.join(pd.DataFrame(np.random.rand(n, 2).round(2)).add_prefix('val'))

In [121]: df
Out[121]:
     key   row   item   col  val0  val1
0   key0  row3  item1  col3  0.81  0.04
1   key1  row2  item1  col2  0.44  0.07
2   key1  row0  item1  col0  0.77  0.01
3   key0  row4  item0  col2  0.15  0.59
4   key1  row0  item2  col1  0.81  0.64
5   key1  row2  item2  col4  0.13  0.88
6   key2  row4  item1  col3  0.88  0.39
..   ...   ...    ...   ...   ...   ...
13  key0  row4  item1  col4  0.24  0.46
14  key1  row3  item2  col3  0.28  0.11
15  key0  row3  item1  col1  0.31  0.23
16  key0  row0  item2  col3  0.86  0.01
17  key0  row4  item0  col3  0.64  0.21
18  key2  row2  item2  col0  0.13  0.45
19  key0  row2  item0  col4  0.37  0.70

[20 rows x 6 columns]
```

Pivoting with Single Aggregations

Suppose we wanted to pivot `df` such that the `col` values are columns, `row` values are the index, and the mean of `val0` are the values? In particular, the resulting DataFrame should look like:

Note: col col0 col1 col2 col3 col4 row row0 0.77 0.605 NaN 0.860 0.65 row2 0.13 NaN 0.395 0.500 0.25 row3 NaN 0.310 NaN 0.545 NaN row4 NaN 0.100 0.395 0.760 0.24

This solution uses *pivot_table()*. Also note that `aggfunc='mean'` is the default. It is included here to be explicit.

```
In [122]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean')
   .....:
Out[122]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65
row2  0.13    NaN  0.395  0.500  0.25
row3   NaN  0.310    NaN  0.545   NaN
row4   NaN  0.100  0.395  0.760  0.24
```

Note that we can also replace the missing values by using the `fill_value` parameter.

```
In [123]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean', fill_value=0)
   .....:
```

(continues on next page)

(continued from previous page)

```
Out[123]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605  0.000  0.860  0.65
row2  0.13  0.000  0.395  0.500  0.25
row3  0.00  0.310  0.000  0.545  0.00
row4  0.00  0.100  0.395  0.760  0.24
```

Also note that we can pass in other aggregation functions as well. For example, we can also pass in `sum`.

```
In [124]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='sum', fill_value=0)
   .....:
Out[124]:
col   col0  col1  col2  col3  col4
row
row0  0.77  1.21  0.00  0.86  0.65
row2  0.13  0.00  0.79  0.50  0.50
row3  0.00  0.31  0.00  1.09  0.00
row4  0.00  0.10  0.79  1.52  0.24
```

Another aggregation we can do is calculate the frequency in which the columns and rows occur together a.k.a. "cross tabulation". To do this, we can pass `size` to the `aggfunc` parameter.

```
In [125]: df.pivot_table(index='row', columns='col', fill_value=0, aggfunc='size')
Out[125]:
col   col0  col1  col2  col3  col4
row
row0     1     2     0     1     1
row2     1     0     2     1     2
row3     0     1     0     2     0
row4     0     1     2     2     1
```

Pivoting with Multiple Aggregations

We can also perform multiple aggregations. For example, to perform both a `sum` and `mean`, we can pass in a list to the `aggfunc` argument.

```
In [126]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc=['mean', 'sum'])
   .....:
Out[126]:
      mean                                sum
col   col0   col1   col2   col3  col4  col0  col1  col2  col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65  0.77  1.21   NaN  0.86  0.65
row2  0.13    NaN  0.395  0.500  0.25  0.13   NaN  0.79  0.50  0.50
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.31   NaN  1.09   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.10  0.79  1.52  0.24
```

Note to aggregate over multiple value columns, we can pass in a list to the `values` parameter.

```
In [127]: df.pivot_table(
   .....:     values=['val0', 'val1'], index='row', columns='col', aggfunc=['mean'])
   .....:
```

(continues on next page)

(continued from previous page)

```
Out[127]:
      mean
      val0                             val1
col   col0   col1   col2   col3  col4  col0   col1  col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65  0.01  0.745   NaN  0.010  0.02
row2  0.13    NaN  0.395  0.500  0.25  0.45    NaN  0.34  0.440  0.79
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.230   NaN  0.075   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.070  0.42  0.300  0.46
```

Note to subdivide over multiple columns we can pass in a list to the `columns` parameter.

```
In [128]: df.pivot_table(
   .....:     values=['val0'], index='row', columns=['item', 'col'], aggfunc=['mean'])
   .....:
Out[128]:
      mean
      val0
item item0             item1                               item2
col   col2  col3  col4  col0  col1  col2  col3  col4  col0   col1  col3  col4
row
row0   NaN   NaN   NaN  0.77   NaN   NaN   NaN   NaN   NaN  0.605  0.86  0.65
row2  0.35   NaN  0.37   NaN   NaN  0.44   NaN   NaN  0.13    NaN  0.50  0.13
row3   NaN   NaN   NaN   NaN  0.31   NaN  0.81   NaN   NaN    NaN  0.28   NaN
row4  0.15  0.64   NaN   NaN  0.10  0.64  0.88  0.24   NaN    NaN   NaN   NaN
```

4.6 Working with Text Data

Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the `str` attribute and generally have names matching the equivalent (scalar) built-in string methods:

```
In [1]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [2]: s.str.lower()
Out[2]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object

In [3]: s.str.upper()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0       A
1       B
2       C
3    AABA
```

(continues on next page)

(continued from previous page)

```
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object

In [4]: s.str.len()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1.0
1    1.0
2    1.0
3    4.0
4    4.0
5    NaN
6    4.0
7    3.0
8    3.0
dtype: float64
```

```
In [5]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

In [6]: idx.str.strip()
Out[6]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [7]: idx.str.lstrip()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: Index(['jack
↪', 'jill ', 'jesse ', 'frank'], dtype='object')

In [8]: idx.str.rstrip()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Index([' jack', 'jill', ' jesse', 'frank'], dtype='object')
```

The string methods on Index are especially useful for cleaning up or transforming DataFrame columns. For instance, you may have columns with leading or trailing whitespace:

```
In [9]: df = pd.DataFrame(np.random.randn(3, 2),
   ...:                   columns=[' Column A ', ' Column B '], index=range(3))
   ...:

In [10]: df
Out[10]:
    Column A    Column B
0   0.469112   -0.282863
1  -1.509059   -1.135632
2   1.212112   -0.173215
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```
In [11]: df.columns.str.strip()
Out[11]: Index(['Column A', 'Column B'], dtype='object')

In [12]: df.columns.str.lower()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]: Index([' column a ',
↪ ' column b '], dtype='object')
```

These string methods can then be used to clean up the columns as needed. Here we are removing leading and trailing white spaces, lower casing all names, and replacing any remaining white spaces with underscores:

```
In [13]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [14]: df
Out[14]:
   column_a  column_b
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
```

Note: If you have a `Series` where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`), it can be faster to convert the original `Series` to one of type `category` and then use `.str.<method>` or `.dt.<property>` on that. The performance difference comes from the fact that, for `Series` of type `category`, the string operations are done on the `.categories` and not on each element of the `Series`.

Please note that a `Series` of type `category` with string `.categories` has some limitations in comparison of `Series` of type string (e.g. you can't add strings to each other: `s + " " + s` won't work if `s` is a `Series` of type `category`). Also, `.str` methods which operate on elements of type `list` are not available on such a `Series`.

4.6.1 Splitting and Replacing Strings

Methods like `split` return a Series of lists:

```
In [15]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [16]: s2.str.split('_')
Out[16]:
0    [a, b, c]
1    [c, d, e]
2          NaN
3    [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using `get` or `[]` notation:

```
In [17]: s2.str.split('_').str.get(1)
Out[17]:
0      b
1      d
2    NaN
3      g
dtype: object

In [18]: s2.str.split('_').str[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]:
0      b
1      d
2    NaN
3      g
dtype: object
```

It is easy to expand this to return a DataFrame using `expand`.

```
In [19]: s2.str.split('_', expand=True)
Out[19]:
     0    1    2
0    a    b    c
1    c    d    e
2  NaN  NaN  NaN
3    f    g    h
```

It is also possible to limit the number of splits:

```
In [20]: s2.str.split('_', expand=True, n=1)
Out[20]:
     0    1
0    a  b_c
1    c  d_e
2  NaN  NaN
3    f  g_h
```

`rsplit` is similar to `split` except it works in the reverse direction, i.e., from the end of the string to the beginning of the string:

```
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
     0    1
0  a_b    c
1  c_d    e
2  NaN  NaN
3  f_g    h
```

`replace` by default replaces regular expressions:

```
In [22]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca',
   ....:                 '', np.nan, 'CABA', 'dog', 'cat'])
   ....:

In [23]: s3
Out[23]:
0       A
1       B
2       C
3    Aaba
4    Baca
5
6     NaN
7    CABA
8     dog
9     cat
dtype: object

In [24]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
```

(continues on next page)

(continued from previous page)

```
5
6          NaN
7     XX-XX BA
8       XX-XX
9      XX-XX t
dtype: object
```

Some caution must be taken to keep regular expressions in mind! For example, the following code will cause trouble because of the regular expression meaning of *$*:

```
# Consider the following badly formatted financial data
In [25]: dollars = pd.Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
Out[26]:
0        12
1       -10
2    10,000
dtype: object

# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[27]:
0         12
1       -$10
2    $10,000
dtype: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'-\$', '-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0         12
1        -10
2    $10,000
dtype: object
```

New in version 0.23.0.

If you do want literal replacement of a string (equivalent to `str.replace()`), you can set the optional `regex` parameter to `False`, rather than escaping each character. In this case both `pat` and `repl` must be strings:

```
# These lines are equivalent
In [29]: dollars.str.replace(r'-\$', '-')
Out[29]:
0         12
1        -10
2    $10,000
dtype: object

In [30]: dollars.str.replace('-$', '-', regex=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]:
0         12
1        -10
2    $10,000
dtype: object
```

New in version 0.20.0.

The replace method can also take a callable as replacement. It is called on every pat using re.sub(). The callable should expect one positional argument (a regex object) and return a string.

```
# Reverse every lowercase alphabetic word
In [31]: pat = r'[a-z]+'

In [32]: def repl(m):
   ....:     return m.group(0)[::-1]
   ....:

In [33]: pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(pat, repl)
Out[33]:
0    oof 123
1    rab zab
2        NaN
dtype: object

# Using regex groups
In [34]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"

In [35]: def repl(m):
   ....:     return m.group('two').swapcase()
   ....:

In [36]: pd.Series(['Foo Bar Baz', np.nan]).str.replace(pat, repl)
Out[36]:
0    bAR
1    NaN
dtype: object
```

New in version 0.20.0.

The replace method also accepts a compiled regular expression object from re.compile() as a pattern. All flags should be included in the compiled regular expression object.

```
In [37]: import re

In [38]: regex_pat = re.compile(r'^.a|dog', flags=re.IGNORECASE)

In [39]: s3.str.replace(regex_pat, 'XX-XX ')
Out[39]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6         NaN
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: object
```

Including a flags argument when calling replace with a compiled regular expression object will raise a ValueError.

```
In [40]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

4.6.2 Concatenation

There are several ways to concatenate a `Series` or `Index`, either with itself or others, all based on *cat()*, resp. `Index.str.cat`.

Concatenating a single Series into a string

The content of a `Series` (or `Index`) can be concatenated:

```
In [41]: s = pd.Series(['a', 'b', 'c', 'd'])

In [42]: s.str.cat(sep=',')
Out[42]: 'a,b,c,d'
```

If not specified, the keyword `sep` for the separator defaults to the empty string, `sep=''`:

```
In [43]: s.str.cat()
Out[43]: 'abcd'
```

By default, missing values are ignored. Using `na_rep`, they can be given a representation:

```
In [44]: t = pd.Series(['a', 'b', np.nan, 'd'])

In [45]: t.str.cat(sep=',')
Out[45]: 'a,b,d'

In [46]: t.str.cat(sep=',', na_rep='-')
\\\\\\\\\\\\\\\\Out[46]: 'a,b,-,d'
```

Concatenating a Series and something list-like into a Series

The first argument to *cat()* can be a list-like object, provided that it matches the length of the calling `Series` (or `Index`).

```
In [47]: s.str.cat(['A', 'B', 'C', 'D'])
Out[47]:
0    aA
1    bB
2    cC
3    dD
dtype: object
```

Missing values on either side will result in missing values in the result as well, *unless* `na_rep` is specified:

```
In [48]: s.str.cat(t)
Out[48]:
0     aa
1     bb
2    NaN
```

(continues on next page)

(continued from previous page)

```
3     dd
dtype: object

In [49]: s.str.cat(t, na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[49]:
0    aa
1    bb
2    c-
3    dd
dtype: object
```

Concatenating a Series and something array-like into a Series

New in version 0.23.0.

The parameter `others` can also be two-dimensional. In this case, the number or rows must match the lengths of the calling `Series` (or `Index`).

```
In [50]: d = pd.concat([t, s], axis=1)

In [51]: s
Out[51]:
0    a
1    b
2    c
3    d
dtype: object

In [52]: d
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[52]:
     0  1
0    a  a
1    b  b
2  NaN  c
3    d  d

In [53]: s.str.cat(d, na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and an indexed object into a Series, with alignment

New in version 0.23.0.

For concatenation with a `Series` or `DataFrame`, it is possible to align the indexes before concatenation by setting the `join`-keyword.

```
In [54]: u = pd.Series(['b', 'd', 'a', 'c'], index=[1, 3, 0, 2])
```

(continues on next page)

(continued from previous page)

```
In [55]: s
Out[55]:
0    a
1    b
2    c
3    d
dtype: object

In [56]: u
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:
1    b
3    d
0    a
2    c
dtype: object

In [57]: s.str.cat(u)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    ab
1    bd
2    ca
3    dc
dtype: object

In [58]: s.str.cat(u, join='left')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    aa
1    bb
2    cc
3    dd
dtype: object
```

Warning: If the `join` keyword is not passed, the method `cat()` will currently fall back to the behavior before version 0.23.0 (i.e. no alignment), but a `FutureWarning` will be raised if any of the involved indexes differ, since this default will change to `join='left'` in a future version.

The usual options are available for `join` (one of `'left', 'outer', 'inner', 'right'`). In particular, alignment also means that the different lengths do not need to coincide anymore.

```
In [59]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4])

In [60]: s
Out[60]:
0    a
1    b
2    c
3    d
dtype: object

In [61]: v
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[61]:
-1    z
```

(continues on next page)

(continued from previous page)

```
 0    a
 1    b
 3    d
 4    e
dtype: object

In [62]: s.str.cat(v, join='left', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    aa
1    bb
2    c-
3    dd
dtype: object

In [63]: s.str.cat(v, join='outer', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
-1    -z
 0    aa
 1    bb
 2    c-
 3    dd
 4    -e
dtype: object
```

The same alignment can be used when `others` is a `DataFrame`:

```
In [64]: f = d.loc[[3, 2, 1, 0], :]

In [65]: s
Out[65]:
0    a
1    b
2    c
3    d
dtype: object

In [66]: f
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[66]:
     0  1
3    d  d
2  NaN  c
1    b  b
0    a  a

In [67]: s.str.cat(f, join='left', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and many objects into a Series

Several array-like items (specifically: `Series`, `Index`, and 1-dimensional variants of `np.ndarray`) can be combined in a list-like container (including iterators, `dict`-views, etc.).

```
In [68]: s
Out[68]:
0    a
1    b
2    c
3    d
dtype: object

In [69]: u
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[69]:
1    b
3    d
0    a
2    c
dtype: object

In [70]: s.str.cat([u, u.to_numpy()], join='left')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    aab
1    bbd
2    cca
3    ddc
dtype: object
```

All elements without an index (e.g. `np.ndarray`) within the passed list-like must match in length to the calling `Series` (or `Index`), but `Series` and `Index` may have arbitrary length (as long as alignment is not disabled with `join=None`):

```
In [71]: v
Out[71]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: object

In [72]: s.str.cat([v, u, u.to_numpy()], join='outer', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[72]:
-1    -z--
 0    aaab
 1    bbbd
 2    c-ca
 3    dddc
 4    -e--
dtype: object
```

If using `join='right'` on a list-like of `others` that contains different indexes, the union of these indexes will be used as the basis for the final concatenation:

```
In [73]: u.loc[[3]]
```

(continues on next page)

(continued from previous page)

```
Out[73]:
3    d
dtype: object

In [74]: v.loc[[-1, 0]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[74]:
-1    z
 0    a
dtype: object

In [75]: s.str.cat([u.loc[[3]], v.loc[[-1, 0]]], join='right', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[75]:
-1    --z
 0    a-a
 3    dd-
dtype: object
```

4.6.3 Indexing with `.str`

You can use `[]` notation to directly index by position locations. If you index past the end of the string, the result will be a `NaN`.

```
In [76]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
   ....:                'CABA', 'dog', 'cat'])
   ....:

In [77]: s.str[0]
Out[77]:
0      A
1      B
2      C
3      A
4      B
5    NaN
6      C
7      d
8      c
dtype: object

In [78]: s.str[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1    NaN
2    NaN
3      a
4      a
5    NaN
6      A
7      o
8      a
dtype: object
```

4.6.4 Extracting Substrings

Extract first match in each subject (extract)

> **Warning:** In version 0.18.0, `extract` gained the `expand` argument. When `expand=False` it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0). When `expand=True` it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user. `expand=True` is the default since version 0.23.0.

The `extract` method accepts a regular expression with at least one capture group.

Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [79]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'([ab])(\d)', expand=False)
Out[79]:
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

Elements that do not match return a row filled with `NaN`. Thus, a Series of messy strings can be "converted" into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects. The dtype of the result is always object, even if no match is found and the result only contains `NaN`.

Named groups like

```
In [80]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'(?P<letter>[ab])(?P<digit>\d)',
   ....:                                           expand=False)
   ....:
Out[80]:
  letter digit
0      a     1
1      b     2
2    NaN   NaN
```

and optional groups like

```
In [81]: pd.Series(['a1', 'b2', '3']).str.extract(r'([ab])?(\d)', expand=False)
Out[81]:
     0  1
0    a  1
1    b  2
2  NaN  3
```

can also be used. Note that any capture group names in the regular expression will be used for column names; otherwise capture group numbers will be used.

Extracting a regular expression with one group returns a `DataFrame` with one column if `expand=True`.

```
In [82]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=True)
Out[82]:
     0
0    1
1    2
2  NaN
```

It returns a Series if `expand=False`.

```
In [83]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=False)
Out[83]:
0      1
1      2
2    NaN
dtype: object
```

Calling on an `Index` with a regex with exactly one capture group returns a `DataFrame` with one column if `expand=True`.

```
In [84]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])

In [85]: s
Out[85]:
A11    a1
B22    b2
C33    c3
dtype: object

In [86]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[86]:
  letter
0      A
1      B
2      C
```

It returns an `Index` if `expand=False`.

```
In [87]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[87]: Index(['A', 'B', 'C'], dtype='object', name='letter')
```

Calling on an `Index` with a regex with more than one capture group returns a `DataFrame` if `expand=True`.

```
In [88]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[88]:
  letter   1
0      A  11
1      B  22
2      C  33
```

It raises `ValueError` if `expand=False`.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

The table below summarizes the behavior of `extract(expand=False)` (input subject in first column, number of groups in regex in first row)

	1 group	>1 group
Index	Index	ValueError
Series	Series	DataFrame

Extract all matches in each subject (extractall)

New in version 0.18.0.

Unlike `extract` (which returns only the first match),

```
In [89]: s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])

In [90]: s
Out[90]:
A    a1a2
B      b1
C      c1
dtype: object

In [91]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'

In [92]: s.str.extract(two_groups, expand=True)
Out[92]:
  letter digit
A      a     1
B      b     1
C      c     1
```

the `extractall` method returns every match. The result of `extractall` is always a `DataFrame` with a `MultiIndex` on its rows. The last level of the `MultiIndex` is named `match` and indicates the order in the subject.

```
In [93]: s.str.extractall(two_groups)
Out[93]:
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
C 0          c     1
```

When each subject string in the Series has exactly one match,

```
In [94]: s = pd.Series(['a3', 'b3', 'c2'])

In [95]: s
Out[95]:
0    a3
1    b3
2    c2
dtype: object
```

then `extractall(pat).xs(0, level='match')` gives the same result as `extract(pat)`.

```
In [96]: extract_result = s.str.extract(two_groups, expand=True)

In [97]: extract_result
Out[97]:
  letter digit
0      a     3
1      b     3
2      c     2

In [98]: extractall_result = s.str.extractall(two_groups)

In [99]: extractall_result
```

(continues on next page)

(continued from previous page)

```
Out[99]: 
        letter digit
  match             
0 0          a     3
1 0          b     3
2 0          c     2

In [100]: extractall_result.xs(0, level="match")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  letter digit
0      a     3
1      b     3
2      c     2
```

Index also supports .str.extractall. It returns a DataFrame which has the same result as a Series.str.extractall with a default index (starts from 0).

New in version 0.19.0.

```
In [101]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[101]: 
        letter digit
  match             
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1

In [102]: pd.Series(["a1a2", "b1", "c1"]).str.extractall(two_groups)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        letter digit
  match             
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1
```

4.6.5 Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

```
In [103]: pattern = r'[0-9][a-z]'

In [104]: pd.Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
Out[104]: 
0    False
1    False
2     True
3     True
4     True
dtype: bool
```

Or whether elements match a pattern:

```
In [105]: pd.Series(['1', '2', '3a', '3b', '03c']).str.match(pattern)
Out[105]:
0    False
1    False
2     True
3     True
4    False
dtype: bool
```

The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match`, while `contains` relies on `re.search`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```
In [106]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat
↪'])

In [107]: s4.str.contains('A', na=False)
Out[107]:
0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
dtype: bool
```

4.6.6 Creating Indicator Variables

You can extract dummy variables from string columns. For example if they are separated by a `'|'`:

```
In [108]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])

In [109]: s.str.get_dummies(sep='|')
Out[109]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1
```

String `Index` also supports `get_dummies` which returns a `MultiIndex`.

New in version 0.18.1.

```
In [110]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])

In [111]: idx.str.get_dummies(sep='|')
Out[111]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
           codes=[[1, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 1]],
           names=['a', 'b', 'c'])
```

See also `get_dummies()`.

4.6.7 Method Summary

Method	Description
`cat()`	Concatenate strings
`split()`	Split strings on delimiter
`rsplit()`	Split strings on delimiter working from the end of the string
`get()`	Index into each element (retrieve i-th element)
`join()`	Join strings in each element of the Series with passed separator
`get_dummies()`	Split strings on the delimiter returning DataFrame of dummy variables
`contains()`	Return boolean array if each string contains pattern/regex
`replace()`	Replace occurrences of pattern/regex/string with some other string or the return value of a callable given the occurrence
`repeat()`	Duplicate values (`s.str.repeat(3)` equivalent to `x * 3`)
`pad()`	Add whitespace to left, right, or both sides of strings
`center()`	Equivalent to `str.center`
`ljust()`	Equivalent to `str.ljust`
`rjust()`	Equivalent to `str.rjust`
`zfill()`	Equivalent to `str.zfill`
`wrap()`	Split long strings into lines with length less than a given width
`slice()`	Slice each string in the Series
`slice_replace()`	Replace slice in each string with passed value
`count()`	Count occurrences of pattern
`startswith()`	Equivalent to `str.startswith(pat)` for each element
`endswith()`	Equivalent to `str.endswith(pat)` for each element
`findall()`	Compute list of all occurrences of pattern/regex for each string
`match()`	Call `re.match` on each element, returning matched groups as list
`extract()`	Call `re.search` on each element, returning DataFrame with one row for each element and one column for each regex capture group
`extractall()`	Call `re.findall` on each element, returning DataFrame with one row for each match and one column for each regex capture group
`len()`	Compute string lengths
`strip()`	Equivalent to `str.strip`
`rstrip()`	Equivalent to `str.rstrip`
`lstrip()`	Equivalent to `str.lstrip`
`partition()`	Equivalent to `str.partition`
`rpartition()`	Equivalent to `str.rpartition`
`lower()`	Equivalent to `str.lower`
`upper()`	Equivalent to `str.upper`
`find()`	Equivalent to `str.find`
`rfind()`	Equivalent to `str.rfind`
`index()`	Equivalent to `str.index`
`rindex()`	Equivalent to `str.rindex`
`capitalize()`	Equivalent to `str.capitalize`
`swapcase()`	Equivalent to `str.swapcase`
`normalize()`	Return Unicode normal form. Equivalent to `unicodedata.normalize`
`translate()`	Equivalent to `str.translate`
`isalnum()`	Equivalent to `str.isalnum`
`isalpha()`	Equivalent to `str.isalpha`

Continued on next page

Table 2 – continued from previous page

Method	Description
isdigit()	Equivalent to `str.isdigit`
isspace()	Equivalent to `str.isspace`
islower()	Equivalent to `str.islower`
isupper()	Equivalent to `str.isupper`
istitle()	Equivalent to `str.istitle`
isnumeric()	Equivalent to `str.isnumeric`
isdecimal()	Equivalent to `str.isdecimal`

4.7 Working with missing data

In this section, we will discuss missing (also referred to as NA) values in pandas.

Note: The choice of using `NaN` internally to denote missing data was largely for simplicity and performance reasons. It differs from the MaskedArray approach of, for example, `scikits.timeseries`. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

See the *cookbook* for some advanced strategies.

4.7.1 Values considered "missing"

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While `NaN` is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python `None` will arise and we wish to also consider that "missing" or "not available" or "NA".

Note: If you want to consider `inf` and `-inf` to be "NA" in computations, you can set `pandas.options.mode.use_inf_as_na = True`.

```
In [1]: df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   ...:                   columns=['one', 'two', 'three'])
   ...:

In [2]: df['four'] = 'bar'

In [3]: df['five'] = df['one'] > 0

In [4]: df
Out[4]:
        one       two     three four   five
a  0.469112 -0.282863 -1.509059  bar   True
c -1.135632  1.212112 -0.173215  bar  False
e  0.119209 -1.044236 -0.861849  bar   True
f -2.104569 -0.494929  1.071804  bar  False
h  0.721555 -0.706771 -1.039575  bar   True

In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
```

(continues on next page)

(continued from previous page)

```
Out[6]:
        one       two     three four   five
a  0.469112 -0.282863 -1.509059  bar   True
b       NaN       NaN       NaN  NaN    NaN
c -1.135632  1.212112 -0.173215  bar  False
d       NaN       NaN       NaN  NaN    NaN
e  0.119209 -1.044236 -0.861849  bar   True
f -2.104569 -0.494929  1.071804  bar  False
g       NaN       NaN       NaN  NaN    NaN
h  0.721555 -0.706771 -1.039575  bar   True
```

To make detecting missing values easier (and across different array dtypes), pandas provides the `isna()` and `notna()` functions, which are also methods on Series and DataFrame objects:

```
In [7]: df2['one']
Out[7]:
a    0.469112
b         NaN
c   -1.135632
d         NaN
e    0.119209
f   -2.104569
g         NaN
h    0.721555
Name: one, dtype: float64

In [8]: pd.isna(df2['one'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool

In [9]: df2['four'].notna()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: four, dtype: bool

In [10]: df2.isna()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     one    two  three   four   five
```

(continues on next page)

(continued from previous page)

```
a  False  False  False  False  False
b   True   True   True   True   True
c  False  False  False  False  False
d   True   True   True   True   True
e  False  False  False  False  False
f  False  False  False  False  False
g   True   True   True   True   True
h  False  False  False  False  False
```

Warning: One has to be mindful that in Python (and NumPy), the `nan's` don't compare equal, but `None's` **do**. Note that pandas/NumPy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`.

```
In [11]: None == None                                                  # noqa: E711
Out[11]: True

In [12]: np.nan == np.nan
\\\\\\\\\\\\\\Out[12]: False
```

So as compared to above, a scalar equality comparison versus a `None/np.nan` doesn't provide useful information.

```
In [13]: df2['one'] == np.nan
Out[13]:
a    False
b    False
c    False
d    False
e    False
f    False
g    False
h    False
Name: one, dtype: bool
```

Integer Dtypes and Missing Data

Because `NaN` is a float, a column of integers with even one missing values is cast to floating-point dtype (see *Support for integer NA* for more). Pandas provides a nullable integer array, which can be used by explicitly requesting the dtype:

```
In [14]: pd.Series([1, 2, np.nan, 4], dtype=pd.Int64Dtype())
Out[14]:
0      1
1      2
2    NaN
3      4
dtype: Int64
```

Alternatively, the string alias `dtype='Int64'` (note the capital `"I"`) can be used.

See *Nullable Integer Data Type* for more.

Datetimes

For datetime64[ns] types, `NaT` represents missing values. This is a pseudo-native sentinel value that can be represented by NumPy in a singular dtype (datetime64[ns]). pandas objects provide compatibility between `NaT` and `NaN`.

```
In [15]: df2 = df.copy()

In [16]: df2['timestamp'] = pd.Timestamp('20120101')

In [17]: df2
Out[17]:
        one       two     three four   five  timestamp
a  0.469112 -0.282863 -1.509059  bar   True 2012-01-01
c -1.135632  1.212112 -0.173215  bar  False 2012-01-01
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h  0.721555 -0.706771 -1.039575  bar   True 2012-01-01

In [18]: df2.loc[['a', 'c', 'h'], ['one', 'timestamp']] = np.nan

In [19]: df2
Out[19]:
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h       NaN -0.706771 -1.039575  bar   True        NaT

In [20]: df2.get_dtype_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
float64           3
object            1
bool              1
datetime64[ns]    1
dtype: int64
```

Inserting missing data

You can insert missing values by simply assigning to containers. The actual missing value used will be chosen based on the dtype.

For example, numeric containers will always use `NaN` regardless of the missing value type chosen:

```
In [21]: s = pd.Series([1, 2, 3])

In [22]: s.loc[0] = None

In [23]: s
Out[23]:
0    NaN
1    2.0
2    3.0
dtype: float64
```

Likewise, datetime containers will always use `NaT`.

For object containers, pandas will use the value given:

```
In [24]: s = pd.Series(["a", "b", "c"])

In [25]: s.loc[0] = None

In [26]: s.loc[1] = np.nan

In [27]: s
Out[27]:
0    None
1     NaN
2       c
dtype: object
```

Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [28]: a
Out[28]:
        one       two
a       NaN -0.282863
c       NaN  1.212112
e  0.119209 -1.044236
f -2.104569 -0.494929
h -2.104569 -0.706771

In [29]: b
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [30]: a + b
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one  three       two
a       NaN    NaN -0.565727
c       NaN    NaN  2.424224
e  0.238417    NaN -2.088472
f -4.209138    NaN -0.989859
h       NaN    NaN -1.413542
```

The descriptive statistics and computational methods discussed in the *data structure overview* (and listed *here* and *here*) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero.
- If the data are all NA, the result will be 0.
- Cumulative methods like *cumsum()* and *cumprod()* ignore NA values by default, but preserve them in the resulting arrays. To override this behaviour and include NA values, use `skipna=False`.

```
In [31]: df
Out[31]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [32]: df['one'].sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪-1.9853605075978744

In [33]: df.mean(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a   -0.895961
c    0.519449
e   -0.595625
f   -0.509232
h   -0.873173
dtype: float64

In [34]: df.cumsum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  0.929249 -1.682273
e  0.119209 -0.114987 -2.544122
f -1.985361 -0.609917 -1.472318
h       NaN -1.316688 -2.511893

In [35]: df.cumsum(skipna=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  0.929249 -1.682273
e  NaN -0.114987 -2.544122
f  NaN -0.609917 -1.472318
h  NaN -1.316688 -2.511893
```

4.7.2 Sum/Prod of Empties/Nans

Warning: This behavior is now standard as of v0.22.0 and is consistent with the default in `numpy`; previously sum/prod of all-NA or empty Series/DataFrames would return NaN. See *v0.22.0 whatsnew* for more.

The sum of an empty or all-NA Series or column of a DataFrame is 0.

```
In [36]: pd.Series([np.nan]).sum()
Out[36]: 0.0
```

(continues on next page)

(continued from previous page)

```
In [37]: pd.Series([]).sum()
\\\\\\\\\\\\Out[37]: 0.0
```

The product of an empty or all-NA Series or column of a DataFrame is 1.

```
In [38]: pd.Series([np.nan]).prod()
Out[38]: 1.0

In [39]: pd.Series([]).prod()
\\\\\\\\\\\\Out[39]: 1.0
```

4.7.3 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example:

```
In [40]: df
Out[40]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [41]: df.groupby('one').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                two     three
one
-2.104569 -0.494929  1.071804
 0.119209 -1.044236 -0.861849
```

See the groupby section *here* for more information.

Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

4.7.4 Filling missing values: fillna

`fillna()` can "fill in" NA values with non-NA data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [42]: df2
Out[42]:
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h       NaN -0.706771 -1.039575  bar   True        NaT
```

(continues on next page)

(continued from previous page)

```
In [43]: df2.fillna(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three four   five           timestamp
a  0.000000 -0.282863 -1.509059  bar   True                   0
c  0.000000  1.212112 -0.173215  bar  False                   0
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01 00:00:00
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01 00:00:00
h  0.000000 -0.706771 -1.039575  bar   True                   0

In [44]: df2['one'].fillna('missing')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a     missing
c     missing
e    0.119209
f    -2.10457
h     missing
Name: one, dtype: object
```

Fill gaps forward or backward

Using the same filling arguments as *reindexing*, we can propagate non-NA values forward or backward:

```
In [45]: df
Out[45]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [46]: df.fillna(method='pad')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h -2.104569 -0.706771 -1.039575
```

Limit the amount of filling

If we only want consecutive gaps filled up to a certain number of data points, we can use the *limit* keyword:

```
In [47]: df
Out[47]:
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN       NaN       NaN
f  NaN       NaN       NaN
h  NaN -0.706771 -1.039575

In [48]: df.fillna(method='pad', limit=1)
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN  1.212112 -0.173215
f  NaN       NaN       NaN
h  NaN -0.706771 -1.039575
```

To remind you, these are the available filling methods:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward

With time series data, using pad/ffill is extremely common so that the "last known value" is available at every time point.

ffill() is equivalent to `fillna(method='ffill')` and *bfill()* is equivalent to `fillna(method='bfill')`

4.7.5 Filling with a PandasObject

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```
In [49]: dff = pd.DataFrame(np.random.randn(10, 3), columns=list('ABC'))

In [50]: dff.iloc[3:5, 0] = np.nan

In [51]: dff.iloc[4:6, 1] = np.nan

In [52]: dff.iloc[5:8, 2] = np.nan

In [53]: dff
Out[53]:
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3       NaN  0.577046 -1.715002
4       NaN       NaN -1.157892
5 -1.344312       NaN       NaN
6 -0.109050  1.643563       NaN
7  0.357021 -0.674600       NaN
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960

In [54]: dff.fillna(dff.mean())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
```

(continues on next page)

(continued from previous page)

```
2  0.113648 -1.478427  0.524988
3 -0.140857  0.577046 -1.715002
4 -0.140857 -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960

In [55]: dff.fillna(dff.mean()['B':'C'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3       NaN  0.577046 -1.715002
4       NaN -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960
```

Same result as above, but is aligning the 'fill' value which is a Series in this case.

```
In [56]: dff.where(pd.notna(dff), dff.mean(), axis='columns')
Out[56]:
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3 -0.140857  0.577046 -1.715002
4 -0.140857 -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960
```

4.7.6 Dropping axis labels with missing data: dropna

You may wish to simply exclude labels from a data set which refer to missing data. To do this, use *dropna()*:

```
In [57]: df
Out[57]:
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN  0.000000  0.000000
f  NaN  0.000000  0.000000
h  NaN -0.706771 -1.039575

In [58]: df.dropna(axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
Empty DataFrame
Columns: [one, two, three]
Index: []

In [59]: df.dropna(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
        two     three
a -0.282863 -1.509059
c  1.212112 -0.173215
e  0.000000  0.000000
f  0.000000  0.000000
h -0.706771 -1.039575

In [60]: df['one'].dropna()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Series([], Name: one, dtype: float64)
```

An equivalent *dropna()* is available for Series. DataFrame.dropna has considerably more options than Series.dropna, which can be examined *in the API*.

4.7.7 Interpolation

New in version 0.21.0: The `limit_area` keyword argument was added.

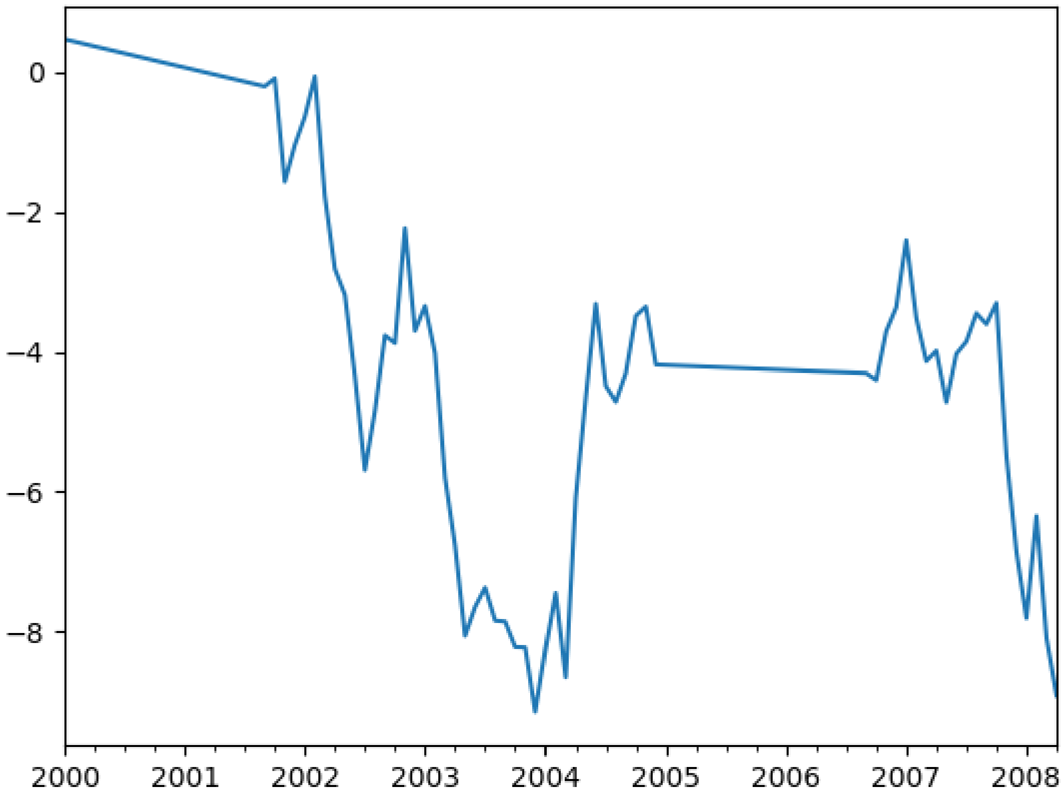
Both Series and DataFrame objects have *interpolate()* that, by default, performs linear interpolation at missing data points.

```
In [61]: ts
Out[61]:
2000-01-31    0.469112
2000-02-29         NaN
2000-03-31         NaN
2000-04-28         NaN
2000-05-31         NaN
2000-06-30         NaN
2000-07-31         NaN
                ...
2007-10-31   -3.305259
2007-11-30   -5.485119
2007-12-31   -6.854968
2008-01-31   -7.809176
2008-02-29   -6.346480
2008-03-31   -8.089641
2008-04-30   -8.916232
Freq: BM, Length: 100, dtype: float64

In [62]: ts.count()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪61

In [63]: ts.interpolate().count()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪100
```

(continues on next page)

(continued from previous page)

```
In [64]: ts.interpolate().plot()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f37e5e58a58>
```



Index aware interpolation is available via the `method` keyword:

```
In [65]: ts2
Out[65]:
2000-01-31    0.469112
2000-02-29         NaN
2002-07-31   -5.689738
2005-01-31         NaN
2008-04-30   -8.916232
dtype: float64

In [66]: ts2.interpolate()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-31    0.469112
2000-02-29   -2.610313
2002-07-31   -5.689738
2005-01-31   -7.302985
2008-04-30   -8.916232
```

(continues on next page)

(continued from previous page)

```
dtype: float64

In [67]: ts2.interpolate(method='time')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-31    0.469112
2000-02-29    0.273272
2002-07-31   -5.689738
2005-01-31   -7.095568
2008-04-30   -8.916232
dtype: float64
```

For a floating-point index, use `method='values'`:

```
In [68]: ser
Out[68]:
0.0      0.0
1.0      NaN
10.0    10.0
dtype: float64

In [69]: ser.interpolate()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[69]:
0.0      0.0
1.0      5.0
10.0    10.0
dtype: float64

In [70]: ser.interpolate(method='values')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0.0      0.0
1.0      1.0
10.0    10.0
dtype: float64
```

You can also interpolate with a DataFrame:

```
In [71]: df = pd.DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
   ....:                    'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
   ....:

In [72]: df
Out[72]:
     A      B
0  1.0   0.25
1  2.1    NaN
2  NaN    NaN
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40

In [73]: df.interpolate()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A      B
0  1.0   0.25
```

(continues on next page)

(continued from previous page)

```
1  2.1   1.50
2  3.4   2.75
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
```

The method argument gives access to fancier interpolation methods. If you have scipy installed, you can pass the name of a 1-d interpolation routine to method. You'll want to consult the full scipy interpolation documentation and reference guide for details. The appropriate interpolation method will depend on the type of data you are working with.

- If you are dealing with a time series that is growing at an increasing rate, method='quadratic' may be appropriate.
- If you have values approximating a cumulative distribution function, then method='pchip' should work well.
- To fill missing values with goal of smooth plotting, consider method='akima'.

Warning: These methods require scipy.

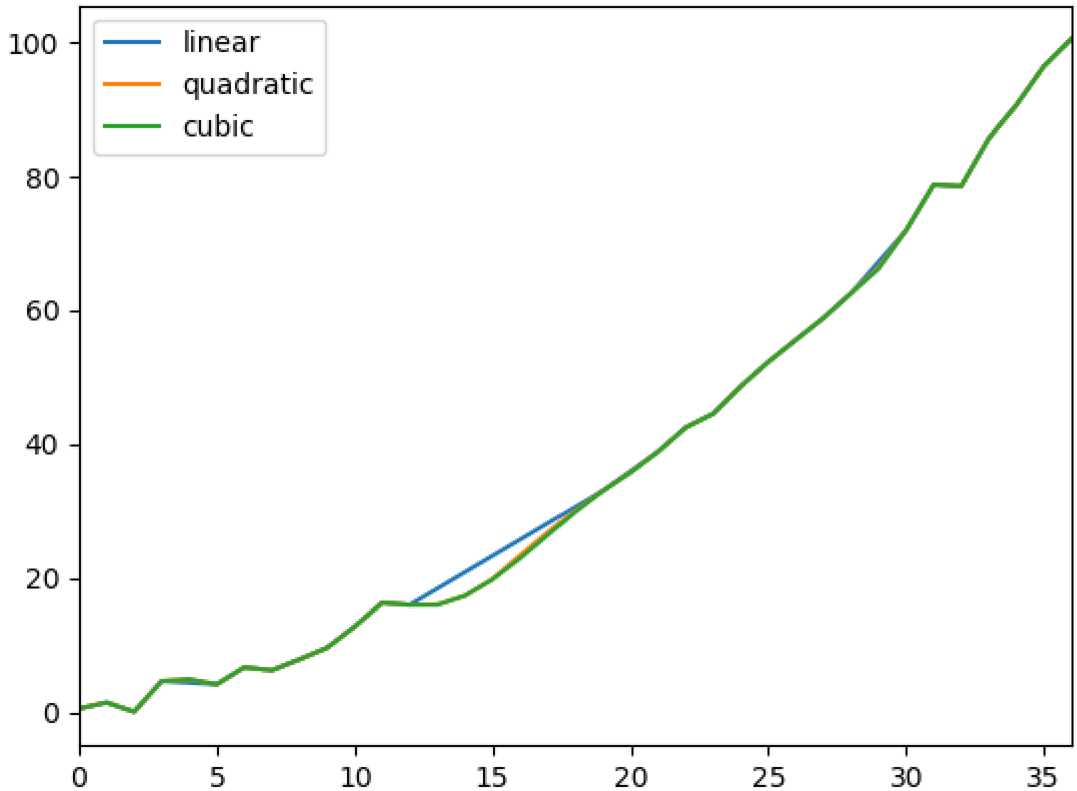
```
In [74]: df.interpolate(method='barycentric')
Out[74]:
      A       B
0  1.00   0.250
1  2.10  -7.660
2  3.53  -4.515
3  4.70   4.000
4  5.60  12.200
5  6.80  14.400

In [75]: df.interpolate(method='pchip')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
         A          B
0  1.00000   0.250000
1  2.10000   0.672808
2  3.43454   1.928950
3  4.70000   4.000000
4  5.60000  12.200000
5  6.80000  14.400000

In [76]: df.interpolate(method='akima')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A          B
0  1.000000   0.250000
1  2.100000  -0.873316
2  3.406667   0.320034
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [77]: df.interpolate(method='spline', order=2)
Out[77]:
          A          B
0  1.000000   0.250000
1  2.100000  -0.428598
2  3.404545   1.206900
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000

In [78]: df.interpolate(method='polynomial', order=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A          B
0  1.000000   0.250000
1  2.100000  -2.703846
2  3.451351  -1.453846
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

Compare several methods:

```
In [79]: np.random.seed(2)

In [80]: ser = pd.Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))

In [81]: bad = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])

In [82]: ser[bad] = np.nan

In [83]: methods = ['linear', 'quadratic', 'cubic']

In [84]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})

In [85]: df.plot()
Out[85]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e5e2c6d8>
```



Another use case is interpolation at *new* values. Suppose you have 100 observations from some distribution. And let's suppose that you're particularly interested in what's happening around the middle. You can mix pandas' `reindex` and `interpolate` methods to interpolate at the new values.

```
In [86]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [87]: new_index = ser.index | pd.Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [88]: interp_s = ser.reindex(new_index).interpolate(method='pchip')

In [89]: interp_s[49:51]
Out[89]:
49.00    0.471410
49.25    0.476841
49.50    0.481780
49.75    0.485998
50.00    0.489266
50.25    0.491814
50.50    0.493995
50.75    0.495763
51.00    0.497074
dtype: float64
```

Like other pandas fill methods, `interpolate()` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive NaN values filled since the last valid observation:

```
In [90]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan,
   ....:                  np.nan, 13, np.nan, np.nan])
   ....:

# fill all consecutive values in a forward direction
In [91]: ser.interpolate()
Out[91]:
0     NaN
1     NaN
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
8    13.0
dtype: float64

# fill one consecutive value in a forward direction
In [92]: ser.interpolate(limit=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5     NaN
6    13.0
7    13.0
8     NaN
dtype: float64
```

By default, NaN values are filled in a forward direction. Use limit_direction parameter to fill backward or from both directions.

```
# fill one consecutive value backwards
In [93]: ser.interpolate(limit=1, limit_direction='backward')
Out[93]:
0     NaN
1     5.0
2     5.0
3     NaN
4     NaN
5    11.0
6    13.0
7     NaN
8     NaN
dtype: float64

# fill one consecutive value in both directions
In [94]: ser.interpolate(limit=1, limit_direction='both')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     NaN
1     5.0
```

(continues on next page)

(continued from previous page)

```
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7    13.0
8     NaN
dtype: float64

# fill all consecutive values in both directions
In [95]: ser.interpolate(limit_direction='both')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     5.0
1     5.0
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
8    13.0
dtype: float64
```

By default, NaN values are filled whether they are inside (surrounded by) existing valid values, or outside existing valid values. Introduced in v0.23 the `limit_area` parameter restricts filling to either inside or outside values.

```
# fill one consecutive inside value in both directions
In [96]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
Out[96]:
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7     NaN
8     NaN
dtype: float64

# fill all consecutive outside values backward
In [97]: ser.interpolate(limit_direction='backward', limit_area='outside')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
8     NaN
dtype: float64
```

(continues on next page)

(continued from previous page)

```
# fill all consecutive outside values in both directions
In [98]: ser.interpolate(limit_direction='both', limit_area='outside')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7    13.0
8    13.0
dtype: float64
```

4.7.8 Replacing Generic Values

Often times we want to replace arbitrary values with other values.

replace() in Series and *replace()* in DataFrame provides an efficient yet flexible way to perform such replacements.

For a Series, you can replace a single value or a list of values by another value:

```
In [99]: ser = pd.Series([0., 1., 2., 3., 4.])

In [100]: ser.replace(0, 5)
Out[100]:
0    5.0
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [101]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[101]:
0    4.0
1    3.0
2    2.0
3    1.0
4    0.0
dtype: float64
```

You can also specify a mapping dict:

```
In [102]: ser.replace({0: 10, 1: 100})
Out[102]:
0     10.0
1    100.0
2      2.0
3      3.0
4      4.0
dtype: float64
```

For a DataFrame, you can specify individual values by column:

```
In [103]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [104]: df.replace({'a': 0, 'b': 5}, 100)
Out[104]:
     a    b
0  100  100
1    1    6
2    2    7
3    3    8
4    4    9
```

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

```
In [105]: ser.replace([1, 2, 3], method='pad')
Out[105]:
0    0.0
1    0.0
2    0.0
3    0.0
4    4.0
dtype: float64
```

4.7.9 String/Regular Expression Replacement

Note: Python strings prefixed with the `r` character such as `r'hello world'` are so-called "raw" strings. They have different semantics regarding backslashes than strings without this prefix. Backslashes in raw strings will be interpreted as an escaped backslash, e.g., `r'\' == '\\'`. You should read about them if this is unclear.

Replace the '.' with `NaN` (str -> str):

```
In [106]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}

In [107]: df = pd.DataFrame(d)

In [108]: df.replace('.', np.nan)
Out[108]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Now do it with a regular expression that removes surrounding whitespace (regex -> regex):

```
In [109]: df.replace(r'\s*\.\s*', np.nan, regex=True)
Out[109]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Replace a few different values (list -> list):

```
In [110]: df.replace(['a', '.'], ['b', np.nan])
Out[110]:
   a    b    c
0  0    b    b
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

list of regex -> list of regex:

```
In [111]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
Out[111]:
   a       b       c
0  0  {stuff  {stuff
1  1       b       b
2  2     dot     NaN
3  3     dot       d
```

Only search in column `'b'` (dict -> dict):

```
In [112]: df.replace({'b': '.'}, {'b': np.nan})
Out[112]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict):

```
In [113]: df.replace({'b': r'\s*\.\s*'}, {'b': np.nan}, regex=True)
Out[113]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can pass nested dictionaries of regular expressions that use `regex=True`:

```
In [114]: df.replace({'b': {'b': r''}}, regex=True)
Out[114]:
   a  b    c
0  0  a    a
1  1       b
2  2  .  NaN
3  3  .    d
```

Alternatively, you can pass the nested dictionary like so:

```
In [115]: df.replace(regex={'b': {r'\s*\.\s*': np.nan}})
Out[115]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well.

```
In [116]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
Out[116]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  .ty  NaN
3  3  .ty    d
```

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex).

```
In [117]: df.replace([r'\s*\.\s*', r'a|b'], np.nan, regex=True)
Out[117]:
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be:

```
In [118]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=np.nan)
Out[118]:
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

This can be convenient if you do not want to pass `regex=True` every time you want to use a regular expression.

Note: Anywhere in the above `replace` examples that you see a regular expression a compiled regular expression is valid as well.

4.7.10 Numeric Replacement

replace() is similar to *fillna()*.

```
In [119]: df = pd.DataFrame(np.random.randn(10, 2))

In [120]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5

In [121]: df.replace(1.5, np.nan)
Out[121]:
          0         1
0 -0.844214 -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
```

(continues on next page)

(continued from previous page)

```
4        NaN       NaN
5        NaN       NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9        NaN       NaN
```

Replacing more than one value is possible by passing a list.

```
In [122]: df00 = df.iloc[0, 0]

In [123]: df.replace([1.5, df00], [np.nan, 'a'])
Out[123]:
          0         1
0         a  -1.02141
1  0.432396  -0.32358
2  0.423825   0.79918
3   1.26261  0.751965
4       NaN       NaN
5       NaN       NaN
6 -0.498174   -1.0608
7  0.591667 -0.183257
8   1.01985  -1.48247
9       NaN       NaN

In [124]: df[1].dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪dtype('float64')
```

You can also operate on the DataFrame in place:

```
In [125]: df.replace(1.5, np.nan, inplace=True)
```

Warning: When replacing multiple `bool` or `datetime64` objects, the first argument to `replace` (`to_replace`) must match the type of the value being replaced. For example,

```
>>> s = pd.Series([True, False, True])
>>> s.replace({'a string': 'new value', True: False})  # raises
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

will raise a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

However, when replacing a *single* object such as,

```
In [126]: s = pd.Series([True, False, True])

In [127]: s.replace('a string', 'another string')
Out[127]:
0     True
1    False
2     True
dtype: bool
```

the original `NDFrame` object will be returned untouched. We're working on unifying this API, but for backwards compatibility reasons we cannot break the latter behavior. See GH6354 for more details.

Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we've established some "casting rules". When a reindexing operation introduces missing data, the Series will be cast according to the rules introduced in the table below.

data type	Cast to
integer	float
boolean	object
float	no cast
object	no cast

For example:

```
In [128]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [129]: s > 0
Out[129]:
0    True
2    True
4    True
6    True
7    True
dtype: bool

In [130]: (s > 0).dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[130]:␣
↪dtype('bool')

In [131]: crit = (s > 0).reindex(list(range(8)))

In [132]: crit
Out[132]:
0    True
1     NaN
2    True
3     NaN
4    True
5     NaN
6    True
7    True
dtype: object

In [133]: crit.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪dtype('O')
```

Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

```
In [134]: reindexed = s.reindex(list(range(8))).fillna(0)

In [135]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-135-0dac417a4890> in <module>
----> 1 reindexed[crit]

/pandas/pandas/core/series.py in __getitem__(self, key)
    906                 key = list(key)
    907
--> 908         if com.is_bool_indexer(key):
    909             key = check_bool_indexer(self.index, key)
    910

/pandas/pandas/core/common.py in is_bool_indexer(key)
    122             if not lib.is_bool_array(key):
    123                 if isna(key).any():
--> 124                     raise ValueError(na_msg)
    125                 return False
    126             return True

ValueError: cannot index with vector containing NA / NaN values
```

However, these can be filled in using `fillna()` and it will work fine:

```
In [136]: reindexed[crit.fillna(False)]
Out[136]:
0    0.126504
2    0.696198
4    0.697416
6    0.601516
7    0.003659
dtype: float64

In [137]: reindexed[crit.fillna(True)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪
0    0.126504
1    0.000000
2    0.696198
3    0.000000
4    0.697416
5    0.000000
6    0.601516
7    0.003659
dtype: float64
```

Pandas provides a nullable integer dtype, but you must explicitly request it when creating the series or column. Notice that we use a capital "I" in the `dtype="Int64"`.

```
In [138]: s = pd.Series([0, 1, np.nan, 3, 4], dtype="Int64")

In [139]: s
Out[139]:
0      0
1      1
2    NaN
3      3
4      4
dtype: Int64
```

See *Nullable Integer Data Type* for more.

4.8 Categorical Data

This is an introduction to pandas categorical data type, including a short comparison with R's `factor`.

Categoricals are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (*categories*; *levels* in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

In contrast to statistical categorical variables, categorical data might have an order (e.g. 'strongly agree' vs 'agree' or 'first observation' vs. 'second observation'), but numerical operations (additions, divisions, . . .) are not possible.

All values of categorical data are either in *categories* or *np.nan*. Order is defined by the order of *categories*, not lexical order of the values. Internally, the data structure consists of a *categories* array and an integer array of *codes* which point to the real value in the *categories* array.

The categorical data type is useful in the following cases:

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory, see *here*.
- The lexical order of a variable is not the same as the logical order ("one", "two", "three"). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order, see *here*.
- As a signal to other Python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

See also the *API docs on categoricals*.

4.8.1 Object Creation

Series Creation

Categorical `Series` or columns in a `DataFrame` can be created in several ways:

By specifying `dtype="category"` when constructing a `Series`:

```
In [1]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [2]: s
Out[2]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]
```

By converting an existing `Series` or column to a `category` dtype:

```
In [3]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [4]: df["B"] = df["A"].astype('category')

In [5]: df
```

(continues on next page)

(continued from previous page)

```
Out[5]:
   A  B
0  a  a
1  b  b
2  c  c
3  a  a
```

By using special functions, such as `cut()`, which groups data into discrete bins. See the *example on tiling* in the docs.

```
In [6]: df = pd.DataFrame({'value': np.random.randint(0, 100, 20)})

In [7]: labels = ["{0} - {1}".format(i, i + 9) for i in range(0, 100, 10)]

In [8]: df['group'] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)

In [9]: df.head(10)
Out[9]:
   value    group
0     65  60 - 69
1     49  40 - 49
2     56  50 - 59
3     43  40 - 49
4     43  40 - 49
5     91  90 - 99
6     32  30 - 39
7     87  80 - 89
8     36  30 - 39
9      8    0 - 9
```

By passing a `pandas.Categorical` object to a `Series` or assigning it to a `DataFrame`.

```
In [10]: raw_cat = pd.Categorical(["a", "b", "c", "a"], categories=["b", "c", "d"],
   ....:                          ordered=False)
   ....:

In [11]: s = pd.Series(raw_cat)

In [12]: s
Out[12]:
0    NaN
1      b
2      c
3    NaN
dtype: category
Categories (3, object): [b, c, d]

In [13]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [14]: df["B"] = raw_cat

In [15]: df
Out[15]:
   A    B
0  a  NaN
1  b    b
```

(continues on next page)

(continued from previous page)

```
2  c    c
3  a  NaN
```

Categorical data has a specific `category` *dtype*:

```
In [16]: df.dtypes
Out[16]:
A      object
B    category
dtype: object
```

DataFrame Creation

Similar to the previous section where a single column was converted to categorical, all columns in a `DataFrame` can be batch converted to categorical either during or after construction.

This can be done during construction by specifying `dtype="category"` in the `DataFrame` constructor:

```
In [17]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')}, dtype="category")

In [18]: df.dtypes
Out[18]:
A    category
B    category
dtype: object
```

Note that the categories present in each column differ; the conversion is done column by column, so only labels present in a given column are categories:

```
In [19]: df['A']
Out[19]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [20]: df['B']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

New in version 0.23.0.

Analogously, all columns in an existing `DataFrame` can be batch converted using *`DataFrame.astype()`*:

```
In [21]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [22]: df_cat = df.astype('category')
```

(continues on next page)

(continued from previous page)

```
In [23]: df_cat.dtypes
Out[23]: 
A    category
B    category
dtype: object
```

This conversion is likewise done column by column:

```
In [24]: df_cat['A']
Out[24]: 
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [25]: df_cat['B']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

Controlling Behavior

In the examples above where we passed `dtype='category'`, we used the default behavior:

1. Categories are inferred from the data.
2. Categories are unordered.

To control those behaviors, instead of passing `'category'`, use an instance of `CategoricalDtype`.

```
In [26]: from pandas.api.types import CategoricalDtype

In [27]: s = pd.Series(["a", "b", "c", "a"])

In [28]: cat_type = CategoricalDtype(categories=["b", "c", "d"],
   ....:                             ordered=True)
   ....: 

In [29]: s_cat = s.astype(cat_type)

In [30]: s_cat
Out[30]: 
0    NaN
1      b
2      c
3    NaN
dtype: category
Categories (3, object): [b < c < d]
```

Similarly, a `CategoricalDtype` can be used with a `DataFrame` to ensure that categories are consistent among all columns.

```
In [31]: from pandas.api.types import CategoricalDtype

In [32]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [33]: cat_type = CategoricalDtype(categories=list('abcd'),
   ....:                             ordered=True)
   ....:

In [34]: df_cat = df.astype(cat_type)

In [35]: df_cat['A']
Out[35]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (4, object): [a < b < c < d]

In [36]: df_cat['B']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (4, object): [a < b < c < d]
```

Note: To perform table-wise conversion, where all labels in the entire `DataFrame` are used as categories for each column, the `categories` parameter can be determined programmatically by `categories = pd.unique(df.to_numpy().ravel())`.

If you already have `codes` and `categories`, you can use the *`from_codes()`* constructor to save the factorize step during normal constructor mode:

```
In [37]: splitter = np.random.choice([0, 1], 5, p=[0.5, 0.5])

In [38]: s = pd.Series(pd.Categorical.from_codes(splitter,
   ....:                                         categories=["train", "test"]))
   ....:
```

Regaining Original Data

To get back to the original `Series` or NumPy array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:

```
In [39]: s = pd.Series(["a", "b", "c", "a"])

In [40]: s
Out[40]:
```

(continues on next page)

(continued from previous page)

```
0    a
1    b
2    c
3    a
dtype: object

In [41]: s2 = s.astype('category')

In [42]: s2
Out[42]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [43]: s2.astype(str)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[43]:␣
↪
0    a
1    b
2    c
3    a
dtype: object

In [44]: np.asarray(s2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array(['a', 'b', 'c', 'a'], dtype=object)
```

Note: In contrast to R's *factor* function, categorical data is not converting input values to strings; categories will end up the same data type as the original values.

Note: In contrast to R's *factor* function, there is currently no way to assign/change labels at creation time. Use *categories* to change the categories after creation time.

4.8.2 CategoricalDtype

Changed in version 0.21.0.

A categorical's type is fully described by

1. `categories`: a sequence of unique values and no missing values
2. `ordered`: a boolean

This information can be stored in a `CategoricalDtype`. The `categories` argument is optional, which implies that the actual categories should be inferred from whatever is present in the data when the *pandas.Categorical* is created. The categories are assumed to be unordered by default.

```
In [45]: from pandas.api.types import CategoricalDtype
```

(continues on next page)

(continued from previous page)

```
In [46]: CategoricalDtype(['a', 'b', 'c'])
Out[46]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=None)

In [47]: CategoricalDtype(['a', 'b', 'c'], ordered=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]:␣
↪CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [48]: CategoricalDtype()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪CategoricalDtype(categories=None, ordered=None)
```

A `CategoricalDtype` can be used in any place pandas expects a *dtype*. For example `pandas.read_csv()`, `pandas.DataFrame.astype()`, or in the `Series` constructor.

Note: As a convenience, you can use the string `'category'` in place of a `CategoricalDtype` when you want the default behavior of the categories being unordered, and equal to the set values present in the array. In other words, `dtype='category'` is equivalent to `dtype=CategoricalDtype()`.

Equality Semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and order. When comparing two unordered categoricals, the order of the `categories` is not considered.

```
In [49]: c1 = CategoricalDtype(['a', 'b', 'c'], ordered=False)

# Equal, since order is not considered when ordered=False
In [50]: c1 == CategoricalDtype(['b', 'c', 'a'], ordered=False)
Out[50]: True

# Unequal, since the second CategoricalDtype is ordered
In [51]: c1 == CategoricalDtype(['a', 'b', 'c'], ordered=True)
\\\\\\\\\\\\\\Out[51]: False
```

All instances of `CategoricalDtype` compare equal to the string `'category'`.

```
In [52]: c1 == 'category'
Out[52]: True
```

Warning: Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances `CategoricalDtype` compare equal to `'category'`, all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of `categories` or `ordered`.

4.8.3 Description

Using `describe()` on categorical data will produce similar output to a `Series` or `DataFrame` of type `string`.

```
In [53]: cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])

In [54]: df = pd.DataFrame({"cat": cat, "s": ["a", "c", "c", np.nan]})
```

(continues on next page)

(continued from previous page)

```
In [55]: df.describe()
Out[55]:
       cat  s
count    3  3
unique   2  2
top      c  c
freq     2  2

In [56]: df["cat"].describe()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:␣
↪
count      3
unique     2
top        c
freq       2
Name: cat, dtype: object
```

4.8.4 Working with categories

Categorical data has a *categories* and a *ordered* property, which list their possible values and whether the ordering matters or not. These properties are exposed as `s.cat.categories` and `s.cat.ordered`. If you don't manually specify categories and ordering, they are inferred from the passed arguments.

```
In [57]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [58]: s.cat.categories
Out[58]: Index(['a', 'b', 'c'], dtype='object')

In [59]: s.cat.ordered
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[59]: False
```

It's also possible to pass in the categories in a specific order:

```
In [60]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"],
   ....:                    categories=["c", "b", "a"]))
   ....:

In [61]: s.cat.categories
Out[61]: Index(['c', 'b', 'a'], dtype='object')

In [62]: s.cat.ordered
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[62]: False
```

Note: New categorical data are **not** automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered `Categorical`.

Note: The result of *unique()* is not always the same as `Series.cat.categories`, because `Series.unique()` has a couple of guarantees, namely that it returns categories in the order of appearance, and it only includes values that are actually present.

```
In [63]: s = pd.Series(list('babc')).astype(CategoricalDtype(list('abcd')))
```

(continues on next page)

(continued from previous page)

```
In [64]: s
Out[64]:
0    b
1    a
2    b
3    c
dtype: category
Categories (4, object): [a, b, c, d]

# categories
In [65]: s.cat.categories
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[65]:
↪Index(['a', 'b', 'c', 'd'], dtype='object')

# uniques
In [66]: s.unique()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
[b, a, c]
Categories (3, object): [b, a, c]
```

Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `rename_categories()` method:

```
In [67]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [68]: s
Out[68]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [69]: s.cat.categories = ["Group %s" % g for g in s.cat.categories]

In [70]: s
Out[70]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]

In [71]: s = s.cat.rename_categories([1, 2, 3])

In [72]: s
Out[72]:
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2    3
3    1
dtype: category
Categories (3, int64): [1, 2, 3]

# You can also pass a dict-like object to map the renaming
In [73]: s = s.cat.rename_categories({1: 'x', 2: 'y', 3: 'z'})

In [74]: s
Out[74]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Note: In contrast to R's *factor*, categorical data can have categories of other types than string.

Note: Be aware that assigning new categories is an inplace operation, while most other operations under `Series.cat` per default return a new `Series` of dtype *category*.

Categories must be unique or a *ValueError* is raised:

```
In [75]: try:
   ....:     s.cat.categories = [1, 1, 1]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorical categories must be unique
```

Categories must also not be `NaN` or a *ValueError* is raised:

```
In [76]: try:
   ....:     s.cat.categories = [1, 2, np.nan]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorial categories cannot be null
```

Appending new categories

Appending categories can be done by using the `add_categories()` method:

```
In [77]: s = s.cat.add_categories([4])

In [78]: s.cat.categories
Out[78]: Index(['x', 'y', 'z', 4], dtype='object')

In [79]: s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[79]:
```

(continues on next page)

(continued from previous page)

```
0    x
1    y
2    z
3    x
dtype: category
Categories (4, object): [x, y, z, 4]
```

Removing categories

Removing categories can be done by using the `remove_categories()` method. Values which are removed are replaced by `np.nan`.:

```
In [80]: s = s.cat.remove_categories([4])

In [81]: s
Out[81]: 
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Removing unused categories

Removing unused categories can also be done:

```
In [82]: s = pd.Series(pd.Categorical(["a", "b", "a"],
   ....:                 categories=["a", "b", "c", "d"]))
   ....: 

In [83]: s
Out[83]: 
0    a
1    b
2    a
dtype: category
Categories (4, object): [a, b, c, d]

In [84]: s.cat.remove_unused_categories()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[84]: 
↪
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]
```

Setting categories

If you want to do remove and add new categories in one step (which has some speed advantage), or simply set the categories to a predefined scale, use `set_categories()`.

```
In [85]: s = pd.Series(["one", "two", "four", "-"], dtype="category")

In [86]: s
Out[86]:
0     one
1     two
2    four
3       -
dtype: category
Categories (4, object): [-, four, one, two]

In [87]: s = s.cat.set_categories(["one", "two", "three", "four"])

In [88]: s
Out[88]:
0     one
1     two
2    four
3     NaN
dtype: category
Categories (4, object): [one, two, three, four]
```

Note: Be aware that `Categorical.set_categories()` cannot know whether some category is omitted intentionally or because it is misspelled or (under Python3) due to a type difference (e.g., NumPy S1 dtype and Python strings). This can result in surprising behaviour!

4.8.5 Sorting and Order

If categorical data is ordered (`s.cat.ordered == True`), then the order of the categories has a meaning and certain operations are possible. If the categorical is unordered, `.min()/.max()` will raise a `TypeError`.

```
In [89]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], ordered=False))

In [90]: s.sort_values(inplace=True)

In [91]: s = pd.Series(["a", "b", "c", "a"]).astype(
   ....:     CategoricalDtype(ordered=True)
   ....: )
   ....:

In [92]: s.sort_values(inplace=True)

In [93]: s
Out[93]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [94]: s.min(), s.max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[94]:
↪('a', 'c')
```

You can set categorical data to be ordered by using `as_ordered()` or unordered by using `as_unordered()`. These will by default return a *new* object.

```
In [95]: s.cat.as_ordered()
Out[95]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [96]: s.cat.as_unordered()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[96]:
↪
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a, b, c]
```

Sorting will use the order defined by categories, not any lexical order present on the data type. This is even true for strings and numeric data:

```
In [97]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [98]: s = s.cat.set_categories([2, 3, 1], ordered=True)

In [99]: s
Out[99]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [100]: s.sort_values(inplace=True)

In [101]: s
Out[101]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [102]: s.min(), s.max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[102]:
↪(2, 1)
```

Reordering

Reordering the categories is possible via the `Categorical.reorder_categories()` and the `Categorical.set_categories()` methods. For `Categorical.reorder_categories()`, all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [103]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [104]: s = s.cat.reorder_categories([2, 3, 1], ordered=True)

In [105]: s
Out[105]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [106]: s.sort_values(inplace=True)

In [107]: s
Out[107]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [108]: s.min(), s.max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[108]:
↪(2, 1)
```

Note: Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the `Series`, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the `Series` are changed.

Note: If the `Categorical` is not ordered, *`Series.min()`* and *`Series.max()`* will raise `TypeError`. Numeric operations like +, -, *, / and operations based on them (e.g. *`Series.median()`*, which would need to compute the mean between two values if the length of an array is even) do not work and raise a `TypeError`.

Multi Column Sorting

A categorical dtyped column will participate in a multi-column sort in a similar manner to other columns. The ordering of the categorical is determined by the `categories` of that column.

```
In [109]: dfs = pd.DataFrame({'A': pd.Categorical(list('bbeebbaa'),
   .....:                                         categories=['e', 'a', 'b'],
   .....:                                         ordered=True),
   .....:                     'B': [1, 2, 1, 2, 2, 1, 2, 1]})
   .....:

In [110]: dfs.sort_values(by=['A', 'B'])
```

(continues on next page)

(continued from previous page)

```
Out[110]:
   A  B
2  e  1
3  e  2
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
```

Reordering the `categories` changes a future sort.

```
In [111]: dfs['A'] = dfs['A'].cat.reorder_categories(['a', 'b', 'e'])

In [112]: dfs.sort_values(by=['A', 'B'])
Out[112]:
   A  B
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
2  e  1
3  e  2
```

4.8.6 Comparisons

Comparing categorical data with other objects is possible in three cases:

- Comparing equality (== and !=) to a list-like object (list, Series, array, . . .) of the same length as the categorical data.
- All comparisons (==, !=, >, >=, <, and <=) of categorical data to another categorical Series, when `ordered==True` and the *categories* are the same.
- All comparisons of a categorical data to a scalar.

All other comparisons, especially "non-equality" comparisons of two categoricals with different categories or a categorical with any list-like object, will raise a `TypeError`.

Note: Any "non-equality" comparisons of categorical data with a `Series`, `np.array`, `list` or categorical data with different categories or ordering will raise a `TypeError` because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [113]: cat = pd.Series([1, 2, 3]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
   .....:

In [114]: cat_base = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
```

(continues on next page)

(continued from previous page)

```
   .....:

In [115]: cat_base2 = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype(ordered=True)
   .....: )
   .....:

In [116]: cat
Out[116]:
0    1
1    2
2    3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [117]: cat_base
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[117]:␣
↪
0    2
1    2
2    2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [118]: cat_base2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2
1    2
2    2
dtype: category
Categories (1, int64): [2]
```

Comparing to a categorical with the same categories and ordering or to a scalar works:

```
In [119]: cat > cat_base
Out[119]:
0     True
1    False
2    False
dtype: bool

In [120]: cat > 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[120]:
0     True
1    False
2    False
dtype: bool
```

Equality comparisons work with any list-like object of same length and scalars:

```
In [121]: cat == cat_base
Out[121]:
0    False
1     True
2    False
dtype: bool
```

(continues on next page)

(continued from previous page)

```
In [122]: cat == np.array([1, 2, 3])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[122]:
0    True
1    True
2    True
dtype: bool

In [123]: cat == 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    False
1     True
2    False
dtype: bool
```

This doesn't work because the categories are not the same:

```
In [124]: try:
   .....:     cat > cat_base2
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categoricals can only be compared if 'categories' are the same. Categories␣
↪are different lengths
```

If you want to do a "non-equality" comparison of a categorical series with a list-like object which is not categorical data, you need to be explicit and convert the categorical data back to the original values:

```
In [125]: base = np.array([1, 2, 3])

In [126]: try:
   .....:     cat > base
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Cannot compare a Categorical for op __gt__ with type <class 'numpy.ndarray
↪'>.
If you want to compare values, use 'np.asarray(cat) <op> other'.

In [127]: np.asarray(cat) > base
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array([False, False, False], dtype=bool)
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```
In [128]: c1 = pd.Categorical(['a', 'b'], categories=['a', 'b'], ordered=False)

In [129]: c2 = pd.Categorical(['a', 'b'], categories=['b', 'a'], ordered=False)

In [130]: c1 == c2
Out[130]: array([ True,  True], dtype=bool)
```

4.8.7 Operations

Apart from *Series.min()*, *Series.max()* and *Series.mode()*, the following operations are possible with categorical data:

Series methods like *Series.value_counts()* will use all categories, even if some categories are not present in the data:

```
In [131]: s = pd.Series(pd.Categorical(["a", "b", "c", "c"],
   .....:                categories=["c", "a", "b", "d"]))
   .....:

In [132]: s.value_counts()
Out[132]:
c    2
b    1
a    1
d    0
dtype: int64
```

Groupby will also show "unused" categories:

```
In [133]: cats = pd.Categorical(["a", "b", "b", "b", "c", "c", "c"],
   .....:                        categories=["a", "b", "c", "d"])
   .....:

In [134]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 2, 3, 4, 5]})

In [135]: df.groupby("cats").mean()
Out[135]:
      values
cats
a        1.0
b        2.0
c        4.0
d        NaN

In [136]: cats2 = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [137]: df2 = pd.DataFrame({"cats": cats2,
   .....:                      "B": ["c", "d", "c", "d"],
   .....:                      "values": [1, 2, 3, 4]})
   .....:

In [138]: df2.groupby(["cats", "B"]).mean()
Out[138]:
        values
cats B
a    c     1.0
     d     2.0
b    c     3.0
     d     4.0
c    c     NaN
     d     NaN
```

Pivot tables:

```
In [139]: raw_cat = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [140]: df = pd.DataFrame({"A": raw_cat,
   .....:                    "B": ["c", "d", "c", "d"],
   .....:                    "values": [1, 2, 3, 4]})
   .....:

In [141]: pd.pivot_table(df, values='values', index=['A', 'B'])
Out[141]:
     values
A B
a c       1
  d       2
b c       3
  d       4
```

4.8.8 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.at`, and `.iat`, work as normal. The only difference is the return type (for getting) and that only values already in *categories* can be assigned.

Getting

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.

```
In [142]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [143]: cats = pd.Series(["a", "b", "b", "b", "c", "c", "c"],
   .....:                  dtype="category", index=idx)
   .....:

In [144]: values = [1, 2, 2, 2, 3, 4, 5]

In [145]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [146]: df.iloc[2:4, :]
Out[146]:
  cats  values
j    b       2
k    b       2

In [147]: df.iloc[2:4, :].dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[147]:
cats      category
values       int64
dtype: object

In [148]: df.loc["h":"j", "cats"]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
h    a
i    b
j    b
Name: cats, dtype: category
```

(continues on next page)

(continued from previous page)

```
Categories (3, object): [a, b, c]

In [149]: df[df["cats"] == "b"]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  cats  values
i    b       2
j    b       2
k    b       2
```

An example where the category type is not preserved is if you take one single row: the resulting `Series` is of dtype `object`:

```
# get the complete "h" row as a Series
In [150]: df.loc["h", :]
Out[150]:
cats      a
values    1
Name: h, dtype: object
```

Returning a single item from categorical data will also return the value, not a categorical of length "1".

```
In [151]: df.iat[0, 0]
Out[151]: 'a'

In [152]: df["cats"].cat.categories = ["x", "y", "z"]

In [153]: df.at["h", "cats"]  # returns a string
Out[153]: 'x'
```

Note: The is in contrast to R's *factor* function, where `factor(c(1,2,3))[1]` returns a single value *factor*.

To get a single value `Series` of type `category`, you pass in a list with a single value:

```
In [154]: df.loc[["h"], "cats"]
Out[154]:
h    x
Name: cats, dtype: category
Categories (3, object): [x, y, z]
```

String and datetime accessors

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```
In [155]: str_s = pd.Series(list('aabb'))

In [156]: str_cat = str_s.astype('category')

In [157]: str_cat
Out[157]:
0    a
1    a
2    b
```

(continues on next page)

(continued from previous page)

```
3    b
dtype: category
Categories (2, object): [a, b]

In [158]: str_cat.str.contains("a")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[158]:␣
↪
0     True
1     True
2    False
3    False
dtype: bool

In [159]: date_s = pd.Series(pd.date_range('1/1/2015', periods=5))

In [160]: date_cat = date_s.astype('category')

In [161]: date_cat
Out[161]:
0   2015-01-01
1   2015-01-02
2   2015-01-03
3   2015-01-04
4   2015-01-05
dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-04, 2015-
↪01-05]

In [162]: date_cat.dt.day
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Note: The returned `Series` (or `DataFrame`) is of the same type as if you used the `.str.<method>` / `.dt.<method>` on a `Series` of that type (and not of type `category`!).

That means, that the returned values from methods and properties on the accessors of a `Series` and the returned values from methods and properties on the accessors of this `Series` transformed to one of type *category* will be equal:

```
In [163]: ret_s = str_s.str.contains("a")

In [164]: ret_cat = str_cat.str.contains("a")

In [165]: ret_s.dtype == ret_cat.dtype
Out[165]: True

In [166]: ret_s == ret_cat
\\\\\\\\\\\\\\Out[166]:
```

(continues on next page)

(continued from previous page)

```
0    True
1    True
2    True
3    True
dtype: bool
```

Note: The work is done on the `categories` and then a new `Series` is constructed. This has some performance implication if you have a `Series` of type string, where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`). In this case it can be faster to convert the original `Series` to one of type `category` and use `.str.<method>` or `.dt.<property>` on that.

Setting

Setting values in a categorical column (or `Series`) works as long as the value is included in the *categories*:

```
In [167]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [168]: cats = pd.Categorical(["a", "a", "a", "a", "a", "a", "a"],
   .....:                       categories=["a", "b"])
   .....:

In [169]: values = [1, 1, 1, 1, 1, 1, 1]

In [170]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [171]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

In [172]: df
Out[172]:
  cats  values
h    a       1
i    a       1
j    b       2
k    b       2
l    a       1
m    a       1
n    a       1

In [173]: try:
   .....:     df.iloc[2:4, :] = [["c", 3], ["c", 3]]
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Cannot setitem on a Categorical with a new category, set the categories first
```

Setting values by assigning categorical data will also check that the *categories* match:

```
In [174]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"], categories=["a", "b"])

In [175]: df
Out[175]:
  cats  values
```

(continues on next page)

(continued from previous page)

```
h    a       1
i    a       1
j    a       2
k    a       2
l    a       1
m    a       1
n    a       1

In [176]: try:
   .....:     df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"],
   .....:                                              categories=["a", "b", "c"])
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Cannot set a Categorical with another, without identical categories
```

Assigning a `Categorical` to parts of a column of other types will use the values:

```
In [177]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a", "a"]})

In [178]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [179]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [180]: df
Out[180]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a

In [181]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[181]:
a    object
b    object
dtype: object
```

Merging

You can concat two `DataFrames` containing categorical data together, but the categories of these categoricals need to be the same:

```
In [182]: cat = pd.Series(["a", "b"], dtype="category")

In [183]: vals = [1, 2]

In [184]: df = pd.DataFrame({"cats": cat, "vals": vals})

In [185]: res = pd.concat([df, df])

In [186]: res
Out[186]:
```

(continues on next page)

(continued from previous page)

```
  cats  vals
0    a     1
1    b     2
0    a     1
1    b     2

In [187]: res.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[187]:
cats    category
vals       int64
dtype: object
```

In this case the categories are not the same, and therefore an error is raised:

```
In [188]: df_different = df.copy()

In [189]: df_different["cats"].cat.categories = ["c", "d"]

In [190]: try:
   .....:     pd.concat([df, df_different])
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
```

The same applies to `df.append(df_different)`.

See also the section on *merge dtypes* for notes about preserving merge dtypes and performance.

Unioning

New in version 0.19.0.

If you want to combine categoricals that do not necessarily have the same categories, the *union_categoricals()* function will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
In [191]: from pandas.api.types import union_categoricals

In [192]: a = pd.Categorical(["b", "c"])

In [193]: b = pd.Categorical(["a", "b"])

In [194]: union_categoricals([a, b])
Out[194]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the data. If you want the categories to be lexsorted, use `sort_categories=True` argument.

```
In [195]: union_categoricals([a, b], sort_categories=True)
Out[195]:
[b, c, a, b]
Categories (3, object): [a, b, c]
```

`union_categoricals` also works with the "easy" case of combining two categoricals of the same categories and order information (e.g. what you could also `append` for).

```
In [196]: a = pd.Categorical(["a", "b"], ordered=True)

In [197]: b = pd.Categorical(["a", "b", "a"], ordered=True)

In [198]: union_categoricals([a, b])
Out[198]:
[a, b, a, b, a]
Categories (2, object): [a < b]
```

The below raises `TypeError` because the categories are ordered and not identical.

```
In [1]: a = pd.Categorical(["a", "b"], ordered=True)
In [2]: b = pd.Categorical(["a", "b", "c"], ordered=True)
In [3]: union_categoricals([a, b])
Out[3]:
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0.

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```
In [199]: a = pd.Categorical(["a", "b", "c"], ordered=True)

In [200]: b = pd.Categorical(["c", "b", "a"], ordered=True)

In [201]: union_categoricals([a, b], ignore_order=True)
Out[201]:
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

`union_categoricals()` also works with a `CategoricalIndex`, or `Series` containing categorical data, but note that the resulting array will always be a plain `Categorical`:

```
In [202]: a = pd.Series(["b", "c"], dtype='category')

In [203]: b = pd.Series(["a", "b"], dtype='category')

In [204]: union_categoricals([a, b])
Out[204]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

Note: `union_categoricals` may recode the integer codes for categories when combining categoricals. This is likely what you want, but if you are relying on the exact numbering of the categories, be aware.

```
In [205]: c1 = pd.Categorical(["b", "c"])

In [206]: c2 = pd.Categorical(["a", "b"])

In [207]: c1
Out[207]:
[b, c]
Categories (2, object): [b, c]

# "b" is coded to 0
```

(continues on next page)

(continued from previous page)

```
In [208]: c1.codes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[208]: array([0, 1], dtype=int8)

In [209]: c2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[209]:␣
↪
[a, b]
Categories (2, object): [a, b]

# "b" is coded to 1
In [210]: c2.codes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array([0, 1], dtype=int8)

In [211]: c = union_categoricals([c1, c2])

In [212]: c
Out[212]:
[b, c, a, b]
Categories (3, object): [b, c, a]

# "b" is coded to 0 throughout, same as c1, different from c2
In [213]: c.codes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[213]: array([0, 1, 2,␣
↪0], dtype=int8)
```

Concatenation

This section describes concatenations specific to `category` dtype. See *Concatenating objects* for general description.

By default, `Series` or `DataFrame` concatenation which contains the same categories results in `category` dtype, otherwise results in `object` dtype. Use `.astype` or `union_categoricals` to get `category` result.

```
# same categories
In [214]: s1 = pd.Series(['a', 'b'], dtype='category')

In [215]: s2 = pd.Series(['a', 'b', 'a'], dtype='category')

In [216]: pd.concat([s1, s2])
Out[216]:
0    a
1    b
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]

# different categories
In [217]: s3 = pd.Series(['b', 'c'], dtype='category')

In [218]: pd.concat([s1, s3])
Out[218]:
0    a
```

(continues on next page)

(continued from previous page)

```
1    b
0    b
1    c
dtype: object

In [219]: pd.concat([s1, s3]).astype('category')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[219]:
0    a
1    b
0    b
1    c
dtype: category
Categories (3, object): [a, b, c]

In [220]: union_categoricals([s1.array, s3.array])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
[a, b, b, c]
Categories (3, object): [a, b, c]
```

Following table summarizes the results of `Categoricals` related concatenations.

arg1	arg2	result
category	category (identical categories)	category
category	category (different categories, both not ordered)	object (dtype is inferred)
category	category (different categories, either one is ordered)	object (dtype is inferred)
category	not category	object (dtype is inferred)

4.8.9 Getting Data In/Out

You can write data that contains `category` dtypes to a `HDFStore`. See *here* for an example and caveats.

It is also possible to write data to and reading data from *Stata* format files. See *here* for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to *category* and assign the right categories and categories ordering.

```
In [221]: import io

In [222]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))

# rename the categories
In [223]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [224]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [225]: df = pd.DataFrame({"cats": s, "vals": [1, 2, 3, 4, 5, 6]})

In [226]: csv = io.StringIO()

In [227]: df.to_csv(csv)

In [228]: df2 = pd.read_csv(io.StringIO(csv.getvalue()))
```

(continues on next page)

(continued from previous page)

```
In [229]: df2.dtypes
Out[229]:
Unnamed: 0     int64
cats          object
vals           int64
dtype: object

In [230]: df2["cats"]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[230]:␣
↪
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: object

# Redo the category
In [231]: df2["cats"] = df2["cats"].astype("category")

In [232]: df2["cats"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                  "good", "very good"],
   .....:                                 inplace=True)
   .....:

In [233]: df2.dtypes
Out[233]:
Unnamed: 0       int64
cats          category
vals             int64
dtype: object

In [234]: df2["cats"]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[234
↪
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

The same holds for writing to a SQL database with `to_sql`.

4.8.10 Missing Data

pandas primarily uses the value *np.nan* to represent missing data. It is by default not included in computations. See the *Missing Data section*.

Missing values should **not** be included in the Categorical's `categories`, only in the `values`. Instead, it is understood that NaN is different, and is always a possibility. When working with the Categorical's `codes`, missing values will always have a code of `-1`.

```
In [235]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")

# only two categories
In [236]: s
Out[236]:
0      a
1      b
2    NaN
3      a
dtype: category
Categories (2, object): [a, b]

In [237]: s.cat.codes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23
↪
0    0
1    1
2   -1
3    0
dtype: int8
```

Methods for working with missing data, e.g. *isna()*, *fillna()*, *dropna()*, all work normally:

```
In [238]: s = pd.Series(["a", "b", np.nan], dtype="category")

In [239]: s
Out[239]:
0      a
1      b
2    NaN
dtype: category
Categories (2, object): [a, b]

In [240]: pd.isna(s)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[240]:
↪
0    False
1    False
2     True
dtype: bool

In [241]: s.fillna("a")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]
```

4.8.11 Differences to R's *factor*

The following differences to R's factor functions can be observed:

- R's *levels* are named *categories*.
- R's *levels* are always of type string, while *categories* in pandas can be of any dtype.

- It's not possible to specify labels at creation time. Use `s.cat.rename_categories(new_labels)` afterwards.
- In contrast to R's *factor* function, using categorical data as the sole input to create a new categorical series will *not* remove unused categories but create a new categorical series which is equal to the passed in one!
- R allows for missing values to be included in its *levels* (pandas' *categories*). Pandas does not allow *NaN* categories, but missing values can still be in the *values*.

4.8.12 Gotchas

Memory Usage

The memory usage of a `Categorical` is proportional to the number of categories plus the length of the data. In contrast, an `object` dtype is a constant times the length of the data.

```
In [242]: s = pd.Series(['foo', 'bar'] * 1000)

# object dtype
In [243]: s.nbytes
Out[243]: 16000

# category dtype
In [244]: s.astype('category').nbytes
\\\\\\\\\\\\\\\\Out[244]: 2016
```

Note: If the number of categories approaches the length of the data, the `Categorical` will use nearly the same or more memory than an equivalent `object` dtype representation.

```
In [245]: s = pd.Series(['foo%04d' % i for i in range(2000)])

# object dtype
In [246]: s.nbytes
Out[246]: 16000

# category dtype
In [247]: s.astype('category').nbytes
\\\\\\\\\\\\\\\\Out[247]: 20000
```

Categorical is not a *numpy* array

Currently, categorical data and the underlying `Categorical` is implemented as a Python object and not as a low-level NumPy array dtype. This leads to some problems.

NumPy itself doesn't know about the new *dtype*:

```
In [248]: try:
   .....:     np.dtype("category")
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type "category" not understood
```

(continues on next page)

(continued from previous page)

```
In [249]: dtype = pd.Categorical(["a"]).dtype

In [250]: try:
   .....:     np.dtype(dtype)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type not understood
```

Dtype comparisons work:

```
In [251]: dtype == np.str_
Out[251]: False

In [252]: np.str_ == dtype
\\\\\\\\\\\\\\\\Out[252]: False
```

To check if a Series contains Categorical data, use `hasattr(s, 'cat')`:

```
In [253]: hasattr(pd.Series(['a'], dtype='category'), 'cat')
Out[253]: True

In [254]: hasattr(pd.Series(['a']), 'cat')
\\\\\\\\\\\\\\\\Out[254]: False
```

Using NumPy functions on a `Series` of type `category` should not work as *Categoricals* are not numeric data (even in the case that `.categories` is numeric).

```
In [255]: s = pd.Series(pd.Categorical([1, 2, 3, 4]))

In [256]: try:
   .....:     np.sum(s)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categorical cannot perform the operation sum
```

Note: If such a function works, please file a bug at https://github.com/pandas-dev/pandas!

dtype in apply

Pandas currently does not preserve the dtype in apply functions: If you apply along rows you get a *Series* of `object` *dtype* (same as getting a row -> getting one element will return a basic type) and applying along columns will also convert to object. `NaN` values are unaffected. You can use `fillna` to handle missing values before applying a function.

```
In [257]: df = pd.DataFrame({"a": [1, 2, 3, 4],
   .....:                    "b": ["a", "b", "c", "d"],
   .....:                    "cats": pd.Categorical([1, 2, 3, 2])})
   .....:

In [258]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[258]:
```

(continues on next page)

(continued from previous page)

```
0    <class 'int'>
1    <class 'int'>
2    <class 'int'>
3    <class 'int'>
dtype: object

In [259]: df.apply(lambda col: col.dtype, axis=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a          int64
b         object
cats    category
dtype: object
```

Categorical Index

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements. See the *advanced indexing docs* for a more detailed explanation.

Setting the index will create a `CategoricalIndex`:

```
In [260]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])

In [261]: strings = ["a", "b", "c", "d"]

In [262]: values = [4, 2, 3, 1]

In [263]: df = pd.DataFrame({"strings": strings, "values": values}, index=cats)

In [264]: df.index
Out[264]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False,␣
↪dtype='category')

# This now sorts by the categories order
In [265]: df.sort_index()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪
  strings  values
4       d       1
2       b       2
3       c       3
1       a       4
```

Side Effects

Constructing a `Series` from a `Categorical` will not copy the input `Categorical`. This means that changes to the `Series` will in most cases change the original `Categorical`:

```
In [266]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [267]: s = pd.Series(cat, name="cat")

In [268]: cat
```

(continues on next page)

(continued from previous page)

```
Out[268]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [269]: s.iloc[0:2] = 10

In [270]: cat
Out[270]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [271]: df = pd.DataFrame(s)

In [272]: df["cat"].cat.categories = [1, 2, 3, 4, 5]

In [273]: cat
Out[273]:
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]
```

Use `copy=True` to prevent such a behaviour or simply don't reuse `Categoricals`:

```
In [274]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [275]: s = pd.Series(cat, name="cat", copy=True)

In [276]: cat
Out[276]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [277]: s.iloc[0:2] = 10

In [278]: cat
Out[278]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
```

Note: This also happens in some cases when you supply a NumPy array instead of a `Categorical`: using an int array (e.g. `np.array([1,2,3,4])`) will exhibit the same behavior, while using a string array (e.g. `np.array(["a","b","c","a"])`) will not.

4.9 Nullable Integer Data Type

New in version 0.24.0.

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

In *Working with missing data*, we saw that pandas primarily uses `NaN` to represent missing data. Because `NaN` is a float, this forces an array of integers with any missing values to become floating point. In some cases, this may not matter much. But if your integer column is, say, an identifier, casting to float can be problematic. Some integers cannot even be represented as floating point numbers.

Pandas can represent integer data with possibly missing values using `arrays.IntegerArray`. This is an *extension types* implemented within pandas. It is not the default dtype for integers, and will not be inferred; you must explicitly pass the dtype into `array()` or `Series`:

```
In [1]: arr = pd.array([1, 2, np.nan], dtype=pd.Int64Dtype())

In [2]: arr
Out[2]:
<IntegerArray>
[1, 2, NaN]
Length: 3, dtype: Int64
```

Or the string alias `"Int64"` (note the capital `"I"`, to differentiate from NumPy's `'int64'` dtype:

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

This array can be stored in a `DataFrame` or `Series` like any NumPy array.

```
In [4]: pd.Series(arr)
Out[4]:
0      1
1      2
2    NaN
dtype: Int64
```

You can also pass the list-like object to the `Series` constructor with the dtype.

```
In [5]: s = pd.Series([1, 2, np.nan], dtype="Int64")

In [6]: s
Out[6]:
0      1
1      2
2    NaN
dtype: Int64
```

By default (if you don't specify `dtype`), NumPy is used, and you'll end up with a `float64` dtype Series:

```
In [7]: pd.Series([1, 2, np.nan])
Out[7]:
0    1.0
1    2.0
2    NaN
dtype: float64
```

Operations involving an integer array will behave similar to NumPy arrays. Missing values will be propagated, and and the data will be coerced to another dtype if needed.

```
# arithmetic
In [8]: s + 1
Out[8]:
```

(continues on next page)

(continued from previous page)

```
0      2
1      3
2    NaN
dtype: Int64

# comparison
In [9]: s == 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
0     True
1    False
2    False
dtype: bool

# indexing
In [10]: s.iloc[1:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
1      2
2    NaN
dtype: Int64

# operate with other dtypes
In [11]: s + s.iloc[1:3].astype('Int8')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1      4
2    NaN
dtype: Int64

# coerce when needed
In [12]: s + 0.01
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1.01
1    2.01
2     NaN
dtype: float64
```

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

```
In [13]: df = pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab')})

In [14]: df
Out[14]:
     A  B  C
0    1  1  a
1    2  1  a
2  NaN  3  b

In [15]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]:
A     Int64
B     int64
C    object
dtype: object
```

These dtypes can be merged & reshaped & casted.

```
In [16]: pd.concat([df[['A']], df[['B', 'C']]], axis=1).dtypes
Out[16]:
A      Int64
B      int64
C     object
dtype: object

In [17]: df['A'].astype(float)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[17]:
0    1.0
1    2.0
2    NaN
Name: A, dtype: float64
```

Reduction and groupby operations such as ‘sum’ work as well.

```
In [18]: df.sum()
Out[18]:
A      3
B      5
C    aab
dtype: object

In [19]: df.groupby('B').A.sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:
B
1    3
3    0
Name: A, dtype: Int64
```

4.10 Visualization

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt

In [2]: plt.close('all')
```

We provide the basics in pandas to easily create decent looking plots. See the *ecosystem* section for visualization libraries that go beyond the basics documented here.

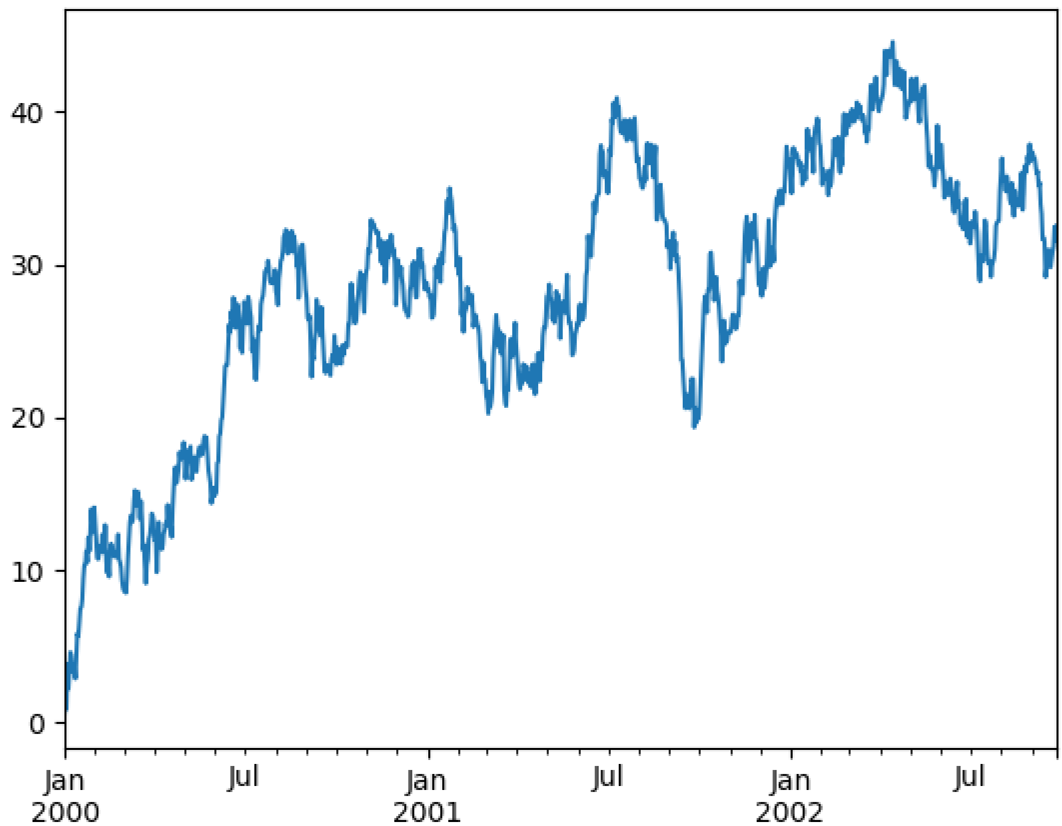
Note: All calls to `np.random` are seeded with 123456.

4.10.1 Basic Plotting: `plot`

We will demonstrate the basics, see the *cookbook* for some advanced strategies.

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`:

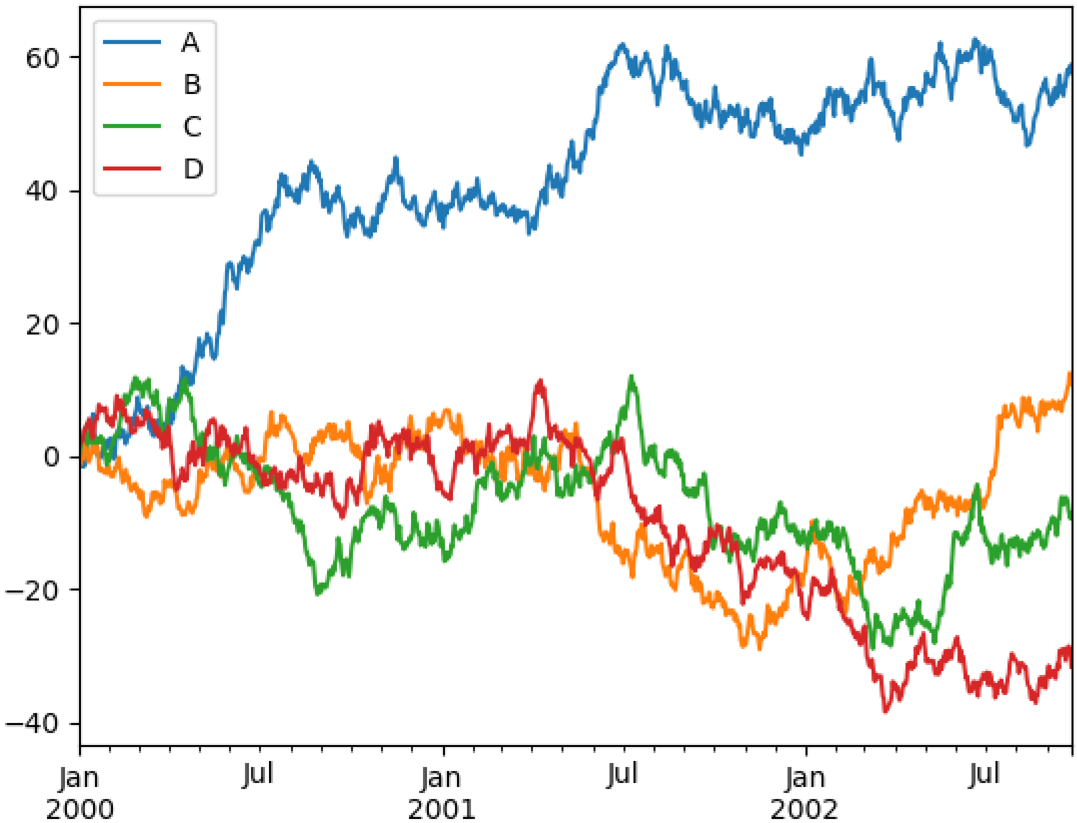
```
In [3]: ts = pd.Series(np.random.randn(1000),
   ...:                index=pd.date_range('1/1/2000', periods=1000))
   ...:

In [4]: ts = ts.cumsum()

In [5]: ts.plot()
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f382055ba58>
```



If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above.

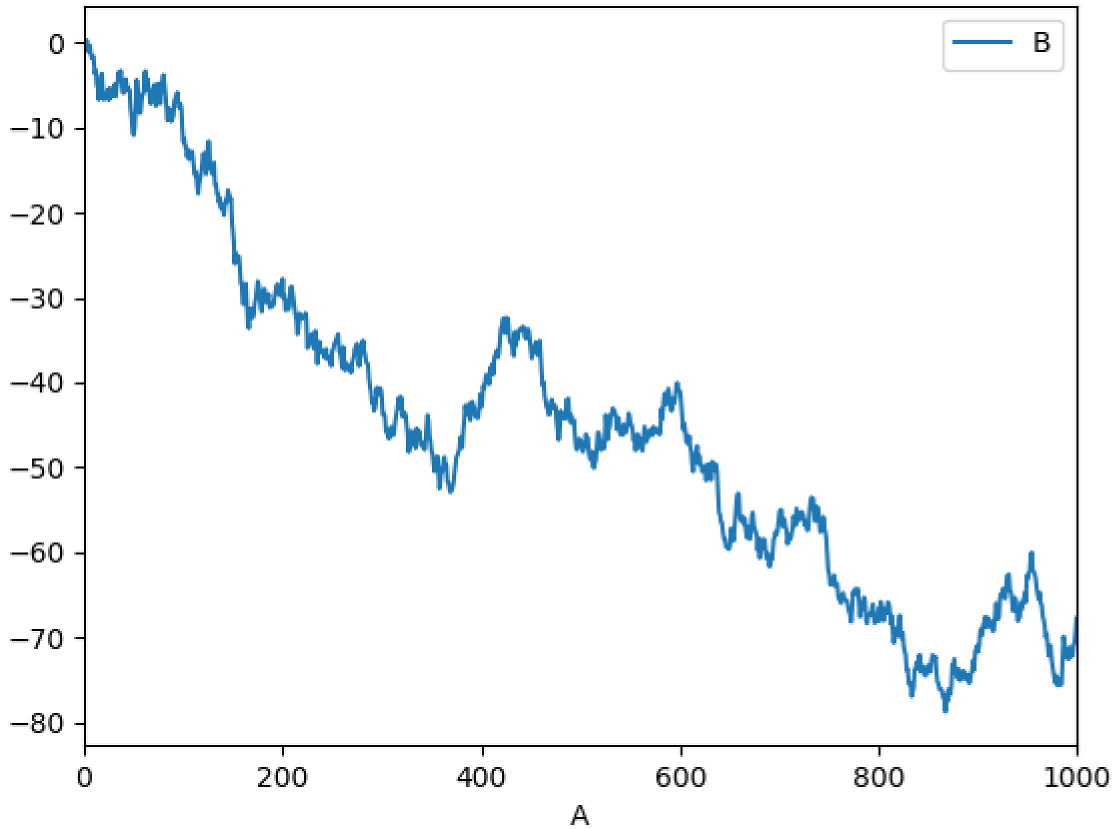
On DataFrame, `plot()` is a convenience to plot all of the columns with labels:

```
In [6]: df = pd.DataFrame(np.random.randn(1000, 4),
   ...:                   index=ts.index, columns=list('ABCD'))
   ...:

In [7]: df = df.cumsum()

In [8]: plt.figure();

In [9]: df.plot();
```



You can plot one column versus another using the *x* and *y* keywords in `plot()`:

```
In [10]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()

In [11]: df3['A'] = pd.Series(list(range(len(df))))

In [12]: df3.plot(x='A', y='B')
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e92959b0>
```



Note: For more formatting and styling options, see *formatting* below.

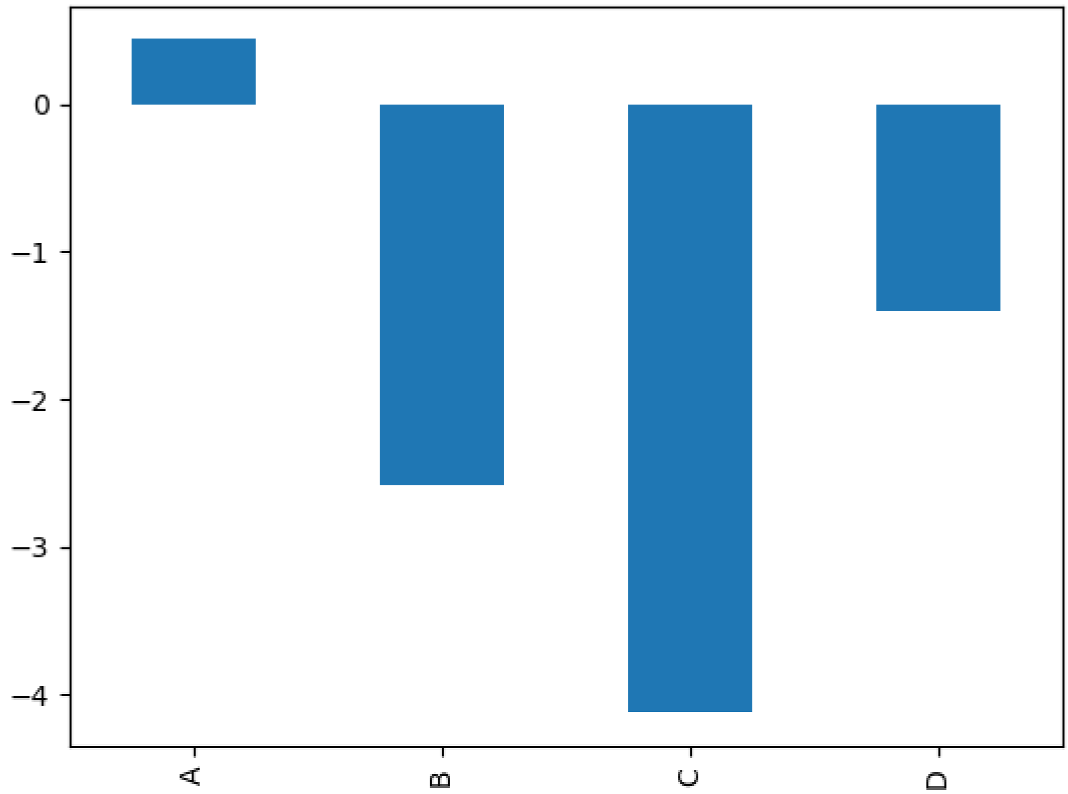
4.10.2 Other Plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to `plot()`, and include:

- *'bar'* or *'barh'* for bar plots
- *'hist'* for histogram
- *'box'* for boxplot
- *'kde'* or *'density'* for density plots
- *'area'* for area plots
- *'scatter'* for scatter plots
- *'hexbin'* for hexagonal bin plots
- *'pie'* for pie plots

For example, a bar plot can be created the following way:

```
In [13]: plt.figure();

In [14]: df.iloc[5].plot(kind='bar');
```



You can also create these other plots using the methods `DataFrame.plot.<kind>` instead of providing the `kind` keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

```
In [15]: df = pd.DataFrame()

In [16]: df.plot.<TAB>  # noqa: E225, E999
df.plot.area      df.plot.barh      df.plot.density  df.plot.hist      df.plot.line    ␣
↪df.plot.scatter
df.plot.bar       df.plot.box       df.plot.hexbin   df.plot.kde       df.plot.pie
```

In addition to these `kind` s, there are the *DataFrame.hist()*, and *DataFrame.boxplot()* methods, which use a separate interface.

Finally, there are several *plotting functions* in `pandas.plotting` that take a `Series` or `DataFrame` as an argument. These include:

- *Scatter Matrix*
- *Andrews Curves*
- *Parallel Coordinates*

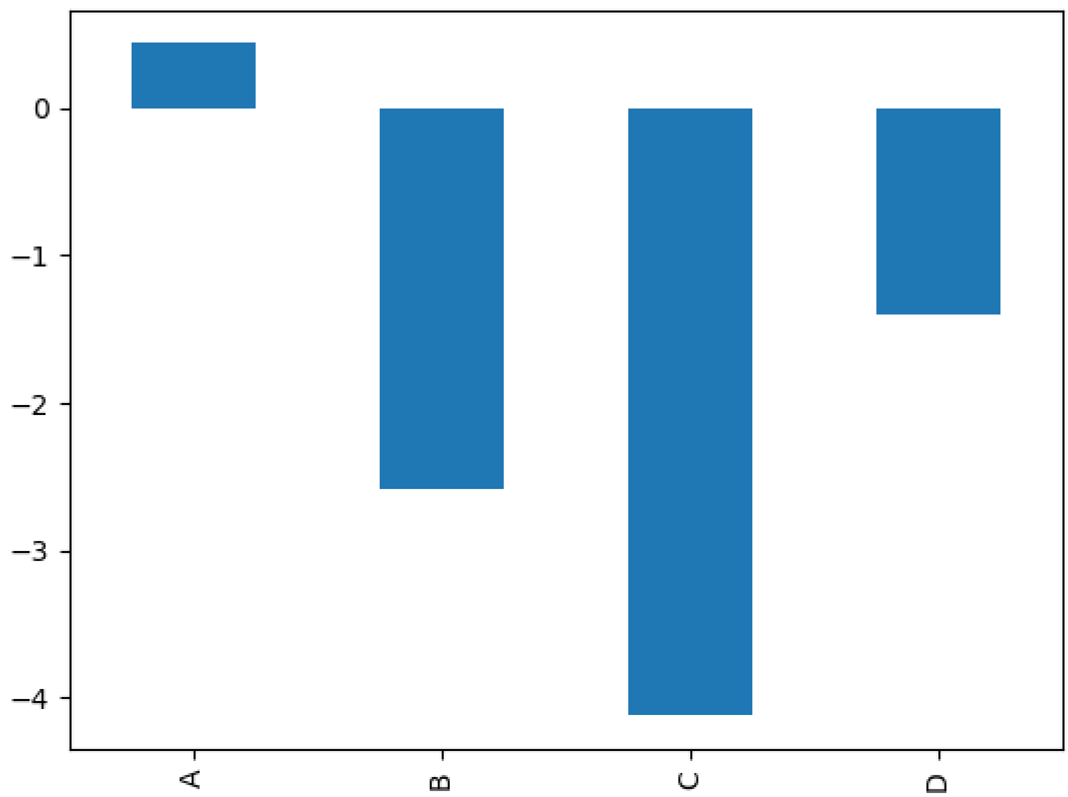
- *Lag Plot*
- *Autocorrelation Plot*
- *Bootstrap Plot*
- *RadViz*

Plots may also be adorned with *errorbars* or *tables*.

Bar plots

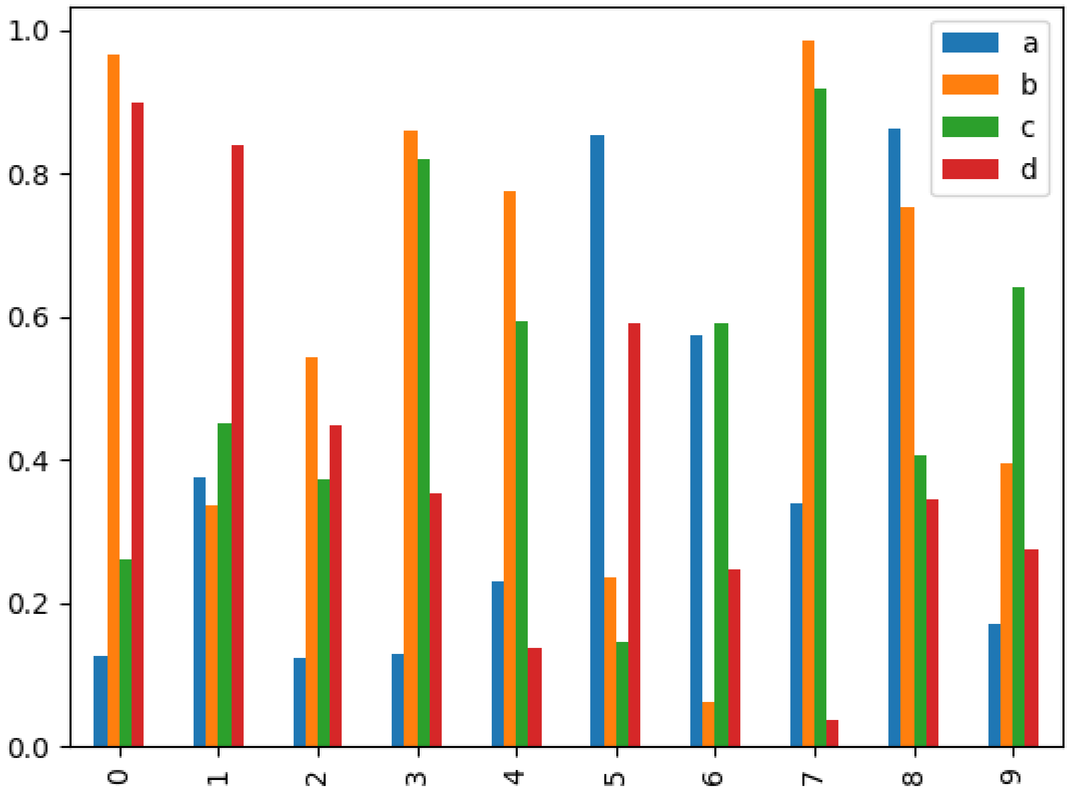
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [17]: plt.figure();

In [18]: df.iloc[5].plot.bar()
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e7492ef0>

In [19]: plt.axhline(0, color='k');
```



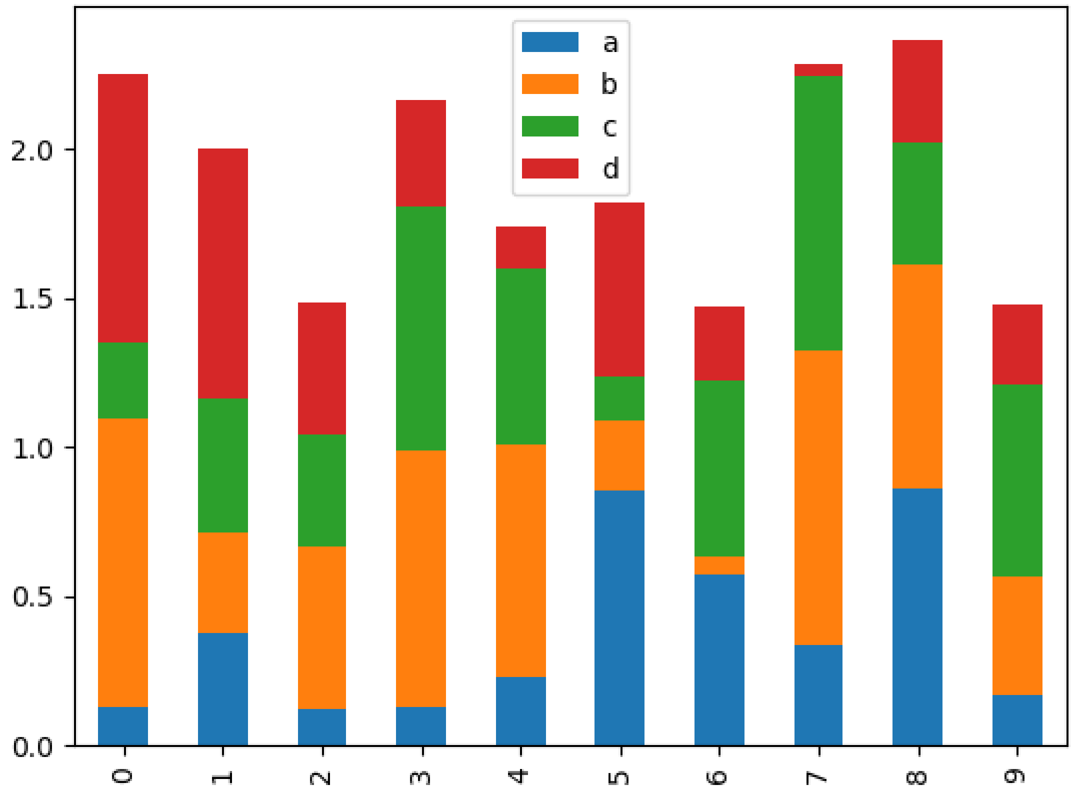
Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

```
In [20]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [21]: df2.plot.bar();
```



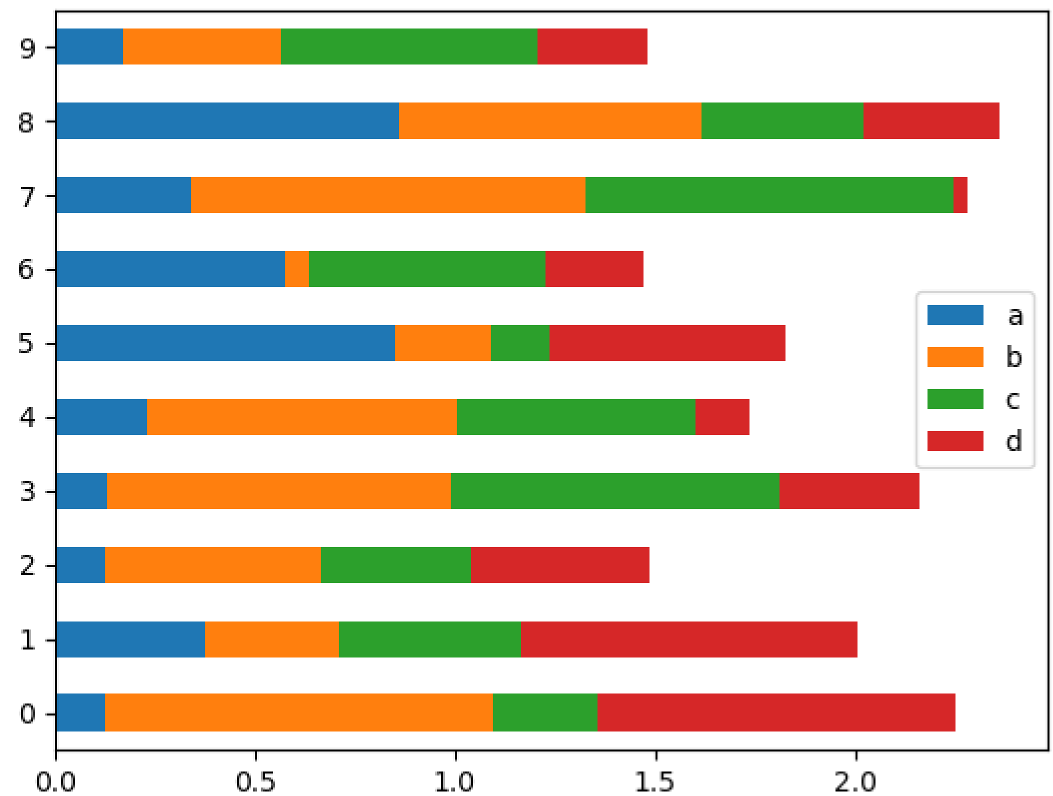
To produce a stacked bar plot, pass `stacked=True`:

```
In [22]: df2.plot.bar(stacked=True);
```



To get horizontal bar plots, use the `barh` method:

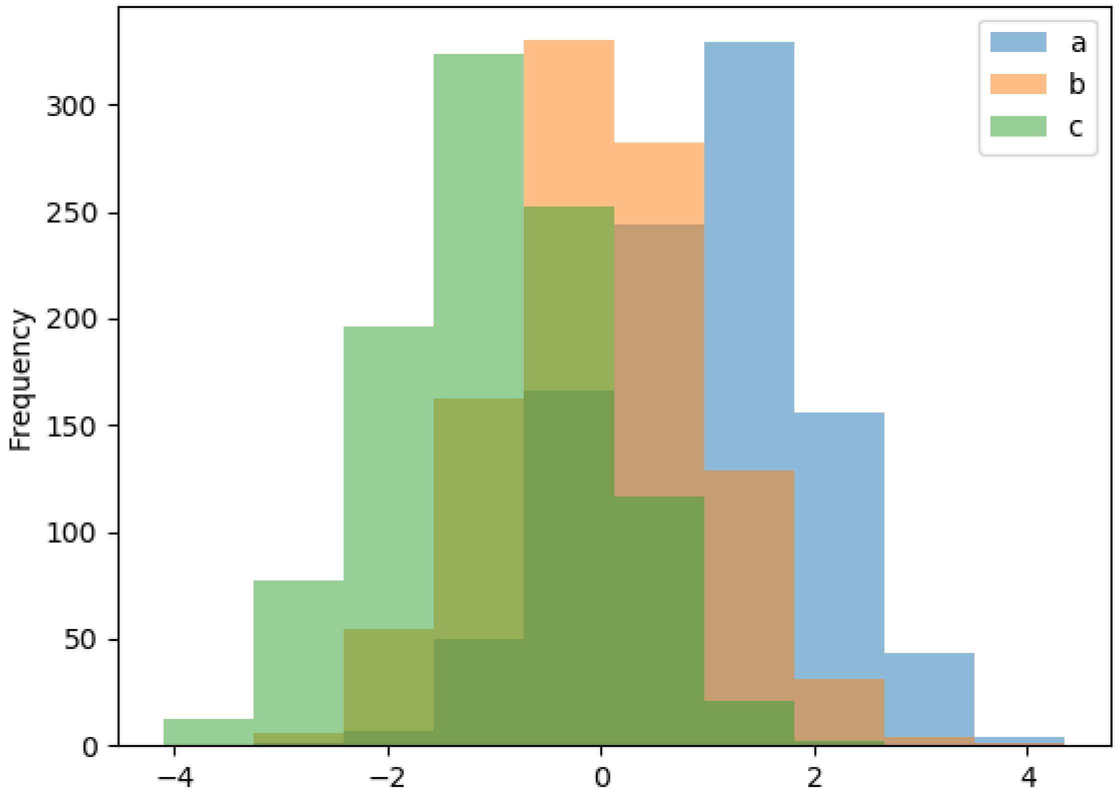
```
In [23]: df2.plot.barh(stacked=True);
```



Histograms

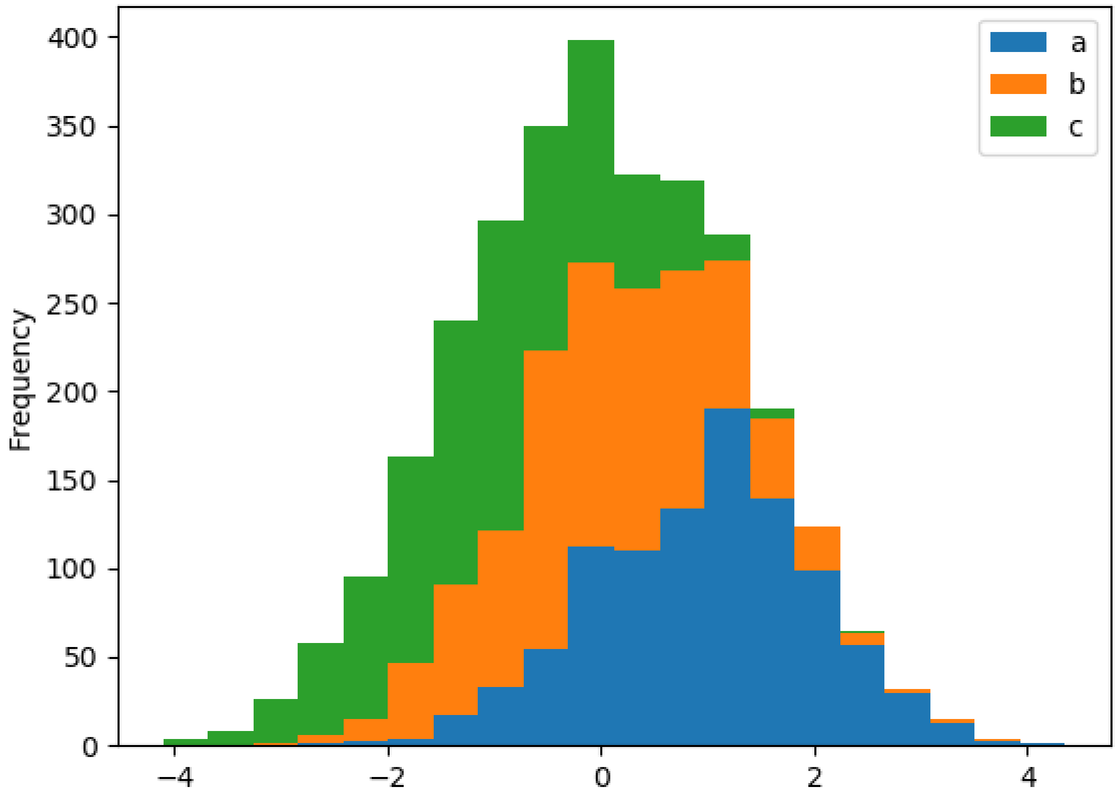
Histograms can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [24]: df4 = pd.DataFrame({'a': np.random.randn(1000) + 1, 'b': np.random.
↪randn(1000),
   ....:                    'c': np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])
   ....:

In [25]: plt.figure();

In [26]: df4.plot.hist(alpha=0.5)
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e5e0a668>
```



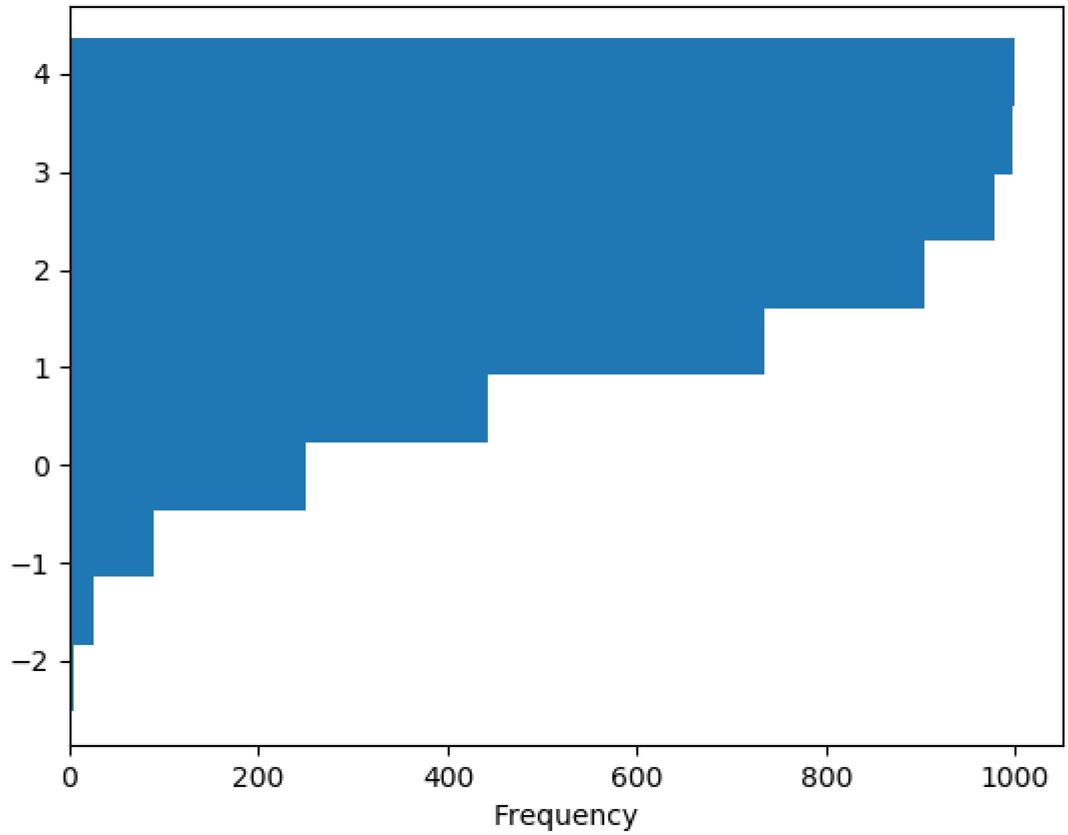
A histogram can be stacked using `stacked=True`. Bin size can be changed using the `bins` keyword.

```
In [27]: plt.figure();

In [28]: df4.plot.hist(stacked=True, bins=20)
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e6c5d2e8>
```



You can pass other keywords supported by matplotlib `hist`. For example, horizontal and cumulative histograms can be drawn by `orientation='horizontal'` and `cumulative=True`.

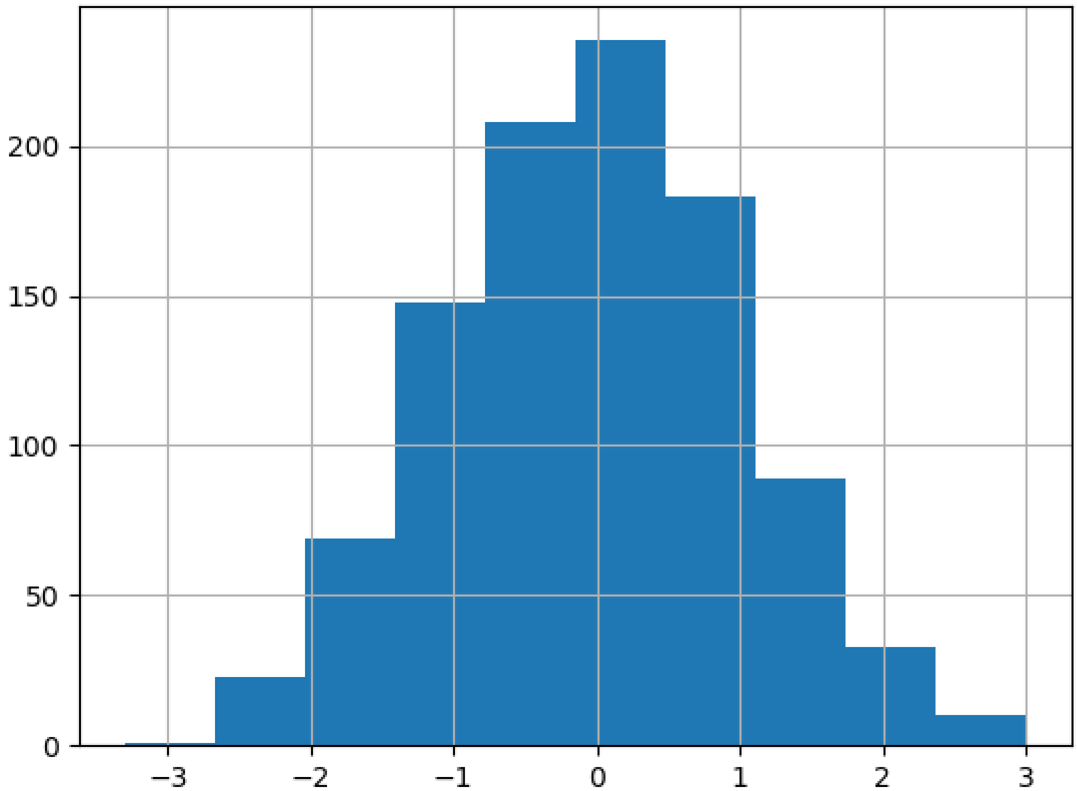
```
In [29]: plt.figure();

In [30]: df4['a'].plot.hist(orientation='horizontal', cumulative=True)
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e67fae48>
```



See the hist method and the matplotlib hist documentation for more.

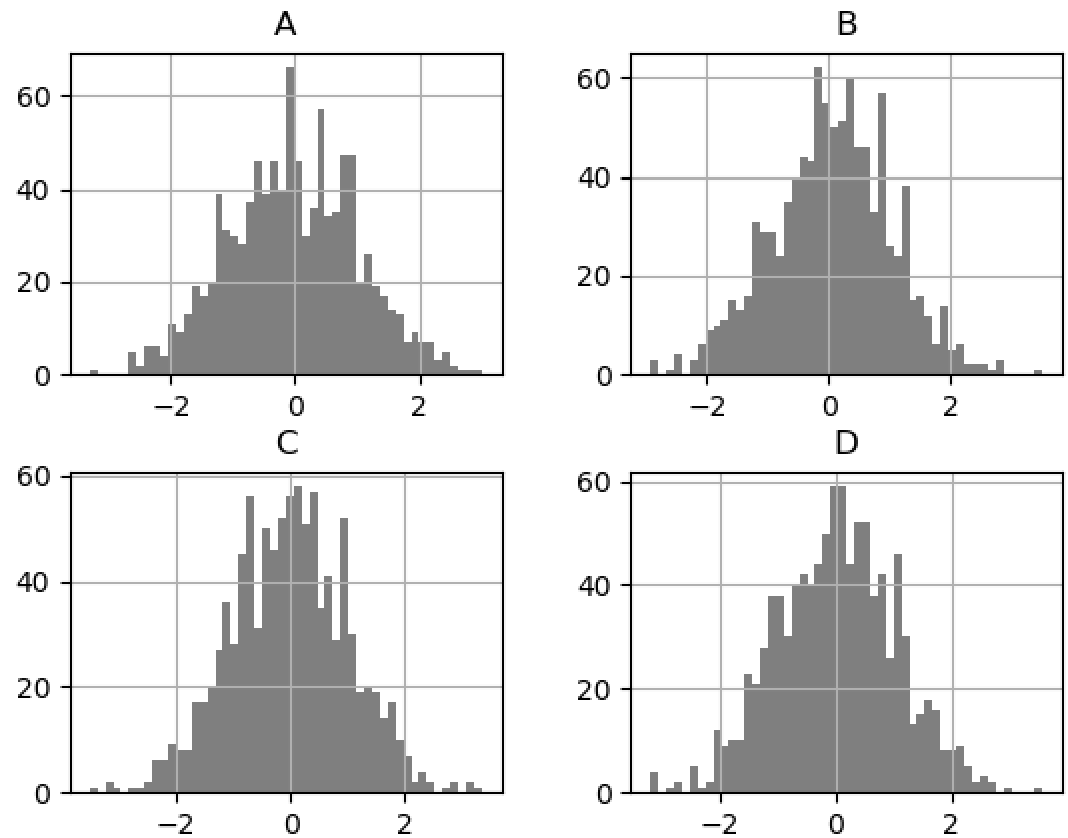
The existing interface DataFrame.hist to plot histogram still can be used.

```
In [31]: plt.figure();

In [32]: df['A'].diff().hist()
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e92bff98>
```



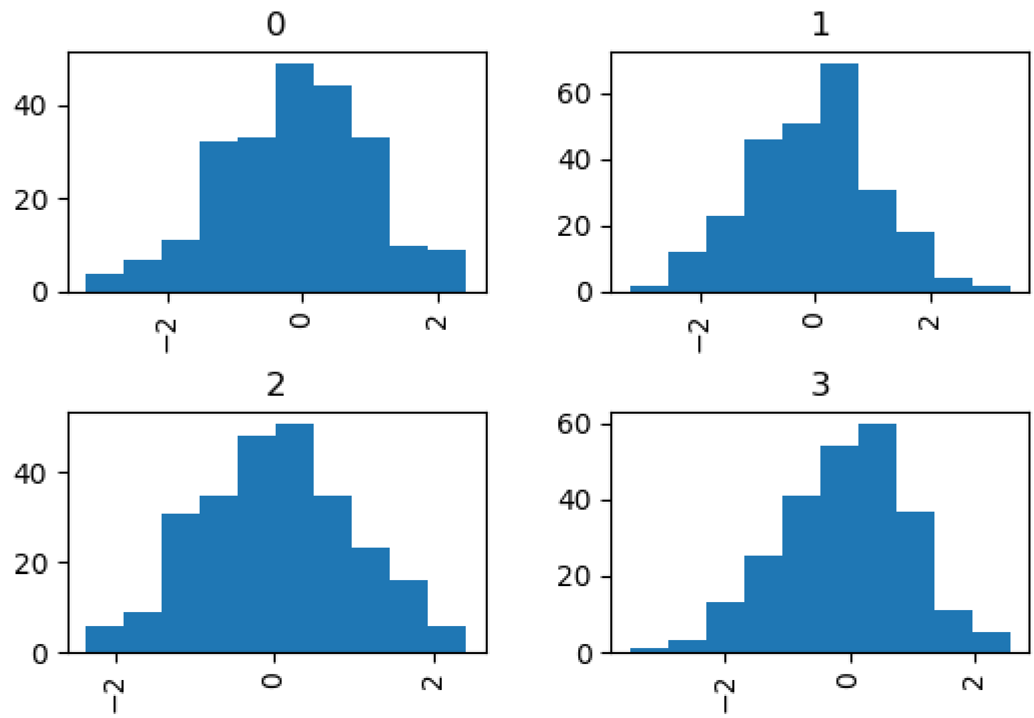
DataFrame.hist() plots the histograms of the columns on multiple subplots:

```
In [33]: plt.figure()
Out[33]: <Figure size 640x480 with 0 Axes>

In [34]: df.diff().hist(color='k', alpha=0.5, bins=50)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[34]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6bc0160>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6ebc3c8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6ec4630>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37e5f7b898>]],
↪dtype=object)
```



The `by` keyword can be specified to plot grouped histograms:

```
In [35]: data = pd.Series(np.random.randn(1000))

In [36]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[36]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6af0e48>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37e71dd0f0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6a70358>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37e6a785c0>]],
↪dtype=object)
```

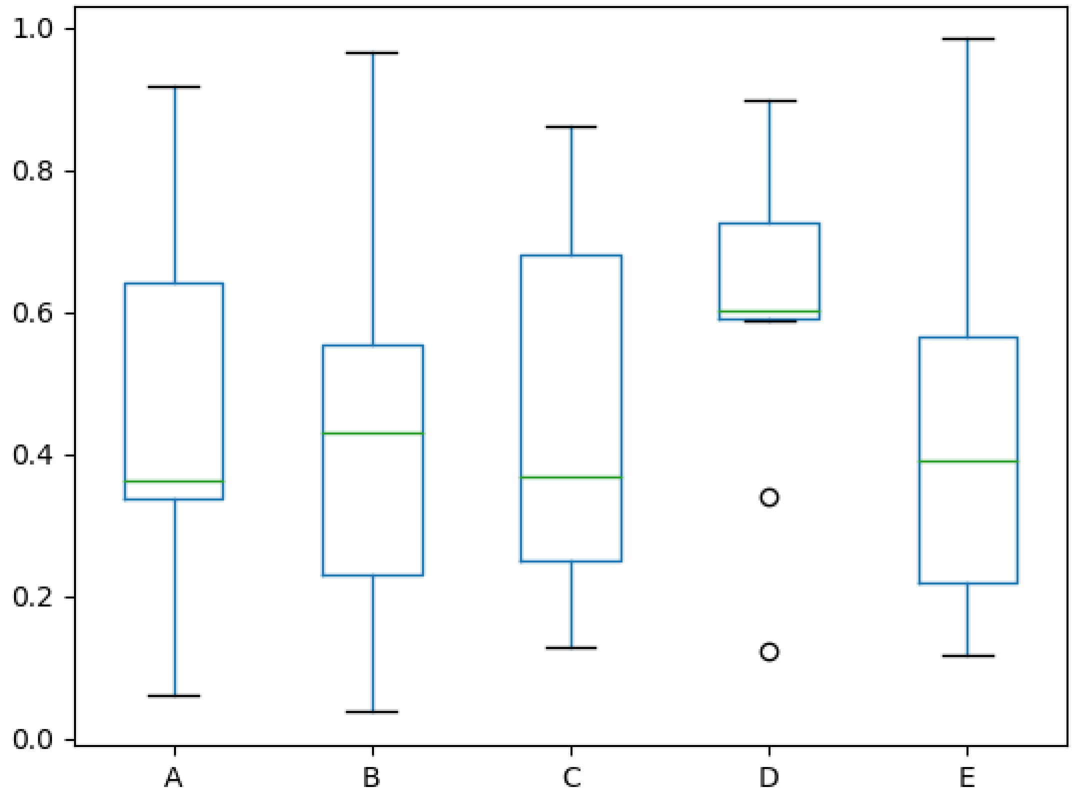


Box Plots

Boxplot can be drawn calling *Series.plot.box()* and *DataFrame.plot.box()*, or *DataFrame.boxplot()* to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

```
In [37]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])

In [38]: df.plot.box()
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e4fe8978>
```

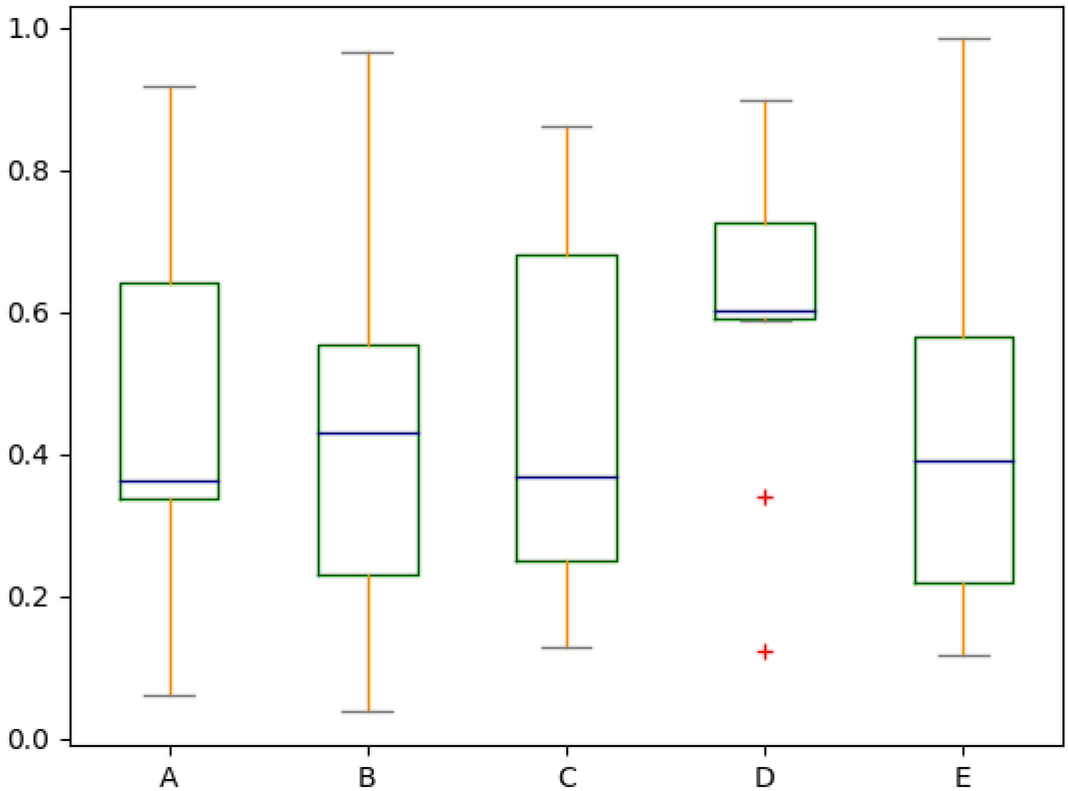


Boxplot can be colorized by passing `color` keyword. You can pass a `dict` whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the `dict`, default colors are used for the corresponding artists. Also, boxplot has `sym` keyword to specify fliers style.

When you pass other type of arguments via `color` keyword, it will be directly passed to matplotlib for all the `boxes`, `whiskers`, `medians` and `caps` colorization.

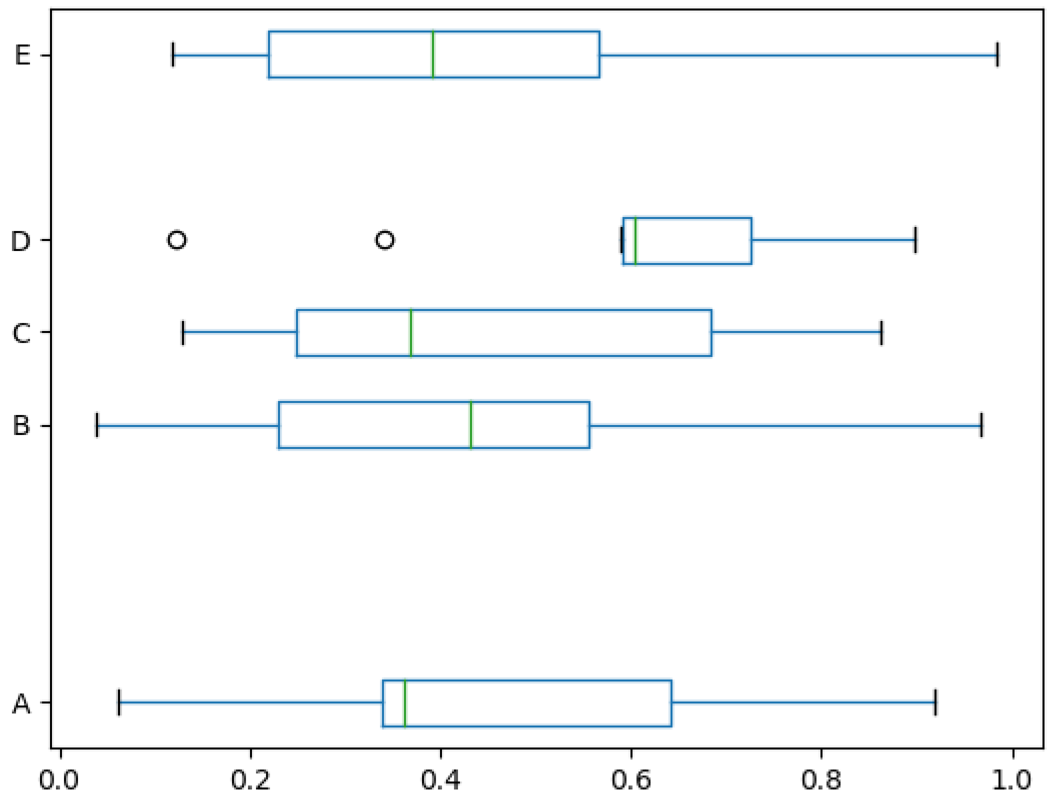
The colors are applied to every boxes to be drawn. If you want more complicated colorization, you can get each drawn artists by passing *return_type*.

```
In [39]: color = {'boxes': 'DarkGreen', 'whiskers': 'DarkOrange',
   ....:          'medians': 'DarkBlue', 'caps': 'Gray'}
   ....:

In [40]: df.plot.box(color=color, sym='r+')
Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e5d42518>
```



Also, you can pass other keywords supported by matplotlib `boxplot`. For example, horizontal and custom-positioned boxplot can be drawn by `vert=False` and `positions` keywords.

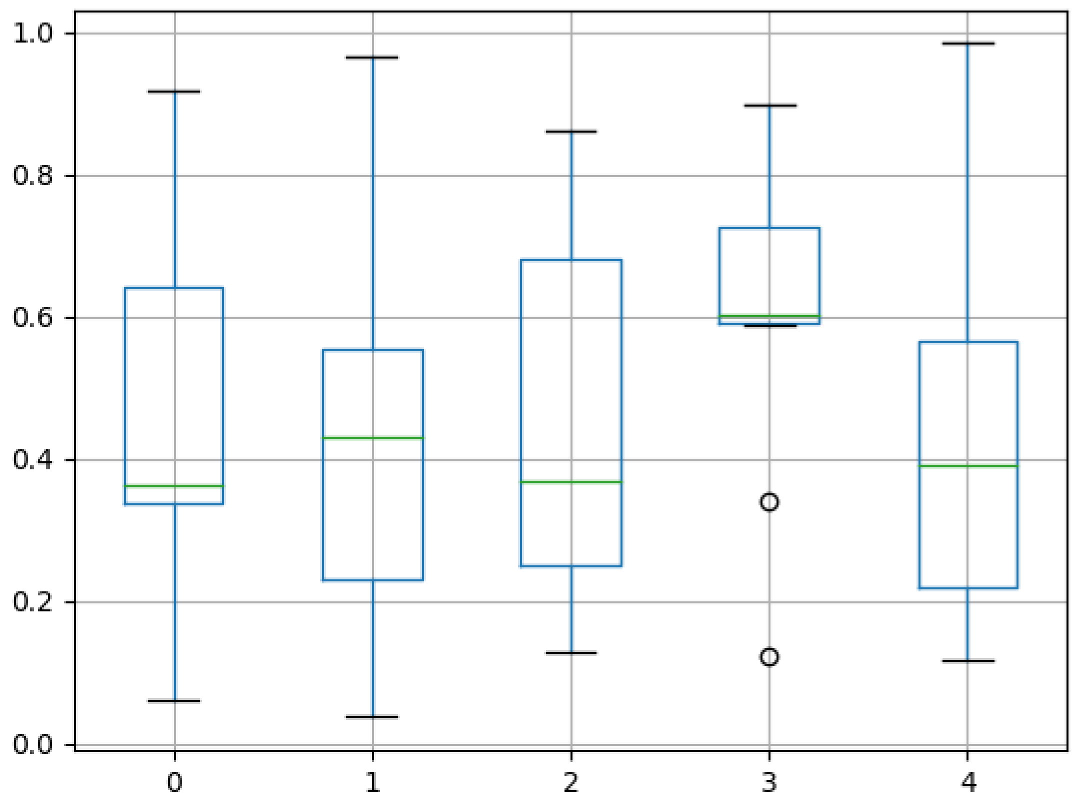
```
In [41]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8])
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e7379320>
```



See the boxplot method and the matplotlib boxplot documentation for more.

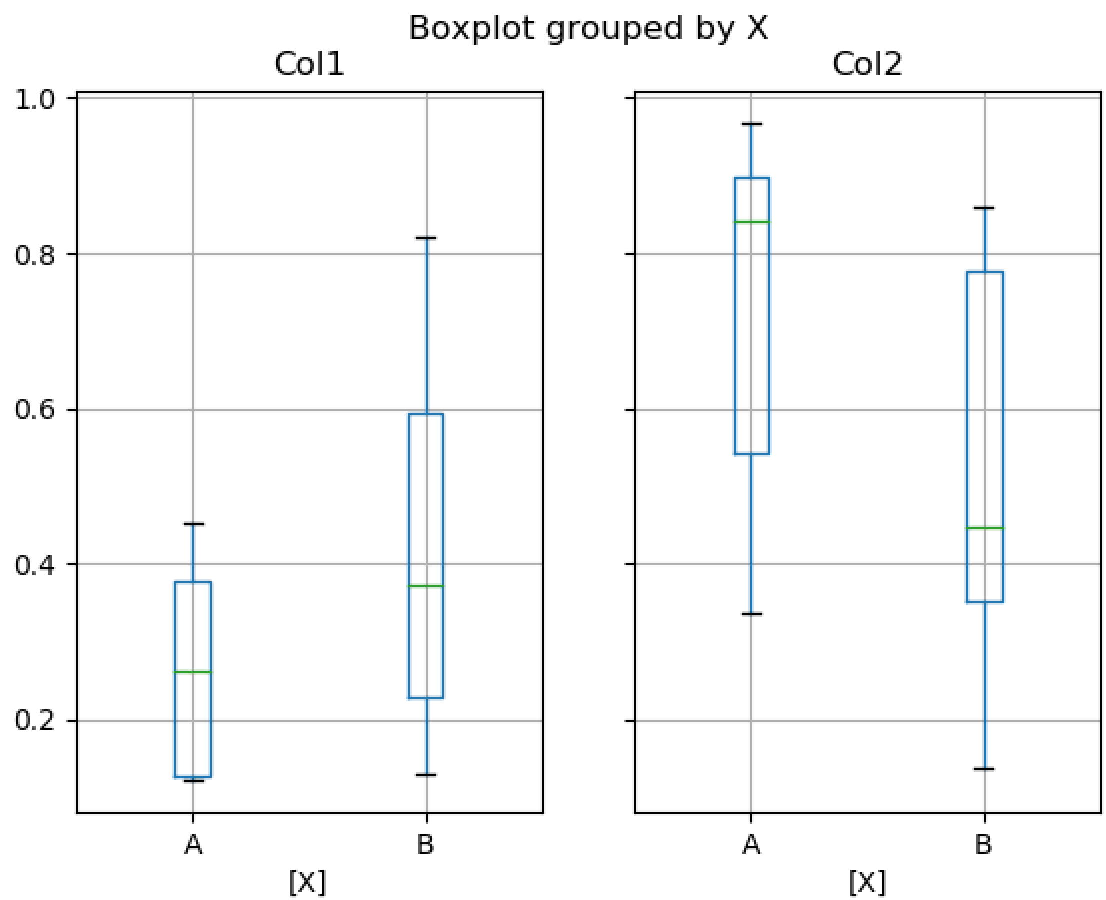
The existing interface DataFrame.boxplot to plot boxplot still can be used.

```
In [42]: df = pd.DataFrame(np.random.rand(10, 5))

In [43]: plt.figure();

In [44]: bp = df.boxplot()
```



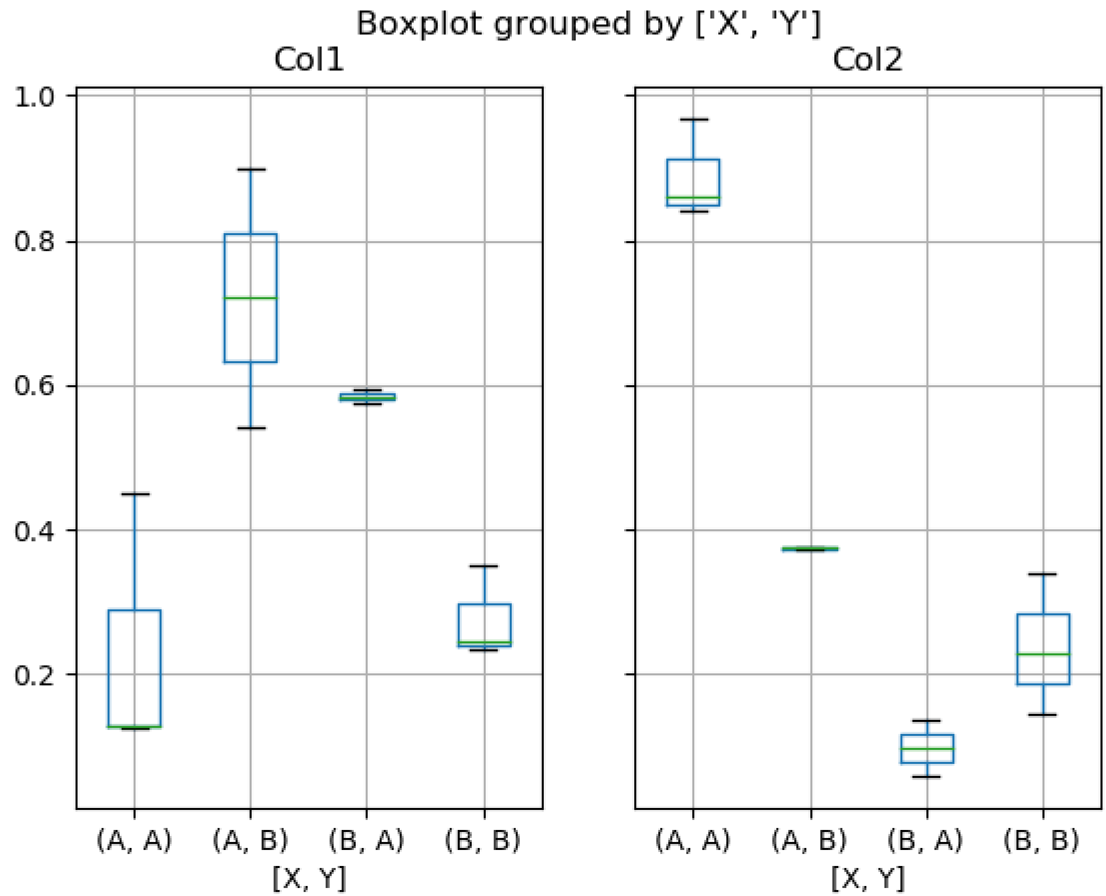
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [45]: df = pd.DataFrame(np.random.rand(10, 2), columns=['Col1', 'Col2'])

In [46]: df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B'])

In [47]: plt.figure();

In [48]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [49]: df = pd.DataFrame(np.random.rand(10, 3), columns=['Col1', 'Col2', 'Col3'])

In [50]: df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B'])

In [51]: df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A', 'B', 'A', 'B', 'A', 'B'])

In [52]: plt.figure();

In [53]: bp = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```



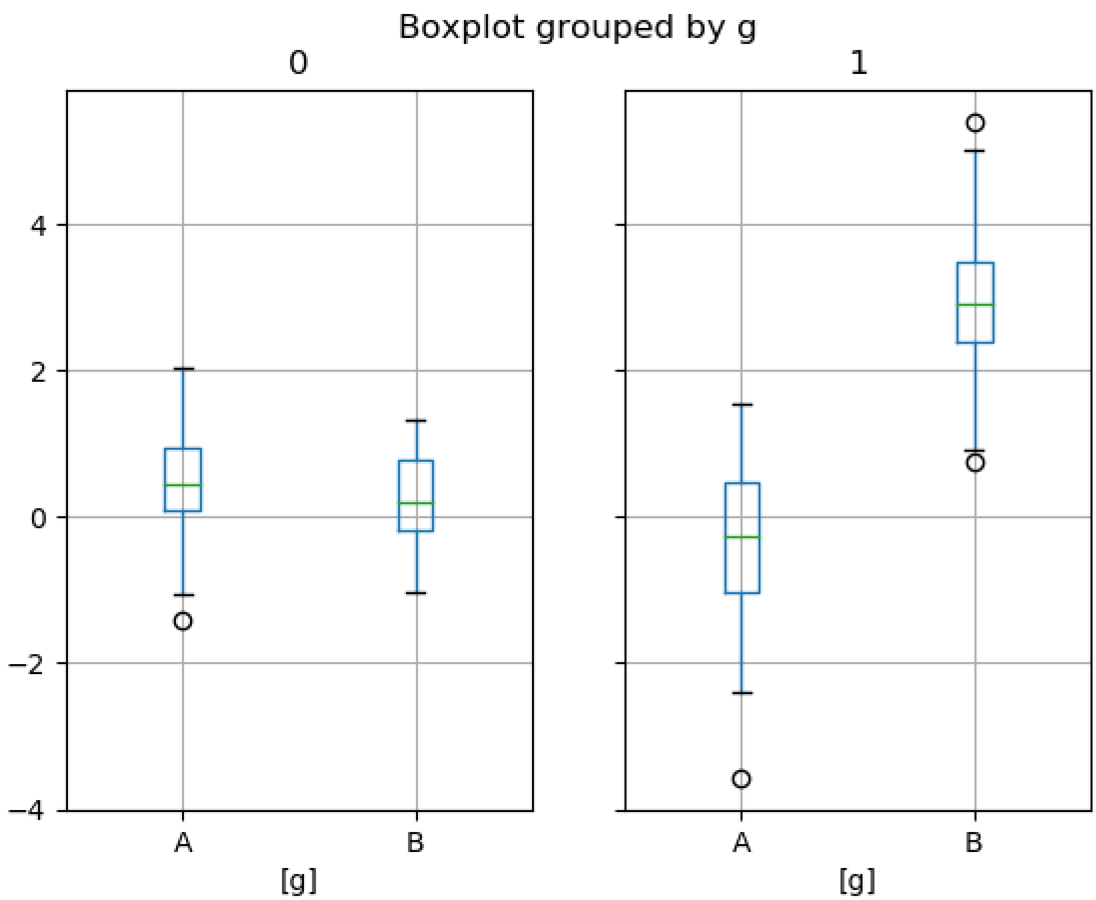
Warning: The default changed from `'dict'` to `'axes'` in version 0.19.0.

In `boxplot`, the return type can be controlled by the `return_type`, keyword. The valid choices are `{"axes", "dict", "both", None}`. Faceting, created by `DataFrame.boxplot` with the `by` keyword, will affect the output type as well:

`return_type=`	Faceted	Output type
`None`	No	axes
`None`	Yes	2-D ndarray of axes
`'axes'`	No	axes
`'axes'`	Yes	Series of axes
`'dict'`	No	dict of artists
`'dict'`	Yes	Series of dicts of artists
`'both'`	No	namedtuple
`'both'`	Yes	Series of namedtuples

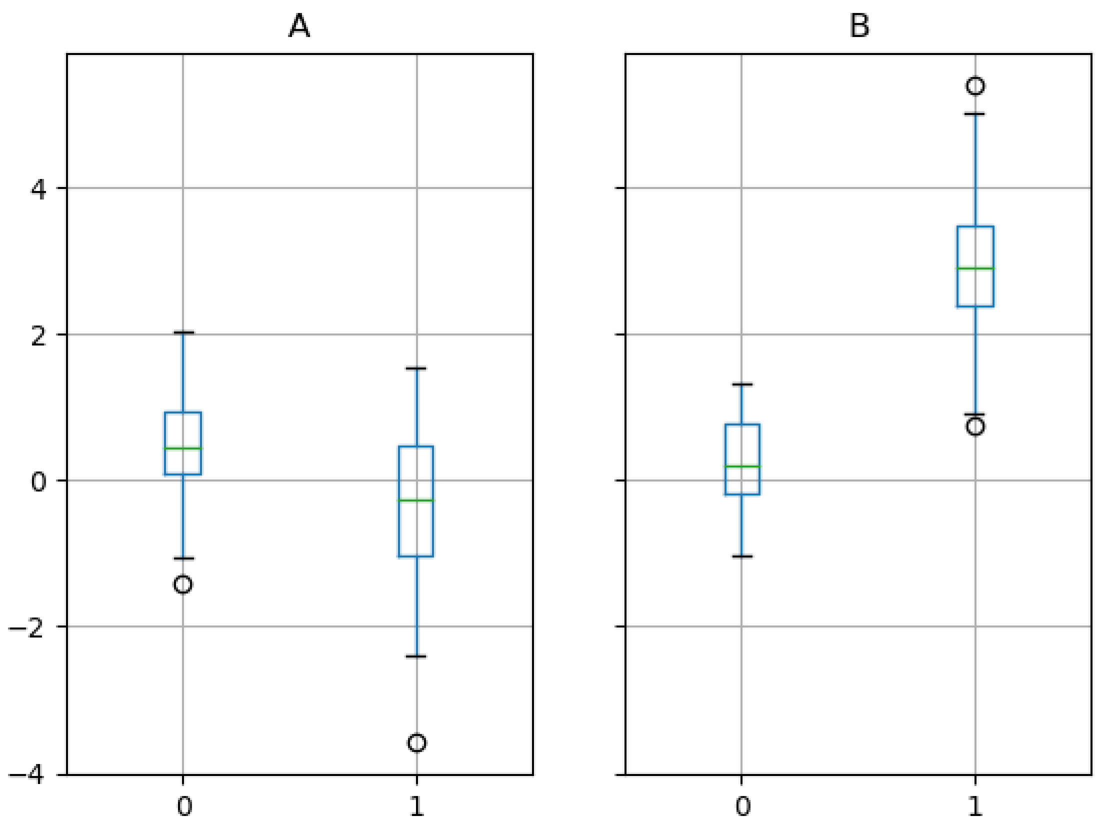
`Groupby.boxplot` always returns a `Series` of `return_type`.

```
In [54]: np.random.seed(1234)

In [55]: df_box = pd.DataFrame(np.random.randn(50, 2))

In [56]: df_box['g'] = np.random.choice(['A', 'B'], size=50)

In [57]: df_box.loc[df_box['g'] == 'B', 1] += 3

In [58]: bp = df_box.boxplot(by='g')
```



The subplots above are split by the numeric columns first, then the value of the `g` column. Below the subplots are first split by the value of `g`, then by the numeric columns.

```
In [59]: bp = df_box.groupby('g').boxplot()
```

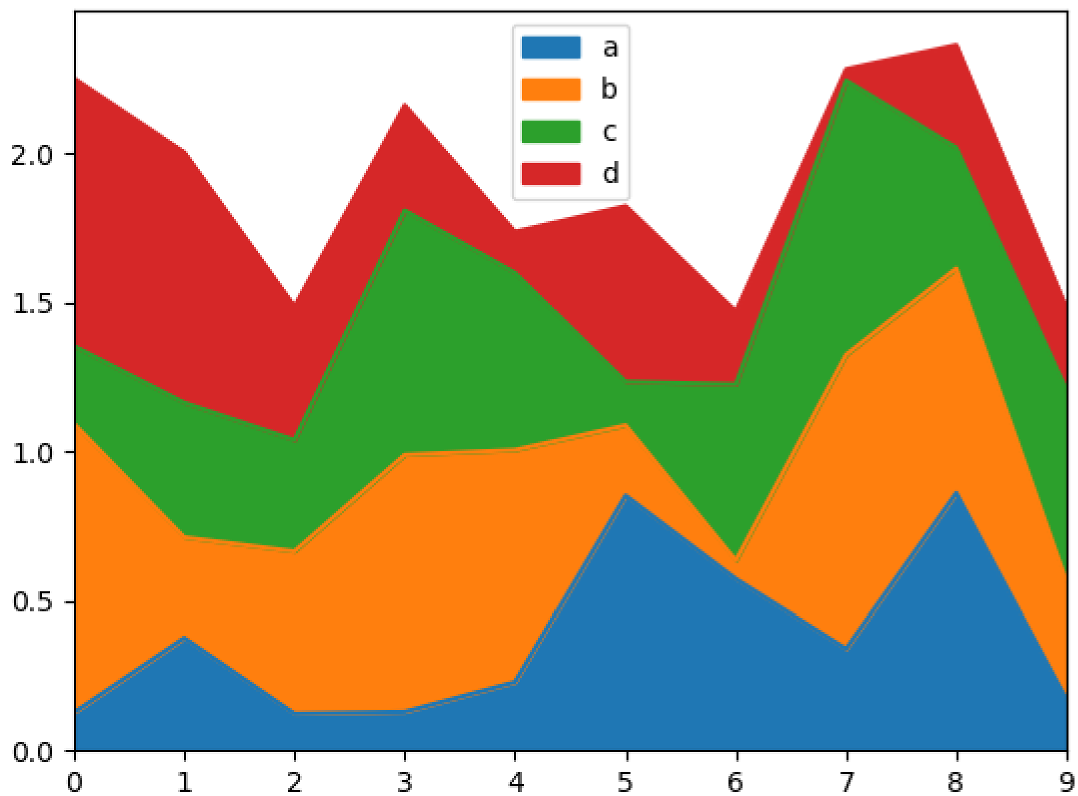


Area Plot

You can create area plots with *Series.plot.area()* and *DataFrame.plot.area()*. Area plots are stacked by default. To produce stacked area plot, each column must be either all positive or all negative values.

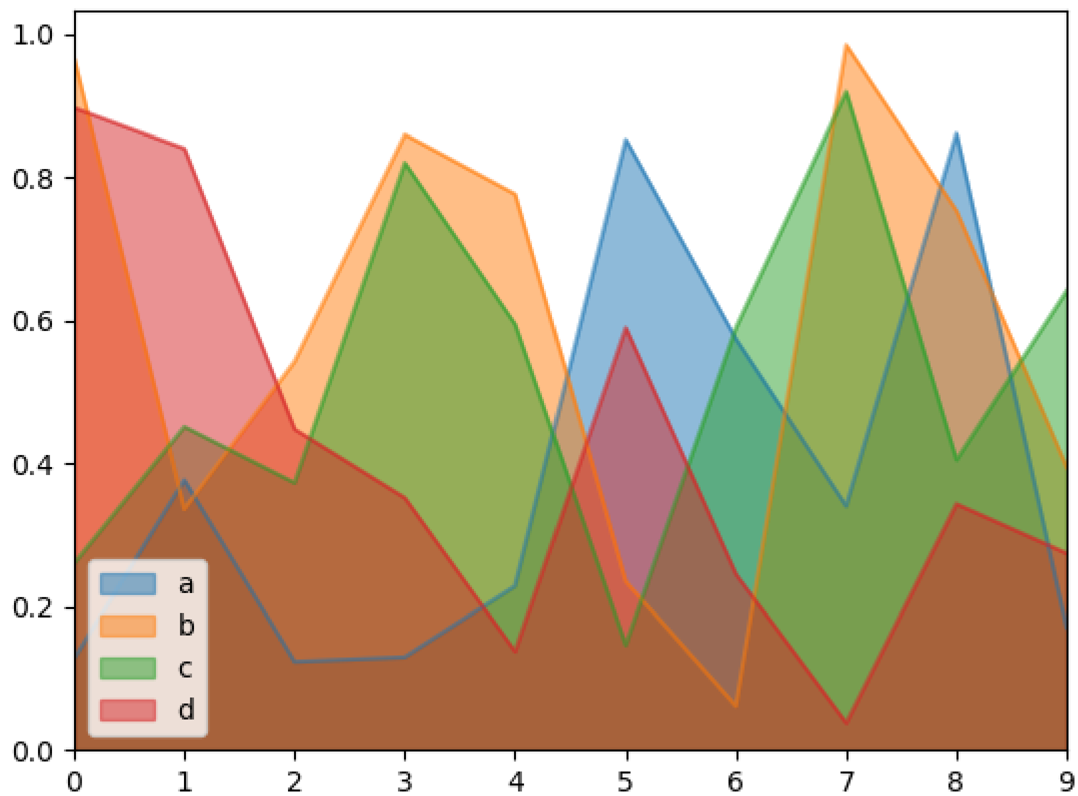
When input data contains *NaN*, it will be automatically filled by 0. If you want to drop or fill by different values, use `dataframe.dropna()` or `dataframe.fillna()` before calling *plot*.

```
In [60]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

In [61]: df.plot.area();
```



To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

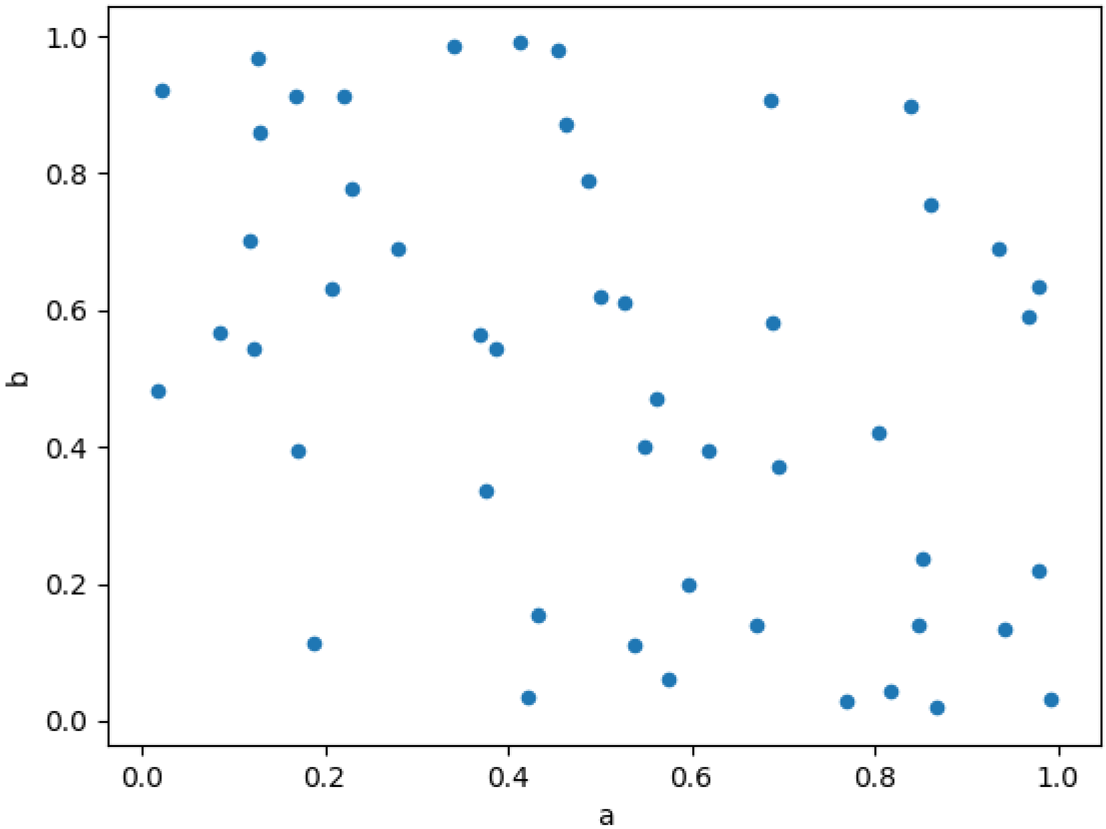
```
In [62]: df.plot.area(stacked=False);
```



Scatter Plot

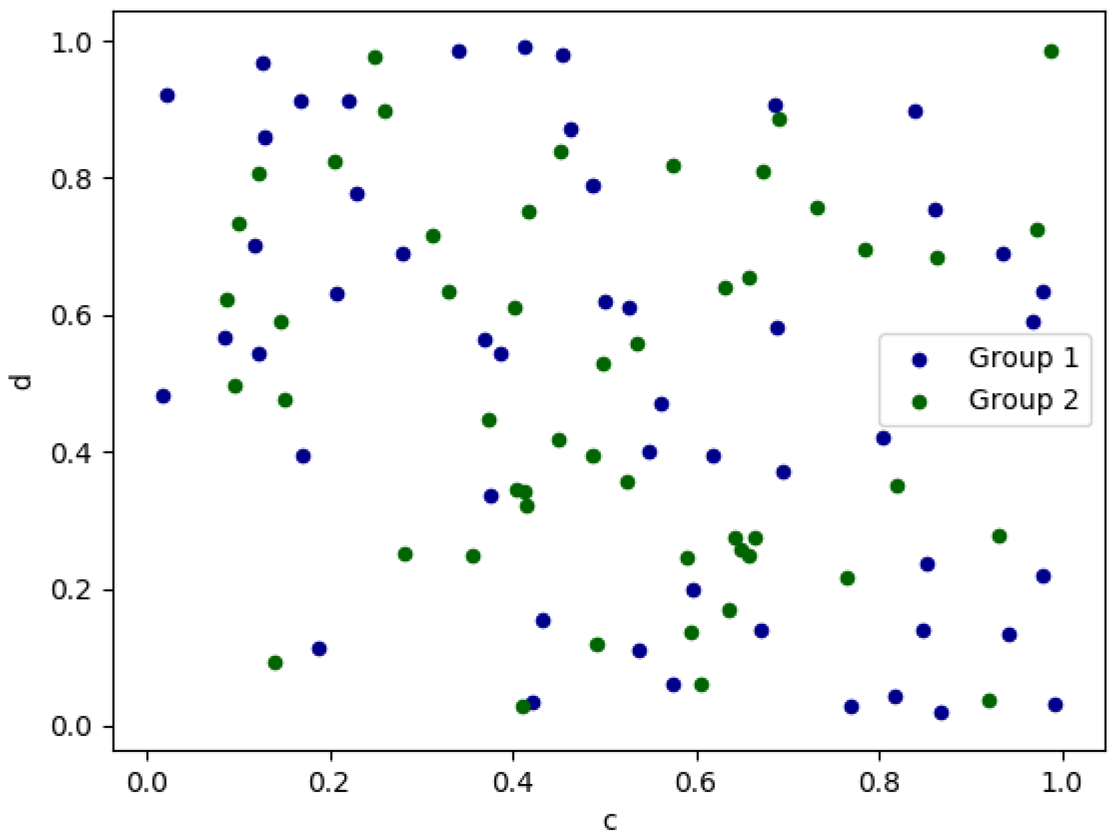
Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for the x and y axes. These can be specified by the `x` and `y` keywords.

```
In [63]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])

In [64]: df.plot.scatter(x='a', y='b');
```



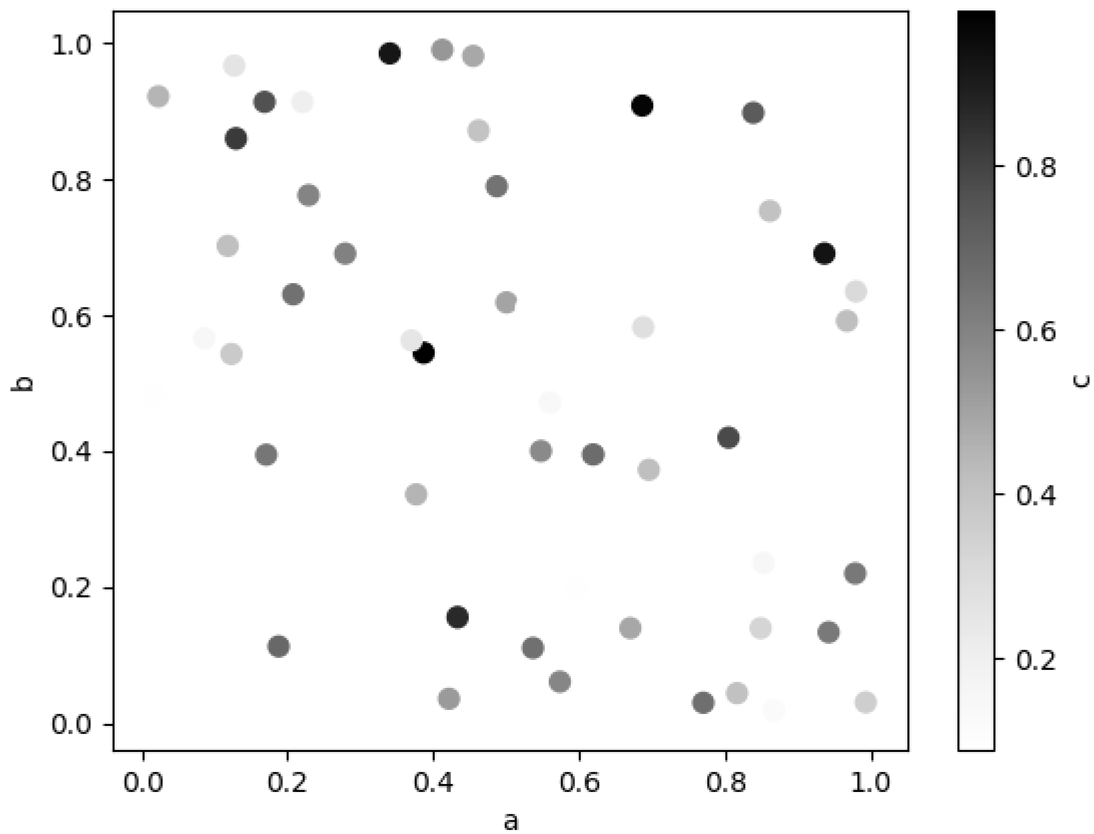
To plot multiple column groups in a single axes, repeat `plot` method specifying target `ax`. It is recommended to specify `color` and `label` keywords to distinguish each groups.

```
In [65]: ax = df.plot.scatter(x='a', y='b', color='DarkBlue', label='Group 1');

In [66]: df.plot.scatter(x='c', y='d', color='DarkGreen', label='Group 2', ax=ax);
```



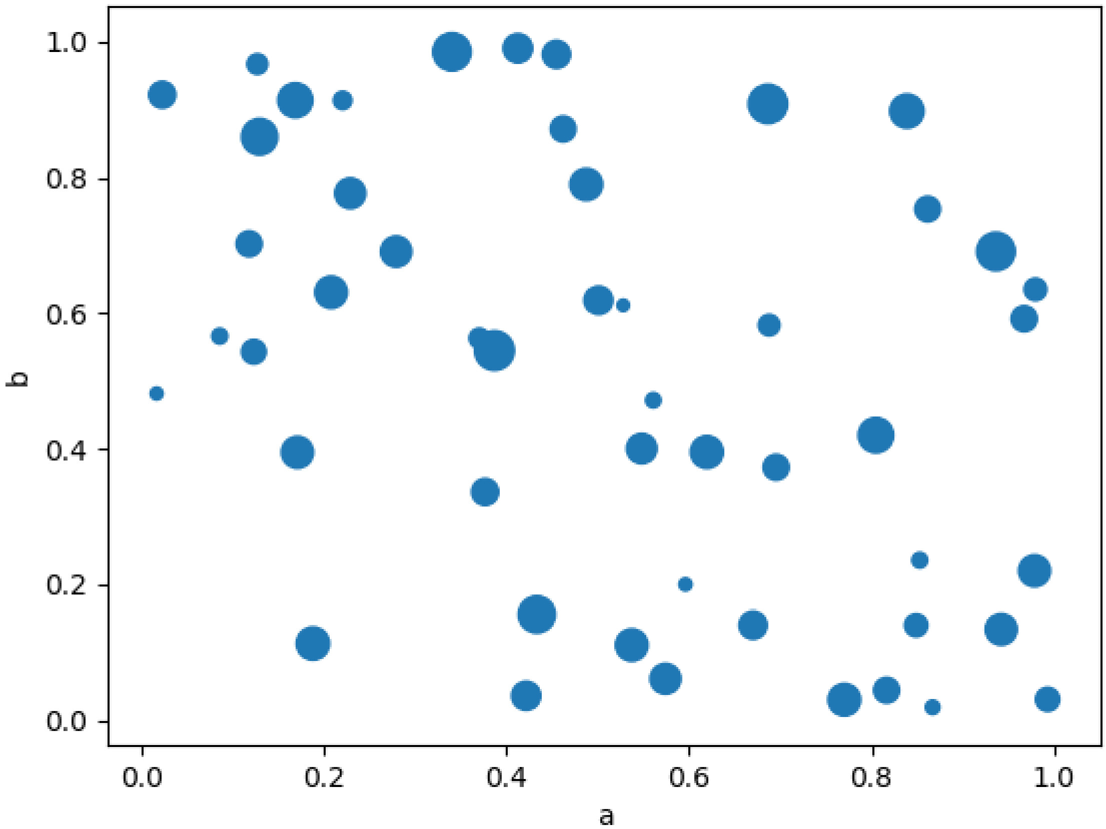
The keyword c may be given as the name of a column to provide colors for each point:

```
In [67]: df.plot.scatter(x='a', y='b', c='c', s=50);
```



You can pass other keywords supported by matplotlib `scatter`. The example below shows a bubble chart using a column of the `DataFrame` as the bubble size.

```
In [68]: df.plot.scatter(x='a', y='b', s=df['c'] * 200);
```

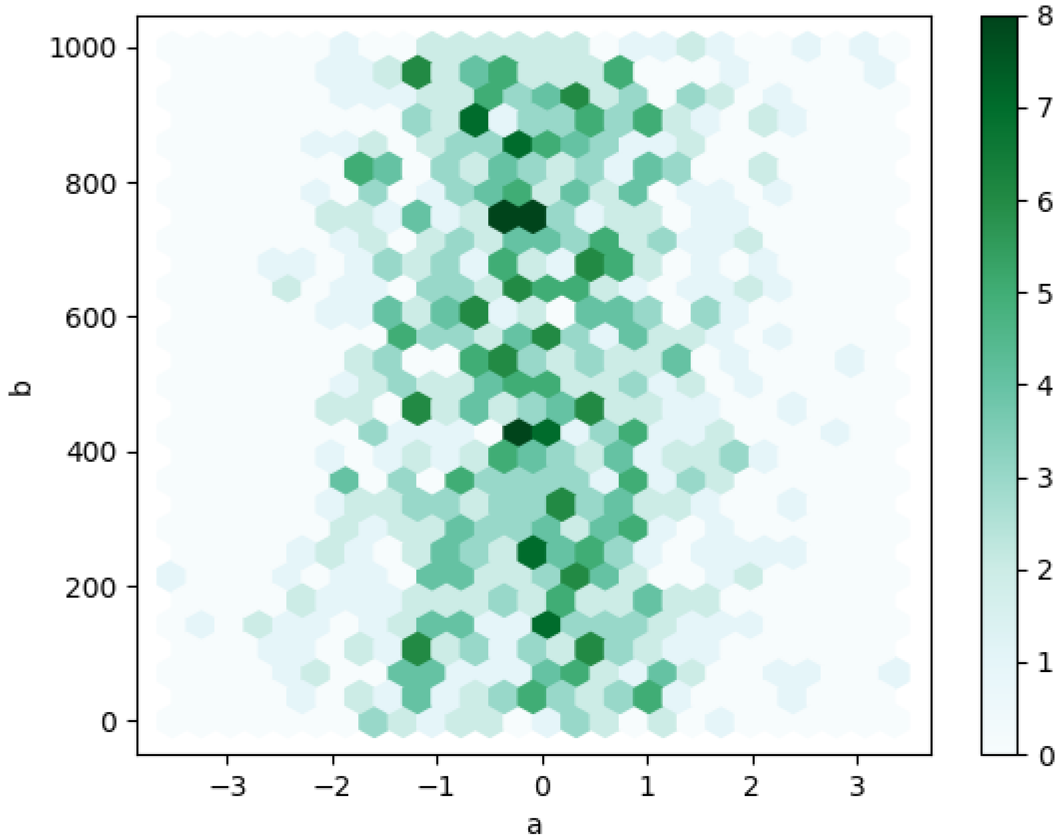


See the `scatter` method and the matplotlib scatter documentation for more.

Hexagonal Bin Plot

You can create hexagonal bin plots with *DataFrame.plot.hexbin()*. Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

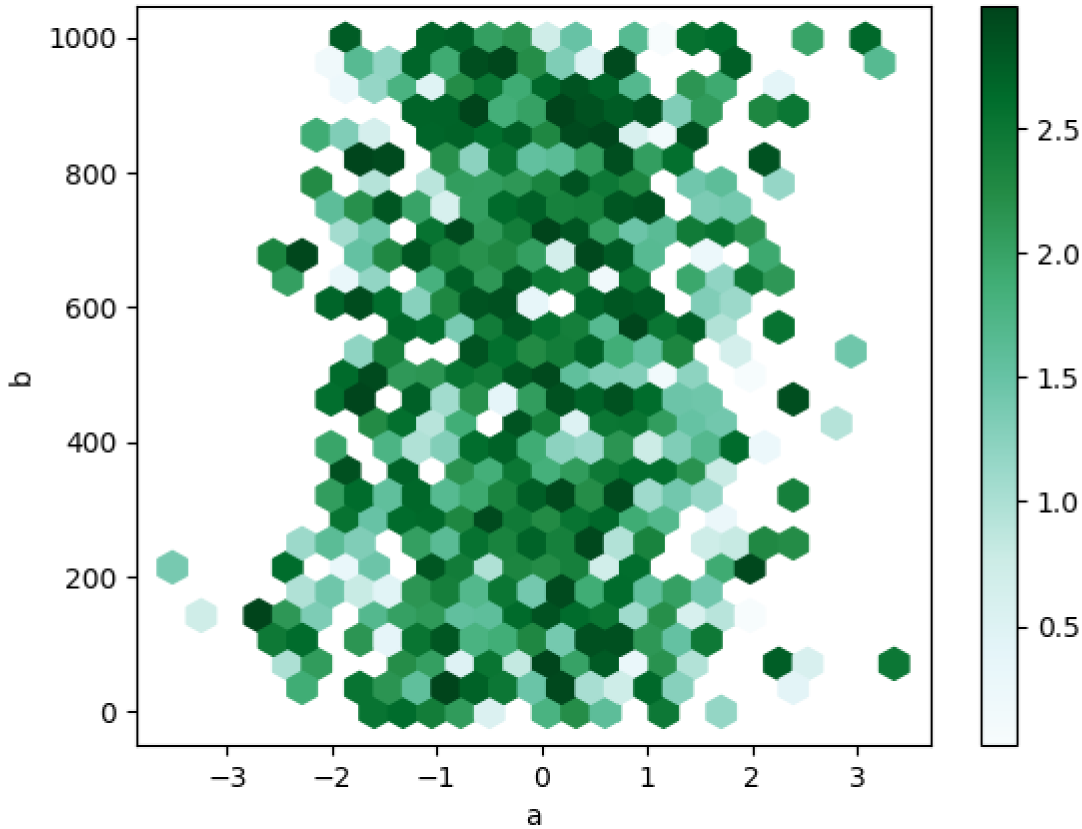
```
In [69]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

In [70]: df['b'] = df['b'] + np.arange(1000)

In [71]: df.plot.hexbin(x='a', y='b', gridsize=25)
Out[71]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e41b9898>
```



A useful keyword argument is `gridsize`; it controls the number of hexagons in the x-direction, and defaults to 100. A larger `gridsize` means more, smaller bins.

By default, a histogram of the counts around each `(x, y)` point is computed. You can specify alternative aggregations by passing values to the `C` and `reduce_C_function` arguments. `C` specifies the value at each `(x, y)` point and `reduce_C_function` is a function of one argument that reduces all the values in a bin to a single number (e.g. `mean`, `max`, `sum`, `std`). In this example the positions are given by columns `a` and `b`, while the value is given by column `z`. The bins are aggregated with NumPy's `max` function.

```
In [72]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

In [73]: df['b'] = df['b'] = df['b'] + np.arange(1000)

In [74]: df['z'] = np.random.uniform(0, 3, 1000)

In [75]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max, gridsize=25)
Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e4158828>
```

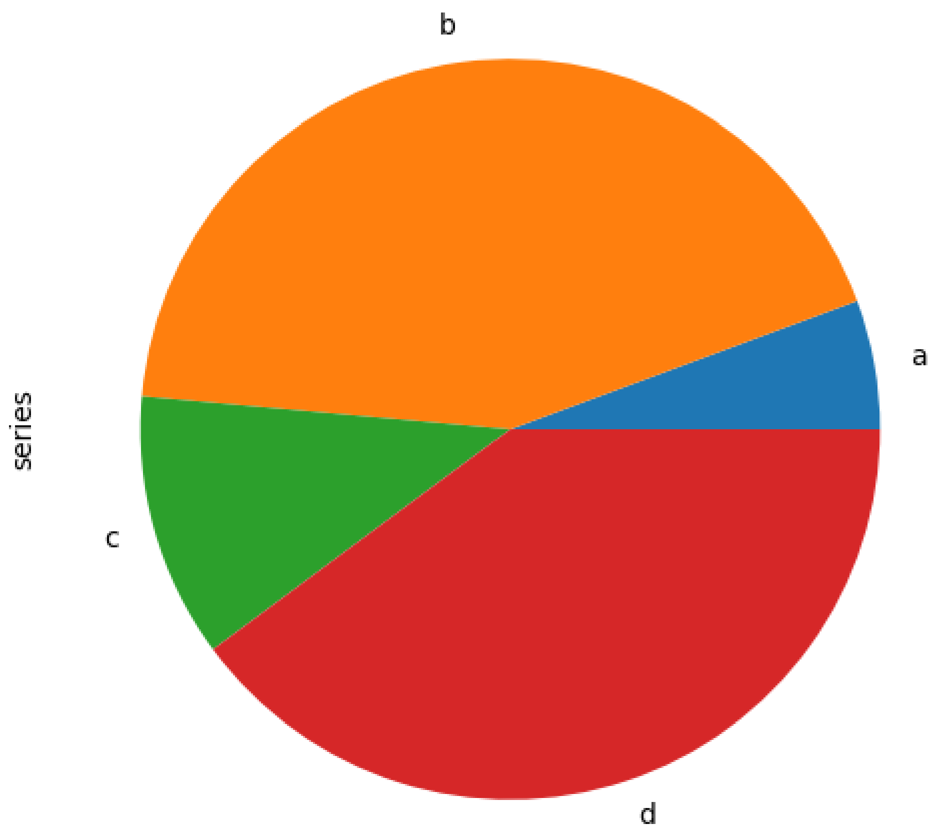


See the hexbin method and the matplotlib hexbin documentation for more.

Pie plot

You can create a pie plot with *DataFrame.plot.pie()* or *Series.plot.pie()*. If your data includes any NaN, they will be automatically filled with 0. A ValueError will be raised if there are any negative values in your data.

```
In [76]: series = pd.Series(3 * np.random.rand(4),
   ....:                    index=['a', 'b', 'c', 'd'], name='series')
   ....:

In [77]: series.plot.pie(figsize=(6, 6))
Out[77]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e4075208>
```



For pie plots it's best to use square figures, i.e. a figure aspect ratio 1. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling `ax.set_aspect('equal')` on the returned `axes` object.

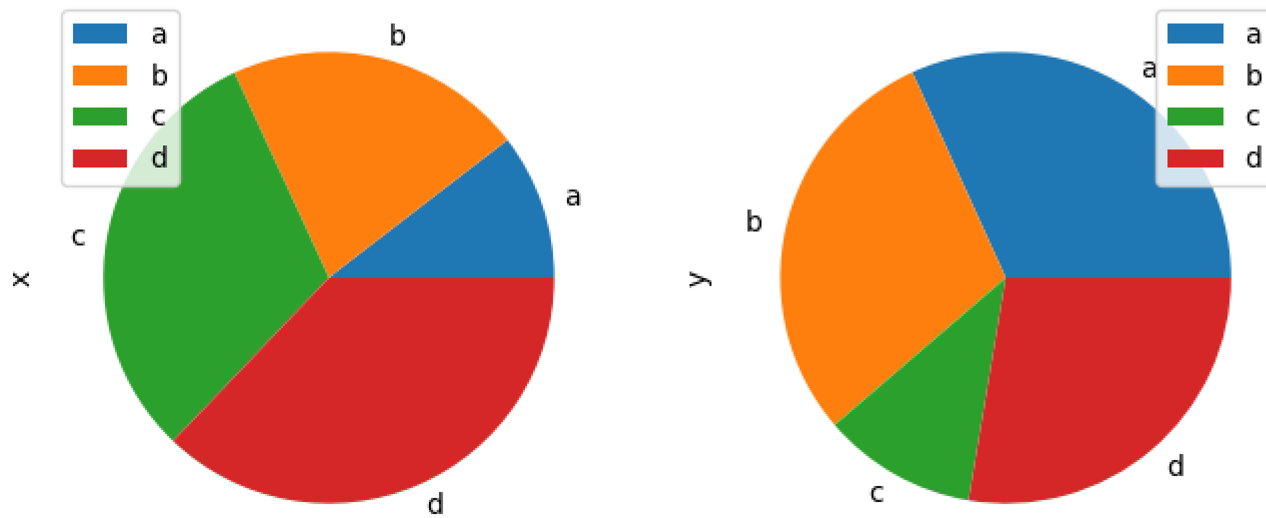
Note that pie plot with *DataFrame* requires that you either specify a target column by the `y` argument or `subplots=True`. When `y` is specified, pie plot of selected column will be drawn. If `subplots=True` is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify `legend=False` to hide it.

```
In [78]: df = pd.DataFrame(3 * np.random.rand(4, 2),
   ....:                   index=['a', 'b', 'c', 'd'], columns=['x', 'y'])
   ....:

In [79]: df.plot.pie(subplots=True, figsize=(8, 4))
Out[79]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f37e4040a58>,
```

(continues on next page)

(continued from previous page)

```
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f37df16a1d0>],
↪dtype=object)
```

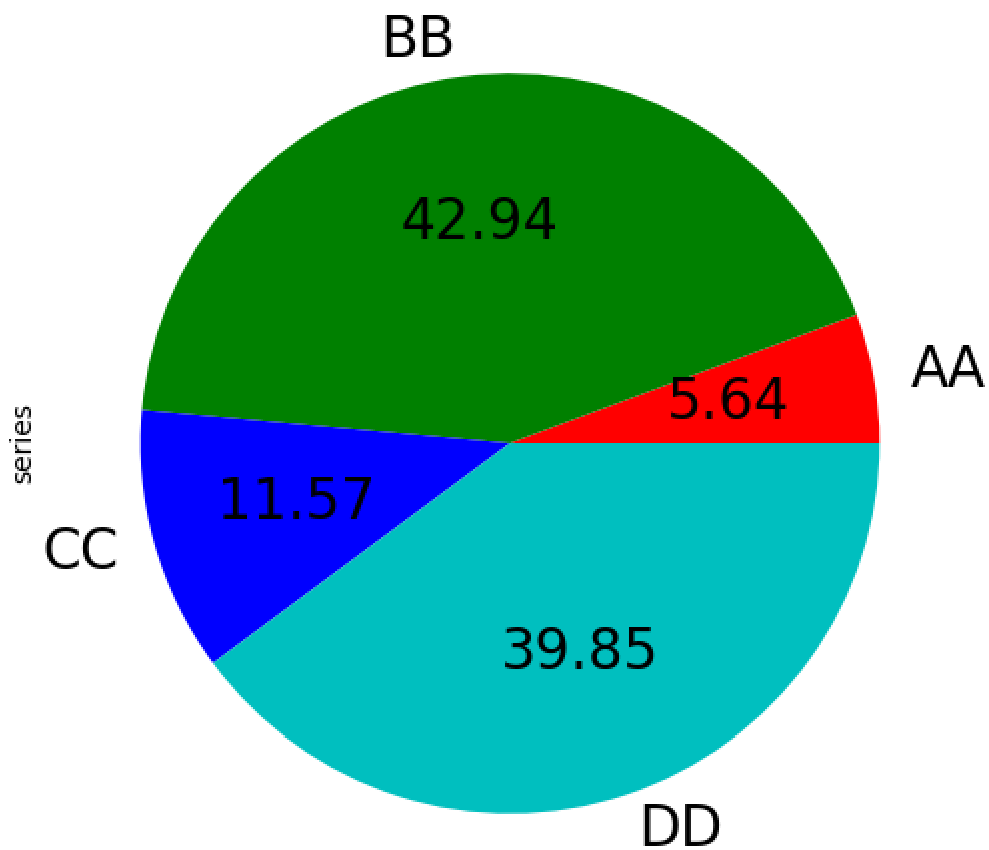


You can use the `labels` and `colors` keywords to specify the labels and colors of each wedge.

Warning: Most pandas plots use the `label` and `color` arguments (note the lack of "s" on those). To be consistent with `matplotlib.pyplot.pie()` you must use `labels` and `colors`.

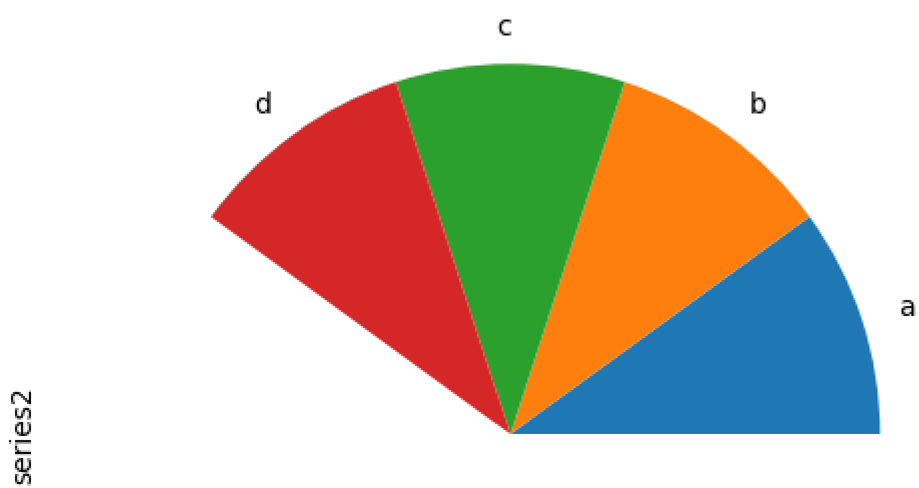
If you want to hide wedge labels, specify `labels=None`. If `fontsize` is specified, the value will be applied to wedge labels. Also, other keywords supported by `matplotlib.pyplot.pie()` can be used.

```
In [80]: series.plot.pie(labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
   ....:                 autopct='%.2f', fontsize=20, figsize=(6, 6))
   ....:
Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37df0dbc18>
```



If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```
In [81]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')

In [82]: series.plot.pie(figsize=(6, 6))
Out[82]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37df0e40f0>
```



See the matplotlib pie documentation for more.

4.10.3 Plotting with Missing Data

Pandas tries to be pragmatic about plotting `DataFrames` or `Series` that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.

Plot Type	NaN Handling
Line	Leave gaps at NaNs
Line (stacked)	Fill 0's
Bar	Fill 0's
Scatter	Drop NaNs
Histogram	Drop NaNs (column-wise)
Box	Drop NaNs (column-wise)
Area	Fill 0's
KDE	Drop NaNs (column-wise)
Hexbin	Drop NaNs
Pie	Fill 0's

If any of these defaults are not what you want, or if you want to be explicit about how missing values are handled, consider using *fillna()* or *dropna()* before plotting.

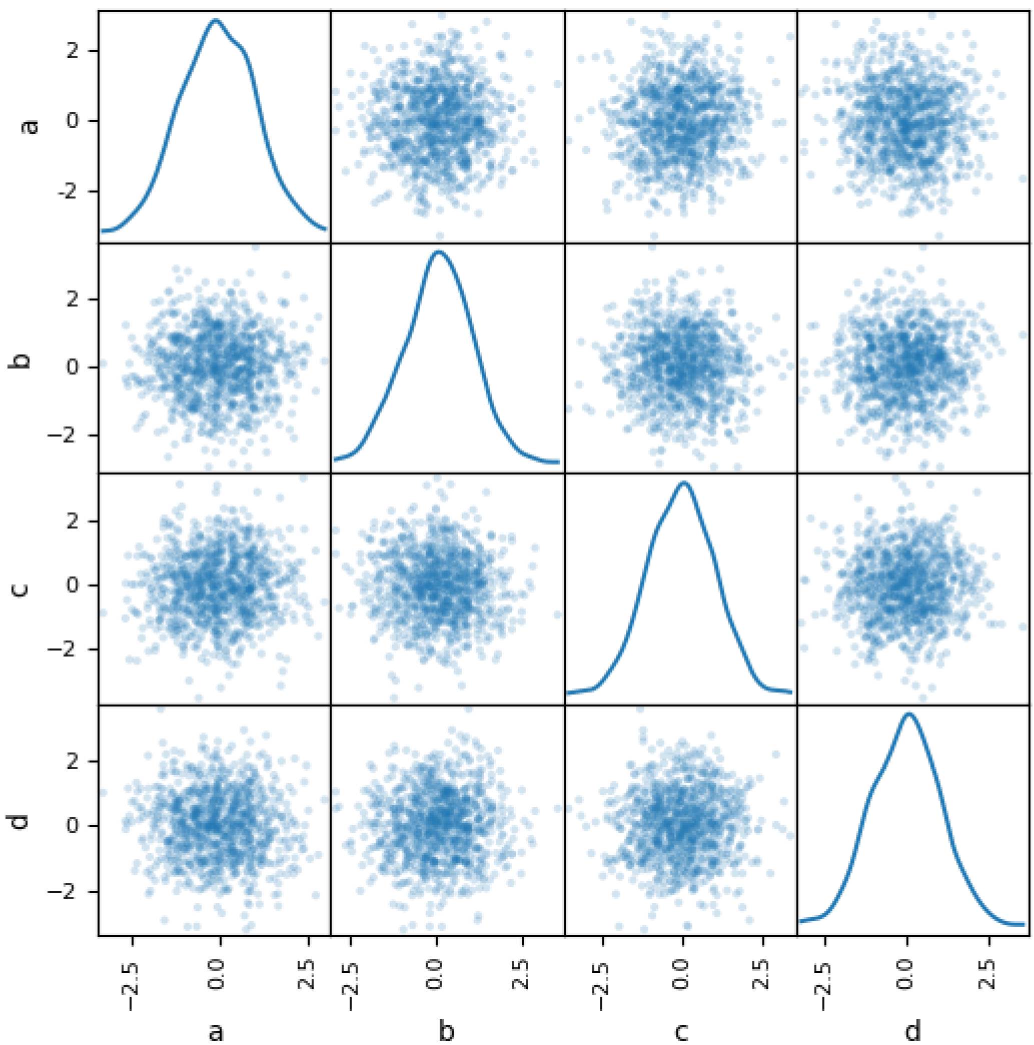
4.10.4 Plotting Tools

These functions can be imported from `pandas.plotting` and take a *Series* or *DataFrame* as an argument.

Scatter Matrix Plot

You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.plotting`:

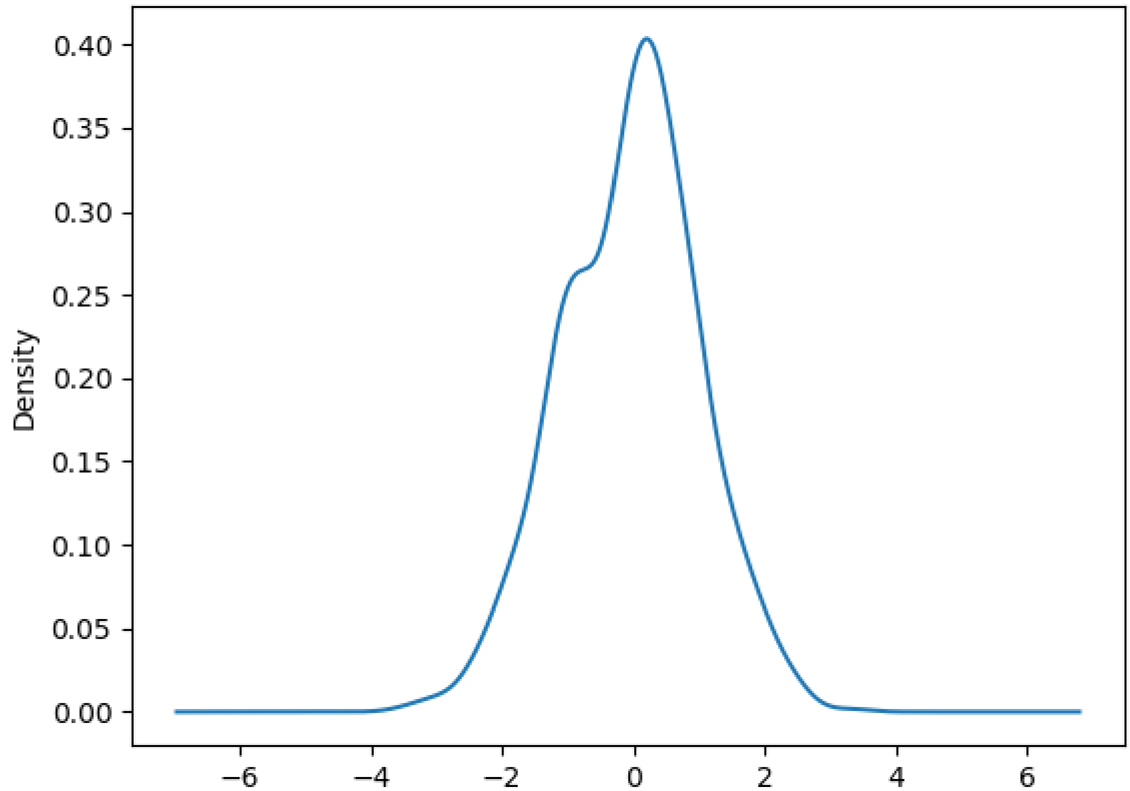
```
In [83]: from pandas.plotting import scatter_matrix

In [84]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

In [85]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
Out[85]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f37df00ca90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37df033cf8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37defdaf60>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37def8d208>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f37defb6470>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37def5f6d8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37def075f8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37def30898>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f37def308d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37dee82d30>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37deeaaf98>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37dee5c240>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f37dee044a8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37dee2c710>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37dedd6978>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f37ded82be0>]],␣
↪dtype=object)
```



Density Plot

You can create density plots using the *Series.plot.kde()* and *DataFrame.plot.kde()* methods.

```
In [86]: ser = pd.Series(np.random.randn(1000))

In [87]: ser.plot.kde()
Out[87]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37dec48d68>
```

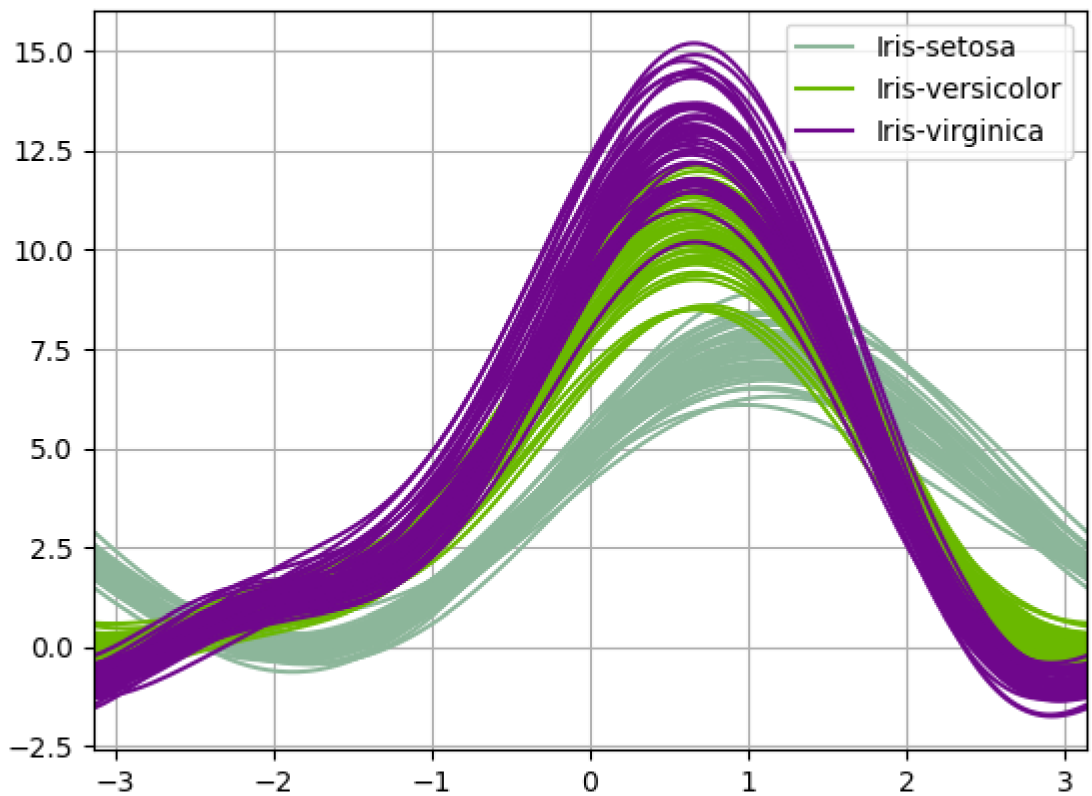


Andrews Curves

Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series, see the Wikipedia entry for more information. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

Note: The "Iris" dataset is available here.

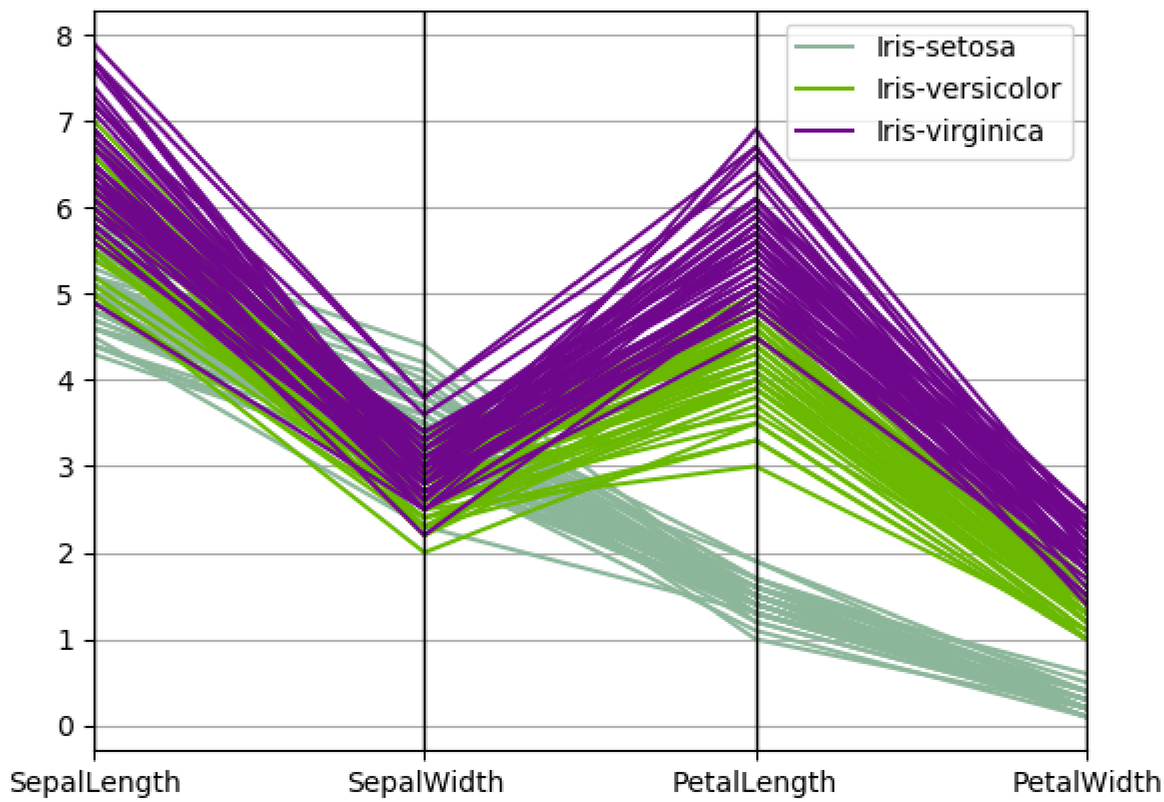
```
In [88]: from pandas.plotting import andrews_curves

In [89]: data = pd.read_csv('data/iris.data')

In [90]: plt.figure()
Out[90]: <Figure size 640x480 with 0 Axes>

In [91]: andrews_curves(data, 'Name')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[91]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e65bb470>
```



Parallel Coordinates

Parallel coordinates is a plotting technique for plotting multivariate data, see the Wikipedia entry for an introduction. Parallel coordinates allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

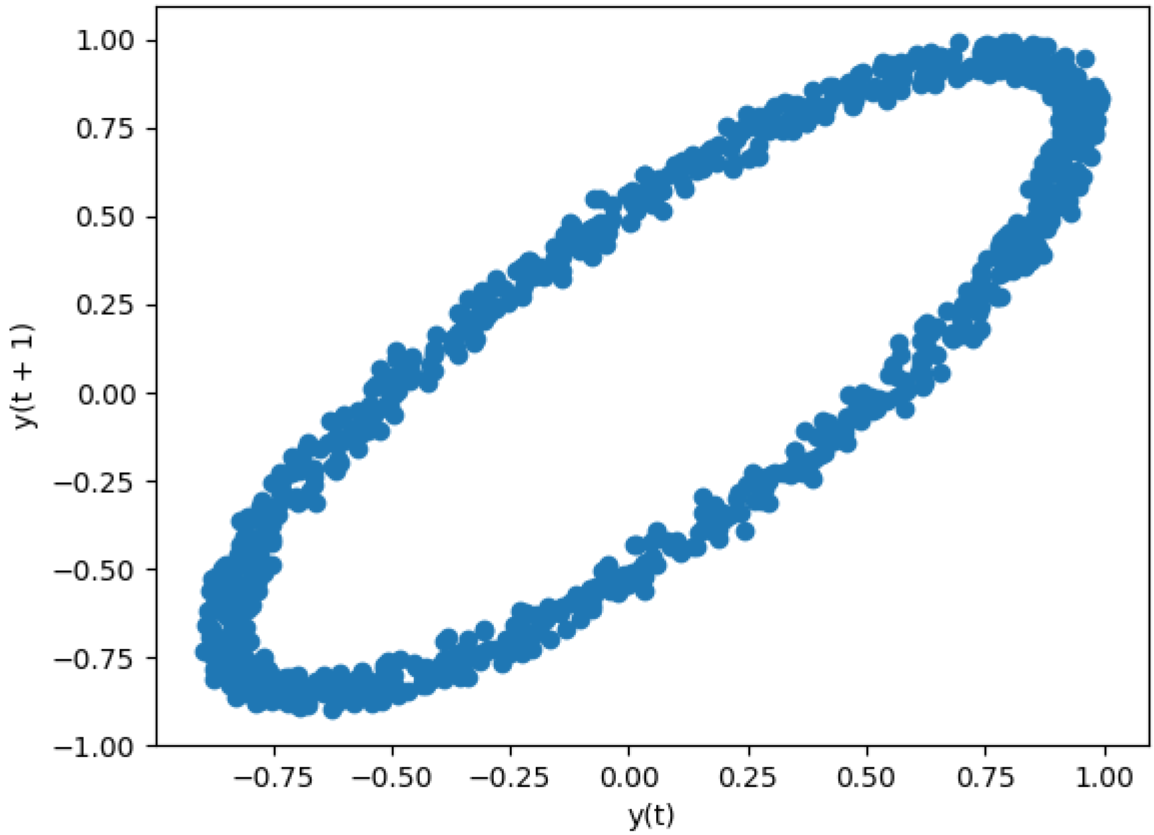
```
In [92]: from pandas.plotting import parallel_coordinates

In [93]: data = pd.read_csv('data/iris.data')

In [94]: plt.figure()
Out[94]: <Figure size 640x480 with 0 Axes>

In [95]: parallel_coordinates(data, 'Name')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[95]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e65bb6d8>
```



Lag Plot

Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random. The `lag` argument may be passed, and when `lag=1` the plot is essentially `data[:-1]` vs. `data[1:]`.

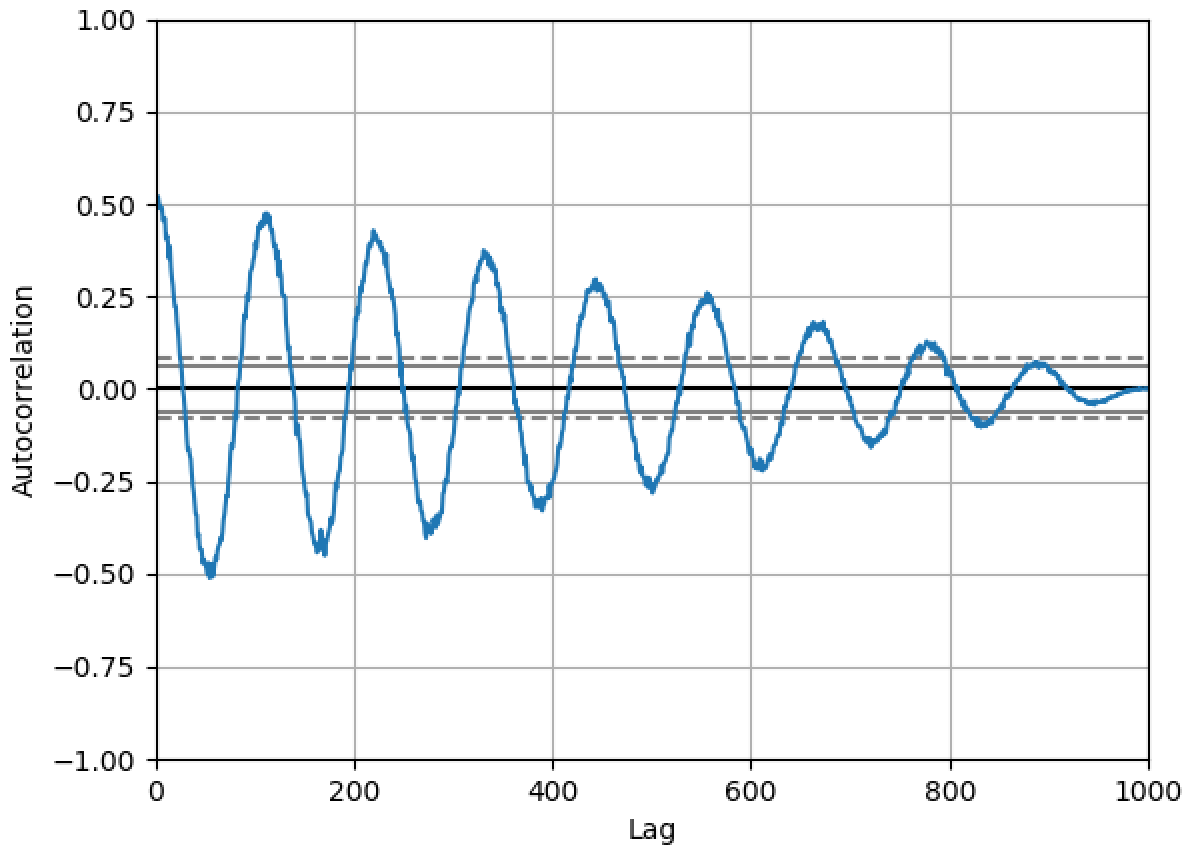
```
In [96]: from pandas.plotting import lag_plot

In [97]: plt.figure()
Out[97]: <Figure size 640x480 with 0 Axes>

In [98]: spacing = np.linspace(-99 * np.pi, 99 * np.pi, num=1000)

In [99]: data = pd.Series(0.1 * np.random.rand(1000) + 0.9 * np.sin(spacing))

In [100]: lag_plot(data)
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381c063be0>
```



Autocorrelation Plot

Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band. See the Wikipedia entry for more about autocorrelation plots.

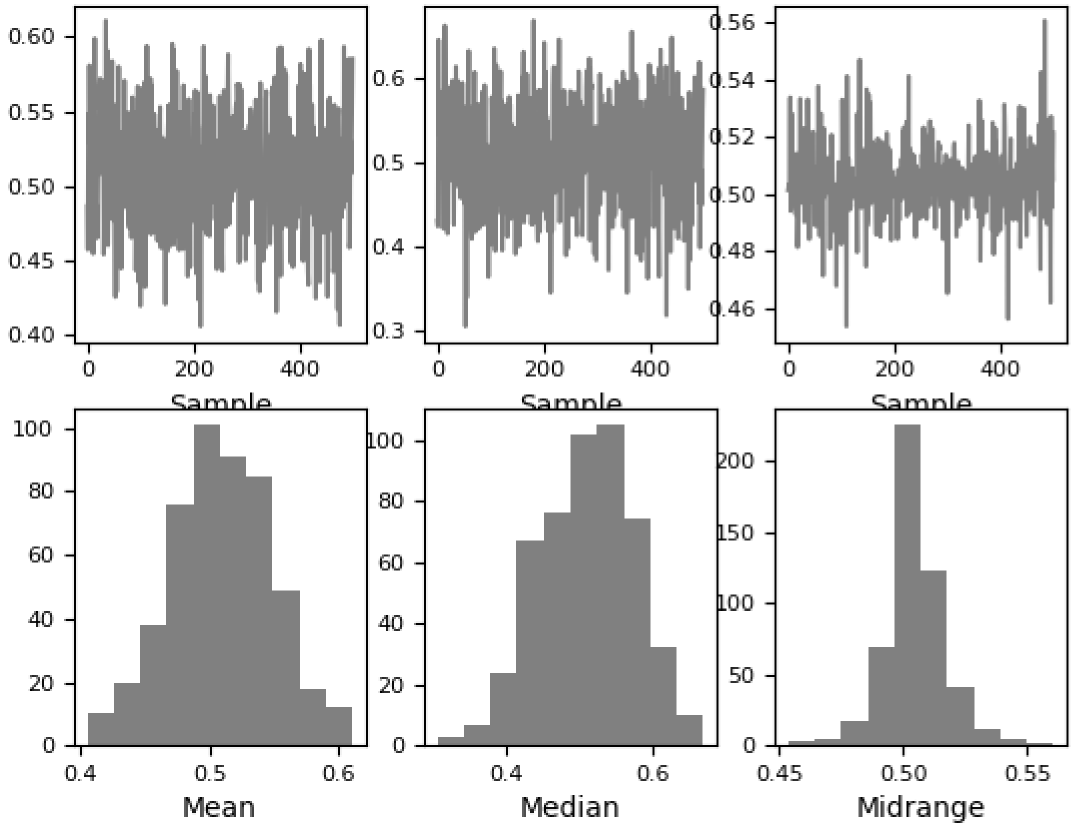
```
In [101]: from pandas.plotting import autocorrelation_plot

In [102]: plt.figure()
Out[102]: <Figure size 640x480 with 0 Axes>

In [103]: spacing = np.linspace(-9 * np.pi, 9 * np.pi, num=1000)

In [104]: data = pd.Series(0.7 * np.random.rand(1000) + 0.3 * np.sin(spacing))

In [105]: autocorrelation_plot(data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37f019f278>
```



Bootstrap Plot

Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc. A random subset of a specified size is selected from a data set, the statistic in question is computed for this subset and the process is repeated a specified number of times. Resulting plots and histograms are what constitutes the bootstrap plot.

```
In [106]: from pandas.plotting import bootstrap_plot

In [107]: data = pd.Series(np.random.rand(1000))

In [108]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[108]: <Figure size 640x480 with 6 Axes>
```

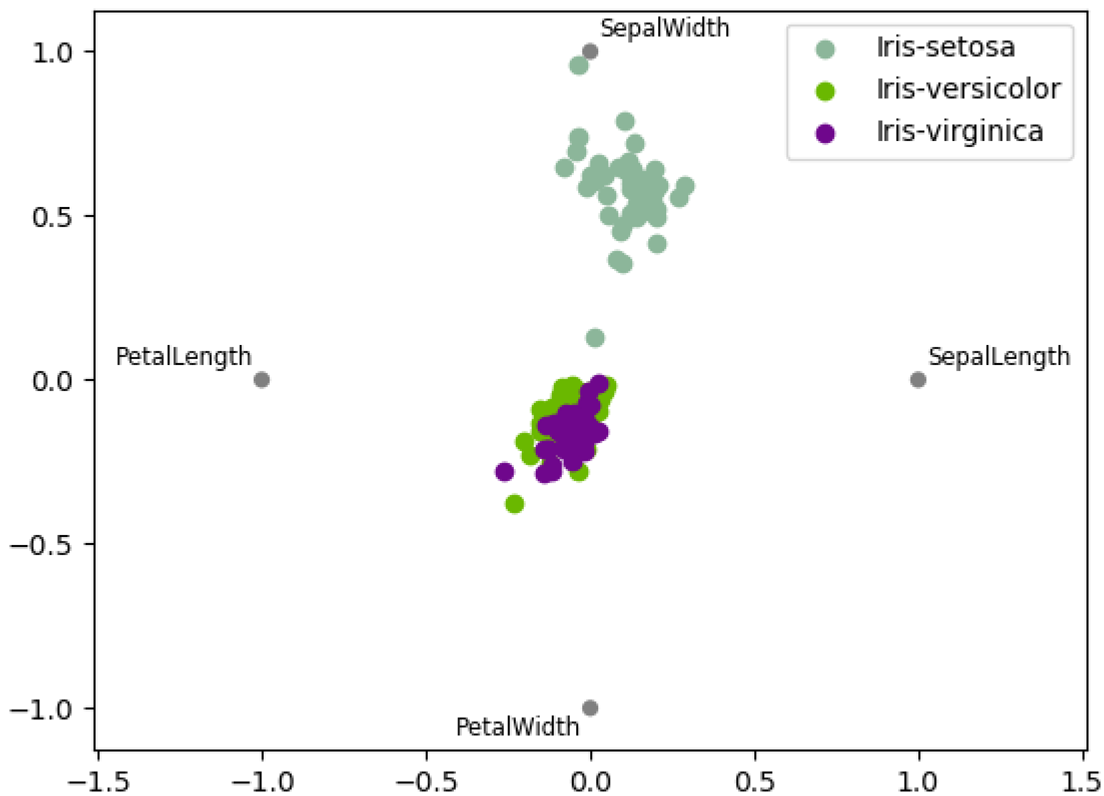


RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs it will be colored differently. See the R package Radviz for more information.

Note: The “Iris” dataset is available here.

```
In [109]: from pandas.plotting import radviz

In [110]: data = pd.read_csv('data/iris.data')

In [111]: plt.figure()
Out[111]: <Figure size 640x480 with 0 Axes>

In [112]: radviz(data, 'Name')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[112]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e4dc0828>
```



4.10.5 Plot Formatting

Setting the plot style

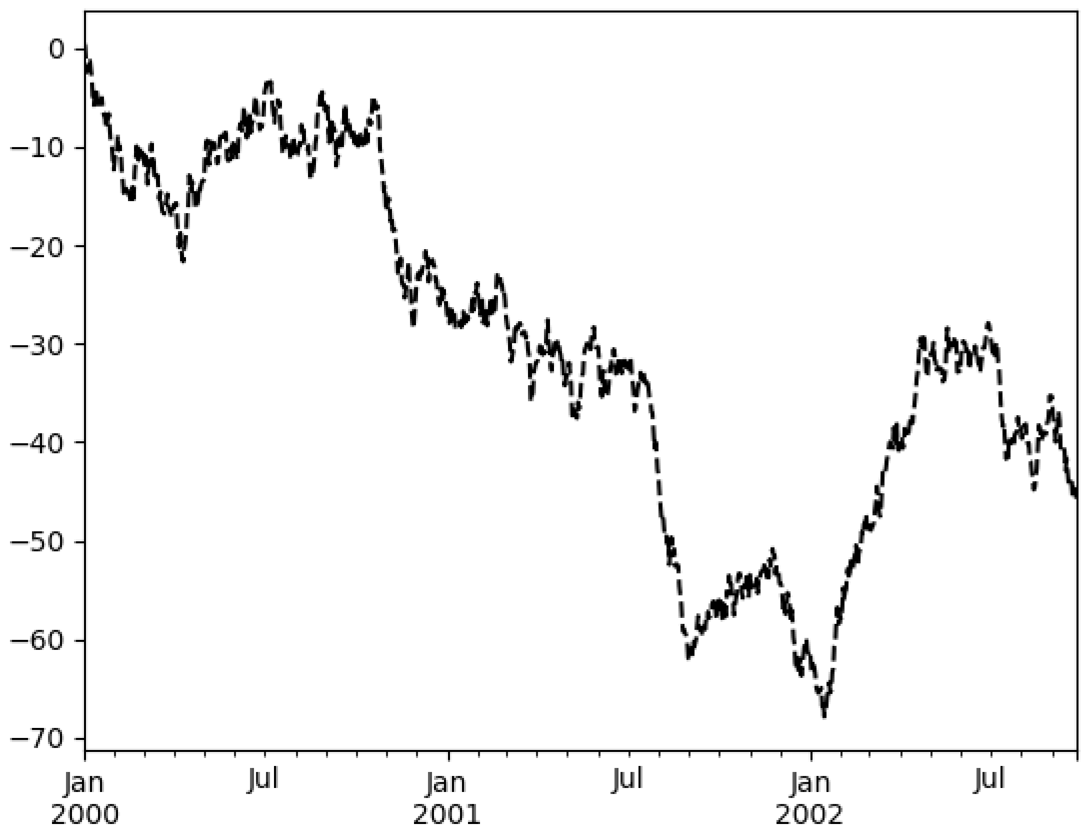
From version 1.5 and up, matplotlib offers a range of pre-configured plotting styles. Setting the style can be used to easily give plots the general look that you want. Setting the style is as easy as calling `matplotlib.style.use(my_plot_style)` before creating your plot. For example you could write `matplotlib.style.use('ggplot')` for ggplot-style plots.

You can see the various available style names at `matplotlib.style.available` and it's very easy to try them out.

General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [113]: plt.figure();

In [114]: ts.plot(style='k--', label='Series');
```

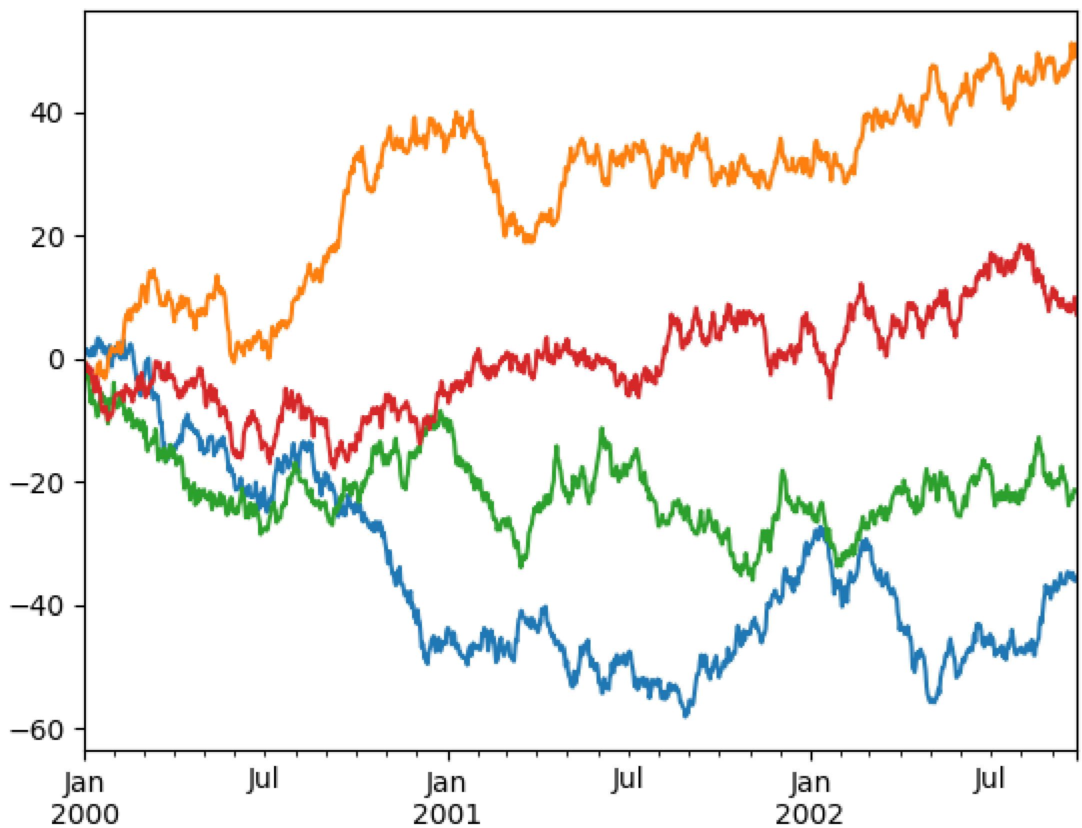


For each kind of plot (e.g. *line*, *bar*, *scatter*) any additional arguments keywords are passed along to the corresponding matplotlib function (`ax.plot()`, `ax.bar()`, `ax.scatter()`). These can be used to control additional styling, beyond what pandas provides.

Controlling the Legend

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

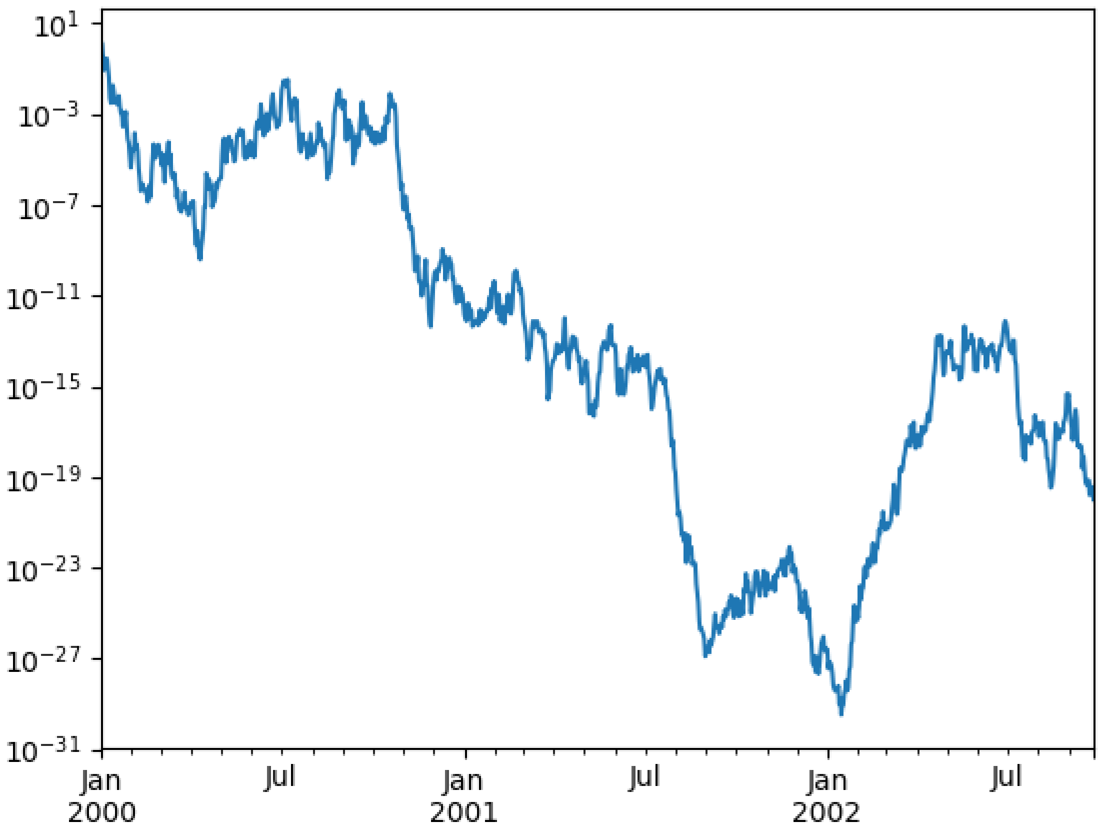
```
In [115]: df = pd.DataFrame(np.random.randn(1000, 4),
   .....:                   index=ts.index, columns=list('ABCD'))
   .....:

In [116]: df = df.cumsum()

In [117]: df.plot(legend=False)
Out[117]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e5b155f8>
```



Scales

You may pass `logy` to get a log-scale Y axis.

```
In [118]: ts = pd.Series(np.random.randn(1000),
   .....:                index=pd.date_range('1/1/2000', periods=1000))
   .....:

In [119]: ts = np.exp(ts.cumsum())

In [120]: ts.plot(logy=True)
Out[120]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381c03afd0>
```

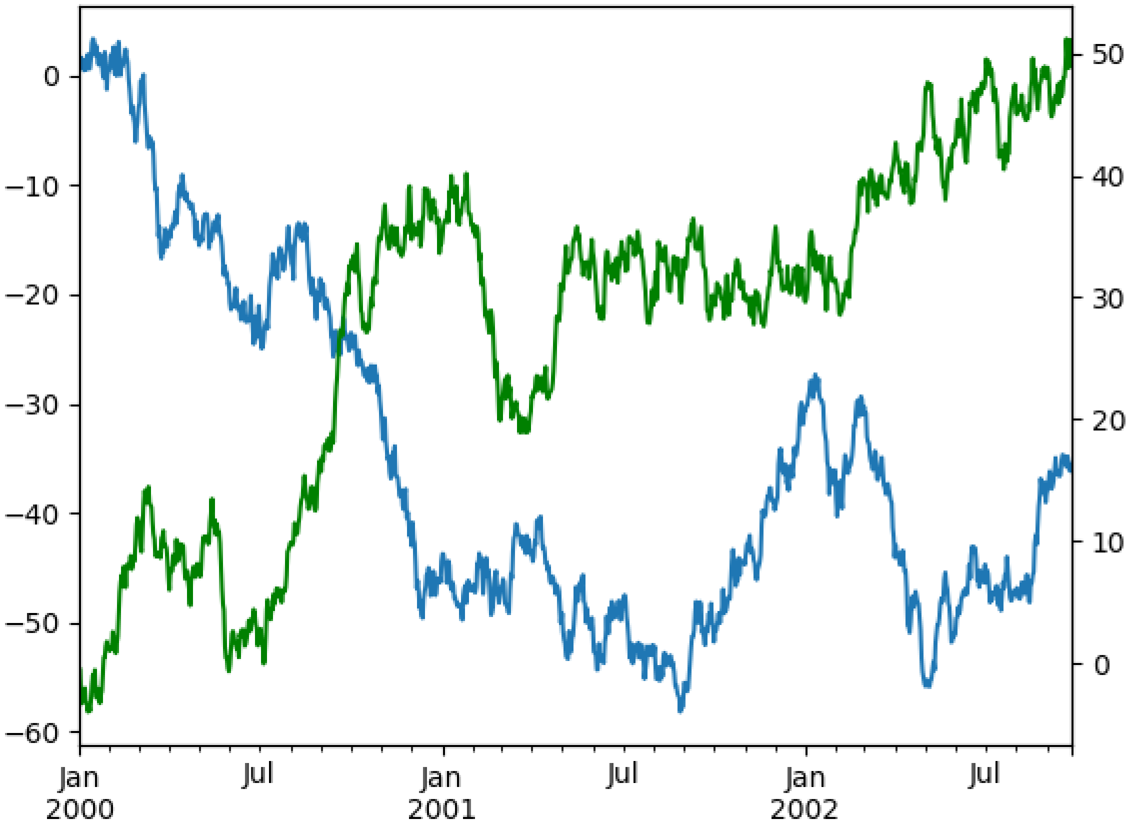


See also the `logx` and `loglog` keyword arguments.

Plotting on a Secondary Y-axis

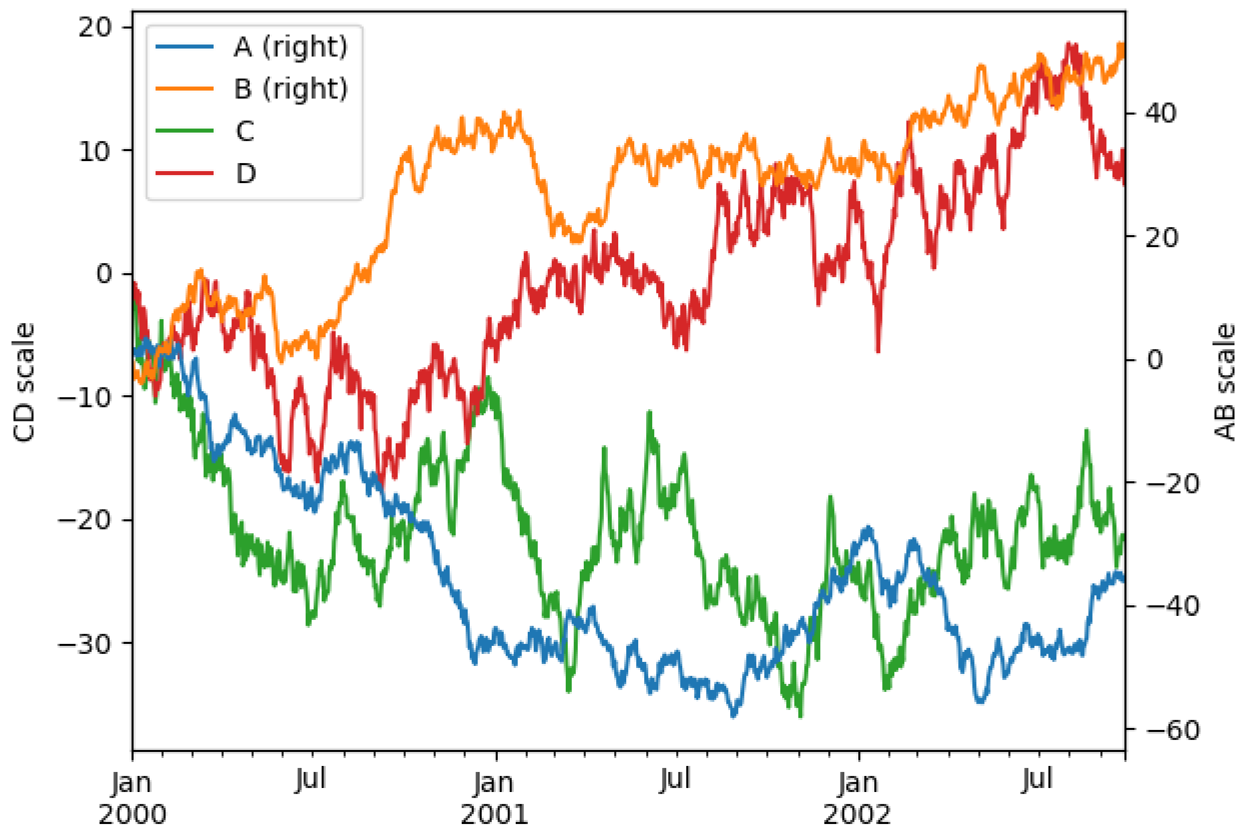
To plot data on a secondary y-axis, use the `secondary_y` keyword:

```
In [121]: df.A.plot()
Out[121]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37e57279e8>

In [122]: df.B.plot(secondary_y=True, style='g')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[122]:
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f37dec6d048>
```



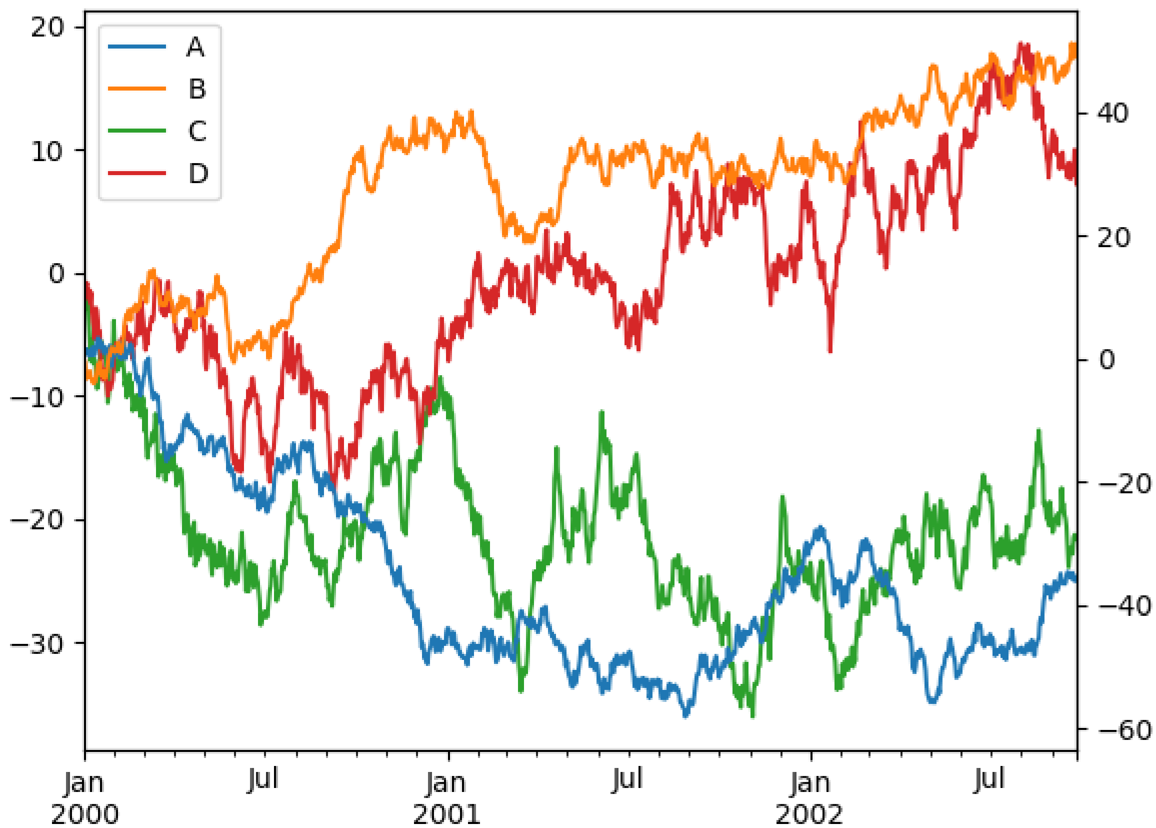
To plot some columns in a `DataFrame`, give the column names to the `secondary_y` keyword:

```
In [123]: plt.figure()
Out[123]: <Figure size 640x480 with 0 Axes>

In [124]: ax = df.plot(secondary_y=['A', 'B'])

In [125]: ax.set_ylabel('CD scale')
Out[125]: Text(0, 0.5, 'CD scale')

In [126]: ax.right_ax.set_ylabel('AB scale')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[126]: Text(0, 0.5, 'AB scale')
```



Note that the columns plotted on the secondary y-axis is automatically marked with "(right)" in the legend. To turn off the automatic marking, use the `mark_right=False` keyword:

```
In [127]: plt.figure()
Out[127]: <Figure size 640x480 with 0 Axes>

In [128]: df.plot(secondary_y=['A', 'B'], mark_right=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[128]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e513b668>
```

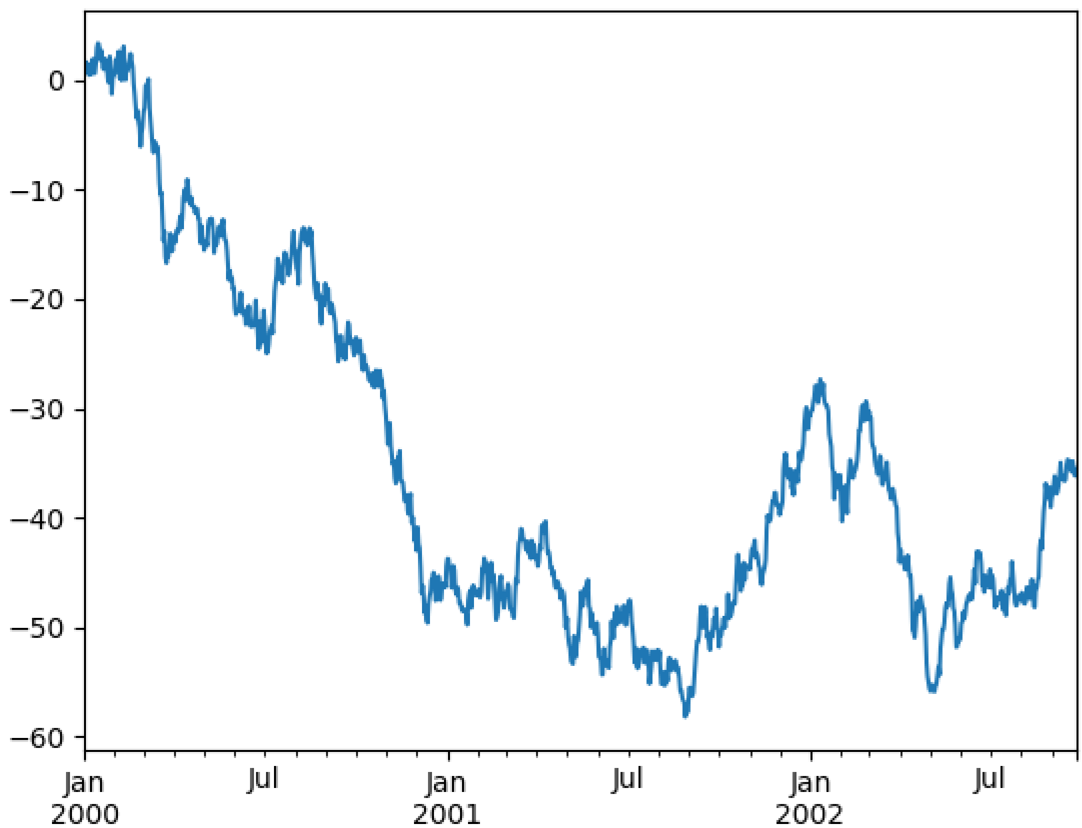


Suppressing Tick Resolution Adjustment

pandas includes automatic tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created `twinx`), you can choose to suppress this behavior for alignment purposes.

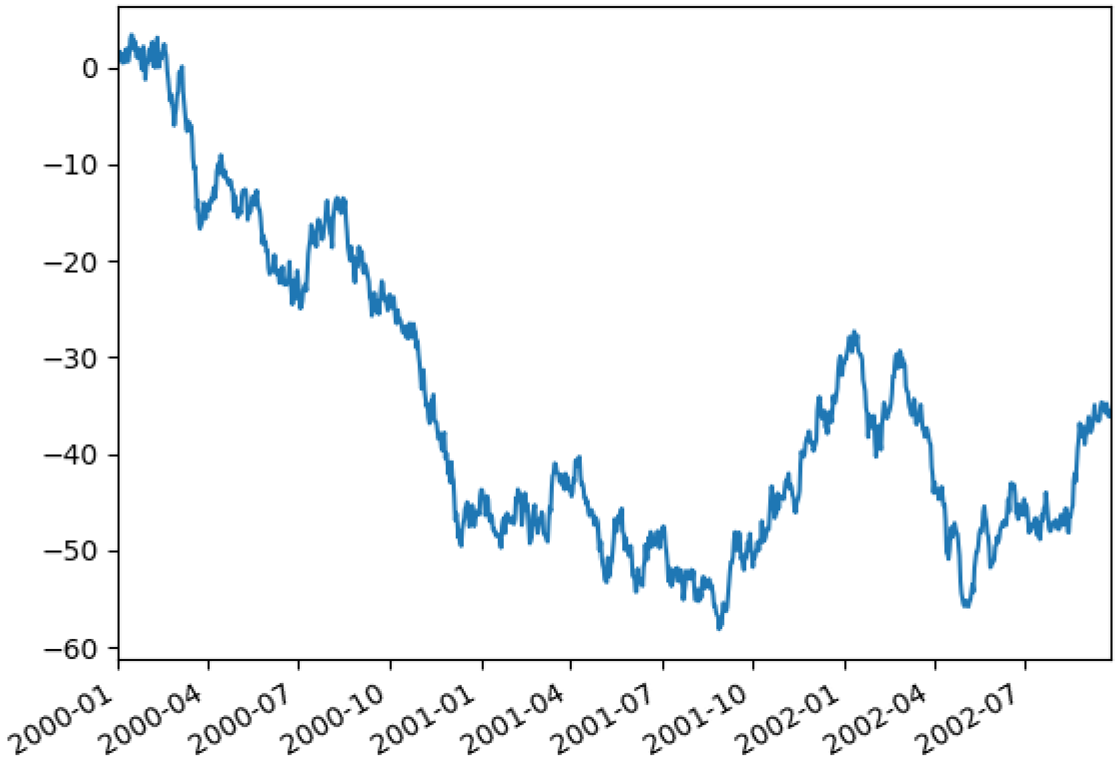
Here is the default behavior, notice how the x-axis tick labeling is performed:

```
In [129]: plt.figure()
Out[129]: <Figure size 640x480 with 0 Axes>

In [130]: df.A.plot()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[130]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e4d497b8>
```



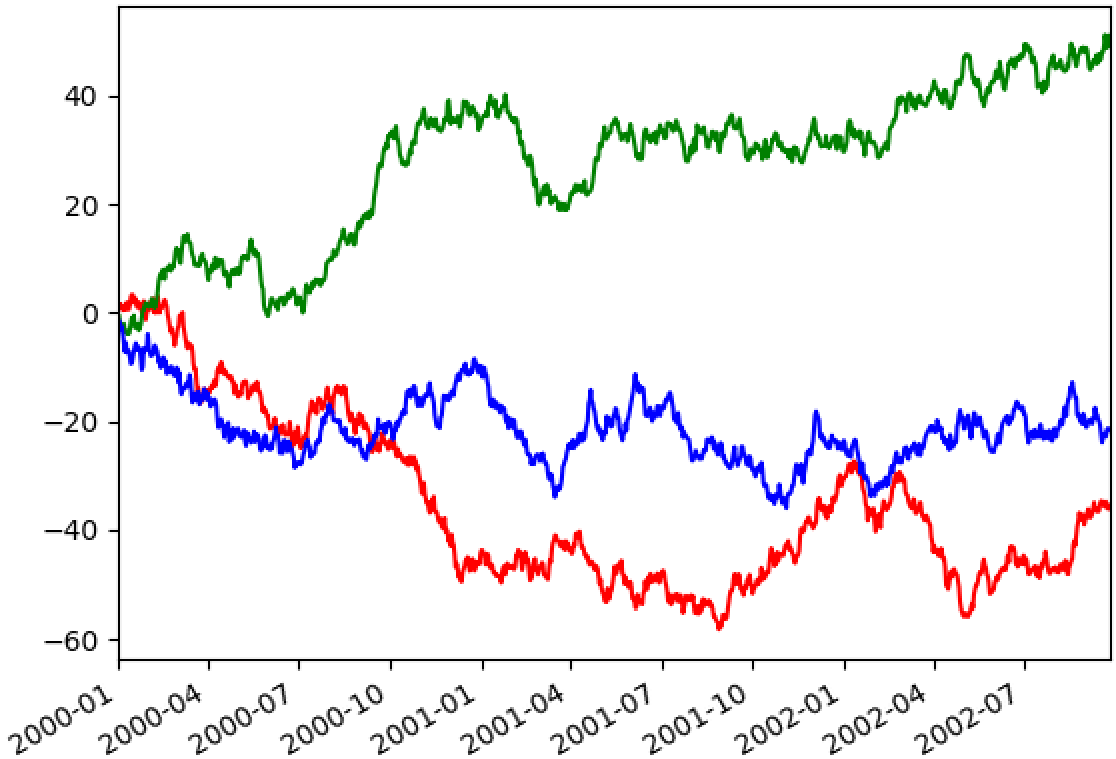
Using the `x_compat` parameter, you can suppress this behavior:

```
In [131]: plt.figure()
Out[131]: <Figure size 640x480 with 0 Axes>

In [132]: df.A.plot(x_compat=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[132]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37e4b42470>
```



If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plotting.plot_params` can be used in a *with statement*:

```
In [133]: plt.figure()
Out[133]: <Figure size 640x480 with 0 Axes>

In [134]: with pd.plotting.plot_params.use('x_compat', True):
   .....:     df.A.plot(color='r')
   .....:     df.B.plot(color='g')
   .....:     df.C.plot(color='b')
   .....:
```



Automatic Date Tick Adjustment

New in version 0.20.0.

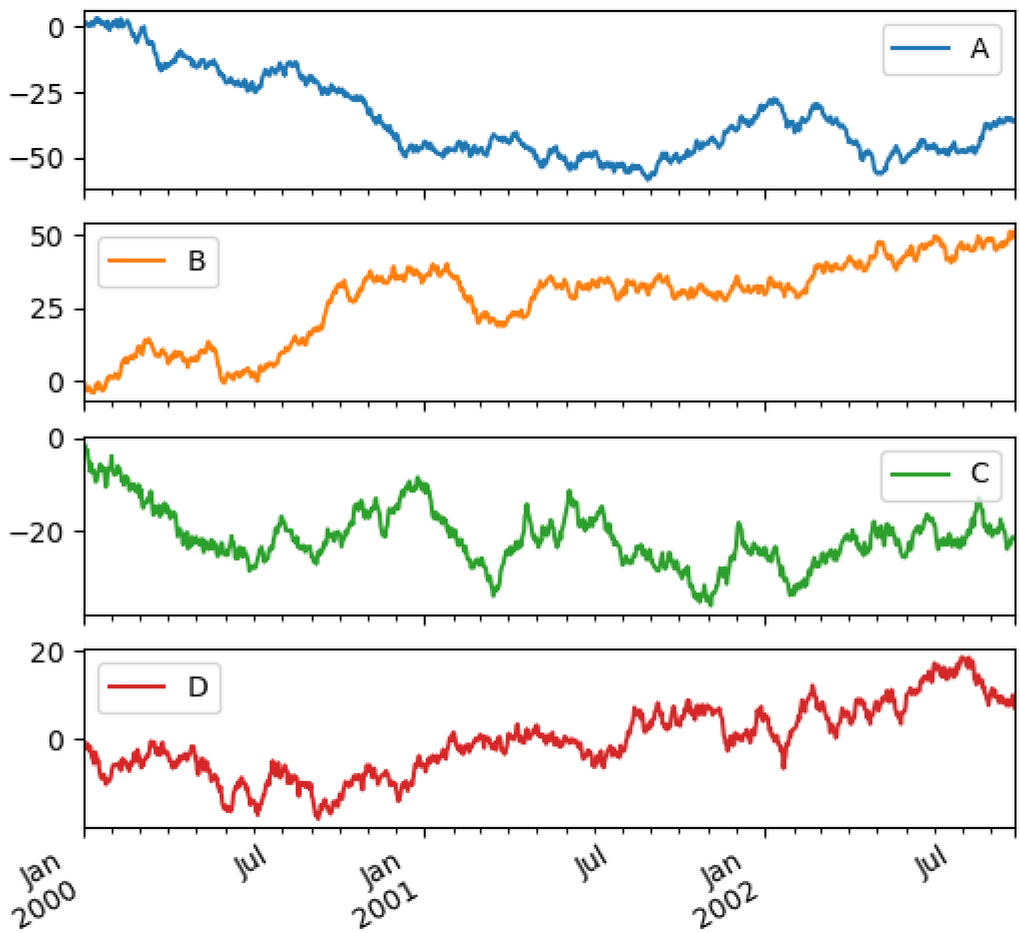
`TimedeltaIndex` now uses the native matplotlib tick locator methods, it is useful to call the automatic date tick adjustment from matplotlib for figures whose ticklabels overlap.

See the `autofmt_xdate` method and the matplotlib documentation for more.

Subplots

Each `Series` in a `DataFrame` can be plotted on a different axis with the `subplots` keyword:

```
In [135]: df.plot(subplots=True, figsize=(6, 6));
```

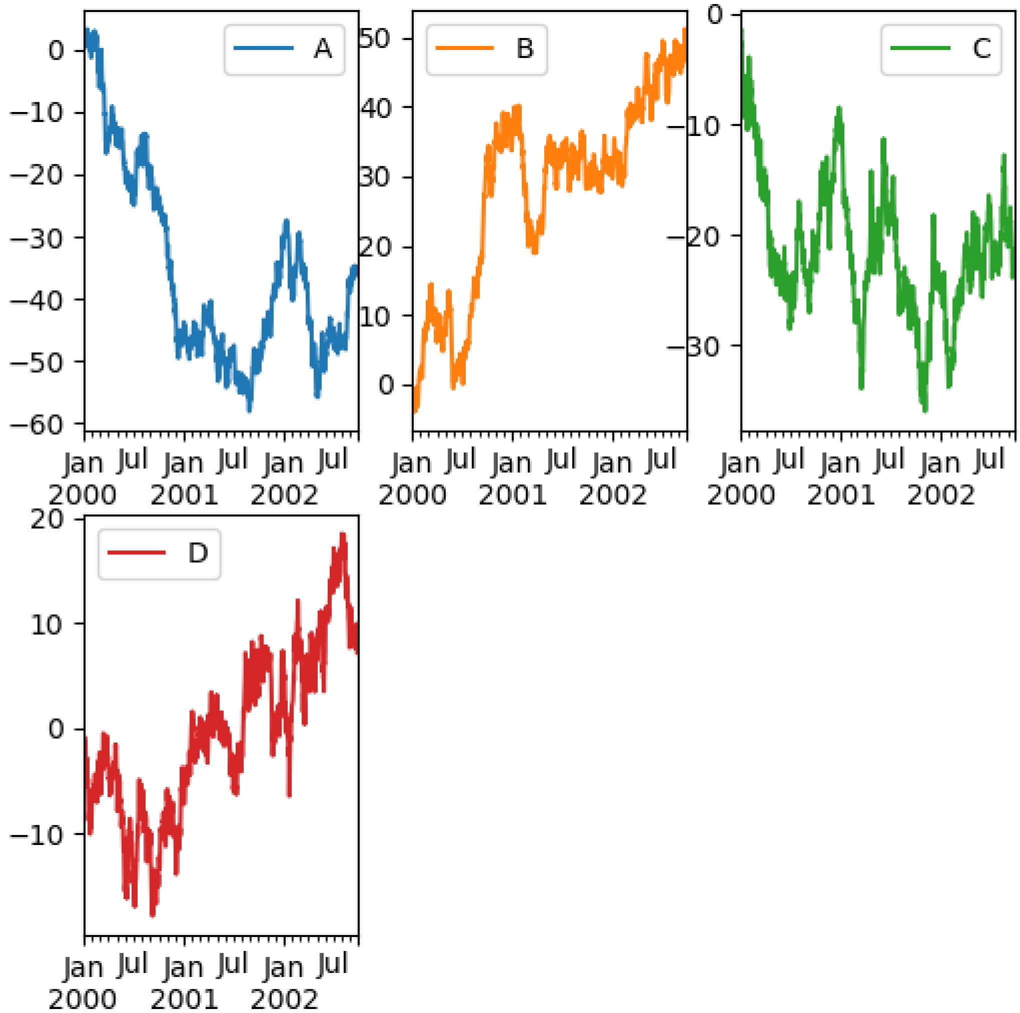


Using Layout and Targeting Multiple Axes

The layout of subplots can be specified by the `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If the input is invalid, a `ValueError` will be raised.

The number of axes which can be contained by rows x columns specified by `layout` must be larger than the number of required subplots. If layout can contain more axes than required, blank axes are not drawn. Similar to a NumPy array's `reshape` method, you can use `-1` for one dimension to automatically calculate the number of rows or columns needed, given the other.

```
In [136]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```



The above example is identical to using:

```
In [137]: df.plot(subplots=True, layout=(2, -1), figsize=(6, 6), sharex=False);
```

The required number of columns (3) is inferred from the number of series to plot and the given number of rows (2).

You can pass multiple axes created beforehand as list-like via `ax` keyword. This allows more complicated layouts. The passed axes must be the same number as the subplots being drawn.

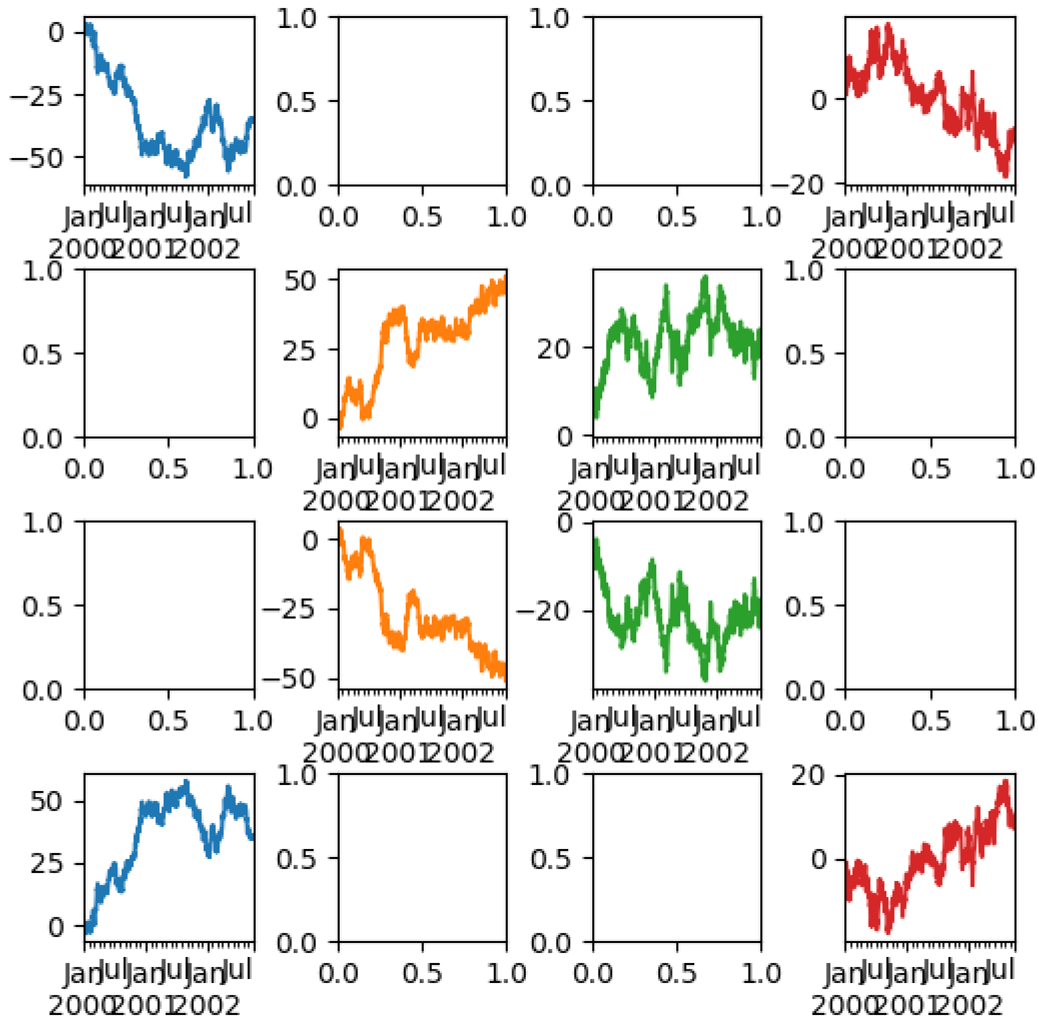
When multiple axes are passed via the `ax` keyword, `layout`, `sharex` and `sharey` keywords don't affect to the output. You should explicitly pass `sharex=False` and `sharey=False`, otherwise you will see a warning.

```
In [138]: fig, axes = plt.subplots(4, 4, figsize=(6, 6))

In [139]: plt.subplots_adjust(wspace=0.5, hspace=0.5)

In [140]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
```

(continues on next page)

(continued from previous page)

```
In [141]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]

In [142]: df.plot(subplots=True, ax=target1, legend=False, sharex=False,␣
↪sharey=False);

In [143]: (-df).plot(subplots=True, ax=target2, legend=False,
   .....:            sharex=False, sharey=False);
   .....:
```



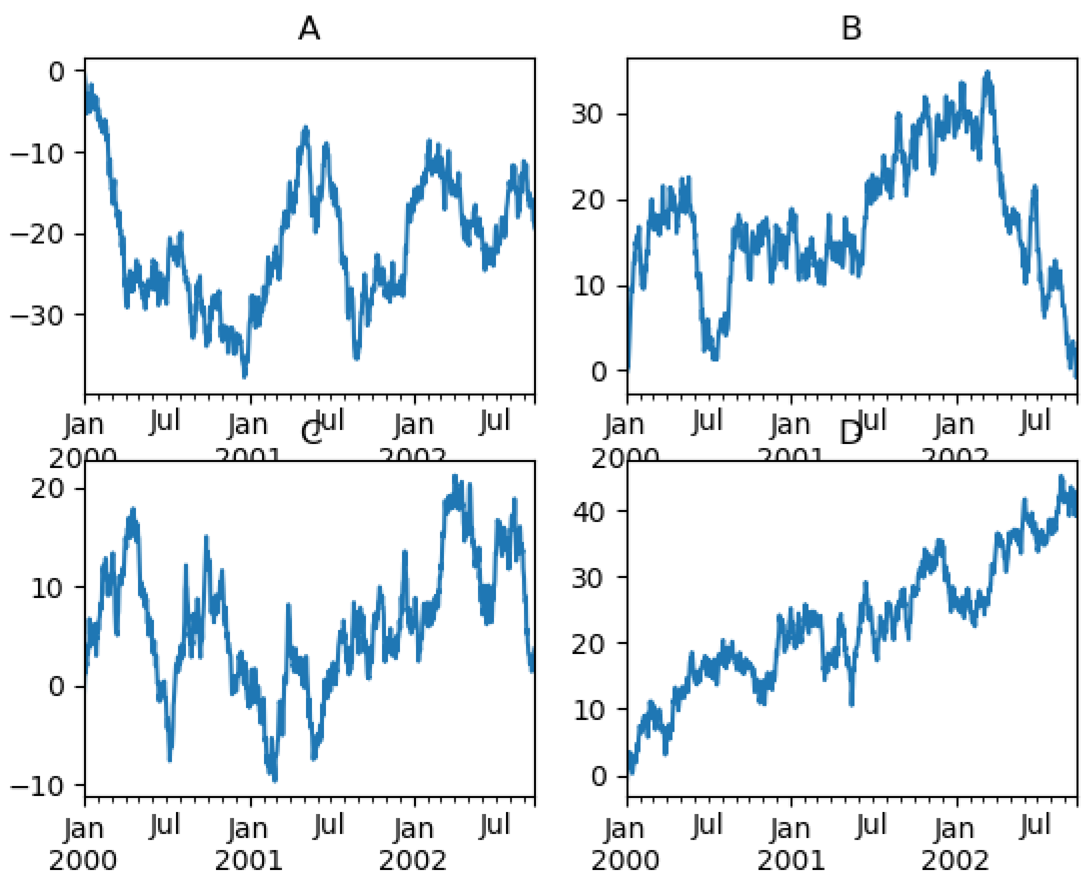
Another option is passing an `ax` argument to *Series.plot()* to plot on a particular axis:

```
In [144]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [145]: df['A'].plot(ax=axes[0, 0]);
```

(continues on next page)

(continued from previous page)

```
In [146]: axes[0, 0].set_title('A');

In [147]: df['B'].plot(ax=axes[0, 1]);

In [148]: axes[0, 1].set_title('B');

In [149]: df['C'].plot(ax=axes[1, 0]);

In [150]: axes[1, 0].set_title('C');

In [151]: df['D'].plot(ax=axes[1, 1]);

In [152]: axes[1, 1].set_title('D');
```



Plotting With Error Bars

Plotting with error bars is supported in *DataFrame.plot()* and *Series.plot()*.

Horizontal and vertical error bars can be supplied to the `xerr` and `yerr` keyword arguments to *plot()*. The error values can be specified using a variety of formats:

- As a *DataFrame* or `dict` of errors with column names matching the `columns` attribute of the plotting

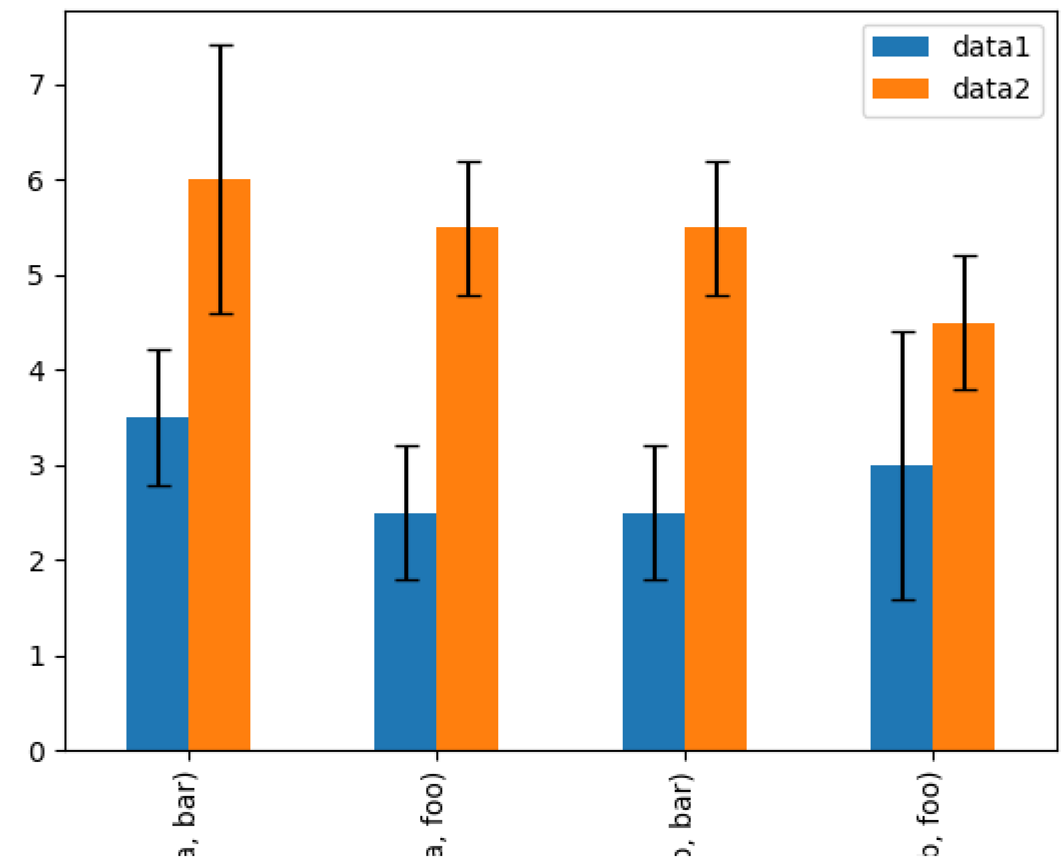
DataFrame or matching the name attribute of the *Series*.

- As a str indicating which of the columns of plotting *DataFrame* contain the error values.
- As raw values (list, tuple, or np.ndarray). Must be the same length as the plotting *DataFrame*/*Series*.

Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a M length *Series*, a Mx2 array should be provided indicating lower and upper (or left and right) errors. For a MxN *DataFrame*, asymmetrical errors should be in a Mx2xN array.

Here is an example of one way to easily plot group means with standard deviations from the raw data.

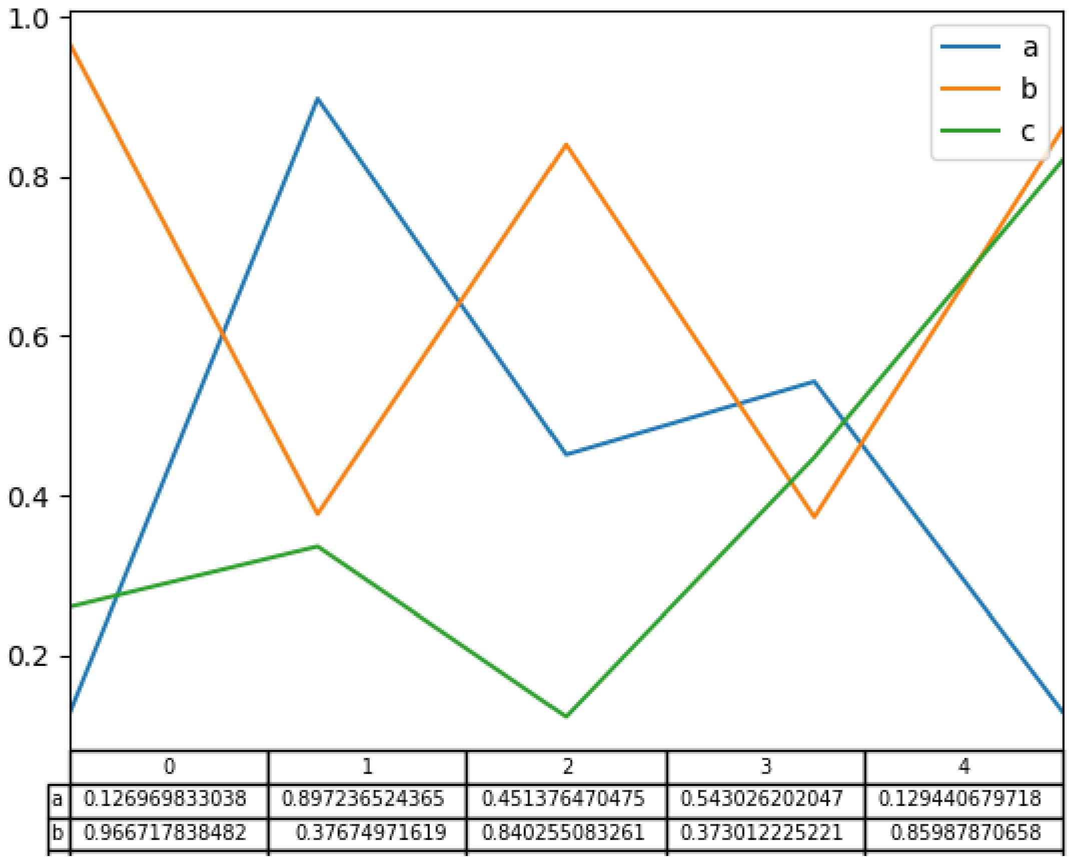
```
# Generate the data
In [153]: ix3 = pd.MultiIndex.from_arrays([
   .....:     ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
   .....:     ['foo', 'foo', 'bar', 'bar', 'foo', 'foo', 'bar', 'bar']],
   .....:     names=['letter', 'word'])
   .....:

In [154]: df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2],
   .....:                     'data2': [6, 5, 7, 5, 4, 5, 6, 5]}, index=ix3)
   .....:

# Group by index labels and take the means and standard deviations
# for each group
In [155]: gp3 = df3.groupby(level=('letter', 'word'))

In [156]: means = gp3.mean()

In [157]: errors = gp3.std()

In [158]: means
Out[158]:
             data1  data2
letter word
a      bar     3.5    6.0
       foo     2.5    5.5
b      bar     2.5    5.5
       foo     3.0    4.5

In [159]: errors
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                data1     data2
letter word
a      bar   0.707107  1.414214
       foo   0.707107  0.707107
b      bar   0.707107  0.707107
       foo   1.414214  0.707107

# Plot
In [160]: fig, ax = plt.subplots()

In [161]: means.plot.bar(yerr=errors, ax=ax, capsize=4)
Out[161]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37ddc4eba8>
```



Plotting Tables

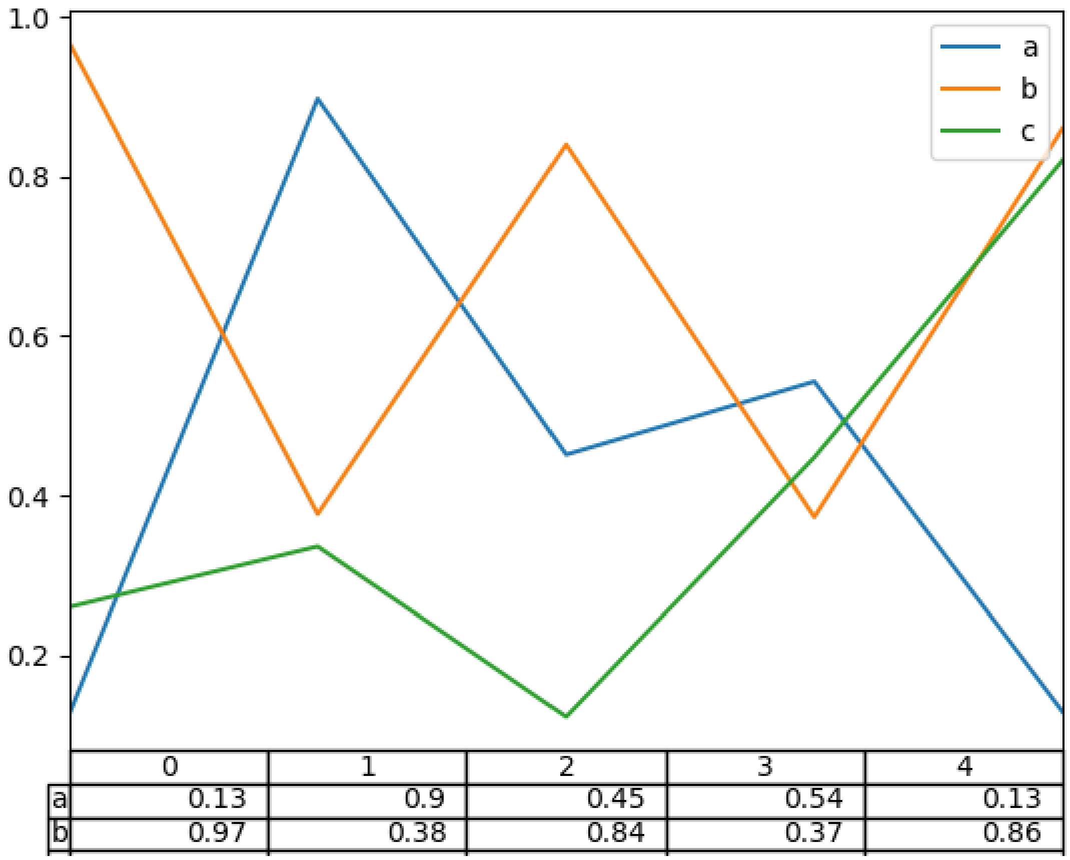
Plotting with matplotlib table is now supported in *DataFrame.plot()* and *Series.plot()* with a `table` keyword. The `table` keyword can accept `bool`, *DataFrame* or *Series*. The simple way to draw a table is to specify `table=True`. Data will be transposed to meet matplotlib's default layout.

```
In [162]: fig, ax = plt.subplots(1, 1)

In [163]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

In [164]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [165]: df.plot(table=True, ax=ax)
Out[165]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37ddbec828>
```



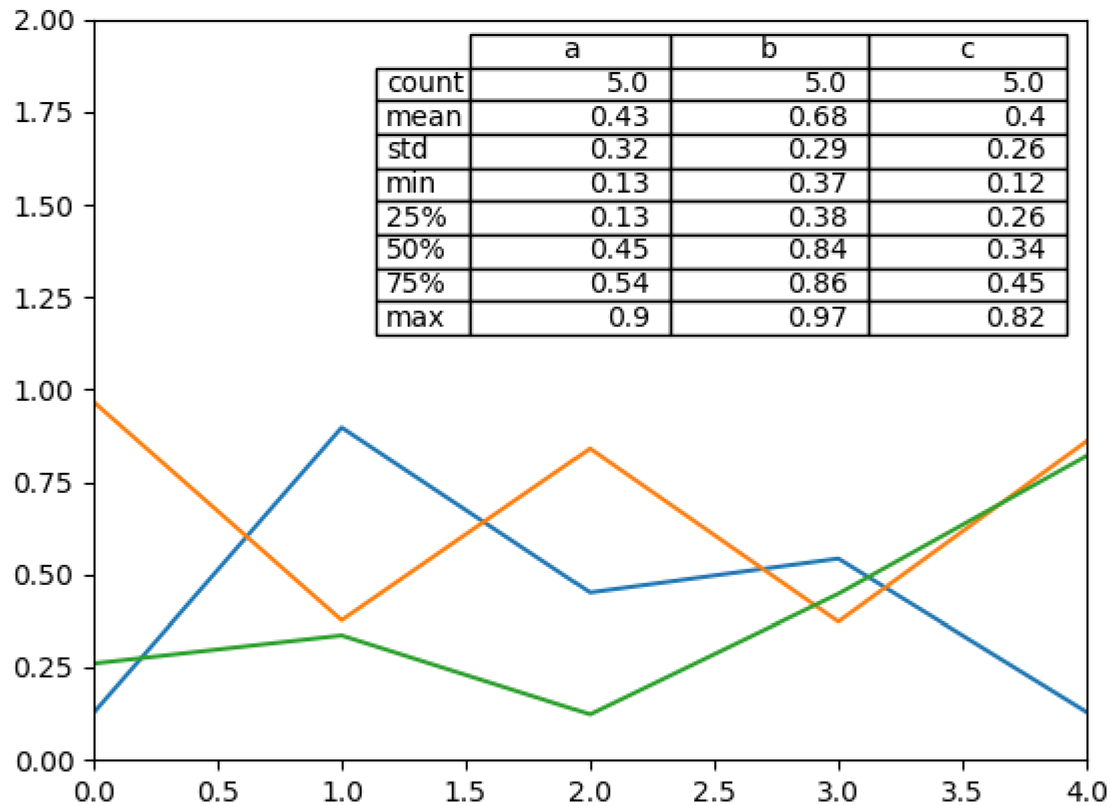
Also, you can pass a different *DataFrame* or *Series* to the `table` keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as seen in the example below.

```
In [166]: fig, ax = plt.subplots(1, 1)

In [167]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [168]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[168]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37ddb6e748>
```



There also exists a helper function `pandas.plotting.table`, which creates a table from `DataFrame` or `Series`, and adds it to an `matplotlib.Axes` instance. This function can accept keywords which the matplotlib table has.

```
In [169]: from pandas.plotting import table

In [170]: fig, ax = plt.subplots(1, 1)

In [171]: table(ax, np.round(df.describe(), 2),
   .....:       loc='upper right', colWidths=[0.2, 0.2, 0.2])
   .....:
Out[171]: <matplotlib.table.Table at 0x7f37ddab17b8>

In [172]: df.plot(ax=ax, ylim=(0, 2), legend=None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[172]: <matplotlib.axes._
↪subplots.AxesSubplot at 0x7f37ddaf9d68>
```



Note: You can get table instances on the axes using `axes.tables` property for further decorations. See the matplotlib table documentation for more.

Colormaps

A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, `DataFrame` plotting supports the use of the `colormap` argument, which accepts either a Matplotlib colormap or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available here.

As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the `DataFrame`. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

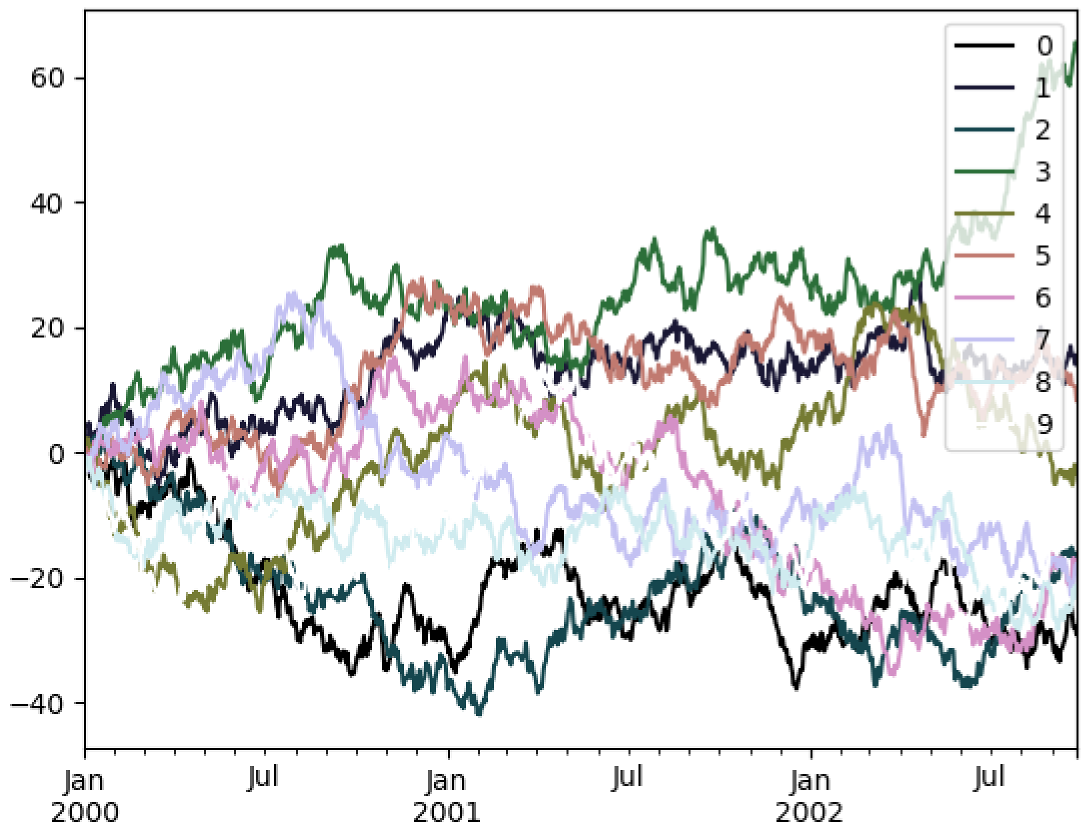
To use the cubehelix colormap, we can pass `colormap='cubehelix'`.

```
In [173]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)

In [174]: df = df.cumsum()

In [175]: plt.figure()
Out[175]: <Figure size 640x480 with 0 Axes>
```

(continues on next page)

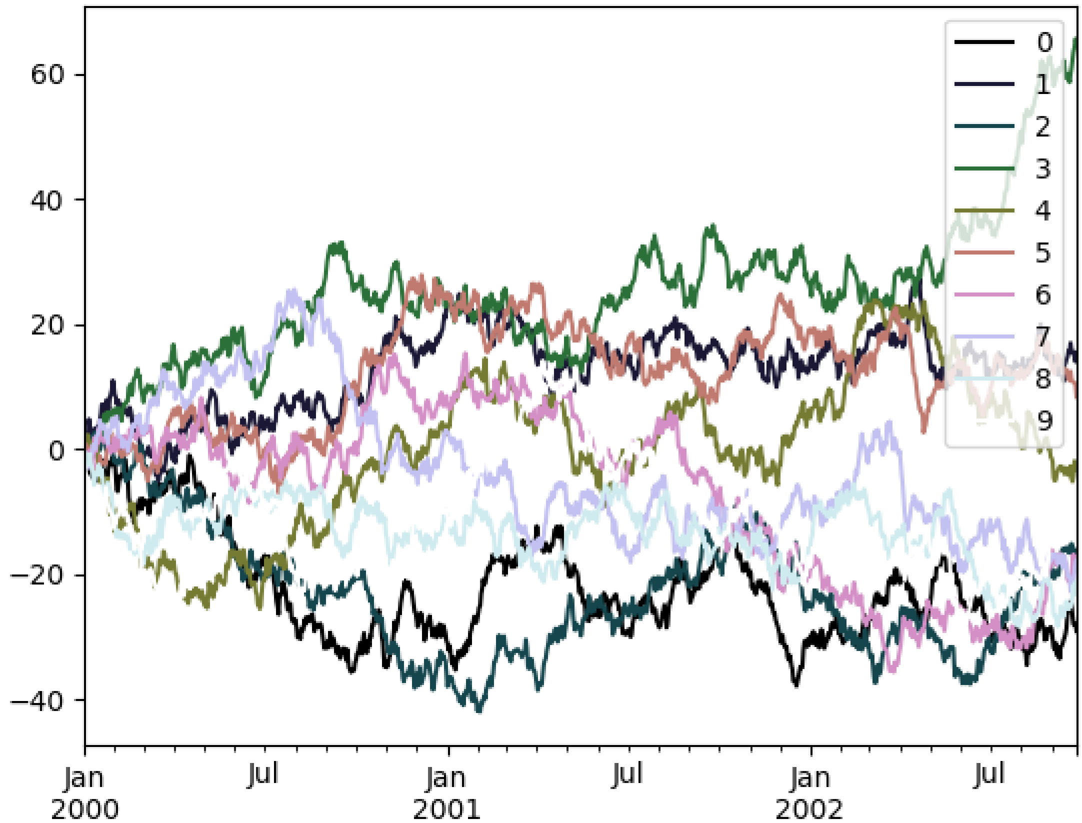
(continued from previous page)

```
In [176]: df.plot(colormap='cubehelix')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[176]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37dda37358>
```



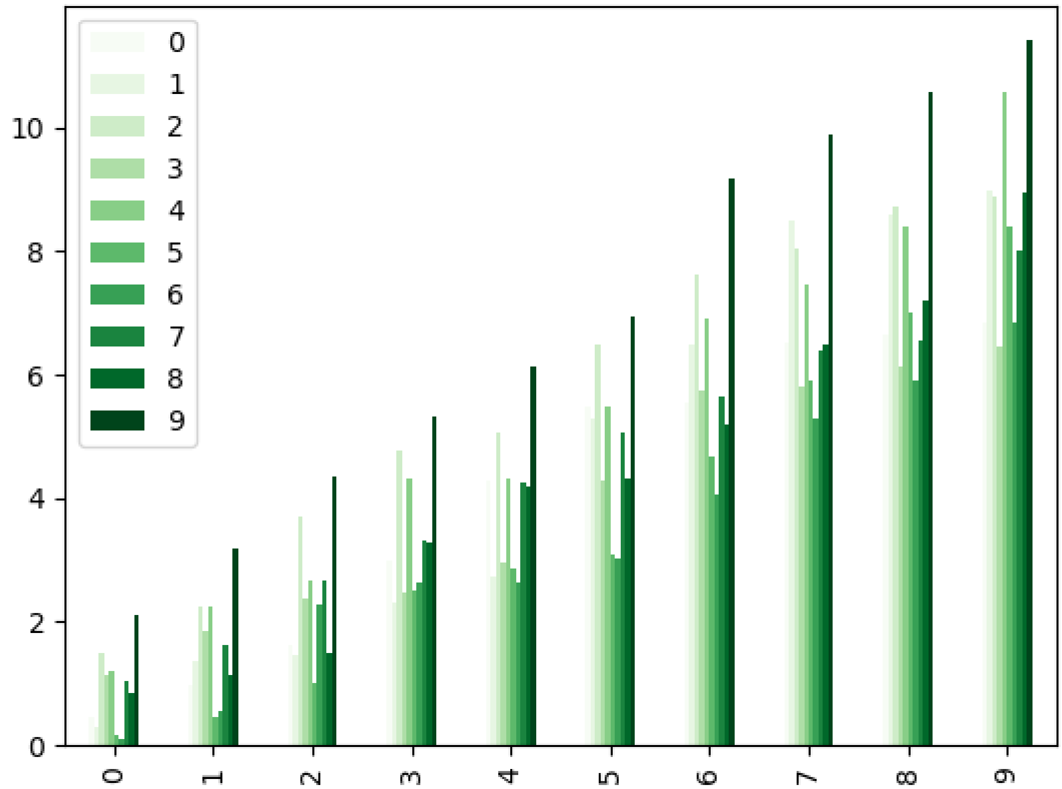
Alternatively, we can pass the colormap itself:

```
In [177]: from matplotlib import cm

In [178]: plt.figure()
Out[178]: <Figure size 640x480 with 0 Axes>

In [179]: df.plot(colormap=cm.cubehelix)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[179]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37dd844fd0>
```



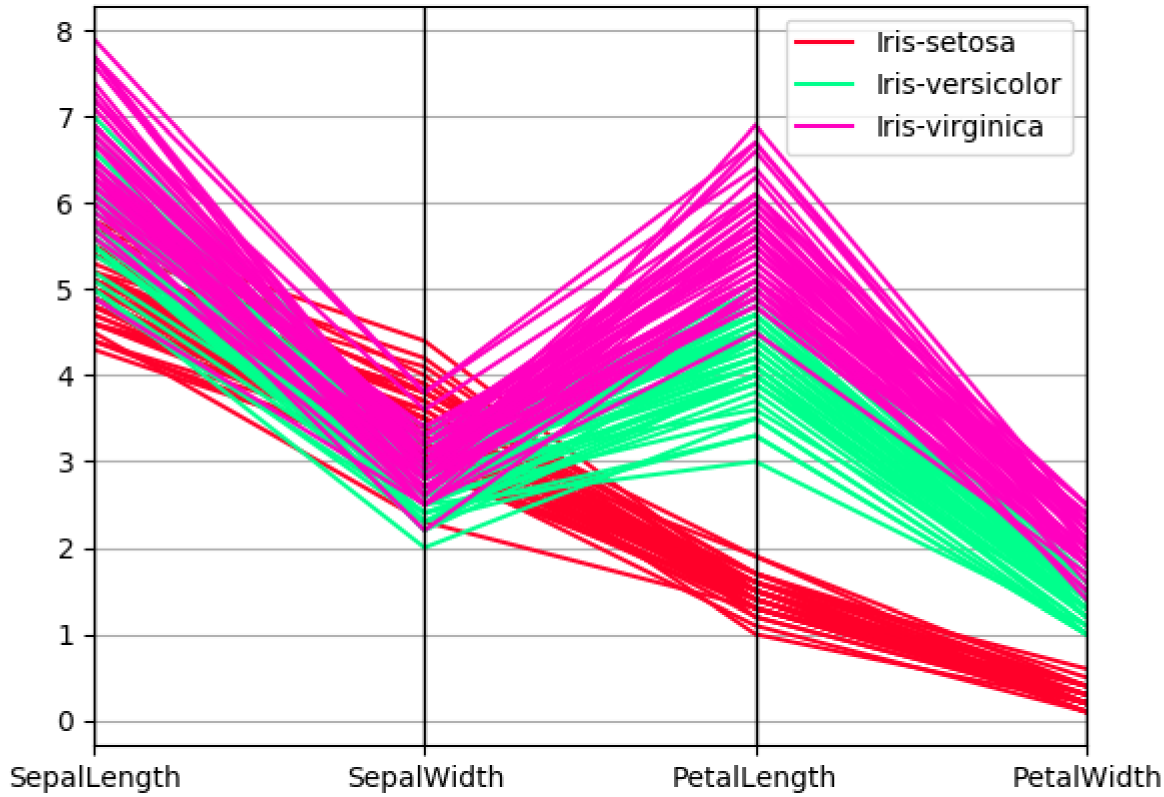
Colormaps can also be used other plot types, like bar charts:

```
In [180]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)

In [181]: dd = dd.cumsum()

In [182]: plt.figure()
Out[182]: <Figure size 640x480 with 0 Axes>

In [183]: dd.plot.bar(colormap='Greens')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[183]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37dd681198>
```



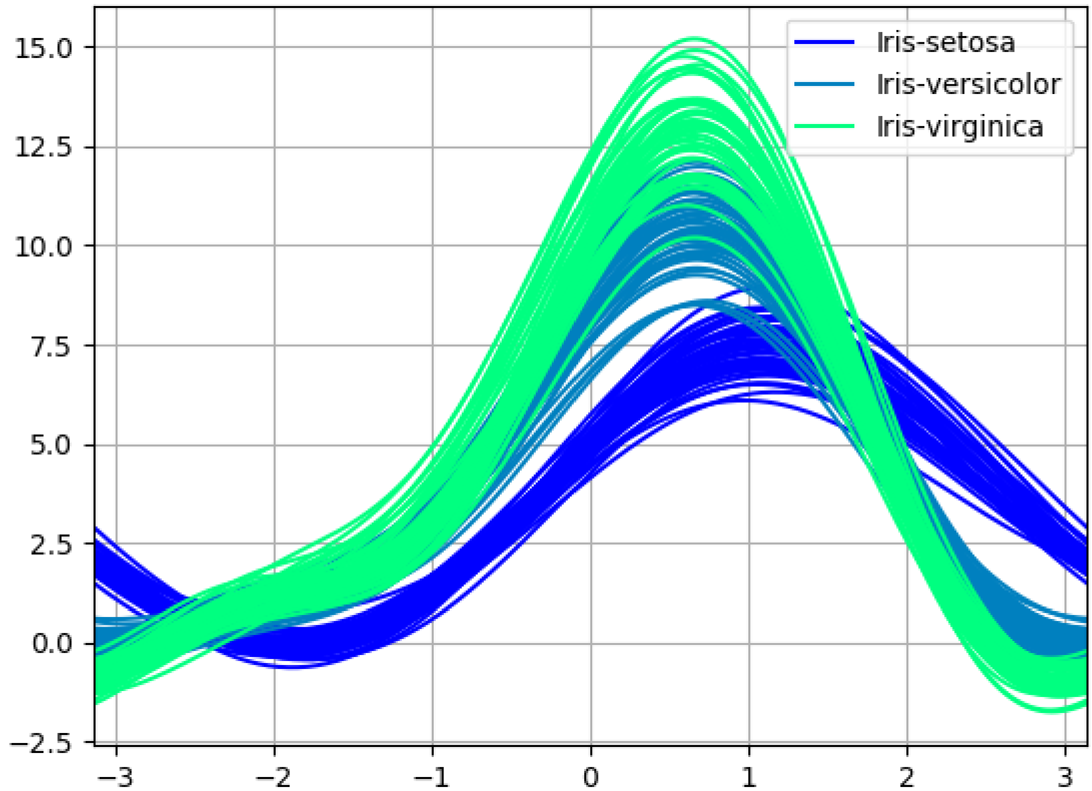
Parallel coordinates charts:

```
In [184]: plt.figure()
Out[184]: <Figure size 640x480 with 0 Axes>

In [185]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[185]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37dd55f8d0>
```



Andrews curves charts:

```
In [186]: plt.figure()
Out[186]: <Figure size 640x480 with 0 Axes>

In [187]: andrews_curves(data, 'Name', colormap='winter')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[187]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37dd3b3d68>
```



4.10.6 Plotting directly with matplotlib

In some situations it may still be preferable or necessary to prepare plots directly with matplotlib, for instance when a certain type of plot or customization is not (yet) supported by pandas. `Series` and `DataFrame` objects behave like arrays and can therefore be passed directly to matplotlib functions without explicit casts.

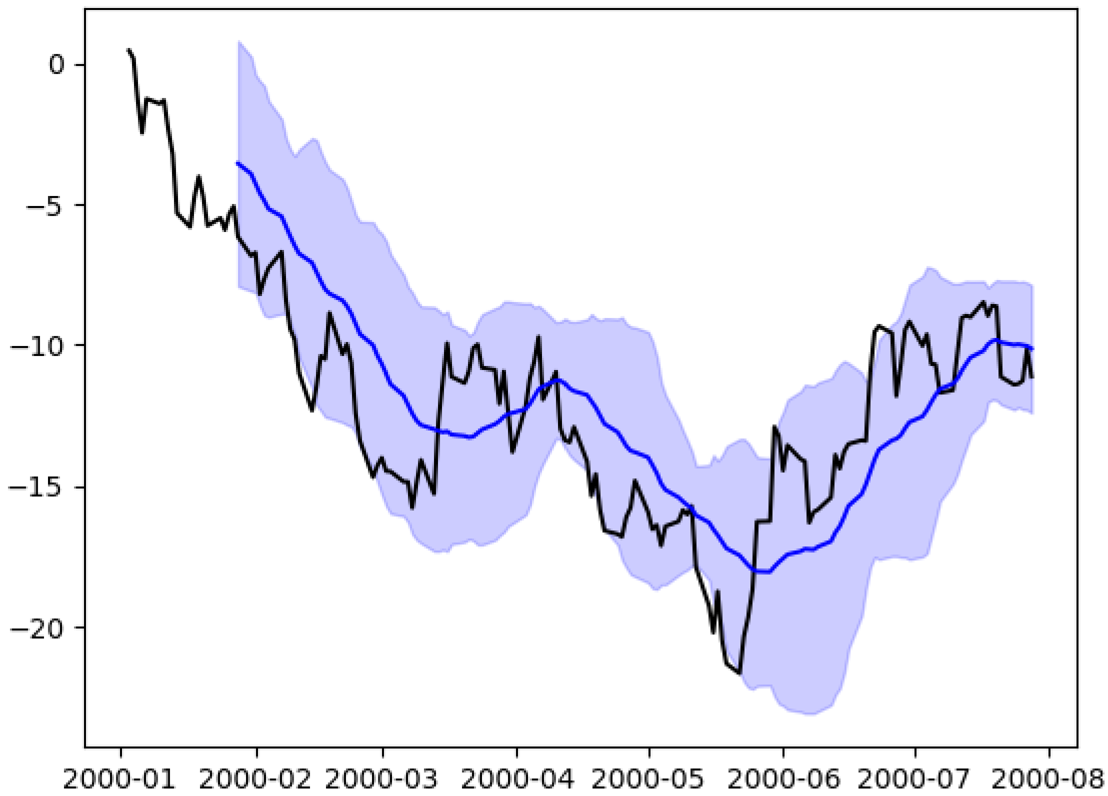
pandas also automatically registers formatters and locators that recognize date indices, thereby extending date and time support to practically all plot types available in matplotlib. Although this formatting does not provide the same level of refinement you would get when plotting via pandas, it can be faster when plotting a large number of points.

```
In [188]: price = pd.Series(np.random.randn(150).cumsum(),
   .....:                   index=pd.date_range('2000-1-1', periods=150, freq='B'))
   .....:

In [189]: ma = price.rolling(20).mean()

In [190]: mstd = price.rolling(20).std()

In [191]: plt.figure()
Out[191]: <Figure size 640x480 with 0 Axes>

In [192]: plt.plot(price.index, price, 'k')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[192]: [<matplotlib.lines.Line2D at␣
↪0x7f37e4e709e8>]
```

(continues on next page)

(continued from previous page)

```
In [193]: plt.plot(ma.index, ma, 'b')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[<matplotlib.lines.Line2D at 0x7f381cdb91d0>]

In [194]: plt.fill_between(mstd.index, ma - 2 * mstd, ma + 2 * mstd,
   .....:                  color='b', alpha=0.2)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.collections.PolyCollection at 0x7f37e5746710>
```



4.10.7 Trellis plotting interface

> **Warning:** The `rplot` trellis plotting interface has been **removed**. Please use external packages like seaborn for similar but more refined functionality and refer to our 0.18.1 documentation here for how to convert to using it.

4.11 Computational tools

4.11.1 Statistical Functions

Percent Change

`Series`, `DataFrame`, and `Panel` all have a method *pct_change()* to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values *before* computing the percent change).

```
In [1]: ser = pd.Series(np.random.randn(8))

In [2]: ser.pct_change()
Out[2]:
0         NaN
1   -1.602976
2    4.334938
3   -0.247456
4   -2.067345
5   -1.142903
6   -1.688214
7   -9.759729
dtype: float64
```

```
In [3]: df = pd.DataFrame(np.random.randn(10, 4))

In [4]: df.pct_change(periods=3)
Out[4]:
          0         1         2         3
0       NaN       NaN       NaN       NaN
1       NaN       NaN       NaN       NaN
2       NaN       NaN       NaN       NaN
3 -0.218320 -1.054001  1.987147 -0.510183
4 -0.439121 -1.816454  0.649715 -4.822809
5 -0.127833 -3.042065 -5.866604 -1.776977
6 -2.596833 -1.959538 -2.111697 -3.798900
7 -0.117826 -2.169058  0.036094 -0.067696
8  2.492606 -1.357320 -1.205802 -1.558697
9 -1.012977  2.324558 -1.003744 -0.371806
```

Covariance

Series.cov() can be used to compute covariance between series (excluding missing values).

```
In [5]: s1 = pd.Series(np.random.randn(1000))

In [6]: s2 = pd.Series(np.random.randn(1000))

In [7]: s1.cov(s2)
Out[7]: 0.000680108817431082
```

Analogously, *DataFrame.cov()* to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

Note: Assuming the missing data are missing at random this results in an estimate for the covariance matrix which is unbiased. However, for many applications this estimate may not be acceptable because the estimated covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimated correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
In [8]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ...:                      columns=['a', 'b', 'c', 'd', 'e'])
   ...:

In [9]: frame.cov()
Out[9]:
          a         b         c         d         e
a  1.000882 -0.003177 -0.002698 -0.006889  0.031912
b -0.003177  1.024721  0.000191  0.009212  0.000857
c -0.002698  0.000191  0.950735 -0.031743 -0.005087
d -0.006889  0.009212 -0.031743  1.002983 -0.047952
e  0.031912  0.000857 -0.005087 -0.047952  1.042487
```

`DataFrame.cov` also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```
In [10]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [11]: frame.loc[frame.index[:5], 'a'] = np.nan

In [12]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [13]: frame.cov()
Out[13]:
          a         b         c
a  1.123670 -0.412851  0.018169
b -0.412851  1.154141  0.305260
c  0.018169  0.305260  1.301149

In [14]: frame.cov(min_periods=12)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b         c
a  1.123670       NaN  0.018169
b       NaN  1.154141  0.305260
c  0.018169  0.305260  1.301149
```

Correlation

Correlation may be computed using the *corr()* method. Using the `method` parameter, several methods for computing correlations are provided:

Method name	Description
`pearson (default)`	Standard correlation coefficient
`kendall`	Kendall Tau correlation coefficient
`spearman`	Spearman rank correlation coefficient

All of these are currently computed using pairwise complete observations. Wikipedia has articles covering the above correlation coefficients:

- Pearson correlation coefficient
- Kendall rank correlation coefficient
- Spearman's rank correlation coefficient

Note: Please see the *caveats* associated with this method of calculating correlation matrices in the *covariance section*.

```
In [15]: frame = pd.DataFrame(np.random.randn(1000, 5),
   ....:                      columns=['a', 'b', 'c', 'd', 'e'])
   ....:

In [16]: frame.iloc[::2] = np.nan

# Series with Series
In [17]: frame['a'].corr(frame['b'])
Out[17]: 0.013479040400098794

In [18]: frame['a'].corr(frame['b'], method='spearman')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]: -0.007289885159540637l

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:
          a         b         c         d         e
a  1.000000  0.013479 -0.049269 -0.042239 -0.028525
b  0.013479  1.000000 -0.020433 -0.011139  0.005654
c -0.049269 -0.020433  1.000000  0.018587 -0.054269
d -0.042239 -0.011139  0.018587  1.000000 -0.017060
e -0.028525  0.005654 -0.054269 -0.017060  1.000000
```

Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```
In [20]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [21]: frame.loc[frame.index[:5], 'a'] = np.nan

In [22]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [23]: frame.corr()
Out[23]:
          a         b         c
a  1.000000 -0.121111  0.069544
b -0.121111  1.000000  0.051742
c  0.069544  0.051742  1.000000

In [24]: frame.corr(min_periods=12)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          a         b         c
a  1.000000       NaN  0.069544
b       NaN  1.000000  0.051742
c  0.069544  0.051742  1.000000
```

New in version 0.24.0.

The `method` argument can also be a callable for a generic correlation calculation. In this case, it should be a single function that produces a single value from two ndarray inputs. Suppose we wanted to compute the correlation based on histogram intersection:

```
# histogram intersection
In [25]: def histogram_intersection(a, b):
   ....:     return np.minimum(np.true_divide(a, a.sum()),
   ....:                       np.true_divide(b, b.sum())).sum()
   ....:

In [26]: frame.corr(method=histogram_intersection)
Out[26]:
          a          b          c
a  1.000000  -6.404882  -2.058431
b -6.404882   1.000000 -19.255743
c -2.058431 -19.255743   1.000000
```

A related method *corrwith()* is implemented on DataFrame to compute the correlation between like-labeled Series contained in different DataFrame objects.

```
In [27]: index = ['a', 'b', 'c', 'd', 'e']

In [28]: columns = ['one', 'two', 'three', 'four']

In [29]: df1 = pd.DataFrame(np.random.randn(5, 4), index=index, columns=columns)

In [30]: df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4], columns=columns)

In [31]: df1.corrwith(df2)
Out[31]:
one     -0.125501
two     -0.493244
three    0.344056
four     0.004183
dtype: float64

In [32]: df2.corrwith(df1, axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪
a   -0.675817
b    0.458296
c    0.190809
d   -0.186275
e         NaN
dtype: float64
```

Data ranking

The *rank()* method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```
In [33]: s = pd.Series(np.random.np.random.randn(5), index=list('abcde'))

In [34]: s['d'] = s['b']  # so there's a tie

In [35]: s.rank()
```

(continues on next page)

(continued from previous page)

```
Out[35]:
a    5.0
b    2.5
c    1.0
d    2.5
e    4.0
dtype: float64
```

`rank()` is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). `NaN` values are excluded from the ranking.

```
In [36]: df = pd.DataFrame(np.random.np.random.randn(10, 6))

In [37]: df[4] = df[2][:5]  # some ties

In [38]: df
Out[38]:
          0         1         2         3         4         5
0 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.204848       NaN  0.463347
6  0.376892  0.959292  0.095572 -0.593740       NaN -0.069180
7 -1.002601  1.957794 -0.120708  0.094214       NaN -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254       NaN -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477       NaN  1.375064

In [39]: df.rank(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     0    1    2    3    4    5
0  4.0  3.0  1.5  5.0  1.5  6.0
1  2.0  6.0  4.5  1.0  4.5  3.0
2  1.0  6.0  3.5  5.0  3.5  2.0
3  4.0  5.0  1.5  3.0  1.5  6.0
4  5.0  3.0  1.5  4.0  1.5  6.0
5  1.0  2.0  5.0  3.0  NaN  4.0
6  4.0  5.0  3.0  1.0  NaN  2.0
7  2.0  5.0  3.0  4.0  NaN  1.0
8  2.0  5.0  3.0  4.0  NaN  1.0
9  2.0  3.0  1.0  4.0  NaN  5.0
```

`rank` optionally takes a parameter `ascending` which by default is true; when false, data is reverse-ranked, with larger values assigned a smaller rank.

`rank` supports different tie-breaking methods, specified with the `method` parameter:

- `average` : average rank of tied group
- `min` : lowest rank in the group
- `max` : highest rank in the group
- `first` : ranks assigned in the order they appear in the array

4.11.2 Window Functions

For working with data, a number of window functions are provided for computing common *window* or *rolling* statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis.

The `rolling()` and `expanding()` functions can be used directly from DataFrameGroupBy objects, see the *groupby docs*.

Note: The API for window statistics is quite similar to the way one works with `GroupBy` objects, see the documentation *here*.

We work with `rolling`, `expanding` and `exponentially weighted` data through the corresponding objects, `Rolling`, `Expanding` and `EWM`.

```
In [40]: s = pd.Series(np.random.randn(1000),
   ....:               index=pd.date_range('1/1/2000', periods=1000))
   ....:

In [41]: s = s.cumsum()

In [42]: s
Out[42]:
2000-01-01    -0.268824
2000-01-02    -1.771855
2000-01-03    -0.818003
2000-01-04    -0.659244
2000-01-05    -1.942133
2000-01-06    -1.869391
2000-01-07     0.563674
                 ...
2002-09-20   -68.233054
2002-09-21   -66.765687
2002-09-22   -67.457323
2002-09-23   -69.253182
2002-09-24   -70.296818
2002-09-25   -70.844674
2002-09-26   -72.475016
Freq: D, Length: 1000, dtype: float64
```

These are created from methods on `Series` and `DataFrame`.

```
In [43]: r = s.rolling(window=60)

In [44]: r
Out[44]: Rolling [window=60,center=False,axis=0]
```

These object provide tab-completion of the available methods and properties.

```
In [14]: r.<TAB>                                          # noqa: E225, E999
r.agg           r.apply         r.count         r.exclusions    r.max           r.median        r.
↪name           r.skew          r.sum
r.aggregate     r.corr          r.cov           r.kurt          r.mean          r.min           r.
↪quantile       r.std           r.var
```

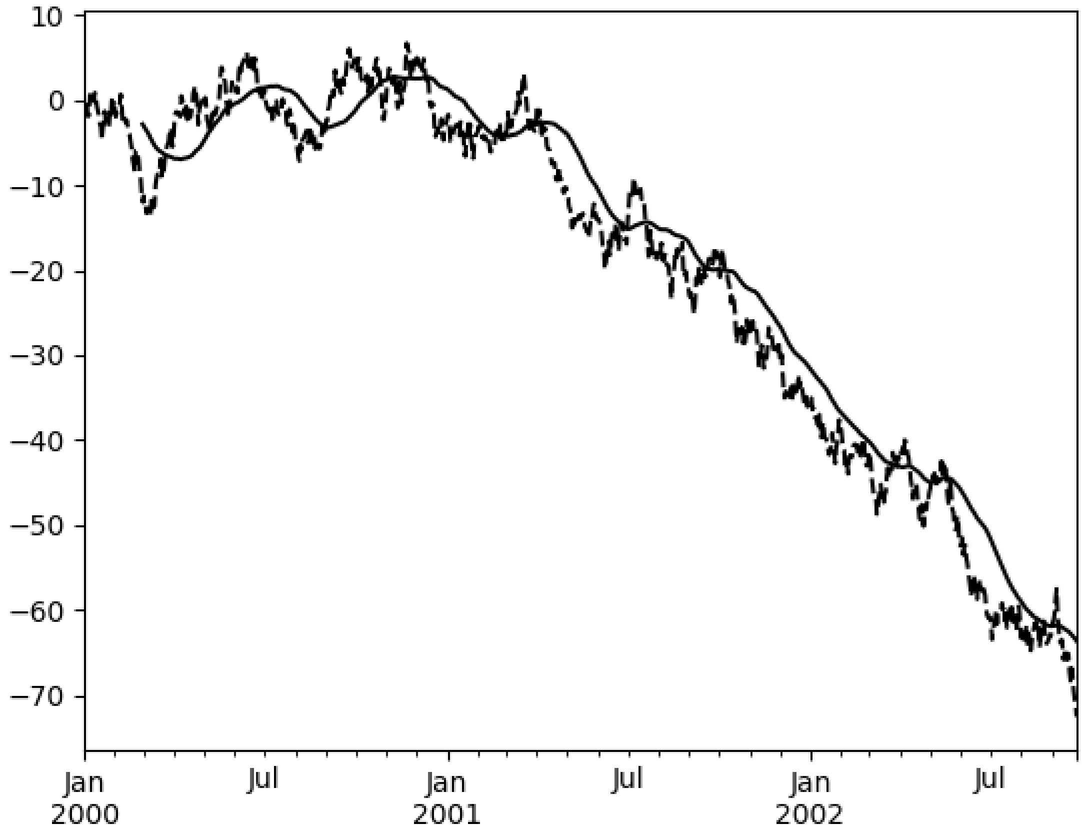
Generally these methods all have the same interface. They all accept the following arguments:

- `window`: size of moving window
- `min_periods`: threshold of non-null data points to require (otherwise result is NA)
- `center`: boolean, whether to set the labels at the center (default is False)

We can then call methods on these `rolling` objects. These return like-indexed objects:

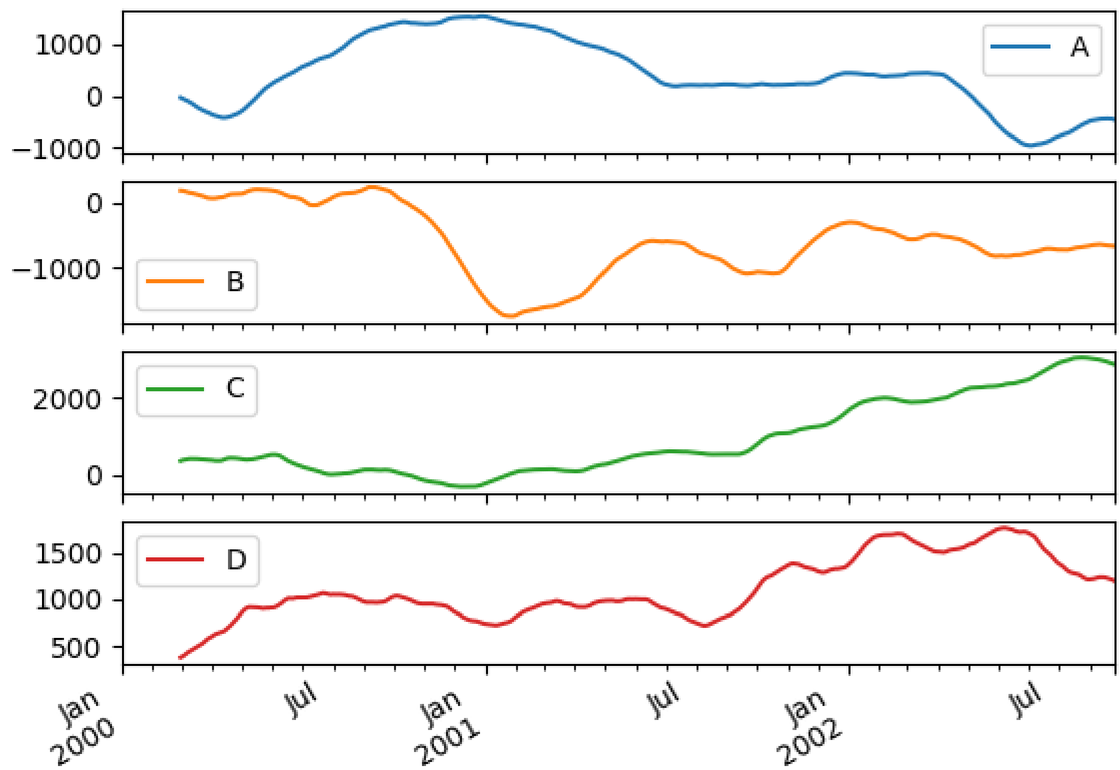
```
In [45]: r.mean()
Out[45]:
2000-01-01          NaN
2000-01-02          NaN
2000-01-03          NaN
2000-01-04          NaN
2000-01-05          NaN
2000-01-06          NaN
2000-01-07          NaN
                ...
2002-09-20   -62.694135
2002-09-21   -62.812190
2002-09-22   -62.914971
2002-09-23   -63.061867
2002-09-24   -63.213876
2002-09-25   -63.375074
2002-09-26   -63.539734
Freq: D, Length: 1000, dtype: float64
```

```
In [46]: s.plot(style='k--')
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381ee87b70>

In [47]: r.mean().plot(style='k')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]:
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f381ee87b70>
```



They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrame's columns:

```
In [48]: df = pd.DataFrame(np.random.randn(1000, 4),
   ....:                   index=pd.date_range('1/1/2000', periods=1000),
   ....:                   columns=['A', 'B', 'C', 'D'])
   ....:

In [49]: df = df.cumsum()

In [50]: df.rolling(window=60).sum().plot(subplots=True)
Out[50]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f381c540b70>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f381c557dd8>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f381c4f3ef0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f381c523048>],␣
↪dtype=object)
```



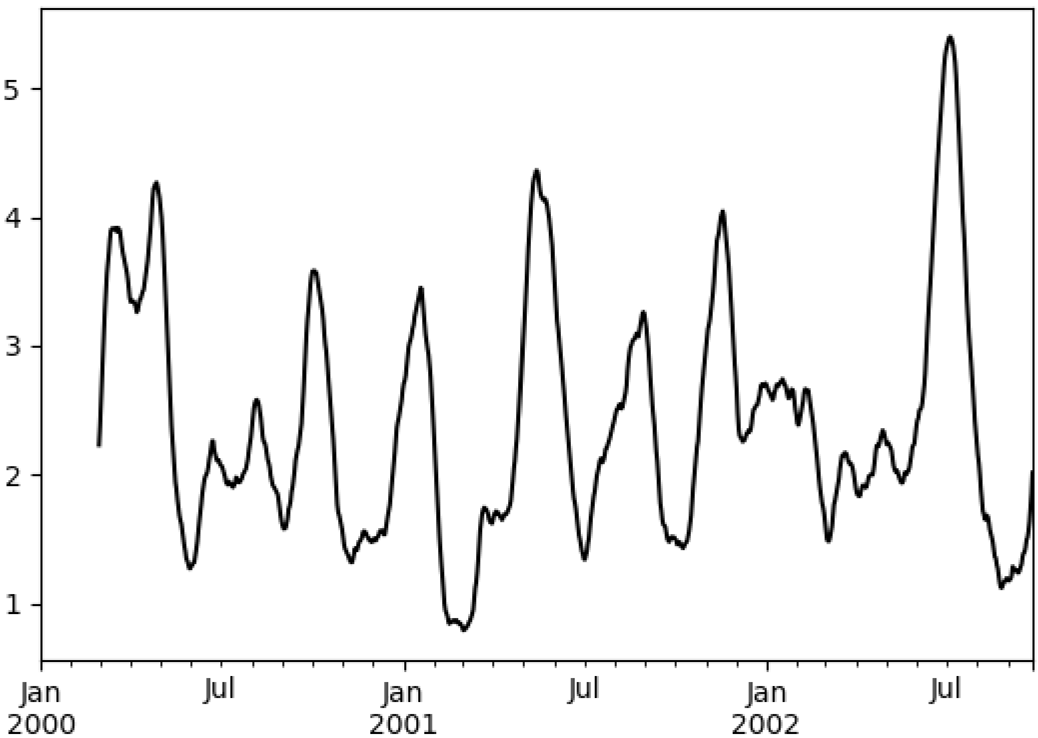
Method Summary

We provide a number of common statistical functions:

Method	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
std()	Bessel-corrected sample standard deviation
var()	Unbiased variance
skew()	Sample skewness (3rd moment)
kurt()	Sample kurtosis (4th moment)
quantile()	Sample quantile (value at %)
apply()	Generic apply
cov()	Unbiased covariance (binary)
corr()	Correlation (binary)

The *apply()* function takes an extra `func` argument and performs generic rolling computations. The `func` argument should be a single function that produces a single value from an ndarray input. Suppose we wanted to compute the mean absolute deviation on a rolling basis:

```
In [51]: def mad(x):
   ....:     return np.fabs(x - x.mean()).mean()
   ....:

In [52]: s.rolling(window=60).apply(mad, raw=True).plot(style='k')
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381c347710>
```



Rolling Windows

Passing `win_type` to `.rolling` generates a generic rolling window computation, that is weighted according the `win_type`. The following methods are available:

Method	Description
sum()	Sum of values
mean()	Mean of values

The weights used in the window are specified by the `win_type` keyword. The list of recognized types are the scipy.signal window functions:

- `boxcar`

- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

```
In [53]: ser = pd.Series(np.random.randn(10),
   ....:                 index=pd.date_range('1/1/2000', periods=10))
   ....:

In [54]: ser.rolling(window=5, win_type='triang').mean()
Out[54]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -1.037870
2000-01-06   -0.767705
2000-01-07   -0.383197
2000-01-08   -0.395513
2000-01-09   -0.558440
2000-01-10   -0.672416
Freq: D, dtype: float64
```

Note that the `boxcar` window is equivalent to *mean()*.

```
In [55]: ser.rolling(window=5, win_type='boxcar').mean()
Out[55]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64

In [56]: ser.rolling(window=5).mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [57]: ser.rolling(window=5, win_type='gaussian').mean(std=0.1)
Out[57]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -1.309989
2000-01-06   -1.153000
2000-01-07    0.606382
2000-01-08   -0.681101
2000-01-09   -0.289724
2000-01-10   -0.996632
Freq: D, dtype: float64
```

Note: For `.sum()` with a `win_type`, there is no normalization done to the weights for the window. Passing custom weights of `[1, 1, 1]` will yield a different result than passing weights of `[2, 2, 2]`, for example. When passing a `win_type` instead of explicitly specifying the weights, the weights are already normalized so that the largest weight is 1.

In contrast, the nature of the `.mean()` calculation is such that the weights are normalized with respect to each other. Weights of `[1, 1, 1]` and `[2, 2, 2]` yield the same result.

Time-aware Rolling

New in version 0.19.0.

New in version 0.19.0 are the ability to pass an offset (or convertible) to a `.rolling()` method and have it produce variable sized windows based on the passed time window. For each time point, this includes all preceding values occurring within the indicated time delta.

This can be particularly useful for a non-regular time frequency index.

```
In [58]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.date_range('20130101 09:00:00',
   ....:                                        periods=5,
   ....:                                        freq='s'))
   ....:

In [59]: dft
Out[59]:
```

(continues on next page)

(continued from previous page)

```
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [60]: dft.rolling(2).sum()
Out[60]:
                       B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN

In [61]: dft.rolling(2, min_periods=1).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [62]: dft.rolling('2s').sum()
Out[62]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0
```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```
In [63]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.Index([pd.Timestamp('20130101 09:00:00'),
   ....:                                    pd.Timestamp('20130101 09:00:02'),
   ....:                                    pd.Timestamp('20130101 09:00:03'),
   ....:                                    pd.Timestamp('20130101 09:00:05'),
   ....:                                    pd.Timestamp('20130101 09:00:06')],
   ....:                                   name='foo'))
   ....:

In [64]: dft
Out[64]:
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
```

(continues on next page)

(continued from previous page)

```
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

In [65]: dft.rolling(2).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                       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
```

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

```
In [66]: dft.rolling('2s').sum()
Out[66]:
                       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
```

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

```
In [67]: dft = dft.reset_index()

In [68]: dft
Out[68]:
                  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

In [69]: dft.rolling('2s', on='foo').sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                  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
```

Rolling Window Endpoints

New in version 0.20.0.

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

closed	Description	Default for
right	close right endpoint	time-based windows
left	close left endpoint	
both	close both endpoints	fixed windows
neither	open endpoints	

For example, having the right endpoint open is useful in many problems that require that there is no contamination from present information back to past information. This allows the rolling window to compute statistics "up to that point in time", but not including that point in time.

```
In [70]: df = pd.DataFrame({'x': 1},
   ....:                   index=[pd.Timestamp('20130101 09:00:01'),
   ....:                          pd.Timestamp('20130101 09:00:02'),
   ....:                          pd.Timestamp('20130101 09:00:03'),
   ....:                          pd.Timestamp('20130101 09:00:04'),
   ....:                          pd.Timestamp('20130101 09:00:06')])
   ....:

In [71]: df["right"] = df.rolling('2s', closed='right').x.sum()  # default

In [72]: df["both"] = df.rolling('2s', closed='both').x.sum()

In [73]: df["left"] = df.rolling('2s', closed='left').x.sum()

In [74]: df["neither"] = df.rolling('2s', closed='neither').x.sum()

In [75]: df
Out[75]:
                     x  right  both  left  neither
2013-01-01 09:00:01  1    1.0   1.0   NaN      NaN
2013-01-01 09:00:02  1    2.0   2.0   1.0      1.0
2013-01-01 09:00:03  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:04  1    2.0   3.0   2.0      1.0
2013-01-01 09:00:06  1    1.0   2.0   1.0      NaN
```

Currently, this feature is only implemented for time-based windows. For fixed windows, the closed parameter cannot be set and the rolling window will always have both endpoints closed.

Time-aware Rolling vs. Resampling

Using `.rolling()` with a time-based index is quite similar to *resampling*. They both operate and perform reductive operations on time-indexed pandas objects.

When using `.rolling()` with an offset. The offset is a time-delta. Take a backwards-in-time looking window, and aggregate all of the values in that window (including the end-point, but not the start-point). This is the new value at that point in the result. These are variable sized windows in time-space for each point of the input. You will get a same sized result as the input.

When using `.resample()` with an offset. Construct a new index that is the frequency of the offset. For each frequency bin, aggregate points from the input within a backwards-in-time looking window that fall in that bin. The result of this aggregation is the output for that frequency point. The windows are fixed size in the frequency space. Your result will have the shape of a regular frequency between the min and the max of the original input object.

To summarize, `.rolling()` is a time-based window operation, while `.resample()` is a frequency-based window operation.

Centering Windows

By default the labels are set to the right edge of the window, but a `center` keyword is available so the labels can be set at the center.

```
In [76]: ser.rolling(window=5).mean()
Out[76]:
2000-01-01         NaN
2000-01-02         NaN
2000-01-03         NaN
2000-01-04         NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64

In [77]: ser.rolling(window=5, center=True).mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-01         NaN
2000-01-02         NaN
2000-01-03   -0.841164
2000-01-04   -0.779948
2000-01-05   -0.565487
2000-01-06   -0.502815
2000-01-07   -0.553755
2000-01-08   -0.472211
2000-01-09         NaN
2000-01-10         NaN
Freq: D, dtype: float64
```

Binary Window Functions

cov() and *corr()* can compute moving window statistics about two `Series` or any combination of `DataFrame`/`Series` or `DataFrame`/`DataFrame`. Here is the behavior in each case:

- two `Series`: compute the statistic for the pairing.
- `DataFrame`/`Series`: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame.
- `DataFrame`/`DataFrame`: by default compute the statistic for matching column names, returning a DataFrame. If the keyword argument `pairwise=True` is passed then computes the statistic for each pair of columns, returning a `MultiIndexed DataFrame` whose `index` are the dates in question (see *the next section*).

For example:

```
In [78]: df = pd.DataFrame(np.random.randn(1000, 4),
   ....:                   index=pd.date_range('1/1/2000', periods=1000),
   ....:                   columns=['A', 'B', 'C', 'D'])
   ....:

In [79]: df = df.cumsum()
```

(continues on next page)

(continued from previous page)

```
In [80]: df2 = df[:20]

In [81]: df2.rolling(window=5).corr(df2['B'])
Out[81]:
                   A    B         C         D
2000-01-01       NaN  NaN       NaN       NaN
2000-01-02       NaN  NaN       NaN       NaN
2000-01-03       NaN  NaN       NaN       NaN
2000-01-04       NaN  NaN       NaN       NaN
2000-01-05  0.768775  1.0 -0.977990  0.800252
2000-01-06  0.744106  1.0 -0.967912  0.830021
2000-01-07  0.683257  1.0 -0.928969  0.384916
...              ...  ...       ...       ...
2000-01-14 -0.392318  1.0  0.570240 -0.591056
2000-01-15  0.017217  1.0  0.649900 -0.896258
2000-01-16  0.691078  1.0  0.807450 -0.939302
2000-01-17  0.274506  1.0  0.582601 -0.902954
2000-01-18  0.330459  1.0  0.515707 -0.545268
2000-01-19  0.046756  1.0 -0.104334 -0.419799
2000-01-20 -0.328241  1.0 -0.650974 -0.777777

[20 rows x 4 columns]
```

Computing rolling pairwise covariances and correlations

Warning: Prior to version 0.20.0 if `pairwise=True` was passed, a `Panel` would be returned. This will now return a 2-level MultiIndexed DataFrame, see the whatsnew *here*.

In financial data analysis and other fields it's common to compute covariance and correlation matrices for a collection of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by passing the `pairwise` keyword argument, which in the case of `DataFrame` inputs will yield a MultiIndexed `DataFrame` whose `index` are the dates in question. In the case of a single DataFrame argument the `pairwise` argument can even be omitted:

Note: Missing values are ignored and each entry is computed using the pairwise complete observations. Please see the *covariance section* for *caveats* associated with this method of calculating covariance and correlation matrices.

```
In [82]: covs = (df[['B', 'C', 'D']].rolling(window=50)
   ....:                              .cov(df[['A', 'B', 'C']], pairwise=True))
   ....:

In [83]: covs.loc['2002-09-22':]
Out[83]:
                     B         C         D
2002-09-22 A  1.367467  8.676734 -8.047366
           B  3.067315  0.865946 -1.052533
           C  0.865946  7.739761 -4.943924
2002-09-23 A  0.910343  8.669065 -8.443062
           B  2.625456  0.565152 -0.907654
           C  0.565152  7.825521 -5.367526
```

(continues on next page)

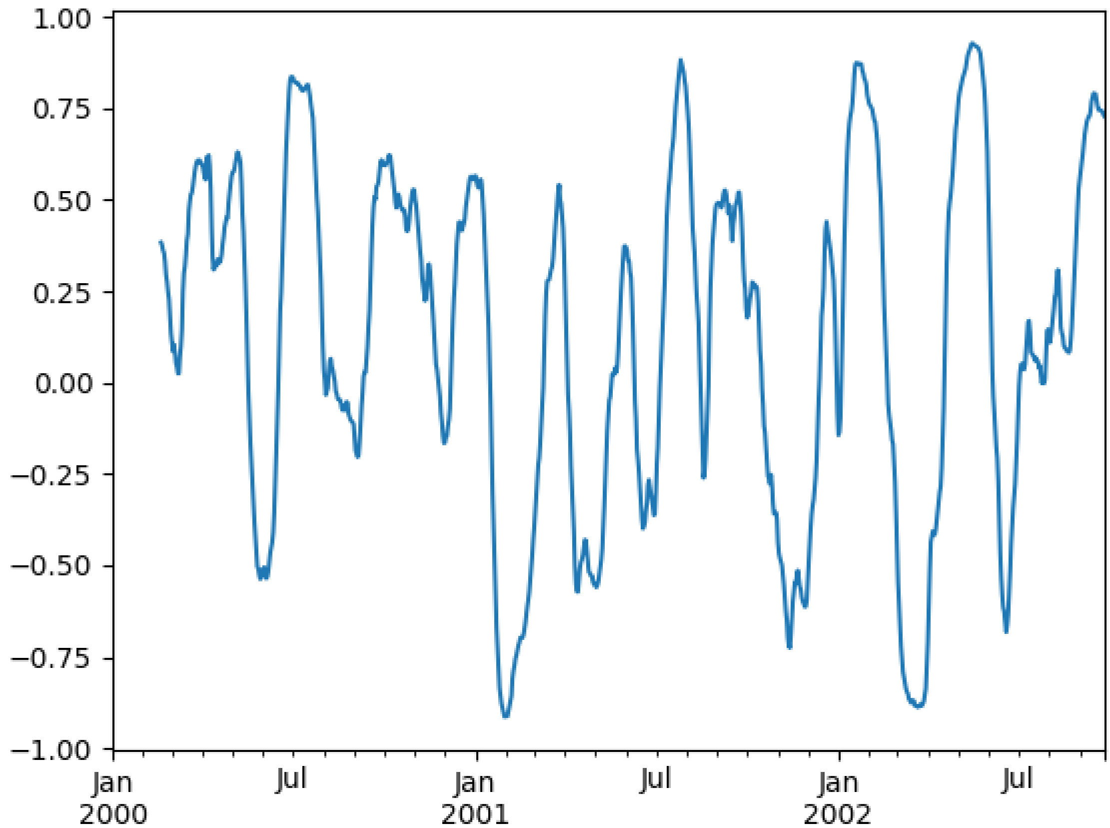
(continued from previous page)

```
2002-09-24 A  0.463332  8.514509 -8.776514
           B  2.306695  0.267746 -0.732186
           C  0.267746  7.771425 -5.696962
2002-09-25 A  0.467976  8.198236 -9.162599
           B  2.307129  0.267287 -0.754080
           C  0.267287  7.466559 -5.822650
2002-09-26 A  0.545781  7.899084 -9.326238
           B  2.311058  0.322295 -0.844451
           C  0.322295  7.038237 -5.684445
```

```
In [84]: correls = df.rolling(window=50).corr()

In [85]: correls.loc['2002-09-22':]
Out[85]: 
                     A         B         C         D
2002-09-22 A  1.000000  0.186397  0.744551 -0.769767
           B  0.186397  1.000000  0.177725 -0.240802
           C  0.744551  0.177725  1.000000 -0.712051
           D -0.769767 -0.240802 -0.712051  1.000000
2002-09-23 A  1.000000  0.134723  0.743113 -0.758758
           B  0.134723  1.000000  0.124683 -0.209934
           C  0.743113  0.124683  1.000000 -0.719088
...                ...       ...       ...       ...
2002-09-25 B  0.075157  1.000000  0.064399 -0.164179
           C  0.731888  0.064399  1.000000 -0.704686
           D -0.739160 -0.164179 -0.704686  1.000000
2002-09-26 A  1.000000  0.087756  0.727792 -0.736562
           B  0.087756  1.000000  0.079913 -0.179477
           C  0.727792  0.079913  1.000000 -0.692303
           D -0.736562 -0.179477 -0.692303  1.000000

[20 rows x 4 columns]
```

You can efficiently retrieve the time series of correlations between two columns by reshaping and indexing:

```
In [86]: correls.unstack(1)[('A', 'C')].plot()
Out[86]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381c074630>
```



4.11.3 Aggregation

Once the `Rolling`, `Expanding` or `EWM` objects have been created, several methods are available to perform multiple computations on the data. These operations are similar to the *aggregating API*, *groupby API*, and *resample API*.

```
In [87]: dfa = pd.DataFrame(np.random.randn(1000, 3),
   ....:                    index=pd.date_range('1/1/2000', periods=1000),
   ....:                    columns=['A', 'B', 'C'])
   ....:

In [88]: r = dfa.rolling(window=60, min_periods=1)

In [89]: r
Out[89]: Rolling [window=60,min_periods=1,center=False,axis=0]
```

We can aggregate by passing a function to the entire DataFrame, or select a Series (or multiple Series) via standard `__getitem__`.

```
In [90]: r.aggregate(np.sum)
Out[90]:
                   A         B         C
2000-01-01 -0.289838  -0.370545 -1.284206
2000-01-02 -0.216612  -1.675528 -1.169415
```

(continues on next page)

(continued from previous page)

```
2000-01-03  1.154661  -1.634017 -1.566620
2000-01-04  2.969393  -4.003274 -1.816179
2000-01-05  4.690630  -4.682017 -2.717209
2000-01-06  3.880630  -4.447700 -1.078947
2000-01-07  4.001957  -2.884072 -3.116903
...              ...        ...       ...
2002-09-20  2.652493 -10.528875  9.867805
2002-09-21  0.844497  -9.280944  9.522649
2002-09-22  2.860036  -9.270337  6.415245
2002-09-23  3.510163  -8.151439  5.177219
2002-09-24  6.524983 -10.168078  5.792639
2002-09-25  6.409626  -9.956226  5.704050
2002-09-26  5.093787  -7.074515  6.905823

[1000 rows x 3 columns]

In [91]: r['A'].aggregate(np.sum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2000-01-01   -0.289838
2000-01-02   -0.216612
2000-01-03    1.154661
2000-01-04    2.969393
2000-01-05    4.690630
2000-01-06    3.880630
2000-01-07    4.001957
                ...
2002-09-20    2.652493
2002-09-21    0.844497
2002-09-22    2.860036
2002-09-23    3.510163
2002-09-24    6.524983
2002-09-25    6.409626
2002-09-26    5.093787
Freq: D, Name: A, Length: 1000, dtype: float64

In [92]: r[['A', 'B']].aggregate(np.sum)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A          B
2000-01-01 -0.289838  -0.370545
2000-01-02 -0.216612  -1.675528
2000-01-03  1.154661  -1.634017
2000-01-04  2.969393  -4.003274
2000-01-05  4.690630  -4.682017
2000-01-06  3.880630  -4.447700
2000-01-07  4.001957  -2.884072
...              ...        ...
2002-09-20  2.652493 -10.528875
2002-09-21  0.844497  -9.280944
2002-09-22  2.860036  -9.270337
2002-09-23  3.510163  -8.151439
2002-09-24  6.524983 -10.168078
2002-09-25  6.409626  -9.956226
2002-09-26  5.093787  -7.074515

[1000 rows x 2 columns]
```

As you can see, the result of the aggregation will have the selected columns, or all columns if none are selected.

Applying multiple functions

With windowed `Series` you can also pass a list of functions to do aggregation with, outputting a DataFrame:

```
In [93]: r['A'].agg([np.sum, np.mean, np.std])
Out[93]:
                 sum      mean       std
2000-01-01 -0.289838 -0.289838       NaN
2000-01-02 -0.216612 -0.108306  0.256725
2000-01-03  1.154661  0.384887  0.873311
2000-01-04  2.969393  0.742348  1.009734
2000-01-05  4.690630  0.938126  0.977914
2000-01-06  3.880630  0.646772  1.128883
2000-01-07  4.001957  0.571708  1.049487
...              ...       ...       ...
2002-09-20  2.652493  0.044208  1.164919
2002-09-21  0.844497  0.014075  1.148231
2002-09-22  2.860036  0.047667  1.132051
2002-09-23  3.510163  0.058503  1.134296
2002-09-24  6.524983  0.108750  1.144204
2002-09-25  6.409626  0.106827  1.142913
2002-09-26  5.093787  0.084896  1.151416

[1000 rows x 3 columns]
```

On a windowed DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [94]: r.agg([np.sum, np.mean])
Out[94]:
                   A                    B                   C
                 sum      mean        sum      mean       sum      mean
2000-01-01 -0.289838 -0.289838  -0.370545 -0.370545 -1.284206 -1.284206
2000-01-02 -0.216612 -0.108306  -1.675528 -0.837764 -1.169415 -0.584708
2000-01-03  1.154661  0.384887  -1.634017 -0.544672 -1.566620 -0.522207
2000-01-04  2.969393  0.742348  -4.003274 -1.000819 -1.816179 -0.454045
2000-01-05  4.690630  0.938126  -4.682017 -0.936403 -2.717209 -0.543442
2000-01-06  3.880630  0.646772  -4.447700 -0.741283 -1.078947 -0.179825
2000-01-07  4.001957  0.571708  -2.884072 -0.412010 -3.116903 -0.445272
...              ...       ...        ...       ...       ...       ...
2002-09-20  2.652493  0.044208 -10.528875 -0.175481  9.867805  0.164463
2002-09-21  0.844497  0.014075  -9.280944 -0.154682  9.522649  0.158711
2002-09-22  2.860036  0.047667  -9.270337 -0.154506  6.415245  0.106921
2002-09-23  3.510163  0.058503  -8.151439 -0.135857  5.177219  0.086287
2002-09-24  6.524983  0.108750 -10.168078 -0.169468  5.792639  0.096544
2002-09-25  6.409626  0.106827  -9.956226 -0.165937  5.704050  0.095068
2002-09-26  5.093787  0.084896  -7.074515 -0.117909  6.905823  0.115097

[1000 rows x 6 columns]
```

Passing a dict of functions has different behavior by default, see the next section.

Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a `DataFrame`:

```
In [95]: r.agg({'A': np.sum, 'B': lambda x: np.std(x, ddof=1)})
Out[95]:
                   A         B
2000-01-01 -0.289838       NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
2000-01-06  3.880630  0.966848
2000-01-07  4.001957  1.240137
...              ...       ...
2002-09-20  2.652493  1.114814
2002-09-21  0.844497  1.113220
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199

[1000 rows x 2 columns]
```

The function names can also be strings. In order for a string to be valid it must be implemented on the windowed object

```
In [96]: r.agg({'A': 'sum', 'B': 'std'})
Out[96]:
                   A         B
2000-01-01 -0.289838       NaN
2000-01-02 -0.216612  0.660747
2000-01-03  1.154661  0.689929
2000-01-04  2.969393  1.072199
2000-01-05  4.690630  0.939657
2000-01-06  3.880630  0.966848
2000-01-07  4.001957  1.240137
...              ...       ...
2002-09-20  2.652493  1.114814
2002-09-21  0.844497  1.113220
2002-09-22  2.860036  1.113208
2002-09-23  3.510163  1.132381
2002-09-24  6.524983  1.080963
2002-09-25  6.409626  1.082911
2002-09-26  5.093787  1.136199

[1000 rows x 2 columns]
```

Furthermore you can pass a nested dict to indicate different aggregations on different columns.

```
In [97]: r.agg({'A': ['sum', 'std'], 'B': ['mean', 'std']})
Out[97]:
                   A                   B
                 sum       std      mean       std
2000-01-01 -0.289838       NaN -0.370545       NaN
2000-01-02 -0.216612  0.256725 -0.837764  0.660747
2000-01-03  1.154661  0.873311 -0.544672  0.689929
2000-01-04  2.969393  1.009734 -1.000819  1.072199
2000-01-05  4.690630  0.977914 -0.936403  0.939657
2000-01-06  3.880630  1.128883 -0.741283  0.966848
```

(continues on next page)

(continued from previous page)

```
2000-01-07  4.001957  1.049487 -0.412010  1.240137
...              ...       ...       ...       ...
2002-09-20  2.652493  1.164919 -0.175481  1.114814
2002-09-21  0.844497  1.148231 -0.154682  1.113220
2002-09-22  2.860036  1.132051 -0.154506  1.113208
2002-09-23  3.510163  1.134296 -0.135857  1.132381
2002-09-24  6.524983  1.144204 -0.169468  1.080963
2002-09-25  6.409626  1.142913 -0.165937  1.082911
2002-09-26  5.093787  1.151416 -0.117909  1.136199

[1000 rows x 4 columns]
```

4.11.4 Expanding Windows

A common alternative to rolling statistics is to use an *expanding* window, which yields the value of the statistic with all the data available up to that point in time.

These follow a similar interface to `.rolling`, with the `.expanding` method returning an `Expanding` object.

As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```
In [98]: df.rolling(window=len(df), min_periods=1).mean()[:5]
Out[98]:
                   A         B         C         D
2000-01-01  0.314226 -0.001675  0.071823  0.892566
2000-01-02  0.654522 -0.171495  0.179278  0.853361
2000-01-03  0.708733 -0.064489 -0.238271  1.371111
2000-01-04  0.987613  0.163472 -0.919693  1.566485
2000-01-05  1.426971  0.288267 -1.358877  1.808650

In [99]: df.expanding(min_periods=1).mean()[:5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-01  0.314226 -0.001675  0.071823  0.892566
2000-01-02  0.654522 -0.171495  0.179278  0.853361
2000-01-03  0.708733 -0.064489 -0.238271  1.371111
2000-01-04  0.987613  0.163472 -0.919693  1.566485
2000-01-05  1.426971  0.288267 -1.358877  1.808650
```

These have a similar set of methods to `.rolling` methods.

Method Summary

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
std()	Unbiased standard deviation
var()	Unbiased variance
skew()	Unbiased skewness (3rd moment)
kurt()	Unbiased kurtosis (4th moment)
quantile()	Sample quantile (value at %)
apply()	Generic apply
cov()	Unbiased covariance (binary)
corr()	Correlation (binary)

Aside from not having a `window` parameter, these functions have the same interfaces as their `.rolling` counterparts. Like above, the parameters they all accept are:

- `min_periods`: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No `NaNs` will be output once `min_periods` non-null data points have been seen.
- `center`: boolean, whether to set the labels at the center (default is False).

Note: The output of the `.rolling` and `.expanding` methods do not return a `NaN` if there are at least `min_periods` non-null values in the current window. For example:

```
In [100]: sn = pd.Series([1, 2, np.nan, 3, np.nan, 4])

In [101]: sn
Out[101]:
0    1.0
1    2.0
2    NaN
3    3.0
4    NaN
5    4.0
dtype: float64

In [102]: sn.rolling(2).max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[102]:
↪
0    NaN
1    2.0
2    NaN
3    NaN
4    NaN
5    NaN
dtype: float64

In [103]: sn.rolling(2, min_periods=1).max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

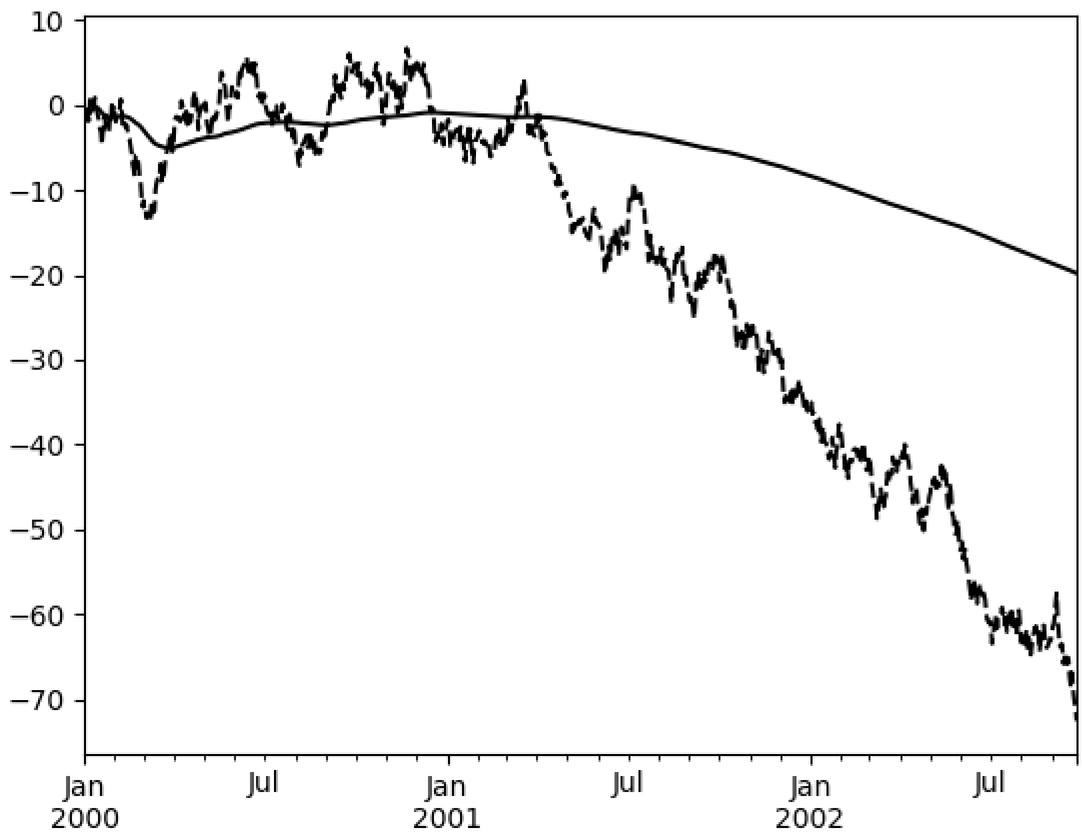
```
0    1.0
1    2.0
2    2.0
3    3.0
4    3.0
5    4.0
dtype: float64
```

In case of expanding functions, this differs from *cumsum()*, *cumprod()*, *cummax()*, and *cummin()*, which return `NaN` in the output wherever a `NaN` is encountered in the input. In order to match the output of `cumsum` with `expanding`, use *fillna()*:

```
In [104]: sn.expanding().sum()
Out[104]:
0     1.0
1     3.0
2     3.0
3     6.0
4     6.0
5    10.0
dtype: float64

In [105]: sn.cumsum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[105]:
↪
0     1.0
1     3.0
2     NaN
3     6.0
4     NaN
5    10.0
dtype: float64

In [106]: sn.cumsum().fillna(method='ffill')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     1.0
1     3.0
2     3.0
3     6.0
4     6.0
5    10.0
dtype: float64
```

An expanding window statistic will be more stable (and less responsive) than its rolling window counterpart as the increasing window size decreases the relative impact of an individual data point. As an example, here is the *mean()* output for the previous time series dataset:

```
In [107]: s.plot(style='k--')
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x7f381c06a080>

In [108]: s.expanding().mean().plot(style='k')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[108]:
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f381c06a080>
```



4.11.5 Exponentially Weighted Windows

A related set of functions are exponentially weighted versions of several of the above statistics. A similar interface to `.rolling` and `.expanding` is accessed through the `.ewm` method to receive an `EWM` object. A number of expanding EW (exponentially weighted) methods are provided:

Function	Description
`mean()`	EW moving average
`var()`	EW moving variance
`std()`	EW moving standard deviation
`corr()`	EW moving correlation
`cov()`	EW moving covariance

In general, a weighted moving average is calculated as

$$y_t = \frac{\sum_{i=0}^{t} w_i x_{t-i}}{\sum_{i=0}^{t} w_i},$$

where x_t is the input, y_t is the result and the w_i are the weights.

The EW functions support two variants of exponential weights. The default, `adjust=True`, uses the weights $w_i =$

$(1-\alpha)^i$ which gives

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ... + (1-\alpha)^t x_0}{1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^t}$$

When `adjust=False` is specified, moving averages are calculated as

$$\begin{split}y_0 &= x_0 \\ y_t &= (1-\alpha)y_{t-1} + \alpha x_t,\end{split}$$

which is equivalent to using weights

$$w_i = \begin{cases} \alpha (1-\alpha)^i & \text{if } i < t \\ (1-\alpha)^i & \text{if } i = t. \end{cases}$$

Note: These equations are sometimes written in terms of $\alpha' = 1 - \alpha$, e.g.

$$y_t = \alpha' y_{t-1} + (1 - \alpha')x_t.$$

The difference between the above two variants arises because we are dealing with series which have finite history. Consider a series of infinite history:

$$y_t = \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{1 + (1-\alpha) + (1-\alpha)^2 + ...}$$

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of $1-\alpha$ we have

$$\begin{split}y_t &= \frac{x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...}{\frac{1}{1-(1-\alpha)}}\\ &= [x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha \\ &= \alpha x_t + [(1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + ...]\alpha \\ &= \alpha x_t + (1-\alpha)[x_{t-1} + (1-\alpha) x_{t-2} + ...]\alpha\\ &= \alpha x_t + (1 - \alpha) y_{t-1}\end{split}$$

which shows the equivalence of the above two variants for infinite series. When `adjust=True` we have $y_0 = x_0$ and from the last representation above we have $y_t = \alpha x_t + (1 - \alpha) y_{t-1}$, therefore there is an assumption that x_0 is not an ordinary value but rather an exponentially weighted moment of the infinite series up to that point.

One must have $0 < \alpha \leq 1$, and while since version 0.18.0 it has been possible to pass α directly, it's often easier to think about either the **span**, **center of mass (com)** or **half-life** of an EW moment:

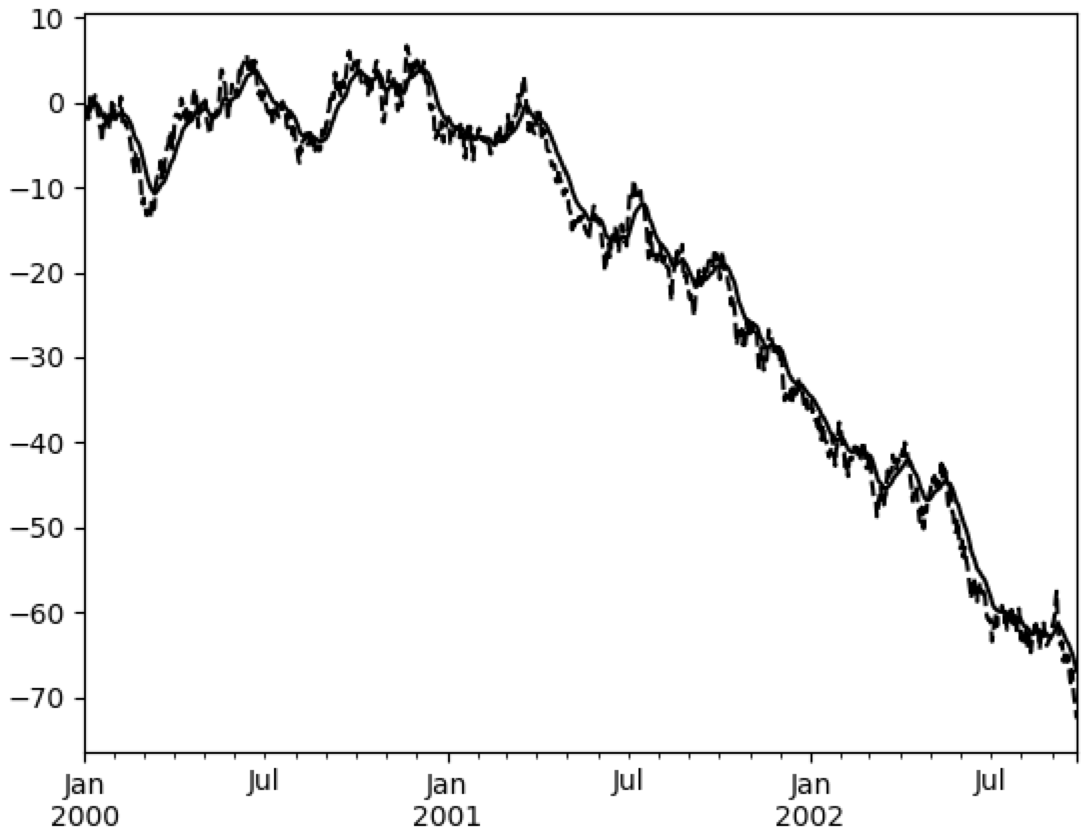
$$\alpha = \begin{cases} \frac{2}{s + 1}, & \text{for span}\ s \geq 1\\ \frac{1}{1 + c}, & \text{for center of mass}\ c \geq 0\\ 1 - \exp^{\frac{\log 0.5}{h}}, & \text{for half-life}\ h > 0 \end{cases}$$

One must specify precisely one of **span**, **center of mass**, **half-life** and **alpha** to the EW functions:

- **Span** corresponds to what is commonly called an "N-day EW moving average".
- **Center of mass** has a more physical interpretation and can be thought of in terms of span: $c = (s - 1)/2$.
- **Half-life** is the period of time for the exponential weight to reduce to one half.
- **Alpha** specifies the smoothing factor directly.

Here is an example for a univariate time series:

```
In [109]: s.plot(style='k--')
Out[109]: <matplotlib.axes._subplots.AxesSubplot at 0x7f380fec60f0>

In [110]: s.ewm(span=20).mean().plot(style='k')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[110]:
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f380fec60f0>
```



EWM has a `min_periods` argument, which has the same meaning it does for all the `.expanding` and `.rolling` methods: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window.

EWM also has an `ignore_na` argument, which determines how intermediate null values affect the calculation of the weights. When `ignore_na=False` (the default), weights are calculated based on absolute positions, so that intermediate null values affect the result. When `ignore_na=True`, weights are calculated by ignoring intermediate null values. For example, assuming `adjust=True`, if `ignore_na=False`, the weighted average of `3, NaN, 5` would be calculated as

$$\frac{(1-\alpha)^2 \cdot 3 + 1 \cdot 5}{(1-\alpha)^2 + 1}.$$

Whereas if `ignore_na=True`, the weighted average would be calculated as

$$\frac{(1-\alpha) \cdot 3 + 1 \cdot 5}{(1-\alpha) + 1}.$$

The `var()`, `std()`, and `cov()` functions have a `bias` argument, specifying whether the result should contain biased or unbiased statistics. For example, if `bias=True`, `ewmvar(x)` is calculated as `ewmvar(x) = ewma(x**2) - ewma(x)**2`; whereas if `bias=False` (the default), the biased variance statistics are scaled by debiasing factors

$$\frac{\left(\sum_{i=0}^{t} w_i\right)^2}{\left(\sum_{i=0}^{t} w_i\right)^2 - \sum_{i=0}^{t} w_i^2}.$$

(For $w_i = 1$, this reduces to the usual $N/(N-1)$ factor, with $N = t + 1$.) See Weighted Sample Variance on Wikipedia for further details.

4.12 Group By: split-apply-combine

By "group by" we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we might wish to one of the following:

- **Aggregation**: compute a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means.
  - Compute group sizes / counts.
- **Transformation**: perform some group-specific computations and return a like-indexed object. Some examples:
  - Standardize data (zscore) within a group.
  - Filling NAs within groups with a value derived from each group.
- **Filtration**: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  - Discard data that belongs to groups with only a few members.
  - Filter out data based on the group sum or mean.
- Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn't fit into either of the above two categories.

Since the set of object instance methods on pandas data structures are generally rich and expressive, we often simply want to invoke, say, a DataFrame function on each group. The name GroupBy should be quite familiar to those who have used a SQL-based tool (or `itertools`), in which you can write code like:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

We aim to make operations like this natural and easy to express using pandas. We'll address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the *cookbook* for some advanced strategies.

4.12.1 Splitting an object into groups

pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you may do the following:

```
In [1]: df = pd.DataFrame([('bird', 'Falconiformes', 389.0),
   ...:                    ('bird', 'Psittaciformes', 24.0),
   ...:                    ('mammal', 'Carnivora', 80.2),
   ...:                    ('mammal', 'Primates', np.nan),
   ...:                    ('mammal', 'Carnivora', 58)],
   ...:                   index=['falcon', 'parrot', 'lion', 'monkey', 'leopard'],
   ...:                   columns=('class', 'order', 'max_speed'))
   ...:

In [2]: df
Out[2]:
          class           order  max_speed
falcon     bird   Falconiformes      389.0
parrot     bird  Psittaciformes       24.0
lion     mammal       Carnivora       80.2
monkey   mammal        Primates        NaN
leopard  mammal       Carnivora       58.0

# default is axis=0
In [3]: grouped = df.groupby('class')

In [4]: grouped = df.groupby('order', axis='columns')

In [5]: grouped = df.groupby(['class', 'order'])
```

The mapping can be specified many different ways:

- A Python function, to be called on each of the axis labels.
- A list or NumPy array of the same length as the selected axis.
- A dict or `Series`, providing a `label -> group name` mapping.
- For `DataFrame` objects, a string indicating a column to be used to group. Of course `df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`, but it makes life simpler.
- For `DataFrame` objects, a string indicating an index level to be used to group.
- A list of any of the above things.

Collectively we refer to the grouping objects as the **keys**. For example, consider the following `DataFrame`:

Note: A string passed to `groupby` may refer to either a column or an index level. If a string matches both a column name and an index level name, a `ValueError` will be raised.

```
In [6]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ...:                          'foo', 'bar', 'foo', 'foo'],
   ...:                    'B': ['one', 'one', 'two', 'three',
   ...:                          'two', 'two', 'one', 'three'],
   ...:                    'C': np.random.randn(8),
   ...:                    'D': np.random.randn(8)})
   ...:
```

(continues on next page)

(continued from previous page)

```
In [7]: df
Out[7]:
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860
```

On a DataFrame, we obtain a GroupBy object by calling *groupby()*. We could naturally group by either the A or B columns, or both:

```
In [8]: grouped = df.groupby('A')

In [9]: grouped = df.groupby(['A', 'B'])
```

New in version 0.24.

If we also have a MultiIndex on columns A and B, we can group by all but the specified columns

```
In [10]: df2 = df.set_index(['A', 'B'])

In [11]: grouped = df2.groupby(level=df2.index.names.difference(['B']))

In [12]: grouped.sum()
Out[12]:
            C         D
A
bar -1.591710 -1.739537
foo -0.752861 -1.402938
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [13]: def get_letter_type(letter):
   ....:     if letter.lower() in 'aeiou':
   ....:         return 'vowel'
   ....:     else:
   ....:         return 'consonant'
   ....:

In [14]: grouped = df.groupby(get_letter_type, axis=1)
```

pandas *Index* objects support duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```
In [15]: lst = [1, 2, 3, 1, 2, 3]

In [16]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)

In [17]: grouped = s.groupby(level=0)

In [18]: grouped.first()
Out[18]:
```

(continues on next page)

(continued from previous page)

```
1    1
2    2
3    3
dtype: int64

In [19]: grouped.last()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:
1    10
2    20
3    30
dtype: int64

In [20]: grouped.sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]:
↪
1    11
2    22
3    33
dtype: int64
```

Note that **no splitting occurs** until it's needed. Creating the GroupBy object only verifies that you've passed a valid mapping.

Note: Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though can't be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.

GroupBy sorting

By default the group keys are sorted during the `groupby` operation. You may however pass `sort=False` for potential speedups:

```
In [21]: df2 = pd.DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [22]: df2.groupby(['X']).sum()
Out[22]:
   Y
X
A  7
B  3

In [23]: df2.groupby(['X'], sort=False).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:
   Y
X
B  3
A  7
```

Note that `groupby` will preserve the order in which *observations* are sorted *within* each group. For example, the groups created by `groupby()` below are in the order they appeared in the original `DataFrame`:

```
In [24]: df3 = pd.DataFrame({'X': ['A', 'B', 'A', 'B'], 'Y': [1, 4, 3, 2]})

In [25]: df3.groupby(['X']).get_group('A')
```

(continues on next page)

(continued from previous page)

```
Out[25]:
   X  Y
0  A  1
2  A  3

In [26]: df3.groupby(['X']).get_group('B')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]:
   X  Y
1  B  4
3  B  2
```

GroupBy object attributes

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```
In [27]: df.groupby('A').groups
Out[27]:
{'bar': Int64Index([1, 3, 5], dtype='int64'),
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}

In [28]: df.groupby(get_letter_type, axis=1).groups
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}
```

Calling the standard Python `len` function on the GroupBy object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [29]: grouped = df.groupby(['A', 'B'])

In [30]: grouped.groups
Out[30]:
{('bar', 'one'): Int64Index([1], dtype='int64'),
 ('bar', 'three'): Int64Index([3], dtype='int64'),
 ('bar', 'two'): Int64Index([5], dtype='int64'),
 ('foo', 'one'): Int64Index([0, 6], dtype='int64'),
 ('foo', 'three'): Int64Index([7], dtype='int64'),
 ('foo', 'two'): Int64Index([2, 4], dtype='int64')}

In [31]: len(grouped)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪6
```

`GroupBy` will tab complete column names (and other attributes):

```
In [32]: df
Out[32]:
               height      weight  gender
2000-01-01  42.849980  157.500553    male
2000-01-02  49.607315  177.340407    male
2000-01-03  56.293531  171.524640    male
2000-01-04  48.421077  144.251986  female
2000-01-05  46.556882  152.526206    male
```

(continues on next page)

(continued from previous page)

```
2000-01-06  68.448851  168.272968  female
2000-01-07  70.757698  136.431469    male
2000-01-08  58.909500  176.499753  female
2000-01-09  76.435631  174.094104  female
2000-01-10  45.306120  177.540920    male

In [33]: gb = df.groupby('gender')
```

```
In [34]: gb.<TAB>  # noqa: E225, E999
gb.agg        gb.boxplot    gb.cummin     gb.describe   gb.filter     gb.get_group  ␣
↪gb.height     gb.last       gb.median     gb.ngroups    gb.plot       gb.rank       ␣
↪gb.std        gb.transform
gb.aggregate  gb.count      gb.cumprod    gb.dtype      gb.first      gb.groups     ␣
↪gb.hist       gb.max        gb.min        gb.nth        gb.prod       gb.resample   ␣
↪gb.sum        gb.var
gb.apply      gb.cummax     gb.cumsum     gb.fillna     gb.gender     gb.head       ␣
↪gb.indices    gb.mean       gb.name       gb.ohlc       gb.quantile   gb.size       ␣
↪gb.tail       gb.weight
```

GroupBy with MultiIndex

With *hierarchically-indexed data*, it's quite natural to group by one of the levels of the hierarchy.

Let's create a Series with a two-level `MultiIndex`.

```
In [35]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [36]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [37]: s = pd.Series(np.random.randn(8), index=index)

In [38]: s
Out[38]:
first  second
bar    one      -0.919854
       two      -0.042379
baz    one       1.247642
       two      -0.009920
foo    one       0.290213
       two       0.495767
qux    one       0.362949
       two       1.548106
dtype: float64
```

We can then group by one of the levels in `s`.

```
In [39]: grouped = s.groupby(level=0)

In [40]: grouped.sum()
Out[40]:
first
bar   -0.962232
baz    1.237723
```

(continues on next page)

(continued from previous page)

```
foo    0.785980
qux    1.911055
dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```
In [41]: s.groupby(level='second').sum()
Out[41]:
second
one    0.980950
two    1.991575
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

```
In [42]: s.sum(level='second')
Out[42]:
second
one    0.980950
two    1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```
In [43]: s
Out[43]:
first  second  third
bar    doo     one     -1.131345
               two     -0.089329
baz    bee     one      0.337863
               two     -0.945867
foo    bop     one     -0.932132
               two      1.956030
qux    bop     one      0.017587
               two     -0.016692
dtype: float64

In [44]: s.groupby(level=['first', 'second']).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
first  second
bar    doo      -1.220674
baz    bee      -0.608004
foo    bop       1.023898
qux    bop       0.000895
dtype: float64
```

New in version 0.20.

Index level names may be supplied as keys.

```
In [45]: s.groupby(['first', 'second']).sum()
Out[45]:
first  second
bar    doo      -1.220674
baz    bee      -0.608004
```

(continues on next page)

(continued from previous page)

```
foo     bop        1.023898
qux     bop        0.000895
dtype: float64
```

More on the `sum` function and aggregation later.

Grouping DataFrame with Index Levels and Columns

A DataFrame may be grouped by a combination of columns and index levels by specifying the column names as strings and the index levels as `pd.Grouper` objects.

```
In [46]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [47]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [48]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
   ....:                    'B': np.arange(8)},
   ....:                   index=index)
   ....:

In [49]: df
Out[49]:
              A  B
first second
bar   one     1  0
      two     1  1
baz   one     1  2
      two     1  3
foo   one     2  4
      two     2  5
qux   one     3  6
      two     3  7
```

The following example groups `df` by the `second` index level and the `A` column.

```
In [50]: df.groupby([pd.Grouper(level=1), 'A']).sum()
Out[50]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

Index levels may also be specified by name.

```
In [51]: df.groupby([pd.Grouper(level='second'), 'A']).sum()
Out[51]:
          B
second A
one    1  2
```

(continues on next page)

(continued from previous page)

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

New in version 0.20.

Index level names may be specified as keys directly to `groupby`.

```
In [52]: df.groupby(['second', 'A']).sum()
Out[52]:
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7
```

DataFrame column selection in GroupBy

Once you have created the GroupBy object from a DataFrame, you might want to do something different for each of the columns. Thus, using `[]` similar to getting a column from a DataFrame, you can do:

```
In [53]: grouped = df.groupby(['A'])

In [54]: grouped_C = grouped['C']

In [55]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [56]: df['C'].groupby(df['A'])
Out[56]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7f381c03a048>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

4.12.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [57]: grouped = df.groupby('A')

In [58]: for name, group in grouped:
   ....:     print(name)
   ....:     print(group)
   ....:
bar
     A      B         C         D
1  bar    one  0.254161  1.511763
3  bar  three  0.215897 -0.990582
```

(continues on next page)

(continued from previous page)

```
5  bar    two -0.077118  1.211526
foo
     A      B         C         D
0  foo    one -0.575247  1.346061
2  foo    two -1.143704  1.627081
4  foo    two  1.193555 -0.441652
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580
```

In the case of grouping by multiple keys, the group name will be a tuple:

```
In [59]: for name, group in df.groupby(['A', 'B']):
   ....:     print(name)
   ....:     print(group)
   ....:
('bar', 'one')
     A    B         C         D
1  bar  one  0.254161  1.511763
('bar', 'three')
     A      B         C         D
3  bar  three  0.215897 -0.990582
('bar', 'two')
     A    B         C         D
5  bar  two -0.077118  1.211526
('foo', 'one')
     A    B         C         D
0  foo  one -0.575247  1.346061
6  foo  one -0.408530  0.268520
('foo', 'three')
     A      B         C        D
7  foo  three -0.862495  0.02458
('foo', 'two')
     A    B         C         D
2  foo  two -1.143704  1.627081
4  foo  two  1.193555 -0.441652
```

See *Iterating through groups*.

4.12.3 Selecting a group

A single group can be selected using `get_group()`:

```
In [60]: grouped.get_group('bar')
Out[60]:
     A      B         C         D
1  bar    one  0.254161  1.511763
3  bar  three  0.215897 -0.990582
5  bar    two -0.077118  1.211526
```

Or for an object grouped on multiple columns:

```
In [61]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[61]:
     A    B         C         D
1  bar  one  0.254161  1.511763
```

4.12.4 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. These operations are similar to the *aggregating API*, *window functions API*, and *resample API*.

An obvious one is aggregation via the `aggregate()` or equivalently `agg()` method:

```
In [62]: grouped = df.groupby('A')

In [63]: grouped.aggregate(np.sum)
Out[63]:
            C         D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [64]: grouped = df.groupby(['A', 'B'])

In [65]: grouped.aggregate(np.sum)
Out[65]:
                  C         D
A   B
bar one    0.254161  1.511763
    three  0.215897 -0.990582
    two   -0.077118  1.211526
foo one   -0.983776  1.614581
    three -0.862495  0.024580
    two    0.049851  1.185429
```

As you can see, the result of the aggregation will have the group names as the new index along the grouped axis. In the case of multiple keys, the result is a *MultiIndex* by default, though this can be changed by using the `as_index` option:

```
In [66]: grouped = df.groupby(['A', 'B'], as_index=False)

In [67]: grouped.aggregate(np.sum)
Out[67]:
     A      B         C         D
0  bar    one  0.254161  1.511763
1  bar  three  0.215897 -0.990582
2  bar    two -0.077118  1.211526
3  foo    one -0.983776  1.614581
4  foo  three -0.862495  0.024580
5  foo    two  0.049851  1.185429

In [68]: df.groupby('A', as_index=False).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A         C         D
0  bar  0.392940  1.732707
1  foo -1.796421  2.824590
```

Note that you could use the `reset_index` DataFrame function to achieve the same result as the column names are stored in the resulting `MultiIndex`:

```
In [69]: df.groupby(['A', 'B']).sum().reset_index()
Out[69]:
     A      B         C         D
```

(continues on next page)

(continued from previous page)

```
0  bar    one  0.254161  1.511763
1  bar  three  0.215897 -0.990582
2  bar    two -0.077118  1.211526
3  foo    one -0.983776  1.614581
4  foo  three -0.862495  0.024580
5  foo    two  0.049851  1.185429
```

Another simple aggregation example is to compute the size of each group. This is included in GroupBy as the `size` method. It returns a Series whose index are the group names and whose values are the sizes of each group.

```
In [70]: grouped.size()
Out[70]:
A    B
bar  one      1
     three    1
     two      1
foo  one      2
     three    1
     two      2
dtype: int64
```

```
In [71]: grouped.describe()
Out[71]:
      C                                                                         D  ␣
↪
  count      mean       std       min       25%       50%       75%       max count  ␣
↪     mean       std       min       25%       50%       75%       max
0   1.0  0.254161       NaN  0.254161  0.254161  0.254161  0.254161  0.254161   1.0 ␣
↪1.511763       NaN  1.511763  1.511763  1.511763  1.511763  1.511763
1   1.0  0.215897       NaN  0.215897  0.215897  0.215897  0.215897  0.215897   1.0 -
↪0.990582       NaN -0.990582 -0.990582 -0.990582 -0.990582 -0.990582
2   1.0 -0.077118       NaN -0.077118 -0.077118 -0.077118 -0.077118 -0.077118   1.0 ␣
↪1.211526       NaN  1.211526  1.211526  1.211526  1.211526  1.211526
3   2.0 -0.491888  0.117887 -0.575247 -0.533567 -0.491888 -0.450209 -0.408530   2.0 ␣
↪0.807291  0.761937  0.268520  0.537905  0.807291  1.076676  1.346061
4   1.0 -0.862495       NaN -0.862495 -0.862495 -0.862495 -0.862495 -0.862495   1.0 ␣
↪0.024580       NaN  0.024580  0.024580  0.024580  0.024580  0.024580
5   2.0  0.024925  1.652692 -1.143704 -0.559389  0.024925  0.609240  1.193555   2.0 ␣
↪0.592714  1.462816 -0.441652  0.075531  0.592714  1.109898  1.627081
```

Note: Aggregation functions **will not** return the groups that you are aggregating over if they are named *columns*, when `as_index=True`, the default. The grouped columns will be the **indices** of the returned object.

Passing `as_index=False` **will** return the groups that you are aggregating over, if they are named *columns*.

Aggregating functions are the ones that reduce the dimension of the returned objects. Some common aggregating functions are tabulated below:

Function	Description
`mean()`	Compute mean of groups
`sum()`	Compute sum of group values
`size()`	Compute group sizes
`count()`	Compute count of group
`std()`	Standard deviation of groups
`var()`	Compute variance of groups
`sem()`	Standard error of the mean of groups
`describe()`	Generates descriptive statistics
`first()`	Compute first of group values
`last()`	Compute last of group values
`nth()`	Take nth value, or a subset if n is a list
`min()`	Compute min of group values
`max()`	Compute max of group values

The aggregating functions above will exclude NA values. Any function which reduces a `Series` to a scalar value is an aggregation function and will work, a trivial example is `df.groupby('A').agg(lambda ser: 1)`. Note that `nth()` can act as a reducer *or* a filter, see *here*.

Applying multiple functions at once

With grouped `Series` you can also pass a list or dict of functions to do aggregation with, outputting a DataFrame:

```
In [72]: grouped = df.groupby('A')

In [73]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[73]:
          sum      mean       std
A
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

On a grouped `DataFrame`, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [74]: grouped.agg([np.sum, np.mean, np.std])
Out[74]:
            C                             D
          sum      mean       std       sum      mean       std
A
bar  0.392940  0.130980  0.181231  1.732707  0.577569  1.366330
foo -1.796421 -0.359284  0.912265  2.824590  0.564918  0.884785
```

The resulting aggregations are named for the functions themselves. If you need to rename, then you can add in a chained operation for a `Series` like this:

```
In [75]: (grouped['C'].agg([np.sum, np.mean, np.std])
   ....:              .rename(columns={'sum': 'foo',
   ....:                               'mean': 'bar',
   ....:                               'std': 'baz'}))
   ....:
Out[75]:
          foo        bar        baz
A
```

(continues on next page)

(continued from previous page)

```
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

For a grouped DataFrame, you can rename in a similar manner:

```
In [76]: (grouped.agg([np.sum, np.mean, np.std])
   ....:         .rename(columns={'sum': 'foo',
   ....:                          'mean': 'bar',
   ....:                          'std': 'baz'}))
   ....: 
Out[76]: 
            C                             D                    
          foo       bar       baz       foo       bar       baz
A                                                              
bar  0.392940  0.130980  0.181231  1.732707  0.577569  1.366330
foo -1.796421 -0.359284  0.912265  2.824590  0.564918  0.884785
```

Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```
In [77]: grouped.agg({'C': np.sum,
   ....:              'D': lambda x: np.std(x, ddof=1)})
   ....: 
Out[77]: 
            C         D
A                      
bar  0.392940  1.366330
foo -1.796421  0.884785
```

The function names can also be strings. In order for a string to be valid it must be either implemented on GroupBy or available via *dispatching*:

```
In [78]: grouped.agg({'C': 'sum', 'D': 'std'})
Out[78]: 
            C         D
A                      
bar  0.392940  1.366330
foo -1.796421  0.884785
```

Note: If you pass a dict to `aggregate`, the ordering of the output columns is non-deterministic. If you want to be sure the output columns will be in a specific order, you can use an `OrderedDict`. Compare the output of the following two commands:

```
In [79]: from collections import OrderedDict

In [80]: grouped.agg({'D': 'std', 'C': 'mean'})
Out[80]: 
            D         C
A                      
bar  1.366330  0.130980
foo  0.884785 -0.359284
```

(continues on next page)

(continued from previous page)

```
In [81]: grouped.agg(OrderedDict([('D', 'std'), ('C', 'mean')]))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            D         C
A
bar  1.366330  0.130980
foo  0.884785 -0.359284
```

Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, `std`, and `sem`, have optimized Cython implementations:

```
In [82]: df.groupby('A').sum()
Out[82]:
            C         D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [83]: df.groupby(['A', 'B']).mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                  C         D
A   B
bar one    0.254161  1.511763
    three  0.215897 -0.990582
    two   -0.077118  1.211526
foo one   -0.491888  0.807291
    three -0.862495  0.024580
    two    0.024925  0.592714
```

Of course `sum` and `mean` are implemented on pandas objects, so the above code would work even without the special versions via dispatching (see below).

4.12.5 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. The transform function must:

- Return a result that is either the same size as the group chunk or broadcastable to the size of the group chunk (e.g., a scalar, `grouped.transform(lambda x: x.iloc[-1])`).
- Operate column-by-column on the group chunk. The transform is applied to the first group chunk using chunk.apply.
- Not perform in-place operations on the group chunk. Group chunks should be treated as immutable, and changes to a group chunk may produce unexpected results. For example, when using `fillna`, `inplace` must be `False` (`grouped.transform(lambda x: x.fillna(inplace=False))`).
- (Optionally) operates on the entire group chunk. If this is supported, a fast path is used starting from the *second* chunk.

For example, suppose we wished to standardize the data within each group:

```
In [84]: index = pd.date_range('10/1/1999', periods=1100)

In [85]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)

In [86]: ts = ts.rolling(window=100, min_periods=100).mean().dropna()

In [87]: ts.head()
Out[87]:
2000-01-08    0.779333
2000-01-09    0.778852
2000-01-10    0.786476
2000-01-11    0.782797
2000-01-12    0.798110
Freq: D, dtype: float64

In [88]: ts.tail()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2002-09-30    0.660294
2002-10-01    0.631095
2002-10-02    0.673601
2002-10-03    0.709213
2002-10-04    0.719369
Freq: D, dtype: float64

In [89]: transformed = (ts.groupby(lambda x: x.year)
   ....:                 .transform(lambda x: (x - x.mean()) / x.std()))
   ....:
```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
# Original Data
In [90]: grouped = ts.groupby(lambda x: x.year)

In [91]: grouped.mean()
Out[91]:
2000    0.442441
2001    0.526246
2002    0.459365
dtype: float64

In [92]: grouped.std()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[92]:
2000    0.131752
2001    0.210945
2002    0.128753
dtype: float64

# Transformed Data
In [93]: grouped_trans = transformed.groupby(lambda x: x.year)

In [94]: grouped_trans.mean()
Out[94]:
2000    1.168208e-15
2001    1.454544e-15
2002    1.726657e-15
```

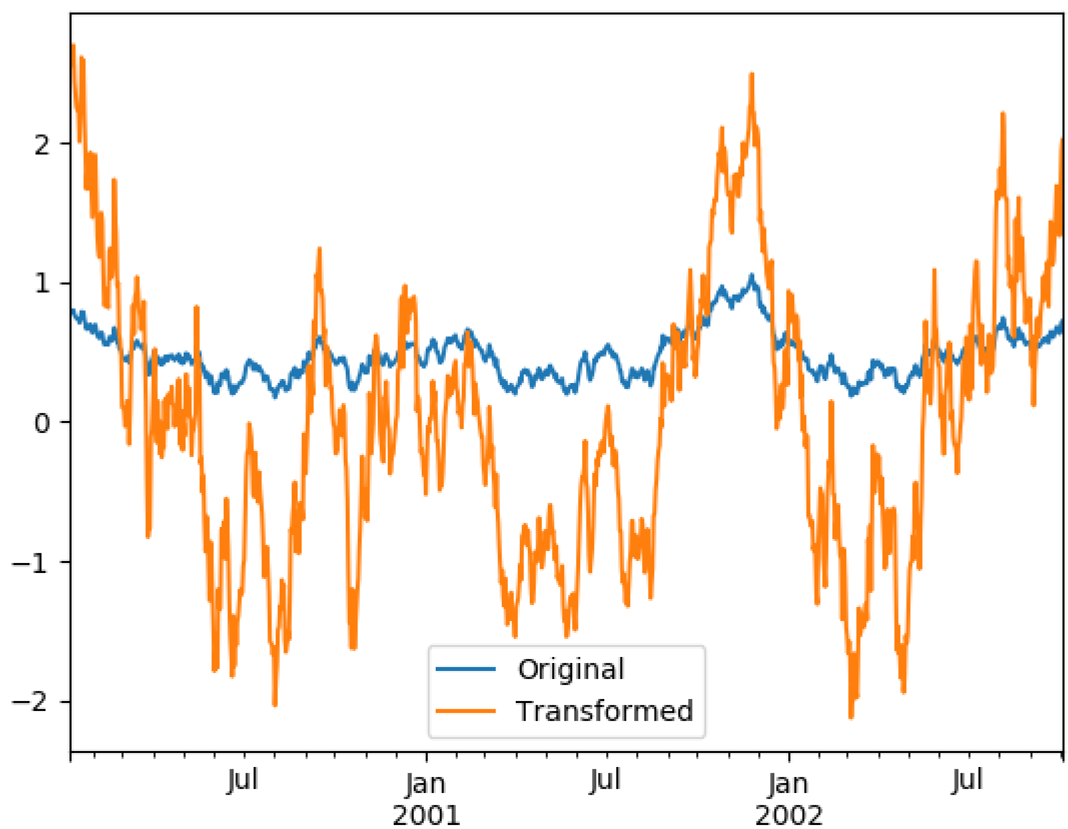
(continues on next page)

(continued from previous page)

```
dtype: float64

In [95]: grouped_trans.std()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[95]:␣
↪
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [96]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})

In [97]: compare.plot()
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x7f380fe0e2b0>
```



Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```
In [98]: ts.groupby(lambda x: x.year).transform(lambda x: x.max() - x.min())
Out[98]:
2000-01-08    0.623893
2000-01-09    0.623893
```

(continues on next page)

(continued from previous page)

```
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
2000-01-13    0.623893
2000-01-14    0.623893
                ...
2002-09-28    0.558275
2002-09-29    0.558275
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Alternatively, the built-in methods could be used to produce the same outputs.

```
In [99]: max = ts.groupby(lambda x: x.year).transform('max')

In [100]: min = ts.groupby(lambda x: x.year).transform('min')

In [101]: max - min
Out[101]:
2000-01-08    0.623893
2000-01-09    0.623893
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
2000-01-13    0.623893
2000-01-14    0.623893
                ...
2002-09-28    0.558275
2002-09-29    0.558275
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Another common data transform is to replace missing data with the group mean.

```
In [102]: data_df
Out[102]:
            A         B         C
0    1.539708 -1.166480  0.533026
1    1.302092 -0.505754       NaN
2   -0.371983  1.104803 -0.651520
3   -1.309622  1.118697 -1.161657
4   -1.924296  0.396437  0.812436
5    0.815643  0.367816 -0.469478
6   -0.030651  1.376106 -0.645129
..        ...       ...       ...
993  0.012359  0.554602 -1.976159
994  0.042312 -1.628835  1.013822
995 -0.093110  0.683847 -0.774753
996 -0.185043  1.438572       NaN
```

(continues on next page)

(continued from previous page)

```
997 -0.394469 -0.642343  0.011374
998 -1.174126  1.857148       NaN
999  0.234564  0.517098  0.393534

[1000 rows x 3 columns]

In [103]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [104]: key = countries[np.random.randint(0, 4, 1000)]

In [105]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [106]: grouped.count()
Out[106]: 
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [107]: transformed = grouped.transform(lambda x: x.fillna(x.mean()))
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains no NAs.

```
In [108]: grouped_trans = transformed.groupby(key)

In [109]: grouped.mean()  # original group means
Out[109]: 
           A         B         C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [110]: grouped_trans.mean()  # transformation did not change group means
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
           A         B         C
GR -0.098371 -0.015420  0.068053
JP  0.069025  0.023100 -0.077324
UK  0.034069 -0.052580 -0.116525
US  0.058664 -0.020399  0.028603

In [111]: grouped.count()  # original has some missing data points
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      A    B    C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217

In [112]: grouped_trans.count()  # counts after transformation
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
      A    B    C
GR  228  228  228
JP  267  267  267
UK  247  247  247
US  258  258  258

In [113]: grouped_trans.size()  # Verify non-NA count equals group size
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
GR    228
JP    267
UK    247
US    258
dtype: int64
```

Note: Some functions will automatically transform the input when applied to a GroupBy object, but returning an object of the same shape as the original. Passing `as_index=False` will not affect these transformation methods.

For example: `fillna, ffill, bfill, shift.`.

```
In [114]: grouped.ffill()
Out[114]:
    NaN         A         B         C
0    US  1.539708 -1.166480  0.533026
1    US  1.302092 -0.505754  0.533026
2    US -0.371983  1.104803 -0.651520
3    JP -1.309622  1.118697 -1.161657
4    JP -1.924296  0.396437  0.812436
5    US  0.815643  0.367816 -0.469478
6    GR -0.030651  1.376106 -0.645129
..   ..       ...       ...       ...
993  US  0.012359  0.554602 -1.976159
994  GR  0.042312 -1.628835  1.013822
995  JP -0.093110  0.683847 -0.774753
996  JP -0.185043  1.438572 -0.774753
997  GR -0.394469 -0.642343  0.011374
998  JP -1.174126  1.857148 -0.774753
999  UK  0.234564  0.517098  0.393534

[1000 rows x 4 columns]
```

New syntax to window and resample operations

New in version 0.18.1.

Working with the resample, expanding or rolling operations on the groupby level used to require the application of helper functions. However, now it is possible to use `resample()`, `expanding()` and `rolling()` as methods on groupbys.

The example below will apply the `rolling()` method on the samples of the column B based on the groups of column A.

```
In [115]: df_re = pd.DataFrame({'A': [1] * 10 + [5] * 10,
   .....:                       'B': np.arange(20)})
```

(continues on next page)

(continued from previous page)

```
   .....:

In [116]: df_re
Out[116]:
    A   B
0   1   0
1   1   1
2   1   2
3   1   3
4   1   4
5   1   5
6   1   6
..  ..  ..
13  5  13
14  5  14
15  5  15
16  5  16
17  5  17
18  5  18
19  5  19

[20 rows x 2 columns]

In [117]: df_re.groupby('A').rolling(4).B.mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
         ...
5  13    11.5
   14    12.5
   15    13.5
   16    14.5
   17    15.5
   18    16.5
   19    17.5
Name: B, Length: 20, dtype: float64
```

The `expanding()` method will accumulate a given operation (`sum()` in the example) for all the members of each particular group.

```
In [118]: df_re.groupby('A').expanding().sum()
Out[118]:
         A     B
A
1 0    1.0   0.0
  1    2.0   1.0
  2    3.0   3.0
  3    4.0   6.0
  4    5.0  10.0
```

(continues on next page)

(continued from previous page)

```
  5    6.0   15.0
  6    7.0   21.0
...    ...    ...
5 13  20.0   46.0
  14  25.0   60.0
  15  30.0   75.0
  16  35.0   91.0
  17  40.0  108.0
  18  45.0  126.0
  19  50.0  145.0

[20 rows x 2 columns]
```

Suppose you want to use the `resample()` method to get a daily frequency in each group of your dataframe and wish to complete the missing values with the `ffill()` method.

```
In [119]: df_re = pd.DataFrame({'date': pd.date_range(start='2016-01-01', periods=4,
   .....:                                             freq='W'),
   .....:                        'group': [1, 1, 2, 2],
   .....:                        'val': [5, 6, 7, 8]}).set_index('date')
   .....:

In [120]: df_re
Out[120]:
            group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

In [121]: df_re.groupby('group').resample('1D').ffill()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                  group  val
group date
1     2016-01-03      1    5
      2016-01-04      1    5
      2016-01-05      1    5
      2016-01-06      1    5
      2016-01-07      1    5
      2016-01-08      1    5
      2016-01-09      1    5
...                 ...  ...
2     2016-01-18      2    7
      2016-01-19      2    7
      2016-01-20      2    7
      2016-01-21      2    7
      2016-01-22      2    7
      2016-01-23      2    7
      2016-01-24      2    8

[16 rows x 2 columns]
```

4.12.6 Filtration

The `filter` method returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [122]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [123]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[123]:
3    3
4    3
5    3
dtype: int64
```

The argument of `filter` must be a function that, applied to the group as a whole, returns `True` or `False`.

Another useful operation is filtering out elements that belong to groups with only a couple members.

```
In [124]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [125]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[125]:
   A  B
2  2  b
3  3  b
4  4  b
5  5  b
```

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [126]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[126]:
     A    B
0  NaN  NaN
1  NaN  NaN
2  2.0    b
3  3.0    b
4  4.0    b
5  5.0    b
6  NaN  NaN
7  NaN  NaN
```

For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [127]: dff['C'] = np.arange(8)

In [128]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[128]:
   A  B  C
2  2  b  2
3  3  b  3
4  4  b  4
5  5  b  5
```

Note: Some functions when applied to a groupby object will act as a **filter** on the input, returning a reduced shape of the original (and potentially eliminating groups), but with the index unchanged. Passing `as_index=False` will not affect these transformation methods.

For example: `head, tail`.

```
In [129]: dff.groupby('B').head(2)
Out[129]:
   A  B  C
0  0  a  0
1  1  a  1
2  2  b  2
3  3  b  3
6  6  c  6
7  7  c  7
```

4.12.7 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```
In [130]: grouped = df.groupby('A')

In [131]: grouped.agg(lambda x: x.std())
Out[131]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

But, it's rather verbose and can be untidy if you need to pass additional arguments. Using a bit of metaprogramming cleverness, GroupBy now has the ability to "dispatch" method calls to the groups:

```
In [132]: grouped.std()
Out[132]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```
In [133]: tsdf = pd.DataFrame(np.random.randn(1000, 3),
   .....:                     index=pd.date_range('1/1/2000', periods=1000),
   .....:                     columns=['A', 'B', 'C'])
   .....:

In [134]: tsdf.iloc[::2] = np.nan

In [135]: grouped = tsdf.groupby(lambda x: x.year)

In [136]: grouped.fillna(method='pad')
Out[136]:
                   A         B         C
```

(continues on next page)

(continued from previous page)

```
2000-01-01       NaN       NaN       NaN
2000-01-02 -0.353501 -0.080957 -0.876864
2000-01-03 -0.353501 -0.080957 -0.876864
2000-01-04  0.050976  0.044273 -0.559849
2000-01-05  0.050976  0.044273 -0.559849
2000-01-06  0.030091  0.186460 -0.680149
2000-01-07  0.030091  0.186460 -0.680149
...              ...       ...       ...
2002-09-20  2.310215  0.157482 -0.064476
2002-09-21  2.310215  0.157482 -0.064476
2002-09-22  0.005011  0.053897 -1.026922
2002-09-23  0.005011  0.053897 -1.026922
2002-09-24 -0.456542 -1.849051  1.559856
2002-09-25 -0.456542 -1.849051  1.559856
2002-09-26  1.123162  0.354660  1.128135

[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

The `nlargest` and `nsmallest` methods work on `Series` style groupbys:

```
In [137]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])

In [138]: g = pd.Series(list('abababab'))

In [139]: gb = s.groupby(g)

In [140]: gb.nlargest(3)
Out[140]:
a  4    19.0
   0     9.0
   2     7.0
b  1     8.0
   3     5.0
   7     3.3
dtype: float64

In [141]: gb.nsmallest(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a  6    4.2
   2    7.0
   0    9.0
b  5    1.0
   7    3.3
   3    5.0
dtype: float64
```

4.12.8 Flexible `apply`

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want GroupBy to infer how to combine the results. For these, use the `apply` function, which can be substituted for both `aggregate` and `transform` in many standard use cases. However, `apply` can handle some exceptional use cases, for example:

```
In [142]: df
Out[142]:
     A      B         C         D
0  foo    one -0.575247  1.346061
1  bar    one  0.254161  1.511763
2  foo    two -1.143704  1.627081
3  bar  three  0.215897 -0.990582
4  foo    two  1.193555 -0.441652
5  bar    two -0.077118  1.211526
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580

In [143]: grouped = df.groupby('A')

# could also just call .describe()
In [144]: grouped['C'].apply(lambda x: x.describe())
Out[144]:
A
bar  count    3.000000
     mean     0.130980
     std      0.181231
     min     -0.077118
     25%      0.069390
     50%      0.215897
     75%      0.235029
                ...
foo  mean    -0.359284
     std      0.912265
     min     -1.143704
     25%     -0.862495
     50%     -0.575247
     75%     -0.408530
     max      1.193555
Name: C, Length: 16, dtype: float64
```

The dimension of the returned result can also change:

```
In [145]: grouped = df.groupby('A')['C']

In [146]: def f(group):
   .....:     return pd.DataFrame({'original': group,
   .....:                          'demeaned': group - group.mean()})
   .....:

In [147]: grouped.apply(f)
Out[147]:
   original  demeaned
0 -0.575247 -0.215962
1  0.254161  0.123181
2 -1.143704 -0.784420
3  0.215897  0.084917
4  1.193555  1.552839
5 -0.077118 -0.208098
6 -0.408530 -0.049245
7 -0.862495 -0.503211
```

`apply` on a Series can operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame:

```
In [148]: def f(x):
   .....:     return pd.Series([x, x ** 2], index=['x', 'x^2'])
   .....:

In [149]: s = pd.Series(np.random.rand(5))

In [150]: s
Out[150]:
0    0.321438
1    0.493496
2    0.139505
3    0.910103
4    0.194158
dtype: float64

In [151]: s.apply(f)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪
          x       x^2
0  0.321438  0.103323
1  0.493496  0.243538
2  0.139505  0.019462
3  0.910103  0.828287
4  0.194158  0.037697
```

Note: `apply` can act as a reducer, transformer, *or* filter function, depending on exactly what is passed to it. So depending on the path taken, and exactly what you are grouping. Thus the grouped columns(s) may be included in the output as well as set the indices.

Warning: In the current implementation apply calls func twice on the first group to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if func has side-effects, as they will take effect twice for the first group.

```
In [152]: d = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})

In [153]: def identity(df):
   .....:     print(df)
   .....:     return df
   .....:

In [154]: d.groupby("a").apply(identity)
   a  b
0  x  1
   a  b
0  x  1
   a  b
1  y  2
Out[154]:
   a  b
0  x  1
1  y  2
```

4.12.9 Other useful features

Automatic exclusion of "nuisance" columns

Again consider the example DataFrame we've been looking at:

```
In [155]: df
Out[155]:
     A      B         C         D
0  foo    one -0.575247  1.346061
1  bar    one  0.254161  1.511763
2  foo    two -1.143704  1.627081
3  bar  three  0.215897 -0.990582
4  foo    two  1.193555 -0.441652
5  bar    two -0.077118  1.211526
6  foo    one -0.408530  0.268520
7  foo  three -0.862495  0.024580
```

Suppose we wish to compute the standard deviation grouped by the `A` column. There is a slight problem, namely that we don't care about the data in column `B`. We refer to this as a "nuisance" column. If the passed aggregation function can't be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [156]: df.groupby('A').std()
Out[156]:
            C         D
A
bar  0.181231  1.366330
foo  0.912265  0.884785
```

Note that `df.groupby('A').colname.std().` is more efficient than `df.groupby('A').std().colname`, so if the result of an aggregation function is only interesting over one column (here `colname`), it may be filtered *before* applying the aggregation function.

Note: Any object column, also if it contains numerical values such as `Decimal` objects, is considered as a "nuisance" columns. They are excluded from aggregate functions automatically in groupby.

If you do wish to include decimal or object columns in an aggregation with other non-nuisance data types, you must do so explicitly.

```
In [157]: from decimal import Decimal

In [158]: df_dec = pd.DataFrame(
   .....:     {'id': [1, 2, 1, 2],
   .....:      'int_column': [1, 2, 3, 4],
   .....:      'dec_column': [Decimal('0.50'), Decimal('0.15'),
   .....:                     Decimal('0.25'), Decimal('0.40')]
   .....:      }
   .....: )
   .....:

# Decimal columns can be sum'd explicitly by themselves...
In [159]: df_dec.groupby(['id'])[['dec_column']].sum()
Out[159]:
   dec_column
```

(continues on next page)

(continued from previous page)

```
id
1        0.75
2        0.55

# ...but cannot be combined with standard data types or they will be excluded
In [160]: df_dec.groupby(['id'])[['int_column', 'dec_column']].sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[160]:
    int_column
id
1            4
2            6

# Use .agg function to aggregate over standard and "nuisance" data types
# at the same time
In [161]: df_dec.groupby(['id']).agg({'int_column': 'sum', 'dec_column': 'sum'})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
    int_column dec_column
id
1            4       0.75
2            6       0.55
```

Handling of (un)observed Categorical values

When using a `Categorical` grouper (as a single grouper, or as part of multiple groupers), the `observed` keyword controls whether to return a cartesian product of all possible groupers values (`observed=False`) or only those that are observed groupers (`observed=True`).

Show all values:

```
In [162]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                             categories=['a', 'b']),
   .....:                               observed=False).count()
   .....:
Out[162]:
a    3
b    0
dtype: int64
```

Show only the observed values:

```
In [163]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                             categories=['a', 'b']),
   .....:                               observed=True).count()
   .....:
Out[163]:
a    3
dtype: int64
```

The returned dtype of the grouped will *always* include *all* of the categories that were grouped.

```
In [164]: s = pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
   .....:                                                 categories=['a', 'b']),
   .....:                                   observed=False).count()
   .....:
```

(continues on next page)

(continued from previous page)

```
In [165]: s.index.dtype
Out[165]: CategoricalDtype(categories=['a', 'b'], ordered=False)
```

NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. In other words, there will never be an "NA group" or "NaT group". This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

Grouping with ordered factors

Categorical variables represented as instance of pandas's `Categorical` class can be used as group keys. If so, the order of the levels will be preserved:

```
In [166]: data = pd.Series(np.random.randn(100))

In [167]: factor = pd.qcut(data, [0, .25, .5, .75, 1.])

In [168]: data.groupby(factor).mean()
Out[168]:
(-2.645, -0.523]   -1.362896
(-0.523, 0.0296]   -0.260266
(0.0296, 0.654]     0.361802
(0.654, 2.21]       1.073801
dtype: float64
```

Grouping with a Grouper specification

You may need to specify a bit more data to properly group. You can use the `pd.Grouper` to provide this local control.

```
In [169]: import datetime

In [170]: df = pd.DataFrame({'Branch': 'A A A A A A A B'.split(),
   .....:                    'Buyer': 'Carl Mark Carl Carl Joe Joe Joe Carl'.split(),
   .....:                    'Quantity': [1, 3, 5, 1, 8, 1, 9, 3],
   .....:                    'Date': [
   .....:                        datetime.datetime(2013, 1, 1, 13, 0),
   .....:                        datetime.datetime(2013, 1, 1, 13, 5),
   .....:                        datetime.datetime(2013, 10, 1, 20, 0),
   .....:                        datetime.datetime(2013, 10, 2, 10, 0),
   .....:                        datetime.datetime(2013, 10, 1, 20, 0),
   .....:                        datetime.datetime(2013, 10, 2, 10, 0),
   .....:                        datetime.datetime(2013, 12, 2, 12, 0),
   .....:                        datetime.datetime(2013, 12, 2, 14, 0)]
   .....:                    })
   .....:

In [171]: df
Out[171]:
  Branch Buyer  Quantity                Date
```

(continues on next page)

(continued from previous page)

```
0      A  Carl         1 2013-01-01 13:00:00
1      A  Mark         3 2013-01-01 13:05:00
2      A  Carl         5 2013-10-01 20:00:00
3      A  Carl         1 2013-10-02 10:00:00
4      A   Joe         8 2013-10-01 20:00:00
5      A   Joe         1 2013-10-02 10:00:00
6      A   Joe         9 2013-12-02 12:00:00
7      B  Carl         3 2013-12-02 14:00:00
```

Groupby a specific column with the desired frequency. This is like resampling.

```
In [172]: df.groupby([pd.Grouper(freq='1M', key='Date'), 'Buyer']).sum()
Out[172]:
                  Quantity
Date       Buyer
2013-01-31 Carl          1
           Mark          3
2013-10-31 Carl          6
           Joe           9
2013-12-31 Carl          3
           Joe           9
```

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

```
In [173]: df = df.set_index('Date')

In [174]: df['Date'] = df.index + pd.offsets.MonthEnd(2)

In [175]: df.groupby([pd.Grouper(freq='6M', key='Date'), 'Buyer']).sum()
Out[175]:
                  Quantity
Date       Buyer
2013-02-28 Carl          1
           Mark          3
2014-02-28 Carl          9
           Joe          18

In [176]: df.groupby([pd.Grouper(freq='6M', level='Date'), 'Buyer']).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                  Quantity
Date       Buyer
2013-01-31 Carl          1
           Mark          3
2014-01-31 Carl          9
           Joe          18
```

Taking the first rows of each group

Just like for a DataFrame or Series you can call head and tail on a groupby:

```
In [177]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [178]: df
Out[178]:
```

(continues on next page)

(continued from previous page)

```
   A  B
0  1  2
1  1  4
2  5  6

In [179]: g = df.groupby('A')

In [180]: g.head(1)
Out[180]:
   A  B
0  1  2
2  5  6

In [181]: g.tail(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[181]:
   A  B
1  1  4
2  5  6
```

This shows the first or last n rows from each group.

Taking the nth row of each group

To select from a DataFrame or Series the nth item, use `nth()`. This is a reduction method, and will return a single row (or no row) per group if you pass an int for n:

```
In [182]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [183]: g = df.groupby('A')

In [184]: g.nth(0)
Out[184]:
     B
A
1  NaN
5  6.0

In [185]: g.nth(-1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[185]:
     B
A
1  4.0
5  6.0

In [186]: g.nth(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[186]:
↪
     B
A
1  4.0
```

If you want to select the nth not-null item, use the `dropna` kwarg. For a DataFrame this should be either `'any'` or `'all'` just like you would pass to dropna:

```
# nth(0) is the same as g.first()
In [187]: g.nth(0, dropna='any')
Out[187]:
     B
A
1  4.0
5  6.0

In [188]: g.first()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[188]:
     B
A
1  4.0
5  6.0

# nth(-1) is the same as g.last()
In [189]: g.nth(-1, dropna='any')  # NaNs denote group exhausted when using dropna
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[189]:
↪
     B
A
1  4.0
5  6.0

In [190]: g.last()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     B
A
1  4.0
5  6.0

In [191]: g.B.nth(0, dropna='all')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A
1    4.0
5    6.0
Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```
In [192]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [193]: g = df.groupby('A', as_index=False)

In [194]: g.nth(0)
Out[194]:
   A    B
0  1  NaN
2  5  6.0

In [195]: g.nth(-1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[195]:
   A    B
1  1  4.0
2  5  6.0
```

You can also select multiple rows from each group by specifying multiple nth values as a list of ints.

```
In [196]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [197]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [198]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[198]:
        a  b
2014 4  1  1
     4  1  1
     4  1  1
     5  1  1
     5  1  1
     5  1  1
     6  1  1
     6  1  1
     6  1  1
```

Enumerate group items

To see the order in which each row appears within its group, use the `cumcount` method:

```
In [199]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [200]: dfg
Out[200]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [201]: dfg.groupby('A').cumcount()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[201]:
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64

In [202]: dfg.groupby('A').cumcount(ascending=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

Enumerate groups

New in version 0.20.2.

To see the ordering of the groups (as opposed to the order of rows within a group given by `cumcount`) you can use `ngroup()`.

Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

```
In [203]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [204]: dfg
Out[204]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [205]: dfg.groupby('A').ngroup()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[205]:
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64

In [206]: dfg.groupby('A').ngroup(ascending=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
```

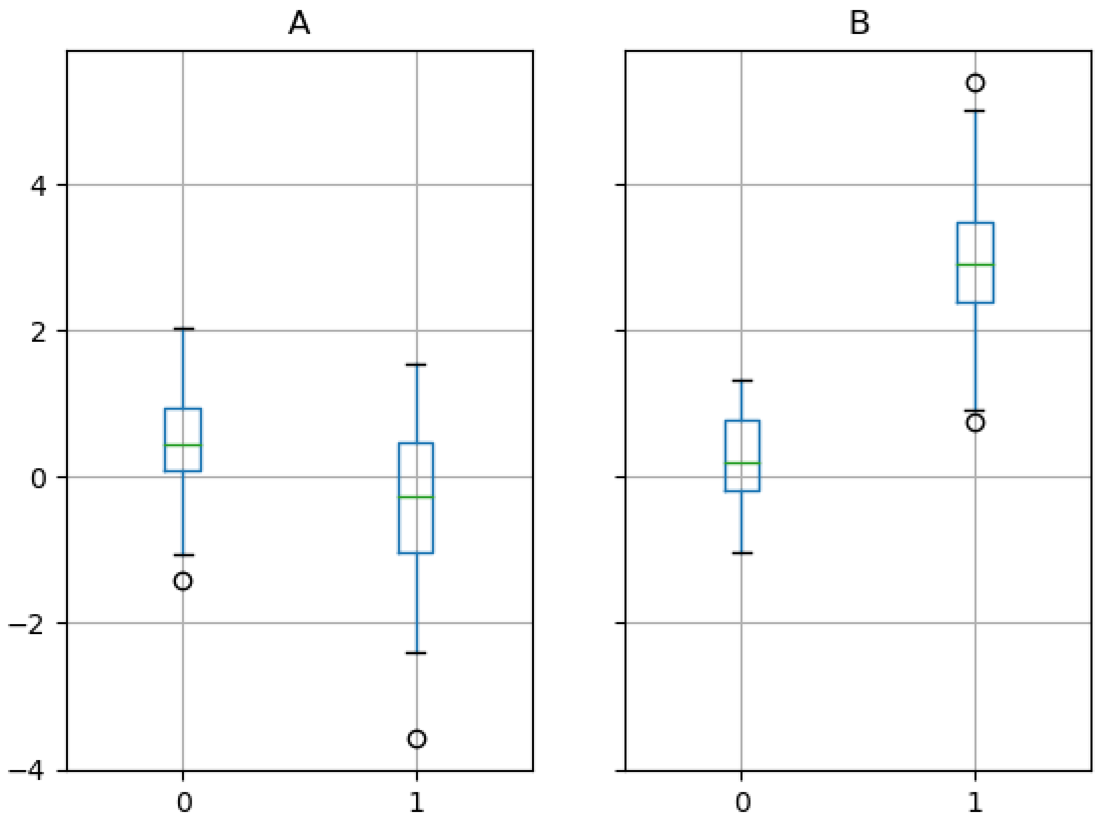
Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is "B" are 3 higher on average.

```
In [207]: np.random.seed(1234)

In [208]: df = pd.DataFrame(np.random.randn(50, 2))

In [209]: df['g'] = np.random.choice(['A', 'B'], size=50)

In [210]: df.loc[df['g'] == 'B', 1] += 3
```

We can easily visualize this with a boxplot:

```
In [211]: df.groupby('g').boxplot()
Out[211]:
A         AxesSubplot(0.1,0.15;0.363636x0.75)
B    AxesSubplot(0.536364,0.15;0.363636x0.75)
dtype: object
```



The result of calling `boxplot` is a dictionary whose keys are the values of our grouping column `g` ("A" and "B"). The values of the resulting dictionary can be controlled by the `return_type` keyword of `boxplot`. See the *visualization documentation* for more.

> **Warning:** For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See *here* for an explanation.

Piping function calls

New in version 0.21.0.

Similar to the functionality provided by `DataFrame` and `Series`, functions that take `GroupBy` objects can be chained together using a `pipe` method to allow for a cleaner, more readable syntax. To read about `.pipe` in general terms, see *here*.

Combining `.groupby` and `.pipe` is often useful when you need to reuse GroupBy objects.

As an example, imagine having a DataFrame with columns for stores, products, revenue and quantity sold. We'd like to do a groupwise calculation of *prices* (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable. First we set the data:

```
In [212]: n = 1000

In [213]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
   .....:                      'Product': np.random.choice(['Product_1',
   .....:                                                   'Product_2'], n),
   .....:                      'Revenue': (np.random.random(n) * 50 + 10).round(2),
   .....:                      'Quantity': np.random.randint(1, 10, size=n)})
   .....: 

In [214]: df.head(2)
Out[214]: 
     Store    Product  Revenue  Quantity
0  Store_2  Product_1    26.12         1
1  Store_2  Product_1    28.86         1
```

Now, to find prices per store/product, we can simply do:

```
In [215]: (df.groupby(['Store', 'Product'])
   .....:    .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   .....:    .unstack().round(2))
   .....: 
Out[215]: 
Product  Product_1  Product_2
Store                        
Store_1       6.82       7.05
Store_2       6.30       6.64
```

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

```
In [216]: def mean(groupby):
   .....:     return groupby.mean()
   .....: 

In [217]: df.groupby(['Store', 'Product']).pipe(mean)
Out[217]: 
                     Revenue  Quantity
Store   Product                       
Store_1 Product_1  34.622727  5.075758
        Product_2  35.482815  5.029630
Store_2 Product_1  32.972837  5.237589
        Product_2  34.684360  5.224000
```

where `mean` takes a GroupBy object and finds the mean of the Revenue and Quantity columns repectively for each Store-Product combination. The `mean` function can be any function that takes in a GroupBy object; the `.pipe` will pass the GroupBy object as a parameter into the function you specify.

4.12.10 Examples

Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

```
In [218]: df = pd.DataFrame({'a': [1, 0, 0], 'b': [0, 1, 0],
   .....:                     'c': [1, 0, 0], 'd': [2, 3, 4]})
   .....:

In [219]: df
Out[219]:
   a  b  c  d
0  1  0  1  2
1  0  1  0  3
2  0  0  0  4

In [220]: df.groupby(df.sum(), axis=1).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[220]:
   1  9
0  2  2
1  1  3
2  0  4
```

Multi-column factorization

By using `ngroup()`, we can extract information about the groups in a way similar to *factorize()* (as described further in the *reshaping API*) but which applies naturally to multiple columns of mixed type and different sources. This can be useful as an intermediate categorical-like step in processing, when the relationships between the group rows are more important than their content, or as input to an algorithm which only accepts the integer encoding. (For more information about support in pandas for full categorical data, see the *Categorical introduction* and the *API documentation*.)

```
In [221]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})

In [222]: dfg
Out[222]:
   A  B
0  1  a
1  1  a
2  2  a
3  3  b
4  2  a

In [223]: dfg.groupby(["A", "B"]).ngroup()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[223]:
0    0
1    0
2    1
3    2
4    1
dtype: int64

In [224]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    0
1    0
2    1
3    3
4    2
dtype: int64
```

Groupby by Indexer to 'resample' data

Resampling produces new hypothetical samples (resamples) from already existing observed data or from a model that generates data. These new samples are similar to the pre-existing samples.

In order to resample to work on indices that are non-datetimelike, the following procedure can be utilized.

In the following examples, **df.index // 5** returns a binary array which is used to determine what gets selected for the groupby operation.

Note: The below example shows how we can downsample by consolidation of samples into fewer samples. Here by using **df.index // 5**, we are aggregating the samples in bins. By applying **std()** function, we aggregate the information contained in many samples into a small subset of values which is their standard deviation thereby reducing the number of samples.

```
In [225]: df = pd.DataFrame(np.random.randn(10, 2))

In [226]: df
Out[226]:
          0         1
0 -0.793893  0.321153
1  0.342250  1.618906
2 -0.975807  1.918201
3 -0.810847 -1.405919
4 -1.977759  0.461659
5  0.730057 -1.316938
6 -0.751328  0.528290
7 -0.257759 -1.081009
8  0.505895 -1.701948
9 -1.006349  0.020208

In [227]: df.index // 5
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Int64Index([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype='int64')

In [228]: df.groupby(df.index // 5).std()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          0         1
0  0.823647  1.312912
1  0.760109  0.942941
```

Returning a Series to propagate names

Group DataFrame columns, compute a set of metrics and return a named Series. The Series name is used as the name for the column index. This is especially useful in conjunction with reshaping operations such as stacking in which the column index name will be used as the name of the inserted column:

```
In [229]: df = pd.DataFrame({'a': [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2],
   .....:                    'b': [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
   .....:                    'c': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
   .....:                    'd': [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]})
   .....:

In [230]: def compute_metrics(x):
```

(continues on next page)

(continued from previous page)

```
   .....:     result = {'b_sum': x['b'].sum(), 'c_mean': x['c'].mean()}
   .....:     return pd.Series(result, name='metrics')
   .....:

In [231]: result = df.groupby('a').apply(compute_metrics)

In [232]: result
Out[232]: 
metrics  b_sum  c_mean
a
0          2.0     0.5
1          2.0     0.5
2          2.0     0.5

In [233]: result.stack()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a  metrics
0  b_sum      2.0
   c_mean     0.5
1  b_sum      2.0
   c_mean     0.5
2  b_sum      2.0
   c_mean     0.5
dtype: float64
```

4.13 Time Series / Date functionality

pandas contains extensive capabilities and features for working with time series data for all domains. Using the NumPy `datetime64` and `timedelta64` dtypes, pandas has consolidated a large number of features from other Python libraries like `scikits.timeseries` as well as created a tremendous amount of new functionality for manipulating time series data.

For example, pandas supports:

Parsing time series information from various sources and formats

```
In [1]: import datetime

In [2]: dti = pd.to_datetime(['1/1/2018', np.datetime64('2018-01-01'),
   ...:                       datetime.datetime(2018, 1, 1)])
   ...:

In [3]: dti
Out[3]: DatetimeIndex(['2018-01-01', '2018-01-01', '2018-01-01'], dtype=
↪'datetime64[ns]', freq=None)
```

Generate sequences of fixed-frequency dates and time spans

```
In [4]: dti = pd.date_range('2018-01-01', periods=3, freq='H')

In [5]: dti
Out[5]: 
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 01:00:00',
```

(continues on next page)

(continued from previous page)

```
               '2018-01-01 02:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Manipulating and converting date times with timezone information

```
In [6]: dti = dti.tz_localize('UTC')

In [7]: dti
Out[7]:
DatetimeIndex(['2018-01-01 00:00:00+00:00', '2018-01-01 01:00:00+00:00',
               '2018-01-01 02:00:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='H')

In [8]: dti.tz_convert('US/Pacific')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2017-12-31 16:00:00-08:00', '2017-12-31 17:00:00-08:00',
               '2017-12-31 18:00:00-08:00'],
              dtype='datetime64[ns, US/Pacific]', freq='H')
```

Resampling or converting a time series to a particular frequency

```
In [9]: idx = pd.date_range('2018-01-01', periods=5, freq='H')

In [10]: ts = pd.Series(range(len(idx)), index=idx)

In [11]: ts
Out[11]:
2018-01-01 00:00:00    0
2018-01-01 01:00:00    1
2018-01-01 02:00:00    2
2018-01-01 03:00:00    3
2018-01-01 04:00:00    4
Freq: H, dtype: int64

In [12]: ts.resample('2H').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2018-01-01 00:00:00    0.5
2018-01-01 02:00:00    2.5
2018-01-01 04:00:00    4.0
Freq: 2H, dtype: float64
```

Performing date and time arithmetic with absolute or relative time increments

```
In [13]: friday = pd.Timestamp('2018-01-05')

In [14]: friday.day_name()
Out[14]: 'Friday'

# Add 1 day
In [15]: saturday = friday + pd.Timedelta('1 day')

In [16]: saturday.day_name()
Out[16]: 'Saturday'

# Add 1 business day (Friday --> Monday)
```

(continues on next page)

(continued from previous page)

```
In [17]: monday = friday + pd.offsets.BDay()

In [18]: monday.day_name()
Out[18]: 'Monday'
```

pandas provides a relatively compact and self-contained set of tools for performing the above tasks and more.

4.13.1 Overview

pandas captures 4 general time related concepts:

1. Date times: A specific date and time with timezone support. Similar to `datetime.datetime` from the standard library.
2. Time deltas: An absolute time duration. Similar to `datetime.timedelta` from the standard library.
3. Time spans: A span of time defined by a point in time and its associated frequency.
4. Date offsets: A relative time duration that respects calendar arithmetic. Similar to `dateutil.relativedelta.relativedelta` from the `dateutil` package.

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`
Time spans	`Period`	`PeriodIndex`	`period[freq]`	`Period` or `period_range`
Date offsets	`DateOffset`	`None`	`None`	`DateOffset`

For time series data, it's conventional to represent the time component in the index of a *Series* or *DataFrame* so manipulations can be performed with respect to the time element.

```
In [19]: pd.Series(range(3), index=pd.date_range('2000', freq='D', periods=3))
Out[19]:
2000-01-01    0
2000-01-02    1
2000-01-03    2
Freq: D, dtype: int64
```

However, *Series* and *DataFrame* can directly also support the time component as data itself.

```
In [20]: pd.Series(pd.date_range('2000', freq='D', periods=3))
Out[20]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
dtype: datetime64[ns]
```

Series and *DataFrame* have extended data type support and functionality for `datetime`, `timedelta` and `Period` data when passed into those constructors. `DateOffset` data however will be stored as `object` data.

```
In [21]: pd.Series(pd.period_range('1/1/2011', freq='M', periods=3))
Out[21]:
0    2011-01
1    2011-02
2    2011-03
dtype: period[M]

In [22]: pd.Series([pd.DateOffset(1), pd.DateOffset(2)])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
0          <DateOffset>
1    <2 * DateOffsets>
dtype: object

In [23]: pd.Series(pd.date_range('1/1/2011', freq='M', periods=3))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2011-01-31
1   2011-02-28
2   2011-03-31
dtype: datetime64[ns]
```

Lastly, pandas represents null date times, time deltas, and time spans as `NaT` which is useful for representing missing or null date like values and behaves similar as `np.nan` does for float data.

```
In [24]: pd.Timestamp(pd.NaT)
Out[24]: NaT

In [25]: pd.Timedelta(pd.NaT)
\\\\\\\\\\\\\Out[25]: NaT

In [26]: pd.Period(pd.NaT)
\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]: NaT

# Equality acts as np.nan would
In [27]: pd.NaT == pd.NaT
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[27]: False
```

4.13.2 Timestamps vs. Time Spans

Timestamped data is the most basic type of time series data that associates values with points in time. For pandas objects it means using the points in time.

```
In [28]: pd.Timestamp(datetime.datetime(2012, 5, 1))
Out[28]: Timestamp('2012-05-01 00:00:00')

In [29]: pd.Timestamp('2012-05-01')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[29]: Timestamp('2012-05-01 00:00:00')

In [30]: pd.Timestamp(2012, 5, 1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]:␣
↪Timestamp('2012-05-01 00:00:00')
```

However, in many cases it is more natural to associate things like change variables with a time span instead. The span represented by `Period` can be specified explicitly, or inferred from datetime string format.

For example:

```
In [31]: pd.Period('2011-01')
Out[31]: Period('2011-01', 'M')

In [32]: pd.Period('2012-05', freq='D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[32]: Period('2012-05-01', 'D')
```

Timestamp and *Period* can serve as an index. Lists of `Timestamp` and `Period` are automatically coerced to *DatetimeIndex* and *PeriodIndex* respectively.

```
In [33]: dates = [pd.Timestamp('2012-05-01'),
   ....:          pd.Timestamp('2012-05-02'),
   ....:          pd.Timestamp('2012-05-03')]
   ....:

In [34]: ts = pd.Series(np.random.randn(3), dates)

In [35]: type(ts.index)
Out[35]: pandas.core.indexes.datetimes.DatetimeIndex

In [36]: ts.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[36]: DatetimeIndex(['2012-05-
↪01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)

In [37]: ts
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-05-01    0.469112
2012-05-02   -0.282863
2012-05-03   -1.509059
dtype: float64

In [38]: periods = [pd.Period('2012-01'), pd.Period('2012-02'), pd.Period('2012-03')]

In [39]: ts = pd.Series(np.random.randn(3), periods)

In [40]: type(ts.index)
Out[40]: pandas.core.indexes.period.PeriodIndex

In [41]: ts.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[41]: PeriodIndex(['2012-01',
↪'2012-02', '2012-03'], dtype='period[M]', freq='M')

In [42]: ts
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01   -1.135632
2012-02    1.212112
2012-03   -0.173215
Freq: M, dtype: float64
```

pandas allows you to capture both representations and convert between them. Under the hood, pandas represents timestamps using instances of `Timestamp` and sequences of timestamps using instances of `DatetimeIndex`. For regular time spans, pandas uses `Period` objects for scalar values and `PeriodIndex` for sequences of spans. Better support for irregular intervals with arbitrary start and end points are forth-coming in future releases.

4.13.3 Converting to Timestamps

To convert a *Series* or list-like object of date-like objects e.g. strings, epochs, or a mixture, you can use the `to_datetime` function. When passed a `Series`, this returns a `Series` (with the same index), while a list-like is converted to a `DatetimeIndex`:

```
In [43]: pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))
Out[43]:
0   2009-07-31
1   2010-01-10
2          NaT
dtype: datetime64[ns]

In [44]: pd.to_datetime(['2005/11/23', '2010.12.31'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[44]:
↪DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None)
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```
In [45]: pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)
Out[45]: DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=None)

In [46]: pd.to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[46]:
↪DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]', freq=None)
```

Warning: You see in the above example that `dayfirst` isn't strict, so if a date can't be parsed with the day being first it will be parsed as if `dayfirst` were False.

If you pass a single string to `to_datetime`, it returns a single `Timestamp`. `Timestamp` can also accept string input, but it doesn't accept string parsing options like `dayfirst` or `format`, so use `to_datetime` if these are required.

```
In [47]: pd.to_datetime('2010/11/12')
Out[47]: Timestamp('2010-11-12 00:00:00')

In [48]: pd.Timestamp('2010/11/12')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[48]: Timestamp('2010-11-12 00:00:00')
```

You can also use the `DatetimeIndex` constructor directly:

```
In [49]: pd.DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'])
Out[49]: DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], dtype=
↪'datetime64[ns]', freq=None)
```

The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation:

```
In [50]: pd.DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], freq='infer')
Out[50]: DatetimeIndex(['2018-01-01', '2018-01-03', '2018-01-05'], dtype=
↪'datetime64[ns]', freq='2D')
```

Providing a Format Argument

In addition to the required datetime string, a `format` argument can be passed to ensure specific parsing. This could also potentially speed up the conversion considerably.

```
In [51]: pd.to_datetime('2010/11/12', format='%Y/%m/%d')
Out[51]: Timestamp('2010-11-12 00:00:00')

In [52]: pd.to_datetime('12-11-2010 00:00', format='%d-%m-%Y %H:%M')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[52]: Timestamp('2010-11-12 00:00:00')
```

For more information on the choices available when specifying the `format` option, see the Python datetime documentation.

Assembling Datetime from Multiple DataFrame Columns

New in version 0.18.1.

You can also pass a `DataFrame` of integer or string columns to assemble into a `Series` of `Timestamps`.

```
In [53]: df = pd.DataFrame({'year': [2015, 2016],
   ....:                    'month': [2, 3],
   ....:                    'day': [4, 5],
   ....:                    'hour': [2, 3]})
   ....:

In [54]: pd.to_datetime(df)
Out[54]:
0   2015-02-04 02:00:00
1   2016-03-05 03:00:00
dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```
In [55]: pd.to_datetime(df[['year', 'month', 'day']])
Out[55]:
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

`pd.to_datetime` looks for standard designations of the datetime component in the column names, including:

- required: `year`, `month`, `day`
- optional: `hour`, `minute`, `second`, `millisecond`, `microsecond`, `nanosecond`

Invalid Data

The default behavior, `errors='raise'`, is to raise when unparseable:

```
In [2]: pd.to_datetime(['2009/07/31', 'asd'], errors='raise')
ValueError: Unknown string format
```

Pass `errors='ignore'` to return the original input when unparseable:

```
In [56]: pd.to_datetime(['2009/07/31', 'asd'], errors='ignore')
Out[56]: Index(['2009/07/31', 'asd'], dtype='object')
```

Pass `errors='coerce'` to convert unparseable data to `NaT` (not a time):

```
In [57]: pd.to_datetime(['2009/07/31', 'asd'], errors='coerce')
Out[57]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

Epoch Timestamps

pandas supports converting integer or float epoch times to `Timestamp` and `DatetimeIndex`. The default unit is nanoseconds, since that is how `Timestamp` objects are stored internally. However, epochs are often stored in another `unit` which can be specified. These are computed from the starting point specified by the `origin` parameter.

```
In [58]: pd.to_datetime([1349720105, 1349806505, 1349892905,
   ....:                 1349979305, 1350065705], unit='s')
   ....:
Out[58]:
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05',
               '2012-10-12 18:15:05'],
              dtype='datetime64[ns]', freq=None)

In [59]: pd.to_datetime([1349720105100, 1349720105200, 1349720105300,
   ....:                 1349720105400, 1349720105500], unit='ms')
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2012-10-08 18:15:05.100000', '2012-10-08 18:15:05.200000',
               '2012-10-08 18:15:05.300000', '2012-10-08 18:15:05.400000',
               '2012-10-08 18:15:05.500000'],
              dtype='datetime64[ns]', freq=None)
```

Note: Epoch times will be rounded to the nearest nanosecond.

Warning: Conversion of float epoch times can lead to inaccurate and unexpected results. Python floats have about 15 digits precision in decimal. Rounding during conversion from float to high precision `Timestamp` is unavoidable. The only way to achieve exact precision is to use a fixed-width types (e.g. an int64).

```
In [60]: pd.to_datetime([1490195805.433, 1490195805.433502912], unit='s')
Out[60]: DatetimeIndex(['2017-03-22 15:16:45.433000088', '2017-03-22 15:16:45.
↪433502913'], dtype='datetime64[ns]', freq=None)

In [61]: pd.to_datetime(1490195805433502912, unit='ns')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2017-03-22 15:16:45.433502912')
```

See also:

Using the origin Parameter

From Timestamps to Epoch

To invert the operation from above, namely, to convert from a `Timestamp` to a 'unix' epoch:

```
In [62]: stamps = pd.date_range('2012-10-08 18:15:05', periods=4, freq='D')

In [63]: stamps
Out[63]:
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05'],
              dtype='datetime64[ns]', freq='D')
```

We subtract the epoch (midnight at January 1, 1970 UTC) and then floor divide by the "unit" (1 second).

```
In [64]: (stamps - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
Out[64]: Int64Index([1349720105, 1349806505, 1349892905, 1349979305], dtype='int64')
```

Using the `origin` Parameter

New in version 0.20.0.

Using the `origin` parameter, one can specify an alternative starting point for creation of a `DatetimeIndex`. For example, to use 1960-01-01 as the starting date:

```
In [65]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[65]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

The default is set at `origin='unix'`, which defaults to `1970-01-01 00:00:00`. Commonly called 'unix epoch' or POSIX time.

```
In [66]: pd.to_datetime([1, 2, 3], unit='D')
Out[66]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

4.13.4 Generating Ranges of Timestamps

To generate an index with timestamps, you can use either the `DatetimeIndex` or `Index` constructor and pass in a list of datetime objects:

```
In [67]: dates = [datetime.datetime(2012, 5, 1),
   ....:          datetime.datetime(2012, 5, 2),
   ....:          datetime.datetime(2012, 5, 3)]
   ....:

# Note the frequency information
In [68]: index = pd.DatetimeIndex(dates)

In [69]: index
Out[69]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↪'datetime64[ns]', freq=None)

# Automatically converted to DatetimeIndex
In [70]: index = pd.Index(dates)

In [71]: index
Out[71]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↪'datetime64[ns]', freq=None)
```

In practice this becomes very cumbersome because we often need a very long index with a large number of timestamps. If we need timestamps on a regular frequency, we can use the *date_range()* and *bdate_range()* functions to create a DatetimeIndex. The default frequency for date_range is a **calendar day** while the default for bdate_range is a **business day**:

```
In [72]: start = datetime.datetime(2011, 1, 1)

In [73]: end = datetime.datetime(2012, 1, 1)

In [74]: index = pd.date_range(start, end)

In [75]: index
Out[75]:
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10',
               ...
               '2011-12-23', '2011-12-24', '2011-12-25', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30',
               '2011-12-31', '2012-01-01'],
              dtype='datetime64[ns]', length=366, freq='D')

In [76]: index = pd.bdate_range(start, end)

In [77]: index
Out[77]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14',
               ...
               '2011-12-19', '2011-12-20', '2011-12-21', '2011-12-22',
               '2011-12-23', '2011-12-26', '2011-12-27', '2011-12-28',
               '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', length=260, freq='B')
```

Convenience functions like date_range and bdate_range can utilize a variety of *frequency aliases*:

```
In [78]: pd.date_range(start, periods=1000, freq='M')
Out[78]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-30',
               '2011-05-31', '2011-06-30', '2011-07-31', '2011-08-31',
               '2011-09-30', '2011-10-31',
               ...
               '2093-07-31', '2093-08-31', '2093-09-30', '2093-10-31',
               '2093-11-30', '2093-12-31', '2094-01-31', '2094-02-28',
               '2094-03-31', '2094-04-30'],
              dtype='datetime64[ns]', length=1000, freq='M')

In [79]: pd.bdate_range(start, periods=250, freq='BQS')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-03', '2011-04-01', '2011-07-01', '2011-10-03',
               '2012-01-02', '2012-04-02', '2012-07-02', '2012-10-01',
               '2013-01-01', '2013-04-01',
               ...
               '2071-01-01', '2071-04-01', '2071-07-01', '2071-10-01',
               '2072-01-01', '2072-04-01', '2072-07-01', '2072-10-03',
               '2073-01-02', '2073-04-03'],
```

(continues on next page)

(continued from previous page)

```
              dtype='datetime64[ns]', length=250, freq='BQS-JAN')
```

`date_range` and `bdate_range` make it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`. The start and end dates are strictly inclusive, so dates outside of those specified will not be generated:

```
In [80]: pd.date_range(start, end, freq='BM')
Out[80]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [81]: pd.date_range(start, end, freq='W')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-02', '2011-01-09', '2011-01-16', '2011-01-23',
               '2011-01-30', '2011-02-06', '2011-02-13', '2011-02-20',
               '2011-02-27', '2011-03-06', '2011-03-13', '2011-03-20',
               '2011-03-27', '2011-04-03', '2011-04-10', '2011-04-17',
               '2011-04-24', '2011-05-01', '2011-05-08', '2011-05-15',
               '2011-05-22', '2011-05-29', '2011-06-05', '2011-06-12',
               '2011-06-19', '2011-06-26', '2011-07-03', '2011-07-10',
               '2011-07-17', '2011-07-24', '2011-07-31', '2011-08-07',
               '2011-08-14', '2011-08-21', '2011-08-28', '2011-09-04',
               '2011-09-11', '2011-09-18', '2011-09-25', '2011-10-02',
               '2011-10-09', '2011-10-16', '2011-10-23', '2011-10-30',
               '2011-11-06', '2011-11-13', '2011-11-20', '2011-11-27',
               '2011-12-04', '2011-12-11', '2011-12-18', '2011-12-25',
               '2012-01-01'],
              dtype='datetime64[ns]', freq='W-SUN')

In [82]: pd.bdate_range(end=end, periods=20)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-12-05', '2011-12-06', '2011-12-07', '2011-12-08',
               '2011-12-09', '2011-12-12', '2011-12-13', '2011-12-14',
               '2011-12-15', '2011-12-16', '2011-12-19', '2011-12-20',
               '2011-12-21', '2011-12-22', '2011-12-23', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', freq='B')

In [83]: pd.bdate_range(start=start, periods=20)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-17', '2011-01-18',
               '2011-01-19', '2011-01-20', '2011-01-21', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28'],
              dtype='datetime64[ns]', freq='B')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced dates from `start` to `end` inclusively, with `periods` number of elements in the resulting `DatetimeIndex`:

```
In [84]: pd.date_range('2018-01-01', '2018-01-05', periods=5)
Out[84]:
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05'],
              dtype='datetime64[ns]', freq=None)

In [85]: pd.date_range('2018-01-01', '2018-01-05', periods=10)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 10:40:00',
               '2018-01-01 21:20:00', '2018-01-02 08:00:00',
               '2018-01-02 18:40:00', '2018-01-03 05:20:00',
               '2018-01-03 16:00:00', '2018-01-04 02:40:00',
               '2018-01-04 13:20:00', '2018-01-05 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Custom Frequency Ranges

Warning: This functionality was originally exclusive to `cdate_range`, which is deprecated as of version 0.21.0 in favor of `bdate_range`. Note that `cdate_range` only utilizes the `weekmask` and `holidays` parameters when custom business day, 'C', is passed as the frequency string. Support has been expanded with `bdate_range` to work with any custom frequency string.

New in version 0.21.0.

`bdate_range` can also generate a range of custom frequency dates by using the `weekmask` and `holidays` parameters. These parameters will only be used if a custom frequency string is passed.

```
In [86]: weekmask = 'Mon Wed Fri'

In [87]: holidays = [datetime.datetime(2011, 1, 5), datetime.datetime(2011, 3, 14)]

In [88]: pd.bdate_range(start, end, freq='C', weekmask=weekmask, holidays=holidays)
Out[88]:
DatetimeIndex(['2011-01-03', '2011-01-07', '2011-01-10', '2011-01-12',
               '2011-01-14', '2011-01-17', '2011-01-19', '2011-01-21',
               '2011-01-24', '2011-01-26',
               ...
               '2011-12-09', '2011-12-12', '2011-12-14', '2011-12-16',
               '2011-12-19', '2011-12-21', '2011-12-23', '2011-12-26',
               '2011-12-28', '2011-12-30'],
              dtype='datetime64[ns]', length=154, freq='C')

In [89]: pd.bdate_range(start, end, freq='CBMS', weekmask=weekmask)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-03', '2011-02-02', '2011-03-02', '2011-04-01',
               '2011-05-02', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-02', '2011-10-03', '2011-11-02', '2011-12-02'],
              dtype='datetime64[ns]', freq='CBMS')
```

See also:

Custom Business Days

4.13.5 Timestamp Limitations

Since pandas represents timestamps in nanosecond resolution, the time span that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [90]: pd.Timestamp.min
Out[90]: Timestamp('1677-09-21 00:12:43.145225')

In [91]: pd.Timestamp.max
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[91]: Timestamp('2262-04-11␣
↪23:47:16.854775807')
```

See also:

Representing Out-of-Bounds Spans

4.13.6 Indexing

One of the main uses for `DatetimeIndex` is as an index for pandas objects. The `DatetimeIndex` class contains many time series related optimizations:

- A large range of dates for various offsets are pre-computed and cached under the hood in order to make generating subsequent date ranges very fast (just have to grab a slice).
- Fast shifting using the `shift` and `tshift` method on pandas objects.
- Unioning of overlapping `DatetimeIndex` objects with the same frequency is very fast (important for fast data alignment).
- Quick access to date fields via properties such as `year`, `month`, etc.
- Regularization functions like `snap` and very fast `asof` logic.

`DatetimeIndex` objects have all the basic functionality of regular `Index` objects, and a smorgasbord of advanced time series specific methods for easy frequency processing.

See also:

Reindexing methods

Note: While pandas does not force you to have a sorted date index, some of these methods may have unexpected or incorrect behavior if the dates are unsorted.

`DatetimeIndex` can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```
In [92]: rng = pd.date_range(start, end, freq='BM')

In [93]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [94]: ts.index
Out[94]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [95]: ts[:5].index
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31'],
              dtype='datetime64[ns]', freq='BM')

In [96]: ts[::2].index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-31', '2011-03-31', '2011-05-31', '2011-07-29',
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```

Partial String Indexing

Dates and strings that parse to timestamps can be passed as indexing parameters:

```
In [97]: ts['1/31/2011']
Out[97]: 0.11920871129693428

In [98]: ts[datetime.datetime(2011, 12, 25):]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[98]:
2011-12-30    0.56702
Freq: BM, dtype: float64

In [99]: ts['10/31/2011':'12/31/2011']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[99]:␣
↪
2011-10-31    0.271860
2011-11-30   -0.424972
2011-12-30    0.567020
Freq: BM, dtype: float64
```

To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

```
In [100]: ts['2011']
Out[100]:
2011-01-31    0.119209
2011-02-28   -1.044236
2011-03-31   -0.861849
2011-04-29   -2.104569
2011-05-31   -0.494929
2011-06-30    1.071804
2011-07-29    0.721555
2011-08-31   -0.706771
2011-09-30   -1.039575
2011-10-31    0.271860
2011-11-30   -0.424972
2011-12-30    0.567020
Freq: BM, dtype: float64

In [101]: ts['2011-6']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2011-06-30    1.071804
```

(continues on next page)

(continued from previous page)

```
Freq: BM, dtype: float64
```

This type of slicing will work on a DataFrame with a DatetimeIndex as well. Since the partial string selection is a form of label slicing, the endpoints **will be** included. This would include matching times on an included date:

```
In [102]: dft = pd.DataFrame(np.random.randn(100000, 1), columns=['A'],
   .....:                    index=pd.date_range('20130101', periods=100000, freq='T
↪'))
   .....:

In [103]: dft
Out[103]:
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-03-11 10:33:00 -1.562855
2013-03-11 10:34:00 -0.776320
2013-03-11 10:35:00 -0.747967
2013-03-11 10:36:00 -0.034523
2013-03-11 10:37:00 -0.201754
2013-03-11 10:38:00 -1.509067
2013-03-11 10:39:00 -1.693043

[100000 rows x 1 columns]

In [104]: dft['2013']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-03-11 10:33:00 -1.562855
2013-03-11 10:34:00 -0.776320
2013-03-11 10:35:00 -0.747967
2013-03-11 10:36:00 -0.034523
2013-03-11 10:37:00 -0.201754
2013-03-11 10:38:00 -1.509067
2013-03-11 10:39:00 -1.693043

[100000 rows x 1 columns]
```

This starts on the very first time in the month, and includes the last date and time for the month:

```
In [105]: dft['2013-1':'2013-2']
```

(continues on next page)

(continued from previous page)

```
Out[105]:
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-02-28 23:53:00  0.096317
2013-02-28 23:54:00  1.069352
2013-02-28 23:55:00  0.850929
2013-02-28 23:56:00  0.976712
2013-02-28 23:57:00 -2.693884
2013-02-28 23:58:00 -1.575535
2013-02-28 23:59:00 -1.573517

[84960 rows x 1 columns]
```

This specifies a stop time **that includes all of the times on the last day**:

```
In [106]: dft['2013-1':'2013-2-28']
Out[106]:
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-02-28 23:53:00  0.096317
2013-02-28 23:54:00  1.069352
2013-02-28 23:55:00  0.850929
2013-02-28 23:56:00  0.976712
2013-02-28 23:57:00 -2.693884
2013-02-28 23:58:00 -1.575535
2013-02-28 23:59:00 -1.573517

[84960 rows x 1 columns]
```

This specifies an **exact** stop time (and is not the same as the above):

```
In [107]: dft['2013-1':'2013-2-28 00:00:00']
Out[107]:
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-02-27 23:54:00  1.604295
```

(continues on next page)

(continued from previous page)

```
2013-02-27 23:55:00  0.885350
2013-02-27 23:56:00  1.197749
2013-02-27 23:57:00  0.720521
2013-02-27 23:58:00 -0.072718
2013-02-27 23:59:00 -0.681192
2013-02-28 00:00:00 -0.557501

[83521 rows x 1 columns]
```

We are stopping on the included end-point as it is part of the index:

```
In [108]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[108]: 
                            A
2013-01-15 00:00:00 -0.984810
2013-01-15 00:01:00  0.941451
2013-01-15 00:02:00  1.559365
2013-01-15 00:03:00  1.034374
2013-01-15 00:04:00 -1.480656
2013-01-15 00:05:00  0.212765
2013-01-15 00:06:00  0.294784
...                       ...
2013-01-15 12:24:00 -0.850073
2013-01-15 12:25:00 -0.526367
2013-01-15 12:26:00  0.371454
2013-01-15 12:27:00 -0.930806
2013-01-15 12:28:00 -0.069177
2013-01-15 12:29:00  0.066510
2013-01-15 12:30:00 -0.003945

[751 rows x 1 columns]
```

New in version 0.18.0.

`DatetimeIndex` partial string indexing also works on a `DataFrame` with a `MultiIndex`:

```
In [109]: dft2 = pd.DataFrame(np.random.randn(20, 1),
   .....:                     columns=['A'],
   .....:                     index=pd.MultiIndex.from_product(
   .....:                         [pd.date_range('20130101', periods=10, freq='12H'),
   .....:                          ['a', 'b']]))
   .....: 

In [110]: dft2
Out[110]: 
                              A
2013-01-01 00:00:00 a -0.298694
                    b  0.823553
2013-01-01 12:00:00 a  0.943285
                    b -1.479399
2013-01-02 00:00:00 a -1.643342
                    b  1.005292
2013-01-02 12:00:00 a -1.562237
...                         ...
2013-01-04 00:00:00 b  0.054993
2013-01-04 12:00:00 a -0.784071
                    b  0.069036
```

(continues on next page)

(continued from previous page)

```
2013-01-05 00:00:00 a  0.122297
                    b  1.422060
2013-01-05 12:00:00 a  0.370079
                    b  1.016331

[20 rows x 1 columns]

In [111]: dft2.loc['2013-01-05']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                              A
2013-01-05 00:00:00 a  0.122297
                    b  1.422060
2013-01-05 12:00:00 a  0.370079
                    b  1.016331

In [112]: idx = pd.IndexSlice

In [113]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [114]: dft2.loc[idx[:, '2013-01-05'], :]
Out[114]:
                              A
a 2013-01-05 00:00:00  0.122297
  2013-01-05 12:00:00  0.370079
b 2013-01-05 00:00:00  1.422060
  2013-01-05 12:00:00  1.016331
```

Slice vs. Exact Match

Changed in version 0.20.0.

The same string used as an indexing parameter can be treated either as a slice or as an exact match depending on the resolution of the index. If the string is less accurate than the index, it will be treated as a slice, otherwise as an exact match.

Consider a `Series` object with a minute resolution index:

```
In [115]: series_minute = pd.Series([1, 2, 3],
   .....:                           pd.DatetimeIndex(['2011-12-31 23:59:00',
   .....:                                             '2012-01-01 00:00:00',
   .....:                                             '2012-01-01 00:02:00']))
   .....:

In [116]: series_minute.index.resolution
Out[116]: 'minute'
```

A timestamp string less accurate than a minute gives a `Series` object.

```
In [117]: series_minute['2011-12-31 23']
Out[117]:
2011-12-31 23:59:00    1
dtype: int64
```

A timestamp string with minute resolution (or more accurate), gives a scalar instead, i.e. it is not casted to a slice.

```
In [118]: series_minute['2011-12-31 23:59']
Out[118]: 1

In [119]: series_minute['2011-12-31 23:59:00']
\\\\\\\\\\\\Out[119]: 1
```

If index resolution is second, then the minute-accurate timestamp gives a `Series`.

```
In [120]: series_second = pd.Series([1, 2, 3],
   .....:                           pd.DatetimeIndex(['2011-12-31 23:59:59',
   .....:                                             '2012-01-01 00:00:00',
   .....:                                             '2012-01-01 00:00:01']))
   .....:

In [121]: series_second.index.resolution
Out[121]: 'second'

In [122]: series_second['2011-12-31 23:59']
\\\\\\\\\\\\\\\\\\Out[122]:
2011-12-31 23:59:59    1
dtype: int64
```

If the timestamp string is treated as a slice, it can be used to index `DataFrame` with `[]` as well.

```
In [123]: dft_minute = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]},
   .....:                           index=series_minute.index)
   .....:

In [124]: dft_minute['2011-12-31 23']
Out[124]:
                     a  b
2011-12-31 23:59:00  1  4
```

Warning: However, if the string is treated as an exact match, the selection in `DataFrame`'s `[]` will be column-wise and not row-wise, see *Indexing Basics*. For example `dft_minute['2011-12-31 23:59']` will raise `KeyError` as `'2012-12-31 23:59'` has the same resolution as the index and there is no column with such name:

To *always* have unambiguous selection, whether the row is treated as a slice or a single selection, use `.loc`.

```
In [125]: dft_minute.loc['2011-12-31 23:59']
Out[125]:
a    1
b    4
Name: 2011-12-31 23:59:00, dtype: int64
```

Note also that `DatetimeIndex` resolution cannot be less precise than day.

```
In [126]: series_monthly = pd.Series([1, 2, 3],
   .....:                            pd.DatetimeIndex(['2011-12', '2012-01', '2012-02
↪']))
   .....:

In [127]: series_monthly.index.resolution
Out[127]: 'day'
```

(continues on next page)

(continued from previous page)

```
In [128]: series_monthly['2011-12']  # returns Series
\\\\\\\\\\\\\\\\Out[128]:
2011-12-01    1
dtype: int64
```

Exact Indexing

As discussed in previous section, indexing a `DatetimeIndex` with a partial string depends on the "accuracy" of the period, in other words how specific the interval is in relation to the resolution of the index. In contrast, indexing with `Timestamp` or `datetime` objects is exact, because the objects have exact meaning. These also follow the semantics of *including both endpoints*.

These `Timestamp` and `datetime` objects have exact `hours, minutes,` and `seconds`, even though they were not explicitly specified (they are `0`).

```
In [129]: dft[datetime.datetime(2013, 1, 1):datetime.datetime(2013, 2, 28)]
Out[129]:
                            A
2013-01-01 00:00:00  0.276232
2013-01-01 00:01:00 -1.087401
2013-01-01 00:02:00 -0.673690
2013-01-01 00:03:00  0.113648
2013-01-01 00:04:00 -1.478427
2013-01-01 00:05:00  0.524988
2013-01-01 00:06:00  0.404705
...                       ...
2013-02-27 23:54:00  1.604295
2013-02-27 23:55:00  0.885350
2013-02-27 23:56:00  1.197749
2013-02-27 23:57:00  0.720521
2013-02-27 23:58:00 -0.072718
2013-02-27 23:59:00 -0.681192
2013-02-28 00:00:00 -0.557501

[83521 rows x 1 columns]
```

With no defaults.

```
In [130]: dft[datetime.datetime(2013, 1, 1, 10, 12, 0):
   .....:     datetime.datetime(2013, 2, 28, 10, 12, 0)]
   .....:
Out[130]:
                            A
2013-01-01 10:12:00  0.565375
2013-01-01 10:13:00  0.068184
2013-01-01 10:14:00  0.788871
2013-01-01 10:15:00 -0.280343
2013-01-01 10:16:00  0.931536
2013-01-01 10:17:00 -0.700648
2013-01-01 10:18:00 -1.166607
...                       ...
2013-02-28 10:06:00 -1.393114
2013-02-28 10:07:00  2.321893
2013-02-28 10:08:00  0.148098
```

(continues on next page)

(continued from previous page)

```
2013-02-28 10:09:00 -0.388138
2013-02-28 10:10:00  0.139348
2013-02-28 10:11:00  0.085288
2013-02-28 10:12:00  0.950146

[83521 rows x 1 columns]
```

Truncating & Fancy Indexing

A `truncate()` convenience function is provided that is similar to slicing. Note that `truncate` assumes a 0 value for any unspecified date component in a `DatetimeIndex` in contrast to slicing which returns any partially matching dates:

```
In [131]: rng2 = pd.date_range('2011-01-01', '2012-01-01', freq='W')

In [132]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)

In [133]: ts2.truncate(before='2011-11', after='2011-12')
Out[133]:
2011-11-06    0.437823
2011-11-13   -0.293083
2011-11-20   -0.059881
2011-11-27    1.252450
Freq: W-SUN, dtype: float64

In [134]: ts2['2011-11':'2011-12']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2011-11-06    0.437823
2011-11-13   -0.293083
2011-11-20   -0.059881
2011-11-27    1.252450
2011-12-04    0.046611
2011-12-11    0.059478
2011-12-18   -0.286539
2011-12-25    0.841669
Freq: W-SUN, dtype: float64
```

Even complicated fancy indexing that breaks the `DatetimeIndex` frequency regularity will result in a `DatetimeIndex`, although frequency is lost:

```
In [135]: ts2[[0, 2, 6]].index
Out[135]: DatetimeIndex(['2011-01-02', '2011-01-16', '2011-02-13'], dtype=
↪'datetime64[ns]', freq=None)
```

4.13.7 Iterating through groups

With the `Resampler` object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [136]: small = pd.Series(
   .....:     range(6),
   .....:     index=pd.to_datetime(['2017-01-01T00:00:00',
```

(continues on next page)

(continued from previous page)

```
   .....:                            '2017-01-01T00:30:00',
   .....:                            '2017-01-01T00:31:00',
   .....:                            '2017-01-01T01:00:00',
   .....:                            '2017-01-01T03:00:00',
   .....:                            '2017-01-01T03:05:00'])
   .....: )
   .....:

In [137]: resampled = small.resample('H')

In [138]: for name, group in resampled:
   .....:     print("Group: ", name)
   .....:     print("-" * 27)
   .....:     print(group, end="\n\n")
   .....:
Group:  2017-01-01 00:00:00
---------------------------
2017-01-01 00:00:00    0
2017-01-01 00:30:00    1
2017-01-01 00:31:00    2
dtype: int64

Group:  2017-01-01 01:00:00
---------------------------
2017-01-01 01:00:00    3
dtype: int64

Group:  2017-01-01 02:00:00
---------------------------
Series([], dtype: int64)

Group:  2017-01-01 03:00:00
---------------------------
2017-01-01 03:00:00    4
2017-01-01 03:05:00    5
dtype: int64
```

See *Iterating through groups* or `Resampler.__iter__` for more.

4.13.8 Time/Date Components

There are several time/date properties that one can access from `Timestamp` or a collection of timestamps like a `DatetimeIndex`.

Property	Description
year	The year of the datetime
month	The month of the datetime
day	The days of the datetime
hour	The hour of the datetime
minute	The minutes of the datetime
second	The seconds of the datetime
microsecond	The microseconds of the datetime
nanosecond	The nanoseconds of the datetime
date	Returns datetime.date (does not contain timezone information)
time	Returns datetime.time (does not contain timezone information)
timetz	Returns datetime.time as local time with timezone information
dayofyear	The ordinal day of year
weekofyear	The week ordinal of the year
week	The week ordinal of the year
dayofweek	The number of the day of the week with Monday=0, Sunday=6
weekday	The number of the day of the week with Monday=0, Sunday=6
weekday_name	The name of the day in a week (ex: Friday)
quarter	Quarter of the date: Jan-Mar = 1, Apr-Jun = 2, etc.
days_in_month	The number of days in the month of the datetime
is_month_start	Logical indicating if first day of month (defined by frequency)
is_month_end	Logical indicating if last day of month (defined by frequency)
is_quarter_start	Logical indicating if first day of quarter (defined by frequency)
is_quarter_end	Logical indicating if last day of quarter (defined by frequency)
is_year_start	Logical indicating if first day of year (defined by frequency)
is_year_end	Logical indicating if last day of year (defined by frequency)
is_leap_year	Logical indicating if the date belongs to a leap year

Furthermore, if you have a `Series` with datetimelike values, then you can access these properties via the `.dt` accessor, as detailed in the section on *.dt accessors*.

4.13.9 DateOffset Objects

In the preceding examples, frequency strings (e.g. `'D'`) were used to specify a frequency that defined:

- how the date times in *DatetimeIndex* were spaced when using *date_range()*
- the frequency of a *Period* or *PeriodIndex*

These frequency strings map to a `DateOffset` object and its subclasses. A `DateOffset` is similar to a *Timedelta* that represents a duration of time but follows specific calendar duration rules. For example, a *Timedelta* day will always increment `datetimes` by 24 hours, while a `DateOffset` day will increment `datetimes` to the same time the next day whether a day represents 23, 24 or 25 hours due to daylight savings time. However, all `DateOffset` subclasses that are an hour or smaller (`Hour`, `Minute`, `Second`, `Milli`, `Micro`, `Nano`) behave like *Timedelta* and respect absolute time.

The basic `DateOffset` acts similar to `dateutil.relativedelta` (relativedelta documentation) that shifts a date time by the corresponding calendar duration specified. The arithmetic operator (+) or the `apply` method can be used to perform the shift.

```
# This particular day contains a day light savings time transition
In [139]: ts = pd.Timestamp('2016-10-30 00:00:00', tz='Europe/Helsinki')
```

(continues on next page)

(continued from previous page)

```
# Respects absolute time
In [140]: ts + pd.Timedelta(days=1)
Out[140]: Timestamp('2016-10-30 23:00:00+0200', tz='Europe/Helsinki')

# Respects calendar time
In [141]: ts + pd.DateOffset(days=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[141]:␣
↪Timestamp('2016-10-31 00:00:00+0200', tz='Europe/Helsinki')

In [142]: friday = pd.Timestamp('2018-01-05')

In [143]: friday.day_name()
Out[143]: 'Friday'

# Add 2 business days (Friday --> Tuesday)
In [144]: two_business_days = 2 * pd.offsets.BDay()

In [145]: two_business_days.apply(friday)
Out[145]: Timestamp('2018-01-09 00:00:00')

In [146]: friday + two_business_days
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[146]: Timestamp('2018-01-09 00:00:00')

In [147]: (friday + two_business_days).day_name()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[147]:␣
↪'Tuesday'
```

Most `DateOffsets` have associated frequencies strings, or offset aliases, that can be passed into `freq` keyword arguments. The available date offsets and associated frequency strings can be found below:

Date Offset	Frequency String	Description
DateOffset	None	Generic offset class, defaults to 1 calendar day
BDay or *BusinessDay*	'B'	business day (weekday)
CDay or *CustomBusinessDay*	'C'	custom business day
Week	'W'	one week, optionally anchored on a day of the week
WeekOfMonth	'WOM'	the x-th day of the y-th week of each month
LastWeekOfMonth	'LWOM'	the x-th day of the last week of each month
MonthEnd	'M'	calendar month end
MonthBegin	'MS'	calendar month begin
BMonthEnd or *BusinessMonthEnd*	'BM'	business month end
BMonthBegin or *BusinessMonthBegin*	'BMS'	business month begin
CBMonthEnd or *CustomBusinessMonthEnd*	'CBM'	custom business month end

Continued on next page

Table 3 – continued from previous page

Date Offset	Frequency String	Description
CBMonthBegin or *CustomBusinessMonthBegin*	'CBMS'	custom business month begin
SemiMonthEnd	'SM'	15th (or other day_of_month) and calendar month end
SemiMonthBegin	'SMS'	15th (or other day_of_month) and calendar month begin
QuarterEnd	'Q'	calendar quarter end
QuarterBegin	'QS'	calendar quarter begin
BQuarterEnd	'BQ	business quarter end
BQuarterBegin	'BQS'	business quarter begin
FY5253Quarter	'REQ'	retail (aka 52-53 week) quarter
YearEnd	'A'	calendar year end
YearBegin	'AS' or 'BYS'	calendar year begin
BYearEnd	'BA'	business year end
BYearBegin	'BAS'	business year begin
FY5253	'RE'	retail (aka 52-53 week) year
Easter	None	Easter holiday
BusinessHour	'BH'	business hour
CustomBusinessHour	'CBH'	custom business hour
Day	'D'	one absolute day
Hour	'H'	one hour
Minute	'T' or 'min'	one minute
Second	'S'	one second
Milli	'L' or 'ms'	one millisecond
Micro	'U' or 'us'	one microsecond
Nano	'N'	one nanosecond

`DateOffsets` additionally have `rollforward()` and `rollback()` methods for moving a date forward or backward respectively to a valid offset date relative to the offset. For example, business offsets will roll dates that land on the weekends (Saturday and Sunday) forward to Monday since business offsets operate on the weekdays.

```
In [148]: ts = pd.Timestamp('2018-01-06 00:00:00')

In [149]: ts.day_name()
Out[149]: 'Saturday'

# BusinessHour's valid offset dates are Monday through Friday
In [150]: offset = pd.offsets.BusinessHour(start='09:00')

# Bring the date to the closest offset date (Monday)
In [151]: offset.rollforward(ts)
Out[151]: Timestamp('2018-01-08 09:00:00')

# Date is brought to the closest offset date first and then the hour is added
In [152]: ts + offset
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[152]: Timestamp('2018-01-08 10:00:00')
```

These operations preserve time (hour, minute, etc) information by default. To reset time to midnight, use `normalize()` before or after applying the operation (depending on whether you want the time information included in the operation).

```
In [153]: ts = pd.Timestamp('2014-01-01 09:00')

In [154]: day = pd.offsets.Day()

In [155]: day.apply(ts)
Out[155]: Timestamp('2014-01-02 09:00:00')

In [156]: day.apply(ts).normalize()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[156]: Timestamp('2014-01-02 00:00:00')

In [157]: ts = pd.Timestamp('2014-01-01 22:00')

In [158]: hour = pd.offsets.Hour()

In [159]: hour.apply(ts)
Out[159]: Timestamp('2014-01-01 23:00:00')

In [160]: hour.apply(ts).normalize()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[160]: Timestamp('2014-01-01 00:00:00')

In [161]: hour.apply(pd.Timestamp("2014-01-01 23:30")).normalize()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[161]:␣
↪Timestamp('2014-01-02 00:00:00')
```

Parametric Offsets

Some of the offsets can be "parameterized" when created to result in different behaviors. For example, the `Week` offset for generating weekly data accepts a `weekday` parameter which results in the generated dates always lying on a particular day of the week:

```
In [162]: d = datetime.datetime(2008, 8, 18, 9, 0)

In [163]: d
Out[163]: datetime.datetime(2008, 8, 18, 9, 0)

In [164]: d + pd.offsets.Week()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[164]: Timestamp('2008-08-25␣
↪09:00:00')

In [165]: d + pd.offsets.Week(weekday=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[165]:␣
↪Timestamp('2008-08-22 09:00:00')

In [166]: (d + pd.offsets.Week(weekday=4)).weekday()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪4

In [167]: d - pd.offsets.Week()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2008-08-11 09:00:00')
```

The `normalize` option will be effective for addition and subtraction.

```
In [168]: d + pd.offsets.Week(normalize=True)
Out[168]: Timestamp('2008-08-25 00:00:00')
```

(continues on next page)

(continued from previous page)

```
In [169]: d - pd.offsets.Week(normalize=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[169]: Timestamp('2008-08-11 00:00:00')
```

Another example is parameterizing `YearEnd` with the specific ending month:

```
In [170]: d + pd.offsets.YearEnd()
Out[170]: Timestamp('2008-12-31 09:00:00')

In [171]: d + pd.offsets.YearEnd(month=6)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[171]: Timestamp('2009-06-30 09:00:00')
```

Using Offsets with `Series` / `DatetimeIndex`

Offsets can be used with either a `Series` or `DatetimeIndex` to apply the offset to each element.

```
In [172]: rng = pd.date_range('2012-01-01', '2012-01-03')

In [173]: s = pd.Series(rng)

In [174]: rng
Out[174]: DatetimeIndex(['2012-01-01', '2012-01-02', '2012-01-03'], dtype=
↪'datetime64[ns]', freq='D')

In [175]: rng + pd.DateOffset(months=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype='datetime64[ns]',␣
↪freq='D')

In [176]: s + pd.DateOffset(months=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-03-01
1   2012-03-02
2   2012-03-03
dtype: datetime64[ns]

In [177]: s - pd.DateOffset(months=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2011-11-01
1   2011-11-02
2   2011-11-03
dtype: datetime64[ns]
```

If the offset class maps directly to a `Timedelta` (`Day`, `Hour`, `Minute`, `Second`, `Micro`, `Milli`, `Nano`) it can be used exactly like a `Timedelta` - see the *Timedelta section* for more examples.

```
In [178]: s - pd.offsets.Day(2)
Out[178]:
0   2011-12-30
1   2011-12-31
2   2012-01-01
dtype: datetime64[ns]

In [179]: td = s - pd.Series(pd.date_range('2011-12-29', '2011-12-31'))
```

(continues on next page)

(continued from previous page)

```
In [180]: td
Out[180]:
0   3 days
1   3 days
2   3 days
dtype: timedelta64[ns]

In [181]: td + pd.offsets.Minute(15)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[181]:
0   3 days 00:15:00
1   3 days 00:15:00
2   3 days 00:15:00
dtype: timedelta64[ns]
```

Note that some offsets (such as `BQuarterEnd`) do not have a vectorized implementation. They can still be used but may calculate significantly slower and will show a `PerformanceWarning`

```
In [182]: rng + pd.offsets.BQuarterEnd()
Out[182]: DatetimeIndex(['2012-03-30', '2012-03-30', '2012-03-30'], dtype=
↪'datetime64[ns]', freq='D')
```

Custom Business Days

The `CDay` or `CustomBusinessDay` class provides a parametric `BusinessDay` class which can be used to create customized business day calendars which account for local holidays and local weekend conventions.

As an interesting example, let's look at Egypt where a Friday-Saturday weekend is observed.

```
In [183]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
# add that for a couple of years
In [184]: holidays = ['2012-05-01',
   .....:             datetime.datetime(2013, 5, 1),
   .....:             np.datetime64('2014-05-01')]
   .....:

In [185]: bday_egypt = pd.offsets.CustomBusinessDay(holidays=holidays,
   .....:                                           weekmask=weekmask_egypt)
   .....:

In [186]: dt = datetime.datetime(2013, 4, 30)

In [187]: dt + 2 * bday_egypt
Out[187]: Timestamp('2013-05-05 00:00:00')
```

Let's map to the weekday names:

```
In [188]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [189]: pd.Series(dts.weekday, dts).map(
   .....:     pd.Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
   .....:
Out[189]:
```

(continues on next page)

(continued from previous page)

```
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
Freq: C, dtype: object
```

Holiday calendars can be used to provide the list of holidays. See the *holiday calendar* section for more information.

```
In [190]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [191]: bday_us = pd.offsets.CustomBusinessDay(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [192]: dt = datetime.datetime(2014, 1, 17)

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [193]: dt + bday_us
Out[193]: Timestamp('2014-01-21 00:00:00')
```

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.

```
In [194]: bmth_us = pd.offsets.CustomBusinessMonthBegin(
   .....:     calendar=USFederalHolidayCalendar())
   .....:

# Skip new years
In [195]: dt = datetime.datetime(2013, 12, 17)

In [196]: dt + bmth_us
Out[196]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [197]: pd.date_range(start='20100101', end='20120101', freq=bmth_us)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[197]:
DatetimeIndex(['2010-01-04', '2010-02-01', '2010-03-01', '2010-04-01',
               '2010-05-03', '2010-06-01', '2010-07-01', '2010-08-02',
               '2010-09-01', '2010-10-01', '2010-11-01', '2010-12-01',
               '2011-01-03', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-05-02', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-01', '2011-10-03', '2011-11-01', '2011-12-01'],
              dtype='datetime64[ns]', freq='CBMS')
```

Note: The frequency string 'C' is used to indicate that a CustomBusinessDay DateOffset is used, it is important to note that since CustomBusinessDay is a parameterised type, instances of CustomBusinessDay may differ and this is not detectable from the 'C' frequency string. The user therefore needs to ensure that the 'C' frequency string is used consistently within the user's application.

Business Hour

The `BusinessHour` class provides a business hour representation on `BusinessDay`, allowing to use specific start and end times.

By default, `BusinessHour` uses 9:00 - 17:00 as business hours. Adding `BusinessHour` will increment

Timestamp by hourly frequency. If target Timestamp is out of business hours, move to the next business hour then increment it. If the result exceeds the business hours end, the remaining hours are added to the next business day.

```
In [198]: bh = pd.offsets.BusinessHour()

In [199]: bh
Out[199]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [200]: pd.Timestamp('2014-08-01 10:00').weekday()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[200]: 4

In [201]: pd.Timestamp('2014-08-01 10:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[201]: Timestamp('2014-08-01␣
↪11:00:00')

# Below example is the same as: pd.Timestamp('2014-08-01 09:00') + bh
In [202]: pd.Timestamp('2014-08-01 08:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2
↪Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
In [203]: pd.Timestamp('2014-08-01 16:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-04 09:00:00')

# Remainings are added to the next day
In [204]: pd.Timestamp('2014-08-01 16:30') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-04 09:30:00')

# Adding 2 business hours
In [205]: pd.Timestamp('2014-08-01 10:00') + pd.offsets.BusinessHour(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-01 12:00:00')

# Subtracting 3 business hours
In [206]: pd.Timestamp('2014-08-01 10:00') + pd.offsets.BusinessHour(-3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-07-31 15:00:00')
```

You can also specify start and end time by keywords. The argument must be a str with an hour:minute representation or a datetime.time instance. Specifying seconds, microseconds and nanoseconds as business hour results in ValueError.

```
In [207]: bh = pd.offsets.BusinessHour(start='11:00', end=datetime.time(20, 0))

In [208]: bh
Out[208]: <BusinessHour: BH=11:00-20:00>

In [209]: pd.Timestamp('2014-08-01 13:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[209]: Timestamp('2014-08-01 14:00:00')

In [210]: pd.Timestamp('2014-08-01 09:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[210]:␣
↪Timestamp('2014-08-01 12:00:00')

In [211]: pd.Timestamp('2014-08-01 18:00') + bh
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-01 19:00:00')
```

Passing `start` time later than `end` represents midnight business hour. In this case, business hour exceeds midnight and overlap to the next day. Valid business hours are distinguished by whether it started from valid `BusinessDay`.

```
In [212]: bh = pd.offsets.BusinessHour(start='17:00', end='09:00')

In [213]: bh
Out[213]: <BusinessHour: BH=17:00-09:00>

In [214]: pd.Timestamp('2014-08-01 17:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[214]: Timestamp('2014-08-01 18:00:00')

In [215]: pd.Timestamp('2014-08-01 23:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[215]:␣
↪Timestamp('2014-08-02 00:00:00')

# Although 2014-08-02 is Satuaday,
# it is valid because it starts from 08-01 (Friday).
In [216]: pd.Timestamp('2014-08-02 04:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-02 05:00:00')

# Although 2014-08-04 is Monday,
# it is out of business hours because it starts from 08-03 (Sunday).
In [217]: pd.Timestamp('2014-08-04 04:00') + bh
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-04 18:00:00')
```

Applying `BusinessHour.rollforward` and `rollback` to out of business hours results in the next business hour start or previous day's end. Different from other offsets, `BusinessHour.rollforward` may output different results from `apply` by definition.

This is because one day's business hour end is equal to next day's business hour start. For example, under the default business hours (9:00 - 17:00), there is no gap (0 minutes) between `2014-08-01 17:00` and `2014-08-04 09:00`.

```
# This adjusts a Timestamp to business hour edge
In [218]: pd.offsets.BusinessHour().rollback(pd.Timestamp('2014-08-02 15:00'))
Out[218]: Timestamp('2014-08-01 17:00:00')

In [219]: pd.offsets.BusinessHour().rollforward(pd.Timestamp('2014-08-02 15:00'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[219]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessHour().apply(pd.Timestamp('2014-08-01 17:00')).
# And it is the same as BusinessHour().apply(pd.Timestamp('2014-08-04 09:00'))
In [220]: pd.offsets.BusinessHour().apply(pd.Timestamp('2014-08-02 15:00'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[220]:␣
↪Timestamp('2014-08-04 10:00:00')

# BusinessDay results (for reference)
In [221]: pd.offsets.BusinessHour().rollforward(pd.Timestamp('2014-08-02'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-04 09:00:00')
```

(continues on next page)

(continued from previous page)

```
# It is the same as BusinessDay().apply(pd.Timestamp('2014-08-01'))
# The result is the same as rollworward because BusinessDay never overlap.
In [222]: pd.offsets.BusinessHour().apply(pd.Timestamp('2014-08-02'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-08-04 10:00:00')
```

BusinessHour regards Saturday and Sunday as holidays. To use arbitrary holidays, you can use CustomBusinessHour offset, as explained in the following subsection.

Custom Business Hour

New in version 0.18.1.

The CustomBusinessHour is a mixture of BusinessHour and CustomBusinessDay which allows you to specify arbitrary holidays. CustomBusinessHour works as the same as BusinessHour except that it skips specified custom holidays.

```
In [223]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [224]: bhour_us = pd.offsets.
↪CustomBusinessHour(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [225]: dt = datetime.datetime(2014, 1, 17, 15)

In [226]: dt + bhour_us
Out[226]: Timestamp('2014-01-17 16:00:00')

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [227]: dt + bhour_us * 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[227]: Timestamp('2014-01-21 09:00:00')
```

You can use keyword arguments supported by either BusinessHour and CustomBusinessDay.

```
In [228]: bhour_mon = pd.offsets.CustomBusinessHour(start='10:00',
   .....:                                           weekmask='Tue Wed Thu Fri')
   .....:

# Monday is skipped because it's a holiday, business hour starts from 10:00
In [229]: dt + bhour_mon * 2
Out[229]: Timestamp('2014-01-21 10:00:00')
```

Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as *offset aliases*.

Alias	Description
B	business day frequency
C	custom business day frequency
D	calendar day frequency
W	weekly frequency
M	month end frequency
SM	semi-month end frequency (15th and end of month)
BM	business month end frequency
CBM	custom business month end frequency
MS	month start frequency
SMS	semi-month start frequency (1st and 15th)
BMS	business month start frequency
CBMS	custom business month start frequency
Q	quarter end frequency
BQ	business quarter end frequency
QS	quarter start frequency
BQS	business quarter start frequency
A, Y	year end frequency
BA, BY	business year end frequency
AS, YS	year start frequency
BAS, BYS	business year start frequency
BH	business hour frequency
H	hourly frequency
T, min	minutely frequency
S	secondly frequency
L, ms	milliseconds
U, us	microseconds
N	nanoseconds

Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```
In [230]: pd.date_range(start, periods=5, freq='B')
Out[230]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B')

In [231]: pd.date_range(start, periods=5, freq=pd.offsets.BDay())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B')
```

You can combine together day and intraday offsets:

```
In [232]: pd.date_range(start, periods=10, freq='2h20min')
Out[232]:
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 02:20:00',
               '2011-01-01 04:40:00', '2011-01-01 07:00:00',
```

(continues on next page)

(continued from previous page)

```
                 '2011-01-01 09:20:00', '2011-01-01 11:40:00',
                 '2011-01-01 14:00:00', '2011-01-01 16:20:00',
                 '2011-01-01 18:40:00', '2011-01-01 21:00:00'],
                dtype='datetime64[ns]', freq='140T')

In [233]: pd.date_range(start, periods=10, freq='1D10U')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex([        '2011-01-01 00:00:00', '2011-01-02 00:00:00.000010',
                 '2011-01-03 00:00:00.000020', '2011-01-04 00:00:00.000030',
                 '2011-01-05 00:00:00.000040', '2011-01-06 00:00:00.000050',
                 '2011-01-07 00:00:00.000060', '2011-01-08 00:00:00.000070',
                 '2011-01-09 00:00:00.000080', '2011-01-10 00:00:00.000090'],
                dtype='datetime64[ns]', freq='86400000010U')
```

Anchored Offsets

For some frequencies you can specify an anchoring suffix:

Alias	Description
W-SUN	weekly frequency (Sundays). Same as 'W'
W-MON	weekly frequency (Mondays)
W-TUE	weekly frequency (Tuesdays)
W-WED	weekly frequency (Wednesdays)
W-THU	weekly frequency (Thursdays)
W-FRI	weekly frequency (Fridays)
W-SAT	weekly frequency (Saturdays)
(B)Q(S)-DEC	quarterly frequency, year ends in December. Same as 'Q'
(B)Q(S)-JAN	quarterly frequency, year ends in January
(B)Q(S)-FEB	quarterly frequency, year ends in February
(B)Q(S)-MAR	quarterly frequency, year ends in March
(B)Q(S)-APR	quarterly frequency, year ends in April
(B)Q(S)-MAY	quarterly frequency, year ends in May
(B)Q(S)-JUN	quarterly frequency, year ends in June
(B)Q(S)-JUL	quarterly frequency, year ends in July
(B)Q(S)-AUG	quarterly frequency, year ends in August
(B)Q(S)-SEP	quarterly frequency, year ends in September
(B)Q(S)-OCT	quarterly frequency, year ends in October
(B)Q(S)-NOV	quarterly frequency, year ends in November

Continued on next page

Table 4 – continued from previous page

Alias	Description
(B)A(S)-DEC	annual frequency, anchored end of December. Same as 'A'
(B)A(S)-JAN	annual frequency, anchored end of January
(B)A(S)-FEB	annual frequency, anchored end of February
(B)A(S)-MAR	annual frequency, anchored end of March
(B)A(S)-APR	annual frequency, anchored end of April
(B)A(S)-MAY	annual frequency, anchored end of May
(B)A(S)-JUN	annual frequency, anchored end of June
(B)A(S)-JUL	annual frequency, anchored end of July
(B)A(S)-AUG	annual frequency, anchored end of August
(B)A(S)-SEP	annual frequency, anchored end of September
(B)A(S)-OCT	annual frequency, anchored end of October
(B)A(S)-NOV	annual frequency, anchored end of November

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

Anchored Offset Semantics

For those offsets that are anchored to the start or end of specific frequency (`MonthEnd`, `MonthBegin`, `WeekEnd`, etc), the following rules apply to rolling forward and backwards.

When `n` is not 0, if the given date is not on an anchor point, it snapped to the next(previous) anchor point, and moved `|n|-1` additional steps forwards or backwards.

```
In [234]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=1)
Out[234]: Timestamp('2014-02-01 00:00:00')

In [235]: pd.Timestamp('2014-01-02') + pd.offsets.MonthEnd(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[235]: Timestamp('2014-01-31 00:00:00')

In [236]: pd.Timestamp('2014-01-02') - pd.offsets.MonthBegin(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[236]:
↪Timestamp('2014-01-01 00:00:00')

In [237]: pd.Timestamp('2014-01-02') - pd.offsets.MonthEnd(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-12-31 00:00:00')

In [238]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-05-01 00:00:00')
```

(continues on next page)

(continued from previous page)

```
In [239]: pd.Timestamp('2014-01-02') - pd.offsets.MonthBegin(n=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-10-01 00:00:00')
```

If the given date *is* on an anchor point, it is moved `|n|` points forwards or backwards.

```
In [240]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=1)
Out[240]: Timestamp('2014-02-01 00:00:00')

In [241]: pd.Timestamp('2014-01-31') + pd.offsets.MonthEnd(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[241]: Timestamp('2014-02-28 00:00:00')

In [242]: pd.Timestamp('2014-01-01') - pd.offsets.MonthBegin(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[242]:␣
↪Timestamp('2013-12-01 00:00:00')

In [243]: pd.Timestamp('2014-01-31') - pd.offsets.MonthEnd(n=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-12-31 00:00:00')

In [244]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-05-01 00:00:00')

In [245]: pd.Timestamp('2014-01-31') - pd.offsets.MonthBegin(n=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-10-01 00:00:00')
```

For the case when `n=0`, the date is not moved if on an anchor point, otherwise it is rolled forward to the next anchor point.

```
In [246]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=0)
Out[246]: Timestamp('2014-02-01 00:00:00')

In [247]: pd.Timestamp('2014-01-02') + pd.offsets.MonthEnd(n=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[247]: Timestamp('2014-01-31 00:00:00')

In [248]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[248]:␣
↪Timestamp('2014-01-01 00:00:00')

In [249]: pd.Timestamp('2014-01-31') + pd.offsets.MonthEnd(n=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2014-01-31 00:00:00')
```

Holidays / Holiday Calendars

Holidays and calendars provide a simple way to define holiday rules to be used with `CustomBusinessDay` or in other analysis that requires a predefined set of holidays. The `AbstractHolidayCalendar` class provides all the necessary methods to return a list of holidays and only `rules` need to be defined in a specific holiday calendar class. Furthermore, the `start_date` and `end_date` class attributes determine over what date range holidays are generated. These should be overwritten on the `AbstractHolidayCalendar` class to have the range apply to all calendar subclasses. `USFederalHolidayCalendar` is the only calendar that exists and primarily serves as an example for developing other calendars.

For holidays that occur on fixed dates (e.g., US Memorial Day or July 4th) an observance rule determines when that holiday is observed if it falls on a weekend or some other non-observed day. Defined observance rules are:

Rule	Description
nearest_workday	move Saturday to Friday and Sunday to Monday
sunday_to_monday	move Sunday to following Monday
next_monday_or_tuesday	move Saturday to Monday and Sunday/Monday to Tuesday
previous_friday	move Saturday and Sunday to previous Friday”
next_monday	move Saturday and Sunday to following Monday

An example of how holidays and holiday calendars are defined:

```
In [250]: from pandas.tseries.holiday import Holiday, USMemorialDay,\
   .....:     AbstractHolidayCalendar, nearest_workday, MO
   .....:

In [251]: class ExampleCalendar(AbstractHolidayCalendar):
   .....:     rules = [
   .....:         USMemorialDay,
   .....:         Holiday('July 4th', month=7, day=4, observance=nearest_workday),
   .....:         Holiday('Columbus Day', month=10, day=1,
   .....:                 offset=pd.DateOffset(weekday=MO(2)))]
   .....:

In [252]: cal = ExampleCalendar()

In [253]: cal.holidays(datetime.datetime(2012, 1, 1), datetime.datetime(2012, 12, 31))
Out[253]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype=
↪'datetime64[ns]', freq=None)
```

hint weekday=MO(2) is same as **2 * Week(weekday=2)**

Using this calendar, creating an index or doing offset arithmetic skips weekends and holidays (i.e., Memorial Day/July 4th). For example, the below defines a custom business day offset using the `ExampleCalendar`. Like any other offset, it can be used to create a `DatetimeIndex` or added to `datetime` or `Timestamp` objects.

```
In [254]: pd.date_range(start='7/1/2012', end='7/10/2012',
   .....:               freq=pd.offsets.CDay(calendar=cal)).to_pydatetime()
   .....:
Out[254]:
array([datetime.datetime(2012, 7, 2, 0, 0),
       datetime.datetime(2012, 7, 3, 0, 0),
       datetime.datetime(2012, 7, 5, 0, 0),
       datetime.datetime(2012, 7, 6, 0, 0),
       datetime.datetime(2012, 7, 9, 0, 0),
       datetime.datetime(2012, 7, 10, 0, 0)], dtype=object)

In [255]: offset = pd.offsets.CustomBusinessDay(calendar=cal)

In [256]: datetime.datetime(2012, 5, 25) + offset
Out[256]: Timestamp('2012-05-29 00:00:00')

In [257]: datetime.datetime(2012, 7, 3) + offset
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[257]: Timestamp('2012-07-05 00:00:00')

In [258]: datetime.datetime(2012, 7, 3) + 2 * offset
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[258]:␣
↪Timestamp('2012-07-06 00:00:00')
```

(continues on next page)

(continued from previous page)

```
In [259]: datetime.datetime(2012, 7, 6) + offset
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2012-07-09 00:00:00')
```

Ranges are defined by the `start_date` and `end_date` class attributes of `AbstractHolidayCalendar`. The defaults are shown below.

```
In [260]: AbstractHolidayCalendar.start_date
Out[260]: Timestamp('1970-01-01 00:00:00')

In [261]: AbstractHolidayCalendar.end_date
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[261]: Timestamp('2030-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as datetime/Timestamp/string.

```
In [262]: AbstractHolidayCalendar.start_date = datetime.datetime(2012, 1, 1)

In [263]: AbstractHolidayCalendar.end_date = datetime.datetime(2012, 12, 31)

In [264]: cal.holidays()
Out[264]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype=
↪'datetime64[ns]', freq=None)
```

Every calendar class is accessible by name using the `get_calendar` function which returns a holiday class instance. Any imported calendar class will automatically be available by this function. Also, `HolidayCalendarFactory` provides an easy interface to create calendars that are combinations of calendars or calendars with additional rules.

```
In [265]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory,\
   .....:     USLaborDay
   .....:

In [266]: cal = get_calendar('ExampleCalendar')

In [267]: cal.rules
Out[267]: 
[Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at␣
↪0x7f37e6dd8048>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]

In [268]: new_cal = HolidayCalendarFactory('NewExampleCalendar', cal, USLaborDay)

In [269]: new_cal.rules
Out[269]: 
[Holiday: Labor Day (month=9, day=1, offset=<DateOffset: weekday=MO(+1)>),
 Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at␣
↪0x7f37e6dd8048>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]
```

4.13.10 Time Series-Related Instance Methods

Shifting / Lagging

One may want to *shift* or *lag* the values in a time series back and forward in time. The method for this is `shift()`, which is available on all of the pandas objects.

```
In [270]: ts = pd.Series(range(len(rng)), index=rng)

In [271]: ts = ts[:5]

In [272]: ts.shift(1)
Out[272]:
2012-01-01    NaN
2012-01-02    0.0
2012-01-03    1.0
Freq: D, dtype: float64
```

The `shift` method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object or also an *offset alias*:

```
In [273]: ts.shift(5, freq=pd.offsets.BDay())
Out[273]:
2012-01-06    0
2012-01-09    1
2012-01-10    2
Freq: B, dtype: int64

In [274]: ts.shift(5, freq='BM')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[274]:
↪
2012-05-31    0
2012-05-31    1
2012-05-31    2
Freq: D, dtype: int64
```

Rather than changing the alignment of the data and the index, `DataFrame` and `Series` objects also have a `tshift()` convenience method that changes all the dates in the index by a specified number of offsets:

```
In [275]: ts.tshift(5, freq='D')
Out[275]:
2012-01-06    0
2012-01-07    1
2012-01-08    2
Freq: D, dtype: int64
```

Note that with `tshift`, the leading entry is no longer NaN because the data is not being realigned.

Frequency Conversion

The primary function for changing frequencies is the `asfreq()` method. For a `DatetimeIndex`, this is basically just a thin, but convenient wrapper around `reindex()` which generates a `date_range` and calls `reindex`.

```
In [276]: dr = pd.date_range('1/1/2010', periods=3, freq=3 * pd.offsets.BDay())

In [277]: ts = pd.Series(np.random.randn(3), index=dr)

In [278]: ts
```

(continues on next page)

(continued from previous page)

```
Out[278]:
2010-01-01    1.494522
2010-01-06   -0.778425
2010-01-11   -0.253355
Freq: 3B, dtype: float64

In [279]: ts.asfreq(pd.offsets.BDay())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2010-01-01    1.494522
2010-01-04         NaN
2010-01-05         NaN
2010-01-06   -0.778425
2010-01-07         NaN
2010-01-08         NaN
2010-01-11   -0.253355
Freq: B, dtype: float64
```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion.

```
In [280]: ts.asfreq(pd.offsets.BDay(), method='pad')
Out[280]:
2010-01-01    1.494522
2010-01-04    1.494522
2010-01-05    1.494522
2010-01-06   -0.778425
2010-01-07   -0.778425
2010-01-08   -0.778425
2010-01-11   -0.253355
Freq: B, dtype: float64
```

Filling Forward / Backward

Related to `asfreq` and `reindex` is *fillna()*, which is documented in the *missing data section*.

Converting to Python Datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.

4.13.11 Resampling

> **Warning:** The interface to `.resample` has changed in 0.18.0 to be more groupby-like and hence more flexible. See the *whatsnew docs* for a comparison with prior versions.

Pandas has a simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications.

`resample()` is a time-based groupby, followed by a reduction method on each of its groups. See some *cookbook examples* for some advanced strategies.

Starting in version 0.18.1, the `resample()` function can be used directly from `DataFrameGroupBy` objects, see the *groupby docs*.

Note: `.resample()` is similar to using a `rolling()` operation with a time-based offset, see a discussion *here*.

Basics

```
In [281]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [282]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [283]: ts.resample('5Min').sum()
Out[283]:
2012-01-01    25103
Freq: 5T, dtype: int64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency conversion and resampling operation.

Any function available via *dispatching* is available as a method of the returned object, including `sum`, `mean`, `std`, `sem`, `max`, `min`, `median`, `first`, `last`, `ohlc`:

```
In [284]: ts.resample('5Min').mean()
Out[284]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [285]: ts.resample('5Min').ohlc()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[285]:
            open  high  low  close
2012-01-01   308   460    9    205

In [286]: ts.resample('5Min').max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01    460
Freq: 5T, dtype: int64
```

For downsampling, `closed` can be set to 'left' or 'right' to specify which end of the interval is closed:

```
In [287]: ts.resample('5Min', closed='right').mean()
Out[287]:
2011-12-31 23:55:00    308.000000
2012-01-01 00:00:00    250.454545
Freq: 5T, dtype: float64

In [288]: ts.resample('5Min', closed='left').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01    251.03
Freq: 5T, dtype: float64
```

Parameters like `label` and `loffset` are used to manipulate the resulting labels. `label` specifies whether the result is labeled with the beginning or the end of the interval. `loffset` performs a time adjustment on the output labels.

```
In [289]: ts.resample('5Min').mean()  # by default label='left'
Out[289]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [290]: ts.resample('5Min', label='left').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[290]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [291]: ts.resample('5Min', label='left', loffset='1s').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01 00:00:01    251.03
dtype: float64
```

Note: The default values for `label` and `closed` is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

```
In [292]: rng2 = pd.date_range('1/1/2012', end='3/31/2012', freq='D')

In [293]: ts2 = pd.Series(range(len(rng2)), index=rng2)

# default: label='right', closed='right'
In [294]: ts2.resample('M').max()
Out[294]:
2012-01-31    30
2012-02-29    59
2012-03-31    90
Freq: M, dtype: int64

# default: label='left', closed='left'
In [295]: ts2.resample('SM').max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[295]:␣
↪
2011-12-31    13
2012-01-15    29
2012-01-31    44
2012-02-15    58
2012-02-29    73
2012-03-15    89
2012-03-31    90
Freq: SM-15, dtype: int64

In [296]: ts2.resample('SM', label='right', closed='right').max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-15    14.0
2012-01-31    30.0
2012-02-15    45.0
2012-02-29    59.0
2012-03-15    74.0
2012-03-31    90.0
2012-04-15     NaN
```

(continues on next page)

(continued from previous page)

```
Freq: SM-15, dtype: float64
```

The `axis` parameter can be set to 0 or 1 and allows you to resample the specified axis for a `DataFrame`.

`kind` can be set to 'timestamp' or 'period' to convert the resulting index to/from timestamp and time span representations. By default `resample` retains the input representation.

`convention` can be set to 'start' or 'end' when resampling period data (detail below). It specifies how low frequency periods are converted to higher frequency periods.

Upsampling

For upsampling, you can specify a way to upsample and the `limit` parameter to interpolate over the gaps that are created:

```
# from secondly to every 250 milliseconds
In [297]: ts[:2].resample('250L').asfreq()
Out[297]:
2012-01-01 00:00:00.000    308.0
2012-01-01 00:00:00.250      NaN
2012-01-01 00:00:00.500      NaN
2012-01-01 00:00:00.750      NaN
2012-01-01 00:00:01.000    204.0
Freq: 250L, dtype: float64

In [298]: ts[:2].resample('250L').ffill()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01 00:00:00.000    308
2012-01-01 00:00:00.250    308
2012-01-01 00:00:00.500    308
2012-01-01 00:00:00.750    308
2012-01-01 00:00:01.000    204
Freq: 250L, dtype: int64

In [299]: ts[:2].resample('250L').ffill(limit=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01 00:00:00.000    308.0
2012-01-01 00:00:00.250    308.0
2012-01-01 00:00:00.500    308.0
2012-01-01 00:00:00.750      NaN
2012-01-01 00:00:01.000    204.0
Freq: 250L, dtype: float64
```

Sparse Resampling

Sparse timeseries are the ones where you have a lot fewer points relative to the amount of time you are looking to resample. Naively upsampling a sparse series can potentially generate lots of intermediate values. When you don't want to use a method to fill these values, e.g. `fill_method` is `None`, then intermediate values will be filled with `NaN`.

Since `resample` is a time-based groupby, the following is a method to efficiently resample only the groups that are not all `NaN`.

```
In [300]: rng = pd.date_range('2014-1-1', periods=100, freq='D') + pd.Timedelta('1s')

In [301]: ts = pd.Series(range(100), index=rng)
```

If we want to resample to the full range of the series:

```
In [302]: ts.resample('3T').sum()
Out[302]:
2014-01-01 00:00:00     0
2014-01-01 00:03:00     0
2014-01-01 00:06:00     0
2014-01-01 00:09:00     0
2014-01-01 00:12:00     0
2014-01-01 00:15:00     0
2014-01-01 00:18:00     0
                       ..
2014-04-09 23:42:00     0
2014-04-09 23:45:00     0
2014-04-09 23:48:00     0
2014-04-09 23:51:00     0
2014-04-09 23:54:00     0
2014-04-09 23:57:00     0
2014-04-10 00:00:00    99
Freq: 3T, Length: 47521, dtype: int64
```

We can instead only resample those groups where we have points as follows:

```
In [303]: from functools import partial

In [304]: from pandas.tseries.frequencies import to_offset

In [305]: def round(t, freq):
   .....:     freq = to_offset(freq)
   .....:     return pd.Timestamp((t.value // freq.delta.value) * freq.delta.value)
   .....:

In [306]: ts.groupby(partial(round, freq='3T')).sum()
Out[306]:
2014-01-01     0
2014-01-02     1
2014-01-03     2
2014-01-04     3
2014-01-05     4
2014-01-06     5
2014-01-07     6
              ..
2014-04-04    93
2014-04-05    94
2014-04-06    95
2014-04-07    96
2014-04-08    97
2014-04-09    98
2014-04-10    99
Length: 100, dtype: int64
```

Aggregation

Similar to the *aggregating API*, *groupby API*, and the *window functions API*, a `Resampler` can be selectively resampled.

Resampling a `DataFrame`, the default will be to act on all columns with the same function.

```
In [307]: df = pd.DataFrame(np.random.randn(1000, 3),
   .....:                   index=pd.date_range('1/1/2012', freq='S', periods=1000),
   .....:                   columns=['A', 'B', 'C'])
   .....:

In [308]: r = df.resample('3T')

In [309]: r.mean()
Out[309]:
                            A         B         C
2012-01-01 00:00:00 -0.033823 -0.121514 -0.081447
2012-01-01 00:03:00  0.056909  0.146731 -0.024320
2012-01-01 00:06:00 -0.058837  0.047046 -0.052021
2012-01-01 00:09:00  0.063123 -0.026158 -0.066533
2012-01-01 00:12:00  0.186340 -0.003144  0.074752
2012-01-01 00:15:00 -0.085954 -0.016287 -0.050046
```

We can select a specific column or columns using standard getitem.

```
In [310]: r['A'].mean()
Out[310]:
2012-01-01 00:00:00   -0.033823
2012-01-01 00:03:00    0.056909
2012-01-01 00:06:00   -0.058837
2012-01-01 00:09:00    0.063123
2012-01-01 00:12:00    0.186340
2012-01-01 00:15:00   -0.085954
Freq: 3T, Name: A, dtype: float64

In [311]: r[['A', 'B']].mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                            A         B
2012-01-01 00:00:00 -0.033823 -0.121514
2012-01-01 00:03:00  0.056909  0.146731
2012-01-01 00:06:00 -0.058837  0.047046
2012-01-01 00:09:00  0.063123 -0.026158
2012-01-01 00:12:00  0.186340 -0.003144
2012-01-01 00:15:00 -0.085954 -0.016287
```

You can pass a list or dict of functions to do aggregation with, outputting a `DataFrame`:

```
In [312]: r['A'].agg([np.sum, np.mean, np.std])
Out[312]:
                           sum      mean       std
2012-01-01 00:00:00  -6.088060 -0.033823  1.043263
2012-01-01 00:03:00  10.243678  0.056909  1.058534
2012-01-01 00:06:00 -10.590584 -0.058837  0.949264
2012-01-01 00:09:00  11.362228  0.063123  1.028096
2012-01-01 00:12:00  33.541257  0.186340  0.884586
2012-01-01 00:15:00  -8.595393 -0.085954  1.035476
```

On a resampled `DataFrame`, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [313]: r.agg([np.sum, np.mean])
Out[313]:
                             A                     B                     C
                           sum      mean       sum      mean       sum      mean
2012-01-01 00:00:00  -6.088060 -0.033823 -21.872530 -0.121514 -14.660515 -0.081447
2012-01-01 00:03:00  10.243678  0.056909  26.411633  0.146731  -4.377642 -0.024320
2012-01-01 00:06:00 -10.590584 -0.058837   8.468289  0.047046  -9.363825 -0.052021
2012-01-01 00:09:00  11.362228  0.063123  -4.708526 -0.026158 -11.975895 -0.066533
2012-01-01 00:12:00  33.541257  0.186340  -0.565895 -0.003144  13.455299  0.074752
2012-01-01 00:15:00  -8.595393 -0.085954  -1.628689 -0.016287  -5.004580 -0.050046
```

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a `DataFrame`:

```
In [314]: r.agg({'A': np.sum,
   .....:        'B': lambda x: np.std(x, ddof=1)})
   .....:
Out[314]:
                             A         B
2012-01-01 00:00:00  -6.088060  1.001294
2012-01-01 00:03:00  10.243678  1.074597
2012-01-01 00:06:00 -10.590584  0.987309
2012-01-01 00:09:00  11.362228  0.944953
2012-01-01 00:12:00  33.541257  1.095025
2012-01-01 00:15:00  -8.595393  1.035312
```

The function names can also be strings. In order for a string to be valid it must be implemented on the resampled object:

```
In [315]: r.agg({'A': 'sum', 'B': 'std'})
Out[315]:
                             A         B
2012-01-01 00:00:00  -6.088060  1.001294
2012-01-01 00:03:00  10.243678  1.074597
2012-01-01 00:06:00 -10.590584  0.987309
2012-01-01 00:09:00  11.362228  0.944953
2012-01-01 00:12:00  33.541257  1.095025
2012-01-01 00:15:00  -8.595393  1.035312
```

Furthermore, you can also specify multiple aggregation functions for each column separately.

```
In [316]: r.agg({'A': ['sum', 'std'], 'B': ['mean', 'std']})
Out[316]:
                             A                   B
                           sum       std      mean       std
2012-01-01 00:00:00  -6.088060  1.043263 -0.121514  1.001294
2012-01-01 00:03:00  10.243678  1.058534  0.146731  1.074597
2012-01-01 00:06:00 -10.590584  0.949264  0.047046  0.987309
2012-01-01 00:09:00  11.362228  1.028096 -0.026158  0.944953
2012-01-01 00:12:00  33.541257  0.884586 -0.003144  1.095025
2012-01-01 00:15:00  -8.595393  1.035476 -0.016287  1.035312
```

If a `DataFrame` does not have a datetimelike index, but instead you want to resample based on datetimelike column in the frame, it can passed to the `on` keyword.

```
In [317]: 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 [318]: df
Out[318]:
                   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

In [319]: df.resample('M', on='date').sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            a
date
2015-01-31  6
2015-02-28  4
```

Similarly, if you instead want to resample by a datetimelike level of `MultiIndex`, its name or location can be passed to the `level` keyword.

```
In [320]: df.resample('M', level='d').sum()
Out[320]:
            a
d
2015-01-31  6
2015-02-28  4
```

4.13.12 Time Span Representation

Regular intervals of time are represented by `Period` objects in pandas while sequences of `Period` objects are collected in a `PeriodIndex`, which can be created with the convenience function `period_range`.

Period

A `Period` represents a span of time (e.g., a day, a month, a quarter, etc). You can specify the span via `freq` keyword using a frequency alias like below. Because `freq` represents a span of `Period`, it cannot be negative like "-3D".

```
In [321]: pd.Period('2012', freq='A-DEC')
Out[321]: Period('2012', 'A-DEC')

In [322]: pd.Period('2012-1-1', freq='D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[322]: Period('2012-01-01', 'D')

In [323]: pd.Period('2012-1-1 19:00', freq='H')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[323]:␣
↪Period('2012-01-01 19:00', 'H')
```

(continues on next page)

(continued from previous page)

```
In [324]: pd.Period('2012-1-1 19:00', freq='5H')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Period('2012-01-01 19:00', '5H')
```

Adding and subtracting integers from periods shifts the period by its own frequency. Arithmetic is not allowed between `Period` with different `freq` (span).

```
In [325]: p = pd.Period('2012', freq='A-DEC')

In [326]: p + 1
Out[326]: Period('2013', 'A-DEC')

In [327]: p - 3
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[327]: Period('2009', 'A-DEC')

In [328]: p = pd.Period('2012-01', freq='2M')

In [329]: p + 2
Out[329]: Period('2012-05', '2M')

In [330]: p - 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[330]: Period('2011-11', '2M')

In [331]: p == pd.Period('2012-01', freq='3M')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\-------------------
↪---------------------------------------------------------
IncompatibleFrequency                     Traceback (most recent call last)
<ipython-input-331-4b67dc0b596c> in <module>
----> 1 p == pd.Period('2012-01', freq='3M')

/pandas/pandas/_libs/tslibs/period.pyx in pandas._libs.tslibs.period._Period.__
↪richcmp__()

IncompatibleFrequency: Input has different freq=3M from Period(freq=2M)
```

If `Period` freq is daily or higher (`D`, `H`, `T`, `S`, `L`, `U`, `N`), `offsets` and `timedelta`-like can be added if the result can have the same freq. Otherwise, `ValueError` will be raised.

```
In [332]: p = pd.Period('2014-07-01 09:00', freq='H')

In [333]: p + pd.offsets.Hour(2)
Out[333]: Period('2014-07-01 11:00', 'H')

In [334]: p + datetime.timedelta(minutes=120)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[334]: Period('2014-07-01 11:00', 'H')

In [335]: p + np.timedelta64(7200, 's')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[335]:␣
↪Period('2014-07-01 11:00', 'H')
```

```
In [1]: p + pd.offsets.Minute(5)
Traceback
   ...
ValueError: Input has different freq from Period(freq=H)
```

If `Period` has other frequencies, only the same `offsets` can be added. Otherwise, `ValueError` will be raised.

```
In [336]: p = pd.Period('2014-07', freq='M')

In [337]: p + pd.offsets.MonthEnd(3)
Out[337]: Period('2014-10', 'M')
```

```
In [1]: p + pd.offsets.MonthBegin(3)
Traceback
   ...
ValueError: Input has different freq from Period(freq=M)
```

Taking the difference of `Period` instances with the same frequency will return the number of frequency units between them:

```
In [338]: pd.Period('2012', freq='A-DEC') - pd.Period('2002', freq='A-DEC')
Out[338]: <10 * YearEnds: month=12>
```

PeriodIndex and period_range

Regular sequences of `Period` objects can be collected in a `PeriodIndex`, which can be constructed using the `period_range` convenience function:

```
In [339]: prng = pd.period_range('1/1/2011', '1/1/2012', freq='M')

In [340]: prng
Out[340]:
PeriodIndex(['2011-01', '2011-02', '2011-03', '2011-04', '2011-05', '2011-06',
             '2011-07', '2011-08', '2011-09', '2011-10', '2011-11', '2011-12',
             '2012-01'],
            dtype='period[M]', freq='M')
```

The `PeriodIndex` constructor can also be used directly:

```
In [341]: pd.PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[341]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```

Passing multiplied frequency outputs a sequence of `Period` which has multiplied span.

```
In [342]: pd.period_range(start='2014-01', freq='3M', periods=4)
Out[342]: PeriodIndex(['2014-01', '2014-04', '2014-07', '2014-10'], dtype='period[3M]
↪', freq='3M')
```

If `start` or `end` are `Period` objects, they will be used as anchor endpoints for a `PeriodIndex` with frequency matching that of the `PeriodIndex` constructor.

```
In [343]: pd.period_range(start=pd.Period('2017Q1', freq='Q'),
   .....:                 end=pd.Period('2017Q2', freq='Q'), freq='M')
   .....:
Out[343]: PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'], dtype='period[M]',
↪ freq='M')
```

Just like `DatetimeIndex`, a `PeriodIndex` can also be used to index pandas objects:

```
In [344]: ps = pd.Series(np.random.randn(len(prng)), prng)

In [345]: ps
```

(continues on next page)

(continued from previous page)

```
Out[345]:
2011-01   -2.916901
2011-02    0.514474
2011-03    1.346470
2011-04    0.816397
2011-05    2.258648
2011-06    0.494789
2011-07    0.301239
2011-08    0.464776
2011-09   -1.393581
2011-10    0.056780
2011-11    0.197035
2011-12    2.261385
2012-01   -0.329583
Freq: M, dtype: float64
```

`PeriodIndex` supports addition and subtraction with the same rule as `Period`.

```
In [346]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [347]: idx
Out[347]:
PeriodIndex(['2014-07-01 09:00', '2014-07-01 10:00', '2014-07-01 11:00',
             '2014-07-01 12:00', '2014-07-01 13:00'],
            dtype='period[H]', freq='H')

In [348]: idx + pd.offsets.Hour(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
             '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]', freq='H')

In [349]: idx = pd.period_range('2014-07', periods=5, freq='M')

In [350]: idx
Out[350]: PeriodIndex(['2014-07', '2014-08', '2014-09', '2014-10', '2014-11'], dtype=
↪'period[M]', freq='M')

In [351]: idx + pd.offsets.MonthEnd(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪PeriodIndex(['2014-10', '2014-11', '2014-12', '2015-01', '2015-02'], dtype=
↪'period[M]', freq='M')
```

`PeriodIndex` has its own dtype named `period`, refer to *Period Dtypes*.

Period Dtypes

New in version 0.19.0.

`PeriodIndex` has a custom `period` dtype. This is a pandas extension dtype similar to the *timezone aware dtype* (`datetime64[ns, tz]`).

The `period` dtype holds the `freq` attribute and is represented with `period[freq]` like `period[D]` or `period[M]`, using *frequency strings*.

```
In [352]: pi = pd.period_range('2016-01-01', periods=3, freq='M')

In [353]: pi
Out[353]: PeriodIndex(['2016-01', '2016-02', '2016-03'], dtype='period[M]', freq='M')

In [354]: pi.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[354]:
↪period[M]
```

The `period` dtype can be used in `.astype(...)`. It allows one to change the `freq` of a `PeriodIndex` like `.asfreq()` and convert a `DatetimeIndex` to `PeriodIndex` like `to_period()`:

```
# change monthly freq to daily freq
In [355]: pi.astype('period[D]')
Out[355]: PeriodIndex(['2016-01-31', '2016-02-29', '2016-03-31'], dtype='period[D]',
↪freq='D')

# convert to DatetimeIndex
In [356]: pi.astype('datetime64[ns]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[35
↪DatetimeIndex(['2016-01-01', '2016-02-01', '2016-03-01'], dtype='datetime64[ns]',
↪freq='MS')

# convert to PeriodIndex
In [357]: dti = pd.date_range('2011-01-01', freq='M', periods=3)

In [358]: dti
Out[358]: DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31'], dtype=
↪'datetime64[ns]', freq='M')

In [359]: dti.astype('period[M]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```

PeriodIndex Partial String Indexing

You can pass in dates and strings to `Series` and `DataFrame` with `PeriodIndex`, in the same manner as `DatetimeIndex`. For details, refer to *DatetimeIndex Partial String Indexing*.

```
In [360]: ps['2011-01']
Out[360]: -2.9169013294054507

In [361]: ps[datetime.datetime(2011, 12, 25):]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[361]:
2011-12    2.261385
2012-01   -0.329583
Freq: M, dtype: float64

In [362]: ps['10/31/2011':'12/31/2011']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2011-10    0.056780
2011-11    0.197035
2011-12    2.261385
Freq: M, dtype: float64
```

Passing a string representing a lower frequency than `PeriodIndex` returns partial sliced data.

```
In [363]: ps['2011']
Out[363]:
2011-01   -2.916901
2011-02    0.514474
2011-03    1.346470
2011-04    0.816397
2011-05    2.258648
2011-06    0.494789
2011-07    0.301239
2011-08    0.464776
2011-09   -1.393581
2011-10    0.056780
2011-11    0.197035
2011-12    2.261385
Freq: M, dtype: float64

In [364]: dfp = pd.DataFrame(np.random.randn(600, 1),
   .....:                    columns=['A'],
   .....:                    index=pd.period_range('2013-01-01 9:00',
   .....:                                          periods=600,
   .....:                                          freq='T'))
   .....:

In [365]: dfp
Out[365]:
                         A
2013-01-01 09:00 -0.538468
2013-01-01 09:01 -1.365819
2013-01-01 09:02 -0.969051
2013-01-01 09:03 -0.331152
2013-01-01 09:04 -0.245334
2013-01-01 09:05  0.182165
2013-01-01 09:06  0.509697
...                    ...
2013-01-01 18:53  0.313342
2013-01-01 18:54  1.741982
2013-01-01 18:55  0.522460
2013-01-01 18:56  0.118710
2013-01-01 18:57  0.167517
2013-01-01 18:58  0.922883
2013-01-01 18:59  1.721104

[600 rows x 1 columns]

In [366]: dfp['2013-01-01 10H']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                         A
2013-01-01 10:00 -0.308975
2013-01-01 10:01  0.542520
2013-01-01 10:02  1.061068
2013-01-01 10:03  0.754005
2013-01-01 10:04  0.352933
2013-01-01 10:05  0.671551
2013-01-01 10:06  0.475667
...                    ...
```

(continues on next page)

(continued from previous page)

```
2013-01-01 10:53 -0.266928
2013-01-01 10:54  0.327742
2013-01-01 10:55 -0.865621
2013-01-01 10:56 -1.167818
2013-01-01 10:57 -2.081748
2013-01-01 10:58 -0.527146
2013-01-01 10:59  0.802298

[60 rows x 1 columns]
```

As with `DatetimeIndex`, the endpoints will be included in the result. The example below slices data starting from 10:00 to 11:59.

```
In [367]: dfp['2013-01-01 10H':'2013-01-01 11H']
Out[367]:
                         A
2013-01-01 10:00 -0.308975
2013-01-01 10:01  0.542520
2013-01-01 10:02  1.061068
2013-01-01 10:03  0.754005
2013-01-01 10:04  0.352933
2013-01-01 10:05  0.671551
2013-01-01 10:06  0.475667
...                    ...
2013-01-01 11:53 -1.219875
2013-01-01 11:54  2.568241
2013-01-01 11:55 -0.590204
2013-01-01 11:56  1.539990
2013-01-01 11:57 -1.224826
2013-01-01 11:58  0.578798
2013-01-01 11:59 -0.685496

[120 rows x 1 columns]
```

Frequency Conversion and Resampling with PeriodIndex

The frequency of `Period` and `PeriodIndex` can be converted via the `asfreq` method. Let's start with the fiscal year 2011, ending in December:

```
In [368]: p = pd.Period('2011', freq='A-DEC')

In [369]: p
Out[369]: Period('2011', 'A-DEC')
```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```
In [370]: p.asfreq('M', how='start')
Out[370]: Period('2011-01', 'M')

In [371]: p.asfreq('M', how='end')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[371]: Period('2011-12', 'M')
```

The shorthands 's' and 'e' are provided for convenience:

```
In [372]: p.asfreq('M', 's')
Out[372]: Period('2011-01', 'M')

In [373]: p.asfreq('M', 'e')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[373]: Period('2011-12', 'M')
```

Converting to a "super-period" (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```
In [374]: p = pd.Period('2011-12', freq='M')

In [375]: p.asfreq('A-NOV')
Out[375]: Period('2012', 'A-NOV')
```

Note that since we converted to an annual frequency that ends the year in November, the monthly period of December 2011 is actually in the 2012 A-NOV period.

Period conversions with anchored frequencies are particularly useful for working with various quarterly data common to economics, business, and other fields. Many organizations define quarters relative to the month in which their fiscal year starts and ends. Thus, first quarter of 2011 could start in 2010 or a few months into 2011. Via anchored frequencies, pandas works for all quarterly frequencies `Q-JAN` through `Q-DEC`.

`Q-DEC` define regular calendar quarters:

```
In [376]: p = pd.Period('2012Q1', freq='Q-DEC')

In [377]: p.asfreq('D', 's')
Out[377]: Period('2012-01-01', 'D')

In [378]: p.asfreq('D', 'e')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[378]: Period('2012-03-31', 'D')
```

`Q-MAR` defines fiscal year end in March:

```
In [379]: p = pd.Period('2011Q4', freq='Q-MAR')

In [380]: p.asfreq('D', 's')
Out[380]: Period('2011-01-01', 'D')

In [381]: p.asfreq('D', 'e')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[381]: Period('2011-03-31', 'D')
```

4.13.13 Converting Between Representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```
In [382]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [383]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [384]: ts
Out[384]:
2012-01-31    1.931253
2012-02-29   -0.184594
2012-03-31    0.249656
```

(continues on next page)

(continued from previous page)

```
2012-04-30   -0.978151
2012-05-31   -0.873389
Freq: M, dtype: float64

In [385]: ps = ts.to_period()

In [386]: ps
Out[386]:
2012-01    1.931253
2012-02   -0.184594
2012-03    0.249656
2012-04   -0.978151
2012-05   -0.873389
Freq: M, dtype: float64

In [387]: ps.to_timestamp()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2012-01-01    1.931253
2012-02-01   -0.184594
2012-03-01    0.249656
2012-04-01   -0.978151
2012-05-01   -0.873389
Freq: MS, dtype: float64
```

Remember that 's' and 'e' can be used to return the timestamps at the start or end of the period:

```
In [388]: ps.to_timestamp('D', how='s')
Out[388]:
2012-01-01    1.931253
2012-02-01   -0.184594
2012-03-01    0.249656
2012-04-01   -0.978151
2012-05-01   -0.873389
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [389]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [390]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [391]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [392]: ts.head()
Out[392]:
1990-03-01 09:00   -0.109291
1990-06-01 09:00   -0.637235
1990-09-01 09:00   -1.735925
1990-12-01 09:00    2.096946
1991-03-01 09:00   -1.039926
Freq: H, dtype: float64
```

4.13.14 Representing Out-of-Bounds Spans

If you have data that is outside of the `Timestamp` bounds, see *Timestamp limitations*, then you can use a `PeriodIndex` and/or `Series` of `Periods` to do computations.

```
In [393]: span = pd.period_range('1215-01-01', '1381-01-01', freq='D')

In [394]: span
Out[394]:
PeriodIndex(['1215-01-01', '1215-01-02', '1215-01-03', '1215-01-04',
             '1215-01-05', '1215-01-06', '1215-01-07', '1215-01-08',
             '1215-01-09', '1215-01-10',
             ...
             '1380-12-23', '1380-12-24', '1380-12-25', '1380-12-26',
             '1380-12-27', '1380-12-28', '1380-12-29', '1380-12-30',
             '1380-12-31', '1381-01-01'],
            dtype='period[D]', length=60632, freq='D')
```

To convert from an `int64` based YYYYMMDD representation.

```
In [395]: s = pd.Series([20121231, 20141130, 99991231])

In [396]: s
Out[396]:
0    20121231
1    20141130
2    99991231
dtype: int64

In [397]: def conv(x):
   .....:     return pd.Period(year=x // 10000, month=x // 100 % 100,
   .....:                      day=x % 100, freq='D')
   .....:

In [398]: s.apply(conv)
Out[398]:
0    2012-12-31
1    2014-11-30
2    9999-12-31
dtype: period[D]

In [399]: s.apply(conv)[2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[399]:␣
↪Period('9999-12-31', 'D')
```

These can easily be converted to a `PeriodIndex`:

```
In [400]: span = pd.PeriodIndex(s.apply(conv))

In [401]: span
Out[401]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='period[D]',␣
↪freq='D')
```

4.13.15 Time Zone Handling

Pandas provides rich support for working with timestamps in different time zones using `pytz` and `dateutil` libraries. `dateutil` currently is only supported for fixed offset and tzfile zones. The default library is `pytz`. Support for `dateutil` is provided for compatibility with other applications e.g. if you use `dateutil` in other Python packages.

Working with Time Zones

By default, pandas objects are time zone unaware:

```
In [402]: rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')

In [403]: rng.tz is None
Out[403]: True
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions. Dateutil time zone strings are distinguished from `pytz` time zones by starting with `dateutil/`.

- In `pytz` you can find a list of common (and less common) time zones using `from pytz import common_timezones, all_timezones`.
- `dateutil` uses the OS timezones so there isn't a fixed list available. For common zones, the names are the same as `pytz`.

```
In [404]: import dateutil

# pytz
In [405]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
   .....:                          tz='Europe/London')
   .....:

In [406]: rng_pytz.tz
Out[406]: <DstTzInfo 'Europe/London' LMT-1 day, 23:59:00 STD>

# dateutil
In [407]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
   .....:                              tz='dateutil/Europe/London')
   .....:

In [408]: rng_dateutil.tz
Out[408]: tzfile('/usr/share/zoneinfo/Europe/London')

# dateutil - utc special case
In [409]: rng_utc = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
   .....:                         tz=dateutil.tz.tzutc())
   .....:

In [410]: rng_utc.tz
Out[410]: tzutc()
```

Note that the `UTC` timezone is a special case in `dateutil` and should be constructed explicitly as an instance of `dateutil.tz.tzutc`. You can also construct other timezones explicitly first, which gives you more control over which time zone is used:

```
In [411]: import pytz

# pytz
In [412]: tz_pytz = pytz.timezone('Europe/London')

In [413]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
```

(continues on next page)

(continued from previous page)

```
   .....:                              tz=tz_pytz)
   .....:

In [414]: rng_pytz.tz == tz_pytz
Out[414]: True

# dateutil
In [415]: tz_dateutil = dateutil.tz.gettz('Europe/London')

In [416]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
   .....:                              tz=tz_dateutil)
   .....:

In [417]: rng_dateutil.tz == tz_dateutil
Out[417]: True
```

Timestamps, like Python's `datetime.datetime` object can be either time zone naive or time zone aware. Naive time series and `DatetimeIndex` objects can be *localized* using `tz_localize`:

```
In [418]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [419]: ts_utc = ts.tz_localize('UTC')

In [420]: ts_utc
Out[420]:
2012-03-06 00:00:00+00:00    0.326152
2012-03-07 00:00:00+00:00    0.455487
2012-03-08 00:00:00+00:00   -0.173426
2012-03-09 00:00:00+00:00    0.832223
2012-03-10 00:00:00+00:00   -0.166404
2012-03-11 00:00:00+00:00   -0.918468
2012-03-12 00:00:00+00:00    0.076835
2012-03-13 00:00:00+00:00    0.039694
2012-03-14 00:00:00+00:00   -1.246487
2012-03-15 00:00:00+00:00   -0.146705
2012-03-16 00:00:00+00:00   -1.392724
2012-03-17 00:00:00+00:00    0.523910
2012-03-18 00:00:00+00:00    1.578829
2012-03-19 00:00:00+00:00    0.654179
2012-03-20 00:00:00+00:00   -1.130643
Freq: D, dtype: float64
```

Again, you can explicitly construct the timezone object first. You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```
In [421]: ts_utc.tz_convert('US/Eastern')
Out[421]:
2012-03-05 19:00:00-05:00    0.326152
2012-03-06 19:00:00-05:00    0.455487
2012-03-07 19:00:00-05:00   -0.173426
2012-03-08 19:00:00-05:00    0.832223
2012-03-09 19:00:00-05:00   -0.166404
2012-03-10 19:00:00-05:00   -0.918468
2012-03-11 20:00:00-04:00    0.076835
2012-03-12 20:00:00-04:00    0.039694
2012-03-13 20:00:00-04:00   -1.246487
```

(continues on next page)

(continued from previous page)

```
2012-03-14 20:00:00-04:00   -0.146705
2012-03-15 20:00:00-04:00   -1.392724
2012-03-16 20:00:00-04:00    0.523910
2012-03-17 20:00:00-04:00    1.578829
2012-03-18 20:00:00-04:00    0.654179
2012-03-19 20:00:00-04:00   -1.130643
Freq: D, dtype: float64
```

Warning: Be wary of conversions between libraries. For some zones `pytz` and `dateutil` have different definitions of the zone. This is more of a problem for unusual timezones than for 'standard' zones like `US/Eastern`.

Warning: Be aware that a timezone definition across versions of timezone libraries may not be considered equal. This may cause problems when working with stored data that is localized using one version and operated on with a different version. See *here* for how to handle such a situation.

Warning: It is incorrect to pass a timezone directly into the `datetime.datetime` constructor (e.g., `datetime.datetime(2011, 1, 1, tz=timezone('US/Eastern'))`. Instead, the datetime needs to be localized using the localize method on the timezone.

Under the hood, all timestamps are stored in UTC. Scalar values from a `DatetimeIndex` with a time zone will have their fields (day, hour, minute) localized to the time zone. However, timestamps with the same UTC value are still considered to be equal even if they are in different time zones:

```
In [422]: rng_eastern = rng_utc.tz_convert('US/Eastern')

In [423]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [424]: rng_eastern[5]
Out[424]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', freq='D')

In [425]: rng_berlin[5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[425]:␣
↪Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', freq='D')

In [426]: rng_eastern[5] == rng_berlin[5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True
```

Like `Series`, `DataFrame`, and `DatetimeIndex`; `Timestamp` objects can be converted to other time zones using `tz_convert`:

```
In [427]: rng_eastern[5]
Out[427]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', freq='D')

In [428]: rng_berlin[5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[428]:␣
↪Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', freq='D')
```

(continues on next page)

(continued from previous page)

```
In [429]: rng_eastern[5].tz_convert('Europe/Berlin')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of `Timestamp` functions just like `DatetimeIndex` and `Series`:

```
In [430]: rng[5]
Out[430]: Timestamp('2012-03-11 00:00:00', freq='D')

In [431]: rng[5].tz_localize('Asia/Shanghai')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[431]: Timestamp('2012-03-11␣
↪00:00:00+0800', tz='Asia/Shanghai')
```

Operations between `Series` in different time zones will yield UTC `Series`, aligning the data on the UTC timestamps:

```
In [432]: eastern = ts_utc.tz_convert('US/Eastern')

In [433]: berlin = ts_utc.tz_convert('Europe/Berlin')

In [434]: result = eastern + berlin

In [435]: result
Out[435]:
2012-03-06 00:00:00+00:00    0.652304
2012-03-07 00:00:00+00:00    0.910974
2012-03-08 00:00:00+00:00   -0.346851
2012-03-09 00:00:00+00:00    1.664446
2012-03-10 00:00:00+00:00   -0.332807
2012-03-11 00:00:00+00:00   -1.836936
2012-03-12 00:00:00+00:00    0.153669
2012-03-13 00:00:00+00:00    0.079388
2012-03-14 00:00:00+00:00   -2.492974
2012-03-15 00:00:00+00:00   -0.293409
2012-03-16 00:00:00+00:00   -2.785448
2012-03-17 00:00:00+00:00    1.047819
2012-03-18 00:00:00+00:00    3.157658
2012-03-19 00:00:00+00:00    1.308359
2012-03-20 00:00:00+00:00   -2.261286
Freq: D, dtype: float64

In [436]: result.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2012-03-06 00:00:00+00:00', '2012-03-07 00:00:00+00:00',
               '2012-03-08 00:00:00+00:00', '2012-03-09 00:00:00+00:00',
               '2012-03-10 00:00:00+00:00', '2012-03-11 00:00:00+00:00',
               '2012-03-12 00:00:00+00:00', '2012-03-13 00:00:00+00:00',
               '2012-03-14 00:00:00+00:00', '2012-03-15 00:00:00+00:00',
               '2012-03-16 00:00:00+00:00', '2012-03-17 00:00:00+00:00',
               '2012-03-18 00:00:00+00:00', '2012-03-19 00:00:00+00:00',
               '2012-03-20 00:00:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

To remove timezone from tz-aware `DatetimeIndex`, use `tz_localize(None)` or `tz_convert(None)`. `tz_localize(None)` will remove timezone holding local time representations. `tz_convert(None)` will remove timezone after converting to UTC time.

```
In [437]: didx = pd.date_range(start='2014-08-01 09:00', freq='H',
   .....:                      periods=10, tz='US/Eastern')
   .....:

In [438]: didx
Out[438]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
               '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
               '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
               '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
               '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

In [439]: didx.tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
               '2014-08-01 11:00:00', '2014-08-01 12:00:00',
               '2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')

In [440]: didx.tz_convert(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [441]: didx.tz_convert('UTC').tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Ambiguous Times when Localizing

In some cases, localize cannot determine the DST and non-DST hours when there are duplicates. This often happens when reading files or database records that simply duplicate the hours. Passing `ambiguous='infer'` into `tz_localize` will attempt to determine the right offset. Below the top example will fail as it contains ambiguous times and the bottom will infer the right offset.

```
In [442]: rng_hourly = pd.DatetimeIndex(['11/06/2011 00:00', '11/06/2011 01:00',
   .....:                                '11/06/2011 01:00', '11/06/2011 02:00',
   .....:                                '11/06/2011 03:00'])
   .....:
```

This will fail as there are ambiguous times

```
In [2]: rng_hourly.tz_localize('US/Eastern')
AmbiguousTimeError: Cannot infer dst time from Timestamp('2011-11-06 01:00:00'), try␣
↪using the 'ambiguous' argument
```

Infer the ambiguous times

```
In [443]: rng_hourly_eastern = rng_hourly.tz_localize('US/Eastern', ambiguous='infer')

In [444]: rng_hourly_eastern.to_list()
Out[444]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]
```

In addition to 'infer', there are several other arguments supported. Passing an array-like of bools or 0s/1s where True represents a DST hour and False a non-DST hour, allows for distinguishing more than one DST transition (e.g., if you have multiple records in a database each with their own DST transition). Or passing 'NaT' will fill in transition times with not-a-time values. These methods are available in the `DatetimeIndex` constructor as well as `tz_localize`.

```
In [445]: rng_hourly_dst = np.array([1, 1, 0, 0, 0])

In [446]: rng_hourly.tz_localize('US/Eastern', ambiguous=rng_hourly_dst).to_list()
Out[446]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]

In [447]: rng_hourly.tz_localize('US/Eastern', ambiguous='NaT').to_list()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 NaT,
 NaT,
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]

In [448]: didx = pd.date_range(start='2014-08-01 09:00', freq='H',
   .....:                      periods=10, tz='US/Eastern')
   .....:

In [449]: didx
Out[449]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
               '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
               '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
               '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
               '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

In [450]: didx.tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

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

In [451]: didx.tz_convert(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [452]: didx.tz_convert('UCT').tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Nonexistent Times when Localizing

A DST transition may also shift the local time ahead by 1 hour creating nonexistent local times. The behavior of localizing a timeseries with nonexistent times can be controlled by the `nonexistent` argument. The following options are available:

- `'raise'`: Raises a `pytz.NonExistentTimeError` (the default behavior)
- `'NaT'`: Replaces nonexistent times with `NaT`
- `'shift_forward'`: Shifts nonexistent times forward to the closest real time
- `'shift_backward'`: Shifts nonexistent times backward to the closest real time
- timedelta object: Shifts nonexistent times by the timedelta duration

```
In [453]: dti = pd.date_range(start='2015-03-29 02:30:00', periods=3, freq='H')

# 2:30 is a nonexistent time
```

Localization of nonexistent times will raise an error by default.

```
In [2]: dti.tz_localize('Europe/Warsaw')
NonExistentTimeError: 2015-03-29 02:30:00
```

Transform nonexistent times to `NaT` or shift the times.

```
In [454]: dti
Out[454]:
```

(continues on next page)

(continued from previous page)

```
DatetimeIndex(['2015-03-29 02:30:00', '2015-03-29 03:30:00',
               '2015-03-29 04:30:00'],
              dtype='datetime64[ns]', freq='H')

In [455]: dti.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2015-03-29 03:00:00+02:00', '2015-03-29 03:30:00+02:00',
               '2015-03-29 04:30:00+02:00'],
              dtype='datetime64[ns, Europe/Warsaw]', freq='H')

In [456]: dti.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2015-03-29 01:59:59.999999999+01:00',
                         '2015-03-29 03:30:00+02:00',
                         '2015-03-29 04:30:00+02:00'],
              dtype='datetime64[ns, Europe/Warsaw]', freq='H')

In [457]: dti.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta(1, unit='H'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2015-03-29 03:30:00+02:00', '2015-03-29 03:30:00+02:00',
               '2015-03-29 04:30:00+02:00'],
              dtype='datetime64[ns, Europe/Warsaw]', freq='H')

In [458]: dti.tz_localize('Europe/Warsaw', nonexistent='NaT')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['NaT', '2015-03-29 03:30:00+02:00',
               '2015-03-29 04:30:00+02:00'],
              dtype='datetime64[ns, Europe/Warsaw]', freq='H')
```

TZ Aware Dtypes

`Series/DatetimeIndex` with a timezone **naive** value are represented with a dtype of `datetime64[ns]`.

```
In [459]: s_naive = pd.Series(pd.date_range('20130101', periods=3))

In [460]: s_naive
Out[460]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
dtype: datetime64[ns]
```

`Series/DatetimeIndex` with a timezone **aware** value are represented with a dtype of `datetime64[ns, tz]`.

```
In [461]: s_aware = pd.Series(pd.date_range('20130101', periods=3, tz='US/Eastern'))

In [462]: s_aware
Out[462]:
0   2013-01-01 00:00:00-05:00
1   2013-01-02 00:00:00-05:00
```

(continues on next page)

(continued from previous page)

```
2   2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

Both of these Series can be manipulated via the .dt accessor, see *here*.

For example, to localize and convert a naive stamp to timezone aware.

```
In [463]: s_naive.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[463]: 
0   2012-12-31 19:00:00-05:00
1   2013-01-01 19:00:00-05:00
2   2013-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

Further more you can .astype(...) timezone aware (and naive). This operation is effectively a localize AND convert on a naive stamp, and a convert on an aware stamp.

```
# localize and convert a naive timezone
In [464]: s_naive.astype('datetime64[ns, US/Eastern]')
Out[464]: 
0   2012-12-31 19:00:00-05:00
1   2013-01-01 19:00:00-05:00
2   2013-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]

# make an aware tz naive
In [465]: s_aware.astype('datetime64[ns]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2013-01-01 05:00:00
1   2013-01-02 05:00:00
2   2013-01-03 05:00:00
dtype: datetime64[ns]

# convert to a new timezone
In [466]: s_aware.astype('datetime64[ns, CET]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2013-01-01 06:00:00+01:00
1   2013-01-02 06:00:00+01:00
2   2013-01-03 06:00:00+01:00
dtype: datetime64[ns, CET]
```

Note: Using *Series.to_numpy()* on a Series, returns a NumPy array of the data. NumPy does not currently support timezones (even though it is *printing* in the local timezone!), therefore an object array of Timestamps is returned for timezone aware data:

```
In [467]: s_naive.to_numpy()
Out[467]: 
array(['2013-01-01T00:00:00.000000000', '2013-01-02T00:00:00.000000000',
       '2013-01-03T00:00:00.000000000'], dtype='datetime64[ns]')

In [468]: s_aware.to_numpy()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
array([Timestamp('2013-01-01 00:00:00-0500', tz='US/Eastern', freq='D'),
```

(continues on next page)

(continued from previous page)

```
       Timestamp('2013-01-02 00:00:00-0500', tz='US/Eastern', freq='D'),
       Timestamp('2013-01-03 00:00:00-0500', tz='US/Eastern', freq='D')],␣
↪dtype=object)
```

By converting to an object array of Timestamps, it preserves the timezone information. For example, when converting back to a Series:

```
In [469]: pd.Series(s_aware.to_numpy())
Out[469]:
0   2013-01-01 00:00:00-05:00
1   2013-01-02 00:00:00-05:00
2   2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

However, if you want an actual NumPy `datetime64[ns]` array (with the values converted to UTC) instead of an array of objects, you can specify the `dtype` argument:

```
In [470]: s_aware.to_numpy(dtype='datetime64[ns]')
Out[470]:
array(['2013-01-01T05:00:00.000000000', '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

4.14 Time Deltas

Timedeltas are differences in times, expressed in difference units, e.g. days, hours, minutes, seconds. They can be both positive and negative.

`Timedelta` is a subclass of `datetime.timedelta`, and behaves in a similar manner, but allows compatibility with `np.timedelta64` types as well as a host of custom representation, parsing, and attributes.

4.14.1 Parsing

You can construct a `Timedelta` scalar through various arguments:

```
In [1]: import datetime

# strings
In [2]: pd.Timedelta('1 days')
Out[2]: Timedelta('1 days 00:00:00')

In [3]: pd.Timedelta('1 days 00:00:00')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]: Timedelta('1 days 00:00:00')

In [4]: pd.Timedelta('1 days 2 hours')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:␣
↪Timedelta('1 days 02:00:00')

In [5]: pd.Timedelta('-1 days 2 min 3us')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-2 days +23:57:59.999997')
```

(continues on next page)

(continued from previous page)

```
# like datetime.timedelta
# note: these MUST be specified as keyword arguments
In [6]: pd.Timedelta(days=1, seconds=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 00:00:01')

# integers with a unit
In [7]: pd.Timedelta(1, unit='d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 00:00:00')

# from a datetime.timedelta/np.timedelta64
In [8]: pd.Timedelta(datetime.timedelta(days=1, seconds=1))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 00:00:01')

In [9]: pd.Timedelta(np.timedelta64(1, 'ms'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('0 days 00:00:00.001000')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [10]: pd.Timedelta('-1us')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-1 days +23:59:59.999999')

# a NaT
In [11]: pd.Timedelta('nan')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪NaT

In [12]: pd.Timedelta('nat')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪NaT

# ISO 8601 Duration strings
In [13]: pd.Timedelta('P0DT0H1M0S')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('0 days 00:01:00')

In [14]: pd.Timedelta('P0DT0H0M0.000000123S')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('0 days 00:00:00.000000')
```

New in version 0.23.0: Added constructor for ISO 8601 Duration strings

DateOffsets (`Day, Hour, Minute, Second, Milli, Micro, Nano`) can also be used in construction.

```
In [15]: pd.Timedelta(pd.offsets.Second(2))
Out[15]: Timedelta('0 days 00:00:02')
```

Further, operations among the scalars yield another scalar `Timedelta`.

```
In [16]: pd.Timedelta(pd.offsets.Day(2)) + pd.Timedelta(pd.offsets.Second(2)) +\
   ....:     pd.Timedelta('00:00:00.000123')
   ....:
Out[16]: Timedelta('2 days 00:00:02.000123')
```

to_timedelta

Using the top-level `pd.to_timedelta`, you can convert a scalar, array, list, or Series from a recognized timedelta format / value into a `Timedelta` type. It will construct Series if the input is a Series, a scalar if the input is scalar-like, otherwise it will output a `TimedeltaIndex`.

You can parse a single string to a Timedelta:

```
In [17]: pd.to_timedelta('1 days 06:05:01.00003')
Out[17]: Timedelta('1 days 06:05:01.000030')

In [18]: pd.to_timedelta('15.5us')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]: Timedelta('0 days 00:00:00.
↪000015')
```

or a list/array of strings:

```
In [19]: pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
Out[19]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT],␣
↪dtype='timedelta64[ns]', freq=None)
```

The `unit` keyword argument specifies the unit of the Timedelta:

```
In [20]: pd.to_timedelta(np.arange(5), unit='s')
Out[20]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'],␣
↪dtype='timedelta64[ns]', freq=None)

In [21]: pd.to_timedelta(np.arange(5), unit='d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↪'timedelta64[ns]', freq=None)
```

Timedelta limitations

Pandas represents `Timedeltas` in nanosecond resolution using 64 bit integers. As such, the 64 bit integer limits determine the `Timedelta` limits.

```
In [22]: pd.Timedelta.min
Out[22]: Timedelta('-106752 days +00:12:43.145224')

In [23]: pd.Timedelta.max
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]: Timedelta('106751 days␣
↪23:47:16.854775')
```

4.14.2 Operations

You can operate on Series/DataFrames and construct `timedelta64[ns]` Series through subtraction operations on `datetime64[ns]` Series, or `Timestamps`.

```
In [24]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [25]: td = pd.Series([pd.Timedelta(days=i) for i in range(3)])

In [26]: df = pd.DataFrame({'A': s, 'B': td})
```

(continues on next page)

(continued from previous page)

```
In [27]: df
Out[27]:
           A      B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [28]: df['C'] = df['A'] + df['B']

In [29]: df
Out[29]:
           A      B          C
0 2012-01-01 0 days 2012-01-01
1 2012-01-02 1 days 2012-01-03
2 2012-01-03 2 days 2012-01-05

In [30]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A     datetime64[ns]
B    timedelta64[ns]
C     datetime64[ns]
dtype: object

In [31]: s - s.max()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [32]: s - datetime.datetime(2011, 1, 1, 3, 5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   364 days 20:55:00
1   365 days 20:55:00
2   366 days 20:55:00
dtype: timedelta64[ns]

In [33]: s + datetime.timedelta(minutes=5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]

In [34]: s + pd.offsets.Minute(5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

(continues on next page)

(continued from previous page)

```
In [35]: s + pd.offsets.Minute(5) + pd.offsets.Milli(5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00.005
1   2012-01-02 00:05:00.005
2   2012-01-03 00:05:00.005
dtype: datetime64[ns]
```

Operations with scalars from a `timedelta64[ns]` series:

```
In [36]: y = s - s[0]

In [37]: y
Out[37]:
0   0 days
1   1 days
2   2 days
dtype: timedelta64[ns]
```

Series of timedeltas with `NaT` values are supported:

```
In [38]: y = s - s.shift()

In [39]: y
Out[39]:
0      NaT
1   1 days
2   1 days
dtype: timedelta64[ns]
```

Elements can be set to `NaT` using `np.nan` analogously to datetimes:

```
In [40]: y[1] = np.nan

In [41]: y
Out[41]:
0      NaT
1      NaT
2   1 days
dtype: timedelta64[ns]
```

Operands can also appear in a reversed order (a singular object operated with a Series):

```
In [42]: s.max() - s
Out[42]:
0   2 days
1   1 days
2   0 days
dtype: timedelta64[ns]

In [43]: datetime.datetime(2011, 1, 1, 3, 5) - s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[43]:
0   -365 days +03:05:00
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]
```

(continues on next page)

(continued from previous page)

```
In [44]: datetime.timedelta(minutes=5) + s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

min, max and the corresponding idxmin, idxmax operations are supported on frames:

```
In [45]: A = s - pd.Timestamp('20120101') - pd.Timedelta('00:05:05')

In [46]: B = s - pd.Series(pd.date_range('2012-1-2', periods=3, freq='D'))

In [47]: df = pd.DataFrame({'A': A, 'B': B})

In [48]: df
Out[48]: 
                  A       B
0 -1 days +23:54:55 -1 days
1   0 days 23:54:55 -1 days
2   1 days 23:54:55 -1 days

In [49]: df.min()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A   -1 days +23:54:55
B   -1 days +00:00:00
dtype: timedelta64[ns]

In [50]: df.min(axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -1 days
1   -1 days
2   -1 days
dtype: timedelta64[ns]

In [51]: df.idxmin()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    0
B    0
dtype: int64

In [52]: df.idxmax()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    2
B    0
dtype: int64
```

min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

```
In [53]: df.min().max()
Out[53]: Timedelta('-1 days +23:54:55')
```

(continues on next page)

(continued from previous page)

```
In [54]: df.min(axis=1).min()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[54]: Timedelta('-1 days +00:00:00')

In [55]: df.min().idxmax()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[55]:␣
↪'A'

In [56]: df.min(axis=1).idxmin()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]
↪0
```

You can fillna on timedeltas. Integers will be interpreted as seconds. You can pass a timedelta to get a particular value.

```
In [57]: y.fillna(0)
Out[57]:
0    0 days
1    0 days
2    1 days
dtype: timedelta64[ns]

In [58]: y.fillna(10)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[58]:
0    0 days 00:00:10
1    0 days 00:00:10
2    1 days 00:00:00
dtype: timedelta64[ns]

In [59]: y.fillna(pd.Timedelta('-1 days, 00:00:05'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -1 days +00:00:05
1   -1 days +00:00:05
2     1 days 00:00:00
dtype: timedelta64[ns]
```

You can also negate, multiply and use `abs` on `Timedeltas`:

```
In [60]: td1 = pd.Timedelta('-1 days 2 hours 3 seconds')

In [61]: td1
Out[61]: Timedelta('-2 days +21:59:57')

In [62]: -1 * td1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[62]: Timedelta('1 days 02:00:03')

In [63]: - td1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[63]:␣
↪Timedelta('1 days 02:00:03')

In [64]: abs(td1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 02:00:03')
```

4.14.3 Reductions

Numeric reduction operation for `timedelta64[ns]` will return `Timedelta` objects. As usual `NaT` are skipped during evaluation.

```
In [65]: y2 = pd.Series(pd.to_timedelta(['-1 days +00:00:05', 'nat',
   ....:                                 '-1 days +00:00:05', '1 days']))
   ....: 

In [66]: y2
Out[66]: 
0   -1 days +00:00:05
1                 NaT
2   -1 days +00:00:05
3     1 days 00:00:00
dtype: timedelta64[ns]

In [67]: y2.mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-1 days +16:00:03.333333')

In [68]: y2.median()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-1 days +00:00:05')

In [69]: y2.quantile(.1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-1 days +00:00:05')

In [70]: y2.sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('-1 days +00:00:10')
```

4.14.4 Frequency Conversion

Timedelta Series, `TimedeltaIndex`, and `Timedelta` scalars can be converted to other 'frequencies' by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield Series and propagate `NaT` -> `nan`. Note that division by the NumPy scalar is true division, while astyping is equivalent of floor division.

```
In [71]: december = pd.Series(pd.date_range('20121201', periods=4))

In [72]: january = pd.Series(pd.date_range('20130101', periods=4))

In [73]: td = january - december

In [74]: td[2] += datetime.timedelta(minutes=5, seconds=3)

In [75]: td[3] = np.nan

In [76]: td
Out[76]: 
0   31 days 00:00:00
1   31 days 00:00:00
2   31 days 00:05:03
3                NaT
dtype: timedelta64[ns]
```

(continues on next page)

(continued from previous page)

```
# to days
In [77]: td / np.timedelta64(1, 'D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    31.000000
1    31.000000
2    31.003507
3          NaN
dtype: float64

In [78]: td.astype('timedelta64[D]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    31.0
1    31.0
2    31.0
3     NaN
dtype: float64

# to seconds
In [79]: td / np.timedelta64(1, 's')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64

In [80]: td.astype('timedelta64[s]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64

# to months (these are constant months)
In [81]: td / np.timedelta64(1, 'M')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1.018501
1    1.018501
2    1.018617
3         NaN
dtype: float64
```

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series yields another `timedelta64[ns]` dtypes Series.

```
In [82]: td * -1
Out[82]:
0   -31 days +00:00:00
1   -31 days +00:00:00
```

(continues on next page)

(continued from previous page)

```
2   -32 days +23:54:57
3                  NaT
dtype: timedelta64[ns]

In [83]: td * pd.Series([1, 2, 3, 4])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   31 days 00:00:00
1   62 days 00:00:00
2   93 days 00:15:09
3                NaT
dtype: timedelta64[ns]
```

Rounded division (floor-division) of a `timedelta64[ns]` Series by a scalar `Timedelta` gives a series of integers.

```
In [84]: td // pd.Timedelta(days=3, hours=4)
Out[84]:
0    9.0
1    9.0
2    9.0
3    NaN
dtype: float64

In [85]: pd.Timedelta(days=3, hours=4) // td
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[85]:
0    0.0
1    0.0
2    0.0
3    NaN
dtype: float64
```

The mod (%) and divmod operations are defined for `Timedelta` when operating with another timedelta-like or with a numeric argument.

```
In [86]: pd.Timedelta(hours=37) % datetime.timedelta(hours=2)
Out[86]: Timedelta('0 days 01:00:00')

# divmod against a timedelta-like returns a pair (int, Timedelta)
In [87]: divmod(datetime.timedelta(hours=2), pd.Timedelta(minutes=11))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[87]: (10, Timedelta('0 days 00:10:00'))

# divmod against a numeric returns a pair (Timedelta, Timedelta)
In [88]: divmod(pd.Timedelta(hours=25), 86400000000000)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[88]:␣
↪(Timedelta('0 days 00:00:00.000000'), Timedelta('0 days 01:00:00'))
```

4.14.5 Attributes

You can access various components of the `Timedelta` or `TimedeltaIndex` directly using the attributes `days,seconds,microseconds,nanoseconds`. These are identical to the values returned by `datetime.timedelta`, in that, for example, the `.seconds` attribute represents the number of seconds >= 0 and < 1 day. These are signed according to whether the `Timedelta` is signed.

These operations can also be directly accessed via the `.dt` property of the `Series` as well.

Note: Note that the attributes are NOT the displayed values of the `Timedelta`. Use `.components` to retrieve the displayed values.

For a `Series`:

```
In [89]: td.dt.days
Out[89]:
0    31.0
1    31.0
2    31.0
3     NaN
dtype: float64

In [90]: td.dt.seconds
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[90]:
0      0.0
1      0.0
2    303.0
3      NaN
dtype: float64
```

You can access the value of the fields for a scalar `Timedelta` directly.

```
In [91]: tds = pd.Timedelta('31 days 5 min 3 sec')

In [92]: tds.days
Out[92]: 31

In [93]: tds.seconds
\\\\\\\\\\\\Out[93]: 303

In [94]: (-tds).seconds
\\\\\\\\\\\\\\\\\\\\\\\\\Out[94]: 86097
```

You can use the `.components` property to access a reduced form of the timedelta. This returns a `DataFrame` indexed similarly to the `Series`. These are the *displayed* values of the `Timedelta`.

```
In [95]: td.dt.components
Out[95]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0  31.0    0.0      0.0      0.0           0.0           0.0          0.0
1  31.0    0.0      0.0      0.0           0.0           0.0          0.0
2  31.0    0.0      5.0      3.0           0.0           0.0          0.0
3   NaN    NaN      NaN      NaN           NaN           NaN          NaN

In [96]: td.dt.components.seconds
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    0.0
1    0.0
2    3.0
3    NaN
Name: seconds, dtype: float64
```

You can convert a `Timedelta` to an ISO 8601 Duration string with the `.isoformat` method

New in version 0.20.0.

```
In [97]: pd.Timedelta(days=6, minutes=50, seconds=3,
   ....:              milliseconds=10, microseconds=10,
   ....:              nanoseconds=12).isoformat()
   ....:
Out[97]: 'P6DT0H50M3.010010012S'
```

4.14.6 TimedeltaIndex

To generate an index with time delta, you can use either the *TimedeltaIndex* or the *timedelta_range()* constructor.

Using `TimedeltaIndex` you can pass string-like, `Timedelta`, `timedelta`, or `np.timedelta64` objects. Passing `np.nan/pd.NaT/nat` will represent missing values.

```
In [98]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05', np.timedelta64(2, 'D'),
   ....:                    datetime.timedelta(days=2, seconds=2)])
   ....:
Out[98]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
                '2 days 00:00:02'],
               dtype='timedelta64[ns]', freq=None)
```

The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation:

```
In [99]: pd.TimedeltaIndex(['0 days', '10 days', '20 days'], freq='infer')
Out[99]: TimedeltaIndex(['0 days', '10 days', '20 days'], dtype='timedelta64[ns]',␣
↪freq='10D')
```

Generating Ranges of Time Deltas

Similar to *date_range()*, you can construct regular ranges of a `TimedeltaIndex` using *timedelta_range()*. The default frequency for `timedelta_range` is calendar day:

```
In [100]: pd.timedelta_range(start='1 days', periods=5)
Out[100]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↪'timedelta64[ns]', freq='D')
```

Various combinations of `start`, `end`, and `periods` can be used with `timedelta_range`:

```
In [101]: pd.timedelta_range(start='1 days', end='5 days')
Out[101]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↪'timedelta64[ns]', freq='D')

In [102]: pd.timedelta_range(end='10 days', periods=4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['7 days', '8 days', '9 days', '10 days'], dtype='timedelta64[ns]',␣
↪freq='D')
```

The `freq` parameter can passed a variety of *frequency aliases*:

```
In [103]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')
Out[103]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
```

(continues on next page)

(continued from previous page)

```
                '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00',
                '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00',
                '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00',
                '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00',
                '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00',
                '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00',
                '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00',
                '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00',
                '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00',
                '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='30T')

In [104]: pd.timedelta_range(start='1 days', periods=5, freq='2D5H')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
TimedeltaIndex(['1 days 00:00:00', '3 days 05:00:00', '5 days 10:00:00',
                '7 days 15:00:00', '9 days 20:00:00'],
               dtype='timedelta64[ns]', freq='53H')
```

New in version 0.23.0.

Specifying `start`, `end`, and `periods` will generate a range of evenly spaced timedeltas from `start` to `end` inclusively, with `periods` number of elements in the resulting `TimedeltaIndex`:

```
In [105]: pd.timedelta_range('0 days', '4 days', periods=5)
Out[105]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↪'timedelta64[ns]', freq=None)

In [106]: pd.timedelta_range('0 days', '4 days', periods=10)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
TimedeltaIndex(['0 days 00:00:00', '0 days 10:40:00', '0 days 21:20:00',
                '1 days 08:00:00', '1 days 18:40:00', '2 days 05:20:00',
                '2 days 16:00:00', '3 days 02:40:00', '3 days 13:20:00',
                '4 days 00:00:00'],
               dtype='timedelta64[ns]', freq=None)
```

Using the TimedeltaIndex

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.

```
In [107]: s = pd.Series(np.arange(100),
   .....:               index=pd.timedelta_range('1 days', periods=100, freq='h'))
   .....:

In [108]: s
Out[108]:
1 days 00:00:00     0
1 days 01:00:00     1
```

(continues on next page)

(continued from previous page)

```
1 days 02:00:00     2
1 days 03:00:00     3
1 days 04:00:00     4
1 days 05:00:00     5
1 days 06:00:00     6
                   ..
4 days 21:00:00    93
4 days 22:00:00    94
4 days 23:00:00    95
5 days 00:00:00    96
5 days 01:00:00    97
5 days 02:00:00    98
5 days 03:00:00    99
Freq: H, Length: 100, dtype: int64
```

Selections work similarly, with coercion on string-likes and slices:

```
In [109]: s['1 day':'2 day']
Out[109]:
1 days 00:00:00     0
1 days 01:00:00     1
1 days 02:00:00     2
1 days 03:00:00     3
1 days 04:00:00     4
1 days 05:00:00     5
1 days 06:00:00     6
                   ..
2 days 17:00:00    41
2 days 18:00:00    42
2 days 19:00:00    43
2 days 20:00:00    44
2 days 21:00:00    45
2 days 22:00:00    46
2 days 23:00:00    47
Freq: H, Length: 48, dtype: int64

In [110]: s['1 day 01:00:00']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1

In [111]: s[pd.Timedelta('1 day 1h')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1
```

Furthermore you can use partial string selection and the range will be inferred:

```
In [112]: s['1 day':'1 day 5 hours']
Out[112]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
Freq: H, dtype: int64
```

Operations

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

```
In [113]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [114]: tdi.to_list()
Out[114]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [115]: dti = pd.date_range('20130101', periods=3)

In [116]: dti.to_list()
Out[116]:
[Timestamp('2013-01-01 00:00:00', freq='D'),
 Timestamp('2013-01-02 00:00:00', freq='D'),
 Timestamp('2013-01-03 00:00:00', freq='D')]

In [117]: (dti + tdi).to_list()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [118]: (dti - tdi).to_list()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

Conversions

Similarly to frequency conversion on a `Series` above, you can convert these indices to yield another Index.

```
In [119]: tdi / np.timedelta64(1, 's')
Out[119]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

In [120]: tdi.astype('timedelta64[s]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[120]:␣
↪Float64Index([86400.0, nan, 172800.0], dtype='float64')
```

Scalars type ops work as well. These can potentially return a *different* type of index.

```
# adding or timedelta and date -> datelike
In [121]: tdi + pd.Timestamp('20130101')
Out[121]: DatetimeIndex(['2013-01-02', 'NaT', '2013-01-03'], dtype='datetime64[ns]',␣
↪freq=None)

# subtraction of a date and a timedelta -> datelike
# note that trying to subtract a date from a Timedelta will raise an exception
In [122]: (pd.Timestamp('20130101') - tdi).to_list()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪[Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')]

# timedelta + timedelta -> timedelta
In [123]: tdi + pd.Timedelta('10 days')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['11 days', NaT, '12 days'], dtype='timedelta64[ns]', freq=None)

# division can result in a Timedelta if the divisor is an integer
```

(continues on next page)

(continued from previous page)

```
In [124]: tdi / 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['0 days 12:00:00', NaT, '1 days 00:00:00'], dtype='timedelta64[ns]',
↪ freq=None)

# or a Float64Index if the divisor is a Timedelta
In [125]: tdi / tdi[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Float64Index([1.0, nan, 2.0], dtype='float64')
```

4.14.7 Resampling

Similar to *timeseries resampling*, we can resample with a `TimedeltaIndex`.

```
In [126]: s.resample('D').mean()
Out[126]:
1 days    11.5
2 days    35.5
3 days    59.5
4 days    83.5
5 days    97.5
Freq: D, dtype: float64
```

4.15 Styling

New in version 0.17.1

Provisional: This is a new feature and still under development. We'll be adding features and possibly making breaking changes in future releases. We'd love to hear your feedback.

This document is written as a Jupyter Notebook, and can be viewed or downloaded here.

You can apply **conditional formatting**, the visual styling of a DataFrame depending on the data within, by using the `DataFrame.style` property. This is a property that returns a `Styler` object, which has useful methods for formatting and displaying DataFrames.

The styling is accomplished using CSS. You write "style functions" that take scalars, `DataFrames` or `Series`, and return *like-indexed* DataFrames or Series with CSS `"attribute: value"` pairs for the values. These functions can be incrementally passed to the `Styler` which collects the styles before rendering.

4.15.1 Building Styles

Pass your style functions into one of the following methods:

- `Styler.applymap`: elementwise
- `Styler.apply`: column-/row-/table-wise

Both of those methods take a function (and some other keyword arguments) and applies your function to the DataFrame in a certain way. `Styler.applymap` works through the DataFrame elementwise. `Styler.apply` passes each column or row into your DataFrame one-at-a-time or the entire table at once, depending on the `axis` keyword argument. For columnwise use `axis=0`, rowwise use `axis=1`, and for the entire table at once use `axis=None`.

For `Styler.applymap` your function should take a scalar and return a single string with the CSS attribute-value pair.

For `Styler.apply` your function should take a Series or DataFrame (depending on the axis parameter), and return a Series or DataFrame with an identical shape where each value is a string with a CSS attribute-value pair.

Let's see some examples.

[2]:
```python
import pandas as pd
import numpy as np

np.random.seed(24)
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[0, 2] = np.nan
```

Here's a boring example of rendering a DataFrame, without any (visible) styles:

[3]:
```python
df.style
```

[3]:
```
<pandas.io.formats.style.Styler at 0x7fb470f574e0>
```

Note: The `DataFrame.style` attribute is a property that returns a `Styler` object. `Styler` has a `_repr_html_` method defined on it so they are rendered automatically. If you want the actual HTML back for further processing or for writing to file call the `.render()` method which returns a string.

The above output looks very similar to the standard DataFrame HTML representation. But we've done some work behind the scenes to attach CSS classes to each cell. We can view these by calling the `.render` method.

[4]:
```python
df.style.highlight_null().render().split('\n')[:10]
```

[4]:
```
['<style  type="text/css" >',
 '    #T_1330f574_20bf_11e9_af5f_9168a5230b87row0_col2 {',
 '            background-color:  red;',
 '        }</style><table id="T_1330f574_20bf_11e9_af5f_9168a5230b87" ><thead>    <tr>
↪        <th class="blank level0" ></th>        <th class="col_heading level0 col0" >
↪A</th>        <th class="col_heading level0 col1" >B</th>        <th class="col_
↪heading level0 col2" >C</th>        <th class="col_heading level0 col3" >D</th>    ␣
↪    <th class="col_heading level0 col4" >E</th>    </tr></thead><tbody>',
 '                <tr>',
 '                        <th id="T_1330f574_20bf_11e9_af5f_9168a5230b87level0_row0"␣
↪class="row_heading level0 row0" >0</th>',
 '                        <td id="T_1330f574_20bf_11e9_af5f_9168a5230b87row0_col0"␣
↪class="data row0 col0" >1</td>',
 '                        <td id="T_1330f574_20bf_11e9_af5f_9168a5230b87row0_col1"␣
↪class="data row0 col1" >1.32921</td>',
 '                        <td id="T_1330f574_20bf_11e9_af5f_9168a5230b87row0_col2"␣
↪class="data row0 col2" >nan</td>',
 '                        <td id="T_1330f574_20bf_11e9_af5f_9168a5230b87row0_col3"␣
↪class="data row0 col3" >-0.31628</td>']
```

The `row0_col2` is the identifier for that particular cell. We've also prepended each row/column identifier with a UUID unique to each DataFrame so that the style from one doesn't collide with the styling from another within the same notebook or page (you can set the `uuid` if you'd like to tie together the styling of two DataFrames).

When writing style functions, you take care of producing the CSS attribute / value pairs you want. Pandas matches those up with the CSS classes that identify each cell.

Let's write a simple style function that will color negative numbers red and positive numbers black.

[5]:
```
def color_negative_red(val):
    """
    Takes a scalar and returns a string with
    the css property `'color: red'` for negative
    strings, black otherwise.
    """
    color = 'red' if val < 0 else 'black'
    return 'color: %s' % color
```

In this case, the cell's style depends only on it's own value. That means we should use the `Styler.applymap` method which works elementwise.

[6]:
```
s = df.style.applymap(color_negative_red)
s
```

[6]: `<pandas.io.formats.style.Styler at 0x7fb470f4c7b8>`

Notice the similarity with the standard `df.applymap`, which operates on DataFrames elementwise. We want you to be able to reuse your existing knowledge of how to interact with DataFrames.

Notice also that our function returned a string containing the CSS attribute and value, separated by a colon just like in a `<style>` tag. This will be a common theme.

Finally, the input shapes matched. `Styler.applymap` calls the function on each scalar input, and the function returns a scalar output.

Now suppose you wanted to highlight the maximum value in each column. We can't use `.applymap` anymore since that operated elementwise. Instead, we'll turn to `.apply` which operates columnwise (or rowwise using the `axis` keyword). Later on we'll see that something like `highlight_max` is already defined on `Styler` so you wouldn't need to write this yourself.

[7]:
```
def highlight_max(s):
    '''
    highlight the maximum in a Series yellow.
    '''
    is_max = s == s.max()
    return ['background-color: yellow' if v else '' for v in is_max]
```

[8]:
```
df.style.apply(highlight_max)
```

[8]: `<pandas.io.formats.style.Styler at 0x7fb452e5d128>`

In this case the input is a `Series`, one column at a time. Notice that the output shape of `highlight_max` matches the input shape, an array with `len(s)` items.

We encourage you to use method chains to build up a style piecewise, before finally rending at the end of the chain.

[9]:
```
df.style.\
    applymap(color_negative_red).\
    apply(highlight_max)
```

[9]: `<pandas.io.formats.style.Styler at 0x7fb452e5d860>`

Above we used `Styler.apply` to pass in each column one at a time.

Debugging Tip: If you're having trouble writing your style function, try just passing it into DataFrame.apply. Internally, Styler.apply uses DataFrame.apply so the result should be the same.

What if you wanted to highlight just the maximum value in the entire table? Use `.apply(function, axis=None)` to indicate that your function wants the entire table, not one column or row at a time. Let's try that next.

We'll rewrite our `highlight-max` to handle either Series (from `.apply(axis=0 or 1)`) or DataFrames (from `.apply(axis=None)`). We'll also allow the color to be adjustable, to demonstrate that `.apply`, and `.applymap` pass along keyword arguments.

[10]:
```
def highlight_max(data, color='yellow'):
    '''
    highlight the maximum in a Series or DataFrame
    '''
    attr = 'background-color: {}'.format(color)
    if data.ndim == 1:  # Series from .apply(axis=0) or axis=1
        is_max = data == data.max()
        return [attr if v else '' for v in is_max]
    else:  # from .apply(axis=None)
        is_max = data == data.max().max()
        return pd.DataFrame(np.where(is_max, attr, ''),
                            index=data.index, columns=data.columns)
```

When using `Styler.apply(func, axis=None)`, the function must return a DataFrame with the same index and column labels.

[11]:
```
df.style.apply(highlight_max, color='darkorange', axis=None)
```

[11]:
```
<pandas.io.formats.style.Styler at 0x7fb452e11198>
```

Building Styles Summary

Style functions should return strings with one or more CSS `attribute: value` delimited by semicolons. Use

- `Styler.applymap(func)` for elementwise styles
- `Styler.apply(func, axis=0)` for columnwise styles
- `Styler.apply(func, axis=1)` for rowwise styles
- `Styler.apply(func, axis=None)` for tablewise styles

And crucially the input and output shapes of `func` must match. If `x` is the input then `func(x).shape == x.shape`.

4.15.2 Finer Control: Slicing

Both `Styler.apply`, and `Styler.applymap` accept a `subset` keyword. This allows you to apply styles to specific rows or columns, without having to code that logic into your `style` function.

The value passed to `subset` behaves similar to slicing a DataFrame.

- A scalar is treated as a column label
- A list (or series or numpy array)
- A tuple is treated as `(row_indexer, column_indexer)`

Consider using `pd.IndexSlice` to construct the tuple for the last one.

[12]:
```
df.style.apply(highlight_max, subset=['B', 'C', 'D'])
```

[12]:
```
<pandas.io.formats.style.Styler at 0x7fb452e2d240>
```

For row and column slicing, any valid indexer to `.loc` will work.

[13]:
```
df.style.applymap(color_negative_red,
                  subset=pd.IndexSlice[2:5, ['B', 'D']])
```

[13]:
```
<pandas.io.formats.style.Styler at 0x7fb452df7630>
```

Only label-based slicing is supported right now, not positional.

If your style function uses a `subset` or `axis` keyword argument, consider wrapping your function in a `functools.partial`, partialing out that keyword.

```
my_func2 = functools.partial(my_func, subset=42)
```

4.15.3 Finer Control: Display Values

We distinguish the *display* value from the *actual* value in `Styler`. To control the display value, the text is printed in each cell, use `Styler.format`. Cells can be formatted according to a format spec string or a callable that takes a single value and returns a string.

[14]:
```
df.style.format("{:.2%}")
```

[14]:
```
<pandas.io.formats.style.Styler at 0x7fb452e28898>
```

Use a dictionary to format specific columns.

[15]:
```
df.style.format({'B': "{:0<4.0f}", 'D': '{:+.2f}'})
```

[15]:
```
<pandas.io.formats.style.Styler at 0x7fb452e09a90>
```

Or pass in a callable (or dictionary of callables) for more flexible handling.

[16]:
```
df.style.format({"B": lambda x: "±{:.2f}".format(abs(x))})
```

[16]:
```
<pandas.io.formats.style.Styler at 0x7fb452e09668>
```

4.15.4 Builtin Styles

Finally, we expect certain styling functions to be common enough that we've included a few "built-in" to the `Styler`, so you don't have to write them yourself.

[17]:
```
df.style.highlight_null(null_color='red')
```

[17]:
```
<pandas.io.formats.style.Styler at 0x7fb452e09828>
```

You can create "heatmaps" with the `background_gradient` method. These require matplotlib, and we'll use Seaborn to get a nice colormap.

[18]:
```
import seaborn as sns

cm = sns.light_palette("green", as_cmap=True)

s = df.style.background_gradient(cmap=cm)
s
```

```
/opt/conda/envs/pandas/lib/python3.6/site-packages/matplotlib/colors.py:512: 
↪RuntimeWarning: invalid value encountered in less
  xa[xa < 0] = -1
```

[18]:
```
<pandas.io.formats.style.Styler at 0x7fb452e11390>
```

`Styler.background_gradient` takes the keyword arguments `low` and `high`. Roughly speaking these extend the range of your data by `low` and `high` percent so that when we convert the colors, the colormap's entire range isn't used. This is useful so that you can actually read the text still.

[19]:
```
# Uses the full color range
df.loc[:4].style.background_gradient(cmap='viridis')
```

[19]:
```
<pandas.io.formats.style.Styler at 0x7fb446e34f28>
```

[20]:
```
# Compress the color range
(df.loc[:4]
    .style
    .background_gradient(cmap='viridis', low=.5, high=0)
    .highlight_null('red'))
```

[20]:
```
<pandas.io.formats.style.Styler at 0x7fb452e09c50>
```

There's also `.highlight_min` and `.highlight_max`.

[21]:
```
df.style.highlight_max(axis=0)
```

[21]:
```
<pandas.io.formats.style.Styler at 0x7fb446e34c50>
```

Use `Styler.set_properties` when the style doesn't actually depend on the values.

[22]:
```
df.style.set_properties(**{'background-color': 'black',
                           'color': 'lawngreen',
                           'border-color': 'white'})
```

[22]:
```
<pandas.io.formats.style.Styler at 0x7fb446e34c88>
```

Bar charts

You can include "bar charts" in your DataFrame.

[23]:
```
df.style.bar(subset=['A', 'B'], color='#d65f5f')
```

[23]:
```
<pandas.io.formats.style.Styler at 0x7fb446e34748>
```

New in version 0.20.0 is the ability to customize further the bar chart: You can now have the `df.style.bar` be centered on zero or midpoint value (in addition to the already existing way of having the min value at the left side of the cell), and you can pass a list of `[color_negative, color_positive]`.

Here's how you can change the above with the new `align='mid'` option:

[24]:
```
df.style.bar(subset=['A', 'B'], align='mid', color=['#d65f5f', '#5fba7d'])
```

[24]:
```
<pandas.io.formats.style.Styler at 0x7fb446ddf828>
```

The following example aims to give a highlight of the behavior of the new align options:

[25]:
```
import pandas as pd
from IPython.display import HTML

# Test series
test1 = pd.Series([-100,-60,-30,-20], name='All Negative')
```

(continues on next page)

(continued from previous page)

```
test2 = pd.Series([10,20,50,100], name='All Positive')
test3 = pd.Series([-10,-5,0,90], name='Both Pos and Neg')

head = """
<table>
    <thead>
        <th>Align</th>
        <th>All Negative</th>
        <th>All Positive</th>
        <th>Both Neg and Pos</th>
    </thead>
    </tbody>

"""

aligns = ['left','zero','mid']
for align in aligns:
    row = "<tr><th>{}</th>".format(align)
    for serie in [test1,test2,test3]:
        s = serie.copy()
        s.name=''
        row += "<td>{}</td>".format(s.to_frame().style.bar(align=align,
                                                           color=['#d65f5f', '#5fba7d
↪'],
                                                           width=100).render())
↪#testn['width']
    row += '</tr>'
    head += row

head+= """
</tbody>
</table>"""


HTML(head)
```

[25]: <IPython.core.display.HTML object>

4.15.5 Sharing Styles

Say you have a lovely style built up for a DataFrame, and now you want to apply the same style to a second DataFrame. Export the style with `df1.style.export`, and import it on the second DataFrame with `df1.style.set`

[26]:
```
df2 = -df
style1 = df.style.applymap(color_negative_red)
style1
```

[26]: <pandas.io.formats.style.Styler at 0x7fb446ddf748>

[27]:
```
style2 = df2.style
style2.use(style1.export())
style2
```

[27]: <pandas.io.formats.style.Styler at 0x7fb446d8e710>

Notice that you're able share the styles even though they're data aware. The styles are re-evaluated on the new DataFrame they've been `used` upon.

4.15.6 Other Options

You've seen a few methods for data-driven styling. `Styler` also provides a few other options for styles that don't depend on the data.

- precision
- captions
- table-wide styles
- hiding the index or columns

Each of these can be specified in two ways:

- A keyword argument to `Styler.__init__`
- A call to one of the `.set_` or `.hide_` methods, e.g. `.set_caption` or `.hide_columns`

The best method to use depends on the context. Use the `Styler` constructor when building many styled DataFrames that should all share the same properties. For interactive use, the `.set_` and `.hide_` methods are more convenient.

Precision

You can control the precision of floats using pandas' regular `display.precision` option.

[28]:
```
with pd.option_context('display.precision', 2):
    html = (df.style
              .applymap(color_negative_red)
              .apply(highlight_max))
html
```

[28]:
```
<pandas.io.formats.style.Styler at 0x7fb446ddf9b0>
```

Or through a `set_precision` method.

[29]:
```
df.style\
  .applymap(color_negative_red)\
  .apply(highlight_max)\
  .set_precision(2)
```

[29]:
```
<pandas.io.formats.style.Styler at 0x7fb446d8e898>
```

Setting the precision only affects the printed number; the full-precision values are always passed to your style functions. You can always use `df.round(2).style` if you'd prefer to round from the start.

Captions

Regular table captions can be added in a few ways.

[30]:
```
df.style.set_caption('Colormaps, with a caption.')\
    .background_gradient(cmap=cm)
```

[30]:
```
<pandas.io.formats.style.Styler at 0x7fb446d869b0>
```

Table Styles

The next option you have are "table styles". These are styles that apply to the table as a whole, but don't look at the data. Certain sytlings, including pseudo-selectors like `:hover` can only be used this way.

[31]:
```
from IPython.display import HTML

def hover(hover_color="#ffff99"):
    return dict(selector="tr:hover",
                props=[("background-color", "%s" % hover_color)])

styles = [
    hover(),
    dict(selector="th", props=[("font-size", "150%"),
                               ("text-align", "center")]),
    dict(selector="caption", props=[("caption-side", "bottom")])
]
html = (df.style.set_table_styles(styles)
          .set_caption("Hover to highlight."))
html
```

[31]:
```
<pandas.io.formats.style.Styler at 0x7fb446d86f98>
```

`table_styles` should be a list of dictionaries. Each dictionary should have the `selector` and `props` keys. The value for `selector` should be a valid CSS selector. Recall that all the styles are already attached to an `id`, unique to each `Styler`. This selector is in addition to that `id`. The value for `props` should be a list of tuples of `('attribute', 'value')`.

`table_styles` are extremely flexible, but not as fun to type out by hand. We hope to collect some useful ones either in pandas, or preferable in a new package that *builds on top* the tools here.

Hiding the Index or Columns

The index can be hidden from rendering by calling `Styler.hide_index`. Columns can be hidden from rendering by calling `Styler.hide_columns` and passing in the name of a column, or a slice of columns.

[32]:
```
df.style.hide_index()
```

[32]:
```
<pandas.io.formats.style.Styler at 0x7fb446ddf208>
```

[33]:
```
df.style.hide_columns(['C','D'])
```

[33]:
```
<pandas.io.formats.style.Styler at 0x7fb446ddfef0>
```

CSS Classes

Certain CSS classes are attached to cells.

- Index and Column names include `index_name` and `level<k>` where `k` is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `row<n>` where `n` is the numeric position of the row
  - `level<k>` where `k` is the level in a MultiIndex
- Column label cells include
  - `col_heading`
  - `col<n>` where `n` is the numeric position of the column
  - `level<k>` where `k` is the level in a MultiIndex

- Blank cells include `blank`
- Data cells include `data`

Limitations

- DataFrame only `(use Series.to_frame().style)`
- The index and columns must be unique
- No large repr, and performance isn't great; this is intended for summary DataFrames
- You can only style the *values*, not the index or columns
- You can only apply styles, you can't insert new HTML entities

Some of these will be addressed in the future.

Terms

- Style function: a function that's passed into `Styler.apply` or `Styler.applymap` and returns values like `'css attribute: value'`
- Builtin style functions: style functions that are methods on `Styler`
- table style: a dictionary with the two keys `selector` and `props`. `selector` is the CSS selector that `props` will apply to. `props` is a list of `(attribute, value)` tuples. A list of table styles passed into `Styler`.

4.15.7 Fun stuff

Here are a few interesting examples.

`Styler` interacts pretty well with widgets. If you're viewing this online instead of running the notebook yourself, you're missing out on interactively adjusting the color palette.

[34]:
```
from IPython.html import widgets
@widgets.interact
def f(h_neg=(0, 359, 1), h_pos=(0, 359), s=(0., 99.9), l=(0., 99.9)):
    return df.style.background_gradient(
        cmap=sns.palettes.diverging_palette(h_neg=h_neg, h_pos=h_pos, s=s, l=l,
                                            as_cmap=True)
    )
```

```
<pandas.io.formats.style.Styler at 0x7fb446d9a6a0>
```

[35]:
```
def magnify():
    return [dict(selector="th",
                 props=[("font-size", "4pt")]),
            dict(selector="td",
                 props=[('padding', "0em 0em")]),
            dict(selector="th:hover",
                 props=[("font-size", "12pt")]),
            dict(selector="tr:hover td:hover",
                 props=[('max-width', '200px'),
                        ('font-size', '12pt')])
]
```

[36]:
```
np.random.seed(25)
cmap = cmap=sns.diverging_palette(5, 250, as_cmap=True)
bigdf = pd.DataFrame(np.random.randn(20, 25)).cumsum()

bigdf.style.background_gradient(cmap, axis=1)\
    .set_properties(**{'max-width': '80px', 'font-size': '1pt'})\
    .set_caption("Hover to magnify")\
    .set_precision(2)\
    .set_table_styles(magnify())
```

[36]:
```
<pandas.io.formats.style.Styler at 0x7fb446d86c88>
```

4.15.8 Export to Excel

New in version 0.20.0

Experimental: This is a new feature and still under development. We'll be adding features and possibly making breaking changes in future releases. We'd love to hear your feedback.

Some support is available for exporting styled `DataFrames` to Excel worksheets using the `OpenPyXL` or `XlsxWriter` engines. CSS2.2 properties handled include:

- `background-color`
- `border-style`, `border-width`, `border-color` and their {`top`, `right`, `bottom`, `left` variants}
- `color`
- `font-family`
- `font-style`
- `font-weight`
- `text-align`
- `text-decoration`
- `vertical-align`
- `white-space: nowrap`
- Only CSS2 named colors and hex colors of the form `#rgb` or `#rrggbb` are currently supported.
- The following pseudo CSS properties are also available to set excel specific style properties:
  - `number-format`

[37]:
```
df.style.\
    applymap(color_negative_red).\
    apply(highlight_max).\
    to_excel('styled.xlsx', engine='openpyxl')
```

A screenshot of the output:

	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

4.15.9 Extensibility

The core of pandas is, and will remain, its "high-performance, easy-to-use data structures". With that in mind, we hope that `DataFrame.style` accomplishes two goals

- Provide an API that is pleasing to use interactively and is "good enough" for many tasks
- Provide the foundations for dedicated libraries to build on

If you build a great library on top of this, let us know and we'll link to it.

Subclassing

If the default template doesn't quite suit your needs, you can subclass Styler and extend or override the template. We'll show an example of extending the default template to insert a custom header before each table.

[38]:
```
from jinja2 import Environment, ChoiceLoader, FileSystemLoader
from IPython.display import HTML
from pandas.io.formats.style import Styler
```

We'll use the following template:

[39]:
```
with open("templates/myhtml.tpl") as f:
    print(f.read())
```

```
{% extends "html.tpl" %}
{% block table %}
<h1>{{ table_title|default("My Table") }}</h1>
{{ super() }}
{% endblock table %}
```

Now that we've created a template, we need to set up a subclass of `Styler` that knows about it.

[40]:
```
class MyStyler(Styler):
    env = Environment(
        loader=ChoiceLoader([
            FileSystemLoader("templates"),  # contains ours
            Styler.loader,  # the default
```

(continues on next page)

(continued from previous page)

```
        ])
    )
    template = env.get_template("myhtml.tpl")
```

Notice that we include the original loader in our environment's loader. That's because we extend the original template, so the Jinja environment needs to be able to find it.

Now we can use that custom styler. It's `__init__` takes a DataFrame.

```
[41]: MyStyler(df)
```

```
[41]: <__main__.MyStyler at 0x7fb446ddf278>
```

Our custom template accepts a `table_title` keyword. We can provide the value in the `.render` method.

```
[42]: HTML(MyStyler(df).render(table_title="Extending Example"))
```

```
[42]: <IPython.core.display.HTML object>
```

For convenience, we provide the `Styler.from_custom_template` method that does the same as the custom subclass.

```
[43]: EasyStyler = Styler.from_custom_template("templates", "myhtml.tpl")
      EasyStyler(df)
```

```
[43]: <pandas.io.formats.style.Styler.from_custom_template.<locals>.MyStyler at␣
      ↪0x7fb4442ca710>
```

Here's the template structure:

```
[44]: with open("templates/template_structure.html") as f:
          structure = f.read()

      HTML(structure)
```

```
[44]: <IPython.core.display.HTML object>
```

See the template in the GitHub repo for more details.

4.16 Options and Settings

4.16.1 Overview

pandas has an options system that lets you customize some aspects of its behaviour, display-related options being those the user is most likely to adjust.

Options have a full "dotted-style", case-insensitive name (e.g. `display.max_rows`). You can get/set options directly as attributes of the top-level `options` attribute:

```
In [1]: import pandas as pd

In [2]: pd.options.display.max_rows
Out[2]: 15

In [3]: pd.options.display.max_rows = 999
```

(continues on next page)

(continued from previous page)

```
In [4]: pd.options.display.max_rows
Out[4]: 999
```

The API is composed of 5 relevant functions, available directly from the `pandas` namespace:

- *get_option()* / *set_option()* - get/set the value of a single option.
- *reset_option()* - reset one or more options to their default value.
- *describe_option()* - print the descriptions of one or more options.
- *option_context()* - execute a codeblock with a set of options that revert to prior settings after execution.

Note: Developers can check out pandas/core/config.py for more information.

All of the functions above accept a regexp pattern (`re.search` style) as an argument, and so passing in a substring will work - as long as it is unambiguous:

```
In [5]: pd.get_option("display.max_rows")
Out[5]: 999

In [6]: pd.set_option("display.max_rows", 101)

In [7]: pd.get_option("display.max_rows")
Out[7]: 101

In [8]: pd.set_option("max_r", 102)

In [9]: pd.get_option("display.max_rows")
Out[9]: 102
```

The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```
In [10]: try:
   ....:     pd.get_option("column")
   ....: except KeyError as e:
   ....:     print(e)
   ....:
'Pattern matched multiple keys'
```

Note: Using this form of shorthand may cause your code to break if new options with similar names are added in future versions.

You can get a list of available options and their descriptions with `describe_option`. When called with no argument `describe_option` will print out the descriptions for all available options.

4.16.2 Getting and Setting Options

As described above, *get_option()* and *set_option()* are available from the pandas namespace. To change an option, call `set_option('option regex', new_value)`.

```
In [11]: pd.get_option('mode.sim_interactive')
Out[11]: False

In [12]: pd.set_option('mode.sim_interactive', True)
```

(continues on next page)

(continued from previous page)

```
In [13]: pd.get_option('mode.sim_interactive')
Out[13]: True
```

Note: The option 'mode.sim_interactive' is mostly used for debugging purposes.

All options also have a default value, and you can use `reset_option` to do just that:

```
In [14]: pd.get_option("display.max_rows")
Out[14]: 60

In [15]: pd.set_option("display.max_rows", 999)

In [16]: pd.get_option("display.max_rows")
Out[16]: 999

In [17]: pd.reset_option("display.max_rows")

In [18]: pd.get_option("display.max_rows")
Out[18]: 60
```

It's also possible to reset multiple options at once (using a regex):

```
In [19]: pd.reset_option("^display")
```

`option_context` context manager has been exposed through the top-level API, allowing you to execute code with given option values. Option values are restored automatically when you exit the *with* block:

```
In [20]: with pd.option_context("display.max_rows", 10, "display.max_columns", 5):
   ....:     print(pd.get_option("display.max_rows"))
   ....:     print(pd.get_option("display.max_columns"))
   ....:
10
5

In [21]: print(pd.get_option("display.max_rows"))
\\\\\60

In [22]: print(pd.get_option("display.max_columns"))
\\\\\\\\0
```

4.16.3 Setting Startup Options in python/ipython Environment

Using startup scripts for the python/ipython environment to import pandas and set options makes working with pandas more efficient. To do this, create a .py or .ipy script in the startup directory of the desired profile. An example where the startup folder is in a default ipython profile can be found at:

```
$IPYTHONDIR/profile_default/startup
```

More information can be found in the ipython documentation. An example startup script for pandas is displayed below:

```
import pandas as pd
pd.set_option('display.max_rows', 999)
pd.set_option('precision', 5)
```

4.16.4 Frequently Used Options

The following is a walk-through of the more frequently used display options.

`display.max_rows` and `display.max_columns` sets the maximum number of rows and columns displayed when a frame is pretty-printed. Truncated lines are replaced by an ellipsis.

```
In [23]: df = pd.DataFrame(np.random.randn(7, 2))

In [24]: pd.set_option('max_rows', 7)

In [25]: df
Out[25]:
          0         1
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569
5 -0.494929  1.071804
6  0.721555 -0.706771

In [26]: pd.set_option('max_rows', 5)

In [27]: df
Out[27]:
           0         1
0   0.469112 -0.282863
1  -1.509059 -1.135632
..       ...       ...
5  -0.494929  1.071804
6   0.721555 -0.706771

[7 rows x 2 columns]

In [28]: pd.reset_option('max_rows')
```

`display.expand_frame_repr` allows for the representation of dataframes to stretch across pages, wrapped over the full column vs row-wise.

```
In [29]: df = pd.DataFrame(np.random.randn(5, 10))

In [30]: pd.set_option('expand_frame_repr', True)

In [31]: df
Out[31]:
          0         1         2         3         4         5         6         7  ␣
↪    8         9
0 -1.039575  0.271860 -0.424972  0.567020  0.276232 -1.087401 -0.673690  0.113648 -1.
↪478427  0.524988
1  0.404705  0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312  0.844885  1.
↪075770 -0.109050
2  1.643563 -1.469388  0.357021 -0.674600 -1.776904 -0.968914 -1.294524  0.413738  0.
↪276662 -0.472035
3 -0.013960 -0.362543 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.
↪431256  1.340309
4 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.
↪130127 -1.436737
```

(continues on next page)

(continued from previous page)

```
In [32]: pd.set_option('expand_frame_repr', False)

In [33]: df
Out[33]:
          0         1         2         3         4         5         6         7         ␣
↪     8         9
0 -1.039575  0.271860 -0.424972  0.567020  0.276232 -1.087401 -0.673690  0.113648 -1.
↪478427  0.524988
1  0.404705  0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312  0.844885  1.
↪075770 -0.109050
2  1.643563 -1.469388  0.357021 -0.674600 -1.776904 -0.968914 -1.294524  0.413738  0.
↪276662 -0.472035
3 -0.013960 -0.362543 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.
↪431256  1.340309
4 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.
↪130127 -1.436737

In [34]: pd.reset_option('expand_frame_repr')
```

`display.large_repr` lets you select whether to display dataframes that exceed `max_columns` or `max_rows` as a truncated frame, or as a summary.

```
In [35]: df = pd.DataFrame(np.random.randn(10, 10))

In [36]: pd.set_option('max_rows', 5)

In [37]: pd.set_option('large_repr', 'truncate')

In [38]: df
Out[38]:
           0         1         2         3         4         5         6         7         ␣
↪      8         9
0  -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747 -0.
↪410001 -0.078638
1   0.545952 -1.219217 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.
↪695775  0.341734
..       ...       ...       ...       ...       ...       ...       ...       ...       ␣
↪   ...       ...
8  -2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349  1.474071 -0.064034 -1.
↪282782  0.781836
9  -1.071357  0.441153  2.353925  0.583787  0.221471 -0.744471  0.758527  1.729689 -0.
↪964980 -0.845696

[10 rows x 10 columns]

In [39]: pd.set_option('large_repr', 'info')

In [40]: df
Out[40]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0    10 non-null float64
1    10 non-null float64
2    10 non-null float64
```

(continues on next page)

(continued from previous page)

```
3    10 non-null float64
4    10 non-null float64
5    10 non-null float64
6    10 non-null float64
7    10 non-null float64
8    10 non-null float64
9    10 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [41]: pd.reset_option('large_repr')

In [42]: pd.reset_option('max_rows')
```

display.max_colwidth sets the maximum width of columns. Cells of this length or longer will be truncated with an ellipsis.

```
In [43]: df = pd.DataFrame(np.array([['foo', 'bar', 'bim', 'uncomfortably long string
↪'],
   ....:                            ['horse', 'cow', 'banana', 'apple']]))
   ....:

In [44]: pd.set_option('max_colwidth', 40)

In [45]: df
Out[45]:
       0    1       2                          3
0    foo  bar     bim  uncomfortably long string
1  horse  cow  banana                      apple

In [46]: pd.set_option('max_colwidth', 6)

In [47]: df
Out[47]:
       0    1      2      3
0    foo  bar    bim  un...
1  horse  cow  ba...  apple

In [48]: pd.reset_option('max_colwidth')
```

display.max_info_columns sets a threshold for when by-column info will be given.

```
In [49]: df = pd.DataFrame(np.random.randn(10, 10))

In [50]: pd.set_option('max_info_columns', 11)

In [51]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0    10 non-null float64
1    10 non-null float64
2    10 non-null float64
3    10 non-null float64
4    10 non-null float64
5    10 non-null float64
```

(continues on next page)

(continued from previous page)

```
6    10 non-null float64
7    10 non-null float64
8    10 non-null float64
9    10 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [52]: pd.set_option('max_info_columns', 5)

In [53]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Columns: 10 entries, 0 to 9
dtypes: float64(10)
memory usage: 880.0 bytes

In [54]: pd.reset_option('max_info_columns')
```

`display.max_info_rows`: `df.info()` will usually show null-counts for each column. For large frames this can be quite slow. `max_info_rows` and `max_info_cols` limit this null check only to frames with smaller dimensions then specified. Note that you can specify the option `df.info(null_counts=True)` to override on showing a particular frame.

```
In [55]: df = pd.DataFrame(np.random.choice([0, 1, np.nan], size=(10, 10)))

In [56]: df
Out[56]:
     0    1    2    3    4    5    6    7    8    9
0  0.0  1.0  1.0  0.0  1.0  1.0  0.0  NaN  1.0  NaN
1  1.0  NaN  0.0  0.0  1.0  1.0  NaN  1.0  0.0  1.0
2  NaN  NaN  NaN  1.0  1.0  0.0  NaN  0.0  1.0  NaN
3  0.0  1.0  1.0  NaN  0.0  NaN  1.0  NaN  NaN  0.0
4  0.0  1.0  0.0  0.0  1.0  0.0  0.0  NaN  0.0  0.0
5  0.0  NaN  1.0  NaN  NaN  NaN  NaN  0.0  1.0  NaN
6  0.0  1.0  0.0  0.0  NaN  1.0  NaN  NaN  0.0  NaN
7  0.0  NaN  1.0  1.0  NaN  1.0  1.0  1.0  1.0  NaN
8  0.0  0.0  NaN  0.0  NaN  1.0  0.0  0.0  NaN  NaN
9  NaN  NaN  0.0  NaN  NaN  NaN  0.0  1.0  1.0  NaN

In [57]: pd.set_option('max_info_rows', 11)

In [58]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0    8 non-null float64
1    5 non-null float64
2    8 non-null float64
3    7 non-null float64
4    5 non-null float64
5    7 non-null float64
6    6 non-null float64
7    6 non-null float64
8    8 non-null float64
9    3 non-null float64
dtypes: float64(10)
```

(continues on next page)

(continued from previous page)

```
memory usage: 880.0 bytes

In [59]: pd.set_option('max_info_rows', 5)

In [60]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0    float64
1    float64
2    float64
3    float64
4    float64
5    float64
6    float64
7    float64
8    float64
9    float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [61]: pd.reset_option('max_info_rows')
```

`display.precision` sets the output display precision in terms of decimal places. This is only a suggestion.

```
In [62]: df = pd.DataFrame(np.random.randn(5, 5))

In [63]: pd.set_option('precision', 7)

In [64]: df
Out[64]:
           0          1          2          3          4
0 -2.0490276  2.8466122 -1.2080493 -0.4503923  2.4239054
1  0.1211080  0.2669165  0.8438259 -0.2225400  2.0219807
2 -0.7167894 -2.2244851 -1.0611370 -0.2328247  0.4307933
3 -0.6654779  1.8298075 -1.4065093  1.0782481  0.3227741
4  0.2003243  0.8900241  0.1948132  0.3516326  0.4488815

In [65]: pd.set_option('precision', 4)

In [66]: df
Out[66]:
        0       1       2       3       4
0 -2.0490  2.8466 -1.2080 -0.4504  2.4239
1  0.1211  0.2669  0.8438 -0.2225  2.0220
2 -0.7168 -2.2245 -1.0611 -0.2328  0.4308
3 -0.6655  1.8298 -1.4065  1.0782  0.3228
4  0.2003  0.8900  0.1948  0.3516  0.4489
```

`display.chop_threshold` sets at what level pandas rounds to zero when it displays a Series of DataFrame. This setting does not change the precision at which the number is stored.

```
In [67]: df = pd.DataFrame(np.random.randn(6, 6))

In [68]: pd.set_option('chop_threshold', 0)
```

(continues on next page)

(continued from previous page)

```
In [69]: df
Out[69]:
        0       1       2       3       4       5
0 -0.1979  0.9657 -1.5229 -0.1166  0.2956 -1.0477
1  1.6406  1.9058  2.7721  0.0888 -1.1442 -0.6334
2  0.9254 -0.0064 -0.8204 -0.6009 -1.0393  0.8248
3 -0.8241 -0.3377 -0.9278 -0.8401  0.2485 -0.1093
4  0.4320 -0.4607  0.3365 -3.2076 -1.5359  0.4098
5 -0.6731 -0.7411 -0.1109 -2.6729  0.8645  0.0609

In [70]: pd.set_option('chop_threshold', .5)

In [71]: df
Out[71]:
        0       1       2       3       4       5
0  0.0000  0.9657 -1.5229  0.0000  0.0000 -1.0477
1  1.6406  1.9058  2.7721  0.0000 -1.1442 -0.6334
2  0.9254  0.0000 -0.8204 -0.6009 -1.0393  0.8248
3 -0.8241  0.0000 -0.9278 -0.8401  0.0000  0.0000
4  0.0000  0.0000  0.0000 -3.2076 -1.5359  0.0000
5 -0.6731 -0.7411  0.0000 -2.6729  0.8645  0.0000

In [72]: pd.reset_option('chop_threshold')
```

`display.colheader_justify` controls the justification of the headers. The options are 'right', and 'left'.

```
In [73]: df = pd.DataFrame(np.array([np.random.randn(6),
   ....:                             np.random.randint(1, 9, 6) * .1,
   ....:                             np.zeros(6)]).T,
   ....:                   columns=['A', 'B', 'C'], dtype='float')
   ....:

In [74]: pd.set_option('colheader_justify', 'right')

In [75]: df
Out[75]:
        A    B    C
0  0.9331  0.3  0.0
1  0.2888  0.2  0.0
2  1.3250  0.2  0.0
3  0.5892  0.7  0.0
4  0.5314  0.1  0.0
5 -1.1987  0.7  0.0

In [76]: pd.set_option('colheader_justify', 'left')

In [77]: df
Out[77]:
   A       B    C
0  0.9331  0.3  0.0
1  0.2888  0.2  0.0
2  1.3250  0.2  0.0
3  0.5892  0.7  0.0
4  0.5314  0.1  0.0
5 -1.1987  0.7  0.0

In [78]: pd.reset_option('colheader_justify')
```

4.16.5 Available Options

Option	Default	Function
display.chop_threshold	None	If set to a float value, all float values smaller then the given threshold will be displaye
display.colheader_justify	right	Controls the justification of column headers. used by DataFrameFormatter.
display.column_space	12	No description available.
display.date_dayfirst	False	When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst	False	When True, prints and parses dates with the year first, eg 2005/01/20
display.encoding	UTF-8	Defaults to the detected encoding of the console. Specifies the encoding to be used fo
display.expand_frame_repr	True	Whether to print out the full DataFrame repr for wide DataFrames across multiple lin
display.float_format	None	The callable should accept a floating point number and return a string with the desire
display.large_repr	truncate	For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show
display.latex.repr	False	Whether to produce a latex DataFrame representation for jupyter frontends that suppo
display.latex.escape	True	Escapes special characters in DataFrames, when using the to_latex method.
display.latex.longtable	False	Specifies if the to_latex method of a DataFrame uses the longtable format.
display.latex.multicolumn	True	Combines columns when using a MultiIndex
display.latex.multicolumn_format	'l'	Alignment of multicolumn labels
display.latex.multirow	False	Combines rows when using a MultiIndex. Centered instead of top-aligned, separated
display.max_columns	0 or 20	max_rows and max_columns are used in __repr__() methods to decide if to_string()
display.max_colwidth	50	The maximum width in characters of a column in the repr of a pandas data structure.
display.max_info_columns	100	max_info_columns is used in DataFrame.info method to decide if per column inform
display.max_info_rows	1690785	df.info() will usually show null-counts for each column. For large frames this can be
display.max_rows	60	This sets the maximum number of rows pandas should output when printing out vario
display.max_seq_items	100	when pretty-printing a long sequence, no more then *max_seq_items* will be printed.
display.memory_usage	True	This specifies if the memory usage of a DataFrame should be displayed when the df.
display.multi_sparse	True	"Sparsify" MultiIndex display (don't display repeated elements in outer levels withi
display.notebook_repr_html	True	When True, IPython notebook will use html representation for pandas objects (if it is
display.pprint_nest_depth	3	Controls the number of nested levels to process when pretty-printing
display.precision	6	Floating point output precision in terms of number of places after the decimal, for re
display.show_dimensions	truncate	Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specifi
display.width	80	Width of the display in characters. In case python/IPython is running in a terminal th
display.html.table_schema	False	Whether to publish a Table Schema representation for frontends that support it.
display.html.border	1	A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTM
display.html.use_mathjax	True	When True, Jupyter notebook will process table contents using MathJax, rendering m
io.excel.xls.writer	xlwt	The default Excel writer engine for 'xls' files.
io.excel.xlsm.writer	openpyxl	The default Excel writer engine for 'xlsm' files. Available options: 'openpyxl' (the d
io.excel.xlsx.writer	openpyxl	The default Excel writer engine for 'xlsx' files.
io.hdf.default_format	None	default format writing format, if None, then put will default to 'fixed' and append wi
io.hdf.dropna_table	True	drop ALL nan rows when appending to a table
io.parquet.engine	None	The engine to use as a default for parquet reading and writing. If None then try 'pyar
mode.chained_assignment	warn	Controls `SettingWithCopyWarning`: 'raise', 'warn', or None. Raise an except
mode.sim_interactive	False	Whether to simulate interactive mode for purposes of testing.
mode.use_inf_as_na	False	True means treat None, NaN, -INF, INF as NA (old way), False means None and Na
compute.use_bottleneck	True	Use the bottleneck library to accelerate computation if it is installed.
compute.use_numexpr	True	Use the numexpr library to accelerate computation if it is installed.
plotting.matplotlib.register_converters	True	Register custom converters with matplotlib. Set to False to de-register.

4.16.6 Number Formatting

pandas also allows you to set how numbers are displayed in the console. This option is not set through the `set_options` API.

Use the `set_eng_float_format` function to alter the floating-point formatting of pandas objects to produce a particular format.

For instance:

```
In [79]: import numpy as np

In [80]: pd.set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [81]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [82]: s / 1.e3
Out[82]: 
a   -236.866u
b    846.974u
c   -685.597u
d    609.099u
e   -303.961u
dtype: float64

In [83]: s / 1.e6
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8
↪
a   -236.866n
b    846.974n
c   -685.597n
d    609.099n
e   -303.961n
dtype: float64
```

To round floats on a case-by-case basis, you can also use *round()* and *round()*.

4.16.7 Unicode Formatting

Warning: Enabling this option will affect the performance for printing of DataFrame and Series (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters whose width corresponds to two Latin characters. If a DataFrame or Series contains these characters, the default output mode may not align them properly.

Note: Screen captures are attached for each output to show the actual results.

```
In [84]: df = pd.DataFrame({u'': ['UK', u''], u'': ['Alice', u'']})

In [85]: df
Out[85]: 

0  UK  Alice
1
```

```
>>> df = pd.DataFrame({u'国籍': ['UK', u'日本'], u'名前': ['Alice', u'しのぶ']})
>>> df
      名前   国籍
0  Alice   UK
1    しのぶ  日本
```

Enabling `display.unicode.east_asian_width` allows pandas to check each character's "East Asian Width" property. These characters can be aligned properly by setting this option to `True`. However, this will result in longer render times than the standard `len` function.

```
In [86]: pd.set_option('display.unicode.east_asian_width', True)

In [87]: df
Out[87]: 

0     UK    Alice
1
```

```
>>> pd.set_option('display.unicode.east_asian_width', True)
>>> df
     名前   国籍
0   Alice    UK
1  しのぶ  日本
```

In addition, Unicode characters whose width is "Ambiguous" can either be 1 or 2 characters wide depending on the terminal setting or encoding. The option `display.unicode.ambiguous_as_wide` can be used to handle the ambiguity.

By default, an "Ambiguous" character's width, such as "¡" (inverted exclamation) in the example below, is taken to be 1.

```
In [88]: df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})

In [89]: df
Out[89]: 
     a    b
0  xxx  yyy
1   ¡¡   ¡¡
```

```
>>> df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
>>> df
     a    b
0  xxx  yyy
1   ¡¡   ¡¡
```

Enabling `display.unicode.ambiguous_as_wide` makes pandas interpret these characters' widths to be 2. (Note that this option will only be effective when `display.unicode.east_asian_width` is enabled.)

However, setting this option incorrectly for your terminal will cause these characters to be aligned incorrectly:

```
In [90]: pd.set_option('display.unicode.ambiguous_as_wide', True)

In [91]: df
Out[91]: 
      a     b
0   xxx   yyy
1  ¡¡  ¡¡
```

```
>>> pd.set_option('display.unicode.ambiguous_as_wide', True)
>>> df
     a     b
0   xxx   yyy
1  ¡¡   ¡¡
```

4.16.8 Table Schema Display

New in version 0.20.0.

`DataFrame` and `Series` will publish a Table Schema representation by default. False by default, this can be enabled globally with the `display.html.table_schema` option:

```
In [92]: pd.set_option('display.html.table_schema', True)
```

Only `'display.max_rows'` are serialized and published.

4.17 Enhancing Performance

In this part of the tutorial, we will investigate how to speed up certain functions operating on pandas `DataFrames` using three different techniques: Cython, Numba and *pandas.eval()*. We will see a speed improvement of ~200 when we use Cython and Numba on a test function operating row-wise on the `DataFrame`. Using *pandas.eval()* we will speed up a sum by an order of ~2.

4.17.1 Cython (Writing C extensions for pandas)

For many use cases writing pandas in pure Python and NumPy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in Python, for example by trying to remove for-loops and making use of NumPy vectorization. It's always worth optimising in Python first.

This tutorial walks through a "typical" process of cythonizing a slow computation. We use an example from the Cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure Python solution.

Pure python

We have a `DataFrame` to which we want to apply a function row-wise.

```
In [1]: df = pd.DataFrame({'a': np.random.randn(1000),
   ...:                    'b': np.random.randn(1000),
   ...:                    'N': np.random.randint(100, 1000, (1000)),
   ...:                    'x': 'x'})
   ...:

In [2]: df
Out[2]:
            a         b    N  x
0    0.469112 -0.218470  585  x
1   -0.282863 -0.061645  841  x
2   -1.509059 -0.723780  251  x
```

(continues on next page)

(continued from previous page)

```
3   -1.135632  0.551225  972  x
4    1.212112 -0.497767  181  x
5   -0.173215  0.837519  458  x
6    0.119209  1.103245  159  x
..        ...       ...  ... ..
993  0.131892  0.290162  190  x
994  0.342097  0.215341  931  x
995 -1.512743  0.874737  374  x
996  0.933753  1.120790  246  x
997 -0.308013  0.198768  157  x
998 -0.079915  1.757555  977  x
999 -1.010589 -1.115680  770  x

[1000 rows x 4 columns]
```

Here's the function in pure Python:

```
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:

In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...:
```

We achieve our result by using `apply` (row-wise):

```
In [7]: %timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 174 ms per loop
```

But clearly this isn't fast enough for us. Let's take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the prun ipython magic function:

```
In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)  #␣
↪noqa E999
         672216 function calls (667196 primitive calls) in 0.226 seconds

   Ordered by: internal time
   List reduced from 219 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.107    0.000    0.164    0.000 <ipython-input-4-c2a74e076cf0>
↪:1(integrate_f)
   552423    0.057    0.000    0.057    0.000 <ipython-input-3-c138bdd570e3>:1(f)
     3000    0.007    0.000    0.040    0.000 base.py:4287(get_value)
        1    0.005    0.005    0.224    0.224 {pandas._libs.reduction.reduce}
```

By far the majority of time is spend inside either `integrate_f` or `f`, hence we'll concentrate our efforts cythonizing these two functions.

Note: In Python 2 replacing the `range` with its generator counterpart (`xrange`) would mean the `range` line would vanish. In Python 3 `range` is already a generator.

Plain Cython

First we're going to need to import the Cython magic function to ipython:

```
In [6]: %load_ext Cython
```

Now, let's simply copy our functions over to Cython as is (the suffix is here to distinguish between function versions):

```
In [7]: %%cython
   ...: def f_plain(x):
   ...:     return x * (x - 1)
   ...: def integrate_f_plain(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_plain(a + i * dx)
   ...:     return s * dx
   ...:
```

Note: If you're having trouble pasting the above into your ipython, you may need to be using bleeding edge ipython for paste to play well with cell magics.

```
In [4]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 85.5 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

Adding type

We get another huge improvement simply by providing type information:

```
In [8]: %%cython
   ...: cdef double f_typed(double x) except? -2:
   ...:     return x * (x - 1)
   ...: cpdef double integrate_f_typed(double a, double b, int N):
   ...:     cdef int i
   ...:     cdef double s, dx
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_typed(a + i * dx)
   ...:     return s * dx
   ...:
```

```
In [4]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 20.3 ms per loop
```

Now, we're talking! It's now over ten times faster than the original python implementation, and we haven't *really* modified the code. Let's have another look at what's eating up time:

```
In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']),␣
↪axis=1)
         119791 function calls (114771 primitive calls) in 0.051 seconds

   Ordered by: internal time
   List reduced from 216 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3000    0.006    0.000    0.032    0.000 base.py:4287(get_value)
     3000    0.003    0.000    0.037    0.000 series.py:865(__getitem__)
        1    0.003    0.003    0.048    0.048 {pandas._libs.reduction.reduce}
     9264    0.003    0.000    0.006    0.000 {built-in method builtins.getattr}
```

Using ndarray

It's calling series… a lot! It's creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in Python, so maybe we could minimize these by cythonizing the apply part.

Note: We are now passing ndarrays into the Cython function, fortunately Cython plays very nicely with NumPy.

```
In [10]: %%cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_
↪b,
   ....:                                            np.ndarray col_N):
   ....:     assert (col_a.dtype == np.float
   ....:             and col_b.dtype == np.float and col_N.dtype == np.int)
   ....:     cdef Py_ssize_t i, n = len(col_N)
   ....:     assert (len(col_a) == len(col_b) == n)
   ....:     cdef np.ndarray[double] res = np.empty(n)
   ....:     for i in range(len(col_a)):
   ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
   ....:     return res
   ....:
```

The implementation is simple, it creates an array of zeros and loops over the rows, applying our `integrate_f_typed`, and putting this in the zeros array.

Warning: You can **not pass** a `Series` directly as a `ndarray` typed parameter to a Cython function. Instead pass the actual `ndarray` using the *Series.to_numpy()*. The reason is that the Cython definition is specific to an ndarray and not the passed `Series`.

So, do not do this:

```
apply_integrate_f(df['a'], df['b'], df['N'])
```

But rather, use *Series.to_numpy()* to get the underlying `ndarray`:

```
apply_integrate_f(df['a'].to_numpy(),
                  df['b'].to_numpy(),
                  df['N'].to_numpy())
```

Note: Loops like this would be *extremely* slow in Python, but in Cython looping over NumPy arrays is *fast*.

```
In [4]: %timeit apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
1000 loops, best of 3: 1.25 ms per loop
```

We've gotten another big improvement. Let's check again where the time is spent:

```
In [11]: %prun -l 4 apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
         197 function calls in 0.002 seconds

   Ordered by: internal time
   List reduced from 53 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.001    0.001 {built-in method _cython_magic_
↪62b00698d71527c516fe264bd42a374f.apply_integrate_f}
        3    0.000    0.000    0.001    0.000 frame.py:2883(__getitem__)
        3    0.000    0.000    0.000    0.000 managers.py:963(iget)
        1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
```

As one might expect, the majority of the time is now spent in `apply_integrate_f`, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

More advanced techniques

There is still hope for improvement. Here's an example of using some more advanced Cython techniques:

```
In [12]: %%cython
   ....: cimport cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....: @cython.boundscheck(False)
   ....: @cython.wraparound(False)
   ....: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a,
```

(continues on next page)

(continued from previous page)

```
   ....:                                                    np.ndarray[double] col_b,
   ....:                                                    np.ndarray[int] col_N):
   ....:     cdef int i, n = len(col_N)
   ....:     assert len(col_a) == len(col_b) == n
   ....:     cdef np.ndarray[double] res = np.empty(n)
   ....:     for i in range(n):
   ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
   ....:     return res
   ....:
```

```
In [4]: %timeit apply_integrate_f_wrap(df['a'].values, df['b'].values, df['N'].values)
1000 loops, best of 3: 987 us per loop
```

Even faster, with the caveat that a bug in our Cython code (an off-by-one error, for example) might cause a segfault because memory access isn't checked. For more about `boundscheck` and `wraparound`, see the Cython docs on compiler directives.

4.17.2 Using Numba

A recent alternative to statically compiling Cython code, is to use a *dynamic jit-compiler*, Numba.

Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.

Note: You will need to install Numba. This is easy with `conda`, by using: `conda install numba`, see *installing using miniconda*.

Note: As of Numba version 0.20, pandas objects cannot be passed directly to Numba-compiled functions. Instead, one must pass the NumPy array underlying the pandas object to the Numba-compiled function as demonstrated below.

Jit

We demonstrate how to use Numba to just-in-time compile our code. We simply take the plain Python code from above and annotate with the `@jit` decorator.

```
import numba


@numba.jit
def f_plain(x):
    return x * (x - 1)


@numba.jit
def integrate_f_numba(a, b, N):
```

(continues on next page)

(continued from previous page)

```
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_plain(a + i * dx)
    return s * dx


@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
    n = len(col_N)
    result = np.empty(n, dtype='float64')
    assert len(col_a) == len(col_b) == n
    for i in range(n):
        result[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
    return result


def compute_numba(df):
    result = apply_integrate_f_numba(df['a'].values, df['b'].values,
                                     df['N'].values)
    return pd.Series(result, index=df.index, name='result')
```

Note that we directly pass NumPy arrays to the Numba function. `compute_numba` is just a wrapper that provides a nicer interface by passing/returning pandas objects.

```
In [4]: %timeit compute_numba(df)
1000 loops, best of 3: 798 us per loop
```

In this example, using Numba was faster than Cython.

Vectorize

Numba can also be used to write vectorized functions that do not require the user to explicitly loop over the observations of a vector; a vectorized function will be applied to each row automatically. Consider the following toy example of doubling each observation:

```
import numba


def double_every_value_nonumba(x):
    return x * 2


@numba.vectorize
def double_every_value_withnumba(x):  # noqa E501
    return x * 2
```

```
# Custom function without numba
In [5]: %timeit df['col1_doubled'] = df.a.apply(double_every_value_nonumba)  # noqa␣
↪E501
1000 loops, best of 3: 797 us per loop

# Standard implementation (faster than a custom function)
In [6]: %timeit df['col1_doubled'] = df.a * 2
1000 loops, best of 3: 233 us per loop
```

(continues on next page)

(continued from previous page)

```
# Custom function with numba
In [7]: %timeit (df['col1_doubled'] = double_every_value_withnumba(df.a.values)
1000 loops, best of 3: 145 us per loop
```

Caveats

Note: Numba will execute on any function, but can only accelerate certain classes of functions.

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. When passed a function that only uses operations it knows how to accelerate, it will execute in `nopython` mode.

If Numba is passed a function that includes something it doesn't know how to work with – a category that currently includes sets, lists, dictionaries, or string functions – it will revert to `object mode`. In `object mode`, Numba will execute but your code will not speed up significantly. If you would prefer that Numba throw an error if it cannot compile a function in a way that speeds up your code, pass Numba the argument `nopython=True` (e.g. `@numba.jit(nopython=True)`). For more on troubleshooting Numba modes, see the Numba troubleshooting page.

Read more in the Numba docs.

4.17.3 Expression Evaluation via `eval()`

The top-level function *`pandas.eval()`* implements expression evaluation of *`Series`* and *`DataFrame`* objects.

Note: To benefit from using *`eval()`* you need to install `numexpr`. See the *recommended dependencies section* for more details.

The point of using *`eval()`* for expression evaluation rather than plain Python is two-fold: 1) large *`DataFrame`* objects are evaluated more efficiently and 2) large arithmetic and boolean expressions are evaluated all at once by the underlying engine (by default `numexpr` is used for evaluation).

Note: You should not use *`eval()`* for simple expressions or for expressions involving small DataFrames. In fact, *`eval()`* is many orders of magnitude slower for smaller expressions/objects than plain ol' Python. A good rule of thumb is to only use *`eval()`* when you have a `DataFrame` with more than 10,000 rows.

`eval()` supports all arithmetic expressions supported by the engine in addition to some extensions available only in pandas.

Note: The larger the frame and the larger the expression the more speedup you will see from using *`eval()`*.

Supported Syntax

These operations are supported by *`pandas.eval()`*:

- Arithmetic operations except for the left shift (<<) and right shift (>>) operators, e.g., `df + 2 * pi / s ** 4 % 42 - the_golden_ratio`

- Comparison operations, including chained comparisons, e.g., `2 < df < df2`
- Boolean operations, e.g., `df < df2 and df3 < df4 or not df_bool`
- `list` and `tuple` literals, e.g., `[1, 2]` or `(1, 2)`
- Attribute access, e.g., `df.a`
- Subscript expressions, e.g., `df[0]`
- Simple variable evaluation, e.g., `pd.eval('df')` (this is not very useful)
- Math functions: *sin*, *cos*, *exp*, *log*, *expm1*, *log1p*, *sqrt*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan*, *arccosh*, *arcsinh*, *arctanh*, *abs*, *arctan2* and *log10*.

This Python syntax is **not** allowed:

- Expressions
  - Function calls other than math functions.
  - `is`/`is not` operations
  - `if` expressions
  - `lambda` expressions
  - `list`/`set`/`dict` comprehensions
  - Literal `dict` and `set` expressions
  - `yield` expressions
  - Generator expressions
  - Boolean expressions consisting of only scalar values
- Statements
  - Neither simple nor compound statements are allowed. This includes things like `for`, `while`, and `if`.

`eval()` Examples

`pandas.eval()` works well with expressions containing large arrays.

First let's create a few decent-sized arrays to play with:

```
In [13]: nrows, ncols = 20000, 100

In [14]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols)) for _ in␣
↪range(4)]
```

Now let's compare adding them together using plain ol' Python versus *`eval()`*:

```
In [15]: %timeit df1 + df2 + df3 + df4
18 ms +- 1.46 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [16]: %timeit pd.eval('df1 + df2 + df3 + df4')
7.59 ms +- 912 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Now let's do the same thing but with comparisons:

```
In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
189 ms +- 8.58 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [18]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
14 ms +- 625 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

`eval()` also works with unaligned pandas objects:

```
In [19]: s = pd.Series(np.random.randn(50))

In [20]: %timeit df1 + df2 + df3 + df4 + s
90.5 ms +- 6.73 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [21]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
8.85 ms +- 911 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Note: Operations such as

```
1 and 2  # would parse to 1 & 2, but should evaluate to 2
3 or 4  # would parse to 3 | 4, but should evaluate to 3
~1  # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type `bool` or `np.bool_`. Again, you should perform these kinds of operations in plain Python.

The `DataFrame.eval` method

In addition to the top level `pandas.eval()` function you can also evaluate an expression in the "context" of a `DataFrame`.

```
In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=['a', 'b'])

In [23]: df.eval('a + b')
Out[23]: 
0   -0.246747
1    0.867786
2   -1.626063
3   -1.134978
4   -1.027798
dtype: float64
```

Any expression that is a valid `pandas.eval()` expression is also a valid `DataFrame.eval()` expression, with the added benefit that you don't have to prefix the name of the `DataFrame` to the column(s) you're interested in evaluating.

In addition, you can perform assignment of columns within an expression. This allows for *formulaic evaluation*. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

New in version 0.18.0.

The `inplace` keyword determines whether this assignment will performed on the original `DataFrame` or return a copy with the new column.

Warning: For backwards compatibility, `inplace` defaults to `True` if not specified. This will change in a future version of pandas - if your code depends on an inplace assignment you should update to explicitly set `inplace=True`.

```
In [24]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [25]: df.eval('c = a + b', inplace=True)

In [26]: df.eval('d = a + b + c', inplace=True)

In [27]: df.eval('a = 1', inplace=True)

In [28]: df
Out[28]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

When `inplace` is set to `False`, a copy of the `DataFrame` with the new or modified columns is returned and the original frame is unchanged.

```
In [29]: df
Out[29]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26

In [30]: df.eval('e = a - c', inplace=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b   c   d   e
0  1  5   5  10  -4
1  1  6   7  14  -6
2  1  7   9  18  -8
3  1  8  11  22 -10
4  1  9  13  26 -12

In [31]: df
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

New in version 0.18.0.

As a convenience, multiple assignments can be performed by using a multi-line string.

```
In [32]: df.eval("""
   ....: c = a + b
   ....: d = a + b + c
   ....: a = 1""", inplace=False)
   ....:
Out[32]:
   a  b   c   d
0  1  5   6  12
1  1  6   7  14
2  1  7   8  16
3  1  8   9  18
4  1  9  10  20
```

The equivalent in standard Python would be

```
In [33]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [34]: df['c'] = df.a + df.b

In [35]: df['d'] = df.a + df.b + df.c

In [36]: df['a'] = 1

In [37]: df
Out[37]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

New in version 0.18.0.

The `query` method gained the `inplace` keyword which determines whether the query modifies the original frame.

```
In [38]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [39]: df.query('a > 2')
Out[39]:
   a  b
3  3  8
4  4  9

In [40]: df.query('a > 2', inplace=True)

In [41]: df
Out[41]:
   a  b
3  3  8
4  4  9
```

> **Warning:** Unlike with `eval`, the default value for `inplace` for `query` is `False`. This is consistent with prior versions of pandas.

Local Variables

You must *explicitly reference* any local variable that you want to use in an expression by placing the `@` character in front of the name. For example,

```
In [42]: df = pd.DataFrame(np.random.randn(5, 2), columns=list('ab'))

In [43]: newcol = np.random.randn(len(df))

In [44]: df.eval('b + @newcol')
Out[44]:
0   -0.173926
1    2.493083
2   -0.881831
3   -0.691045
4    1.334703
dtype: float64

In [45]: df.query('b < @newcol')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[45]:
↪
          a         b
0  0.863987 -0.115998
2 -2.621419 -1.297879
```

If you don't prefix the local variable with `@`, pandas will raise an exception telling you the variable is undefined.

When using *DataFrame.eval()* and *DataFrame.query()*, this allows you to have a local variable and a *DataFrame* column with the same name in an expression.

```
In [46]: a = np.random.randn()

In [47]: df.query('@a < a')
Out[47]:
          a         b
0  0.863987 -0.115998

In [48]: df.loc[a < df.a]  # same as the previous expression
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[48]:
          a         b
0  0.863987 -0.115998
```

With *pandas.eval()* you cannot use the `@` prefix *at all*, because it isn't defined in that context. `pandas` will let you know this if you try to use `@` in a top-level call to *pandas.eval()*. For example,

```
In [49]: a, b = 1, 2

In [50]: pd.eval('@a + b')
Traceback (most recent call last):

  File "/opt/conda/envs/pandas/lib/python3.6/site-packages/IPython/core/
↪interactiveshell.py", line 3267, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-50-af17947a194f>", line 1, in <module>
    pd.eval('@a + b')

  File "/pandas/pandas/core/computation/eval.py", line 286, in eval
```

(continues on next page)

(continued from previous page)

```
    _check_for_locals(expr, level, parser)

  File "/pandas/pandas/core/computation/eval.py", line 153, in _check_for_locals
    raise SyntaxError(msg)

  File "<string>", line unknown
SyntaxError: The '@' prefix is not allowed in top-level eval calls,
please refer to your variables by name without the '@' prefix
```

In this case, you should simply refer to the variables like you would in standard Python.

```
In [51]: pd.eval('a + b')
Out[51]: 3
```

pandas.eval() Parsers

There are two different parsers and two different engines you can use as the backend.

The default `'pandas'` parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the `&` and `|` operators is made equal to the precedence of the corresponding boolean operations `and` and `or`.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the `'python'` parser to enforce strict Python semantics.

```
In [52]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'

In [53]: x = pd.eval(expr, parser='python')

In [54]: expr_no_parens = 'df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0'

In [55]: y = pd.eval(expr_no_parens, parser='pandas')

In [56]: np.all(x == y)
Out[56]: True
```

The same expression can be "anded" together with the word `and` as well:

```
In [57]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'

In [58]: x = pd.eval(expr, parser='python')

In [59]: expr_with_ands = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'

In [60]: y = pd.eval(expr_with_ands, parser='pandas')

In [61]: np.all(x == y)
Out[61]: True
```

The `and` and `or` operators here have the same precedence that they would in vanilla Python.

pandas.eval() Backends

There's also the option to make `eval()` operate identical to plain ol' Python.

Note: Using the `'python'` engine is generally *not* useful, except for testing other evaluation engines against it. You will achieve **no** performance benefits using *eval()* with `engine='python'` and in fact may incur a performance hit.

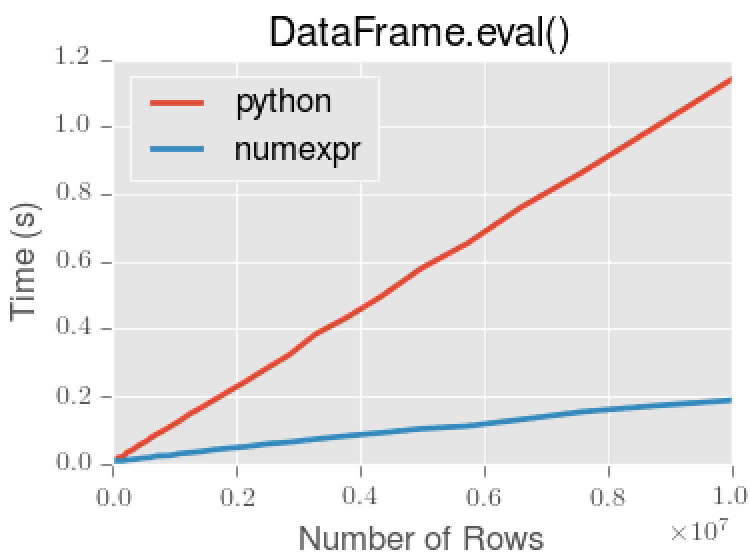
You can see this by using *pandas.eval()* with the `'python'` engine. It is a bit slower (not by much) than evaluating the same expression in Python

```
In [62]: %timeit df1 + df2 + df3 + df4
17.9 ms +- 1.92 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

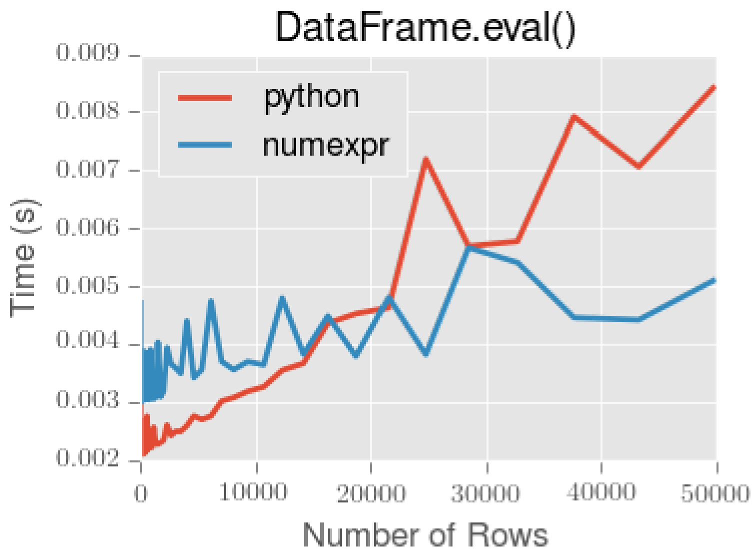
```
In [63]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
19 ms +- 2.34 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

`pandas.eval()` Performance

eval() is intended to speed up certain kinds of operations. In particular, those operations involving complex expressions with large *DataFrame*/*Series* objects should see a significant performance benefit. Here is a plot showing the running time of *pandas.eval()* as function of the size of the frame involved in the computation. The two lines are two different engines.



Note: Operations with smallish objects (around 15k-20k rows) are faster using plain Python:



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

Technical Minutia Regarding Expression Evaluation

Expressions that would result in an object dtype or involve datetime operations (because of `NaT`) must be evaluated in Python space. The main reason for this behavior is to maintain backwards compatibility with versions of NumPy < 1.7. In those versions of NumPy a call to `ndarray.astype(str)` will truncate any strings that are more than 60 characters in length. Second, we can't pass `object` arrays to `numexpr` thus string comparisons must be evaluated in Python space.

The upshot is that this *only* applies to object-dtype expressions. So, if you have an expression–for example

```
In [64]: df = pd.DataFrame({'strings': np.repeat(list('cba'), 3),
   ....:                    'nums': np.repeat(range(3), 3)})
   ....:

In [65]: df
Out[65]:
  strings  nums
0       c     0
1       c     0
2       c     0
3       b     1
4       b     1
5       b     1
6       a     2
7       a     2
8       a     2

In [66]: df.query('strings == "a" and nums == 1')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Empty DataFrame
```

(continues on next page)

(continued from previous page)

```
Columns: [strings, nums]
Index: []
```

the numeric part of the comparison (`nums == 1`) will be evaluated by `numexpr`.

In general, *`DataFrame.query()`*/*`pandas.eval()`* will evaluate the subexpressions that *can* be evaluated by `numexpr` and those that must be evaluated in Python space transparently to the user. This is done by inferring the result type of an expression from its arguments and operators.

4.18 Sparse data structures

Note: The `SparsePanel` class has been removed in 0.19.0

We have implemented "sparse" versions of `Series` and `DataFrame`. These are not sparse in the typical "mostly 0". Rather, you can view these objects as being "compressed" where any data matching a specific value (`NaN` / missing value, though any value can be chosen) is omitted. A special `SparseIndex` object tracks where data has been "sparsified". This will make much more sense with an example. All of the standard pandas data structures have a `to_sparse` method:

```
In [1]: ts = pd.Series(np.random.randn(10))

In [2]: ts[2:-2] = np.nan

In [3]: sts = ts.to_sparse()

In [4]: sts
Out[4]:
0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The `to_sparse` method takes a `kind` argument (for the sparse index, see below) and a `fill_value`. So if we had a mostly zero `Series`, we could convert it to sparse with `fill_value=0`:

```
In [5]: ts.fillna(0).to_sparse(fill_value=0)
Out[5]:
0    0.469112
1   -0.282863
2    0.000000
3    0.000000
4    0.000000
```

(continues on next page)

(continued from previous page)

```
5    0.000000
6    0.000000
7    0.000000
8   -0.861849
9   -2.104569
dtype: Sparse[float64, 0]
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA `DataFrame`:

```
In [6]: df = pd.DataFrame(np.random.randn(10000, 4))

In [7]: df.iloc[:9998] = np.nan

In [8]: sdf = df.to_sparse()

In [9]: sdf
Out[9]:
             0         1         2         3
0          NaN       NaN       NaN       NaN
1          NaN       NaN       NaN       NaN
2          NaN       NaN       NaN       NaN
3          NaN       NaN       NaN       NaN
4          NaN       NaN       NaN       NaN
5          NaN       NaN       NaN       NaN
6          NaN       NaN       NaN       NaN
...        ...       ...       ...       ...
9993       NaN       NaN       NaN       NaN
9994       NaN       NaN       NaN       NaN
9995       NaN       NaN       NaN       NaN
9996       NaN       NaN       NaN       NaN
9997       NaN       NaN       NaN       NaN
9998  0.509184 -0.774928 -1.369894 -0.382141
9999  0.280249 -1.648493  1.490865 -0.890819

[10000 rows x 4 columns]

In [10]: sdf.density
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪0.0002
```

As you can see, the density (% of values that have not been "compressed") is extremely low. This sparse object takes up much less memory on disk (pickled) and in the Python interpreter. Functionally, their behavior should be nearly identical to their dense counterparts.

Any sparse object can be converted back to the standard dense form by calling `to_dense`:

```
In [11]: sts.to_dense()
Out[11]:
0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
```

(continues on next page)

(continued from previous page)

```
6          NaN
7          NaN
8    -0.861849
9    -2.104569
dtype: float64
```

4.18.1 Sparse Accessor

New in version 0.24.0.

Pandas provides a `.sparse` accessor, similar to `.str` for string data, `.cat` for categorical data, and `.dt` for datetime-like data. This namespace provides attributes and methods that are specific to sparse data.

```
In [12]: s = pd.Series([0, 0, 1, 2], dtype="Sparse[int]")

In [13]: s.sparse.density
Out[13]: 0.5

In [14]: s.sparse.fill_value
\\\\\\\\\\\\\Out[14]: 0
```

This accessor is available only on data with `SparseDtype`, and on the *`Series`* class itself for creating a Series with sparse data from a scipy COO matrix with.

4.18.2 SparseArray

`SparseArray` is the base layer for all of the sparse indexed data structures. It is a 1-dimensional ndarray-like object storing only values distinct from the `fill_value`:

```
In [15]: arr = np.random.randn(10)

In [16]: arr[2:5] = np.nan

In [17]: arr[7:8] = np.nan

In [18]: sparr = pd.SparseArray(arr)

In [19]: sparr
Out[19]:
[-1.95566352972, -1.6588664276, nan, nan, nan, 1.15893288864, 0.145297113733, nan, 0.
↪606027190513, 1.33421134013]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

Like the indexed objects (SparseSeries, SparseDataFrame), a `SparseArray` can be converted back to a regular ndarray by calling `to_dense`:

```
In [20]: sparr.to_dense()
Out[20]:
array([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
          nan,  0.606 ,  1.3342])
```

4.18.3 SparseIndex objects

Two kinds of `SparseIndex` are implemented, `block` and `integer`. We recommend using `block` as it's more memory efficient. The `integer` format keeps an arrays of all of the locations where the data are not equal to the fill value. The `block` format tracks only the locations and sizes of blocks of data.

4.18.4 Sparse Dtypes

Sparse data should have the same dtype as its dense representation. Currently, `float64`, `int64` and `bool` dtypes are supported. Depending on the original dtype, `fill_value` default changes:

- `float64`: `np.nan`
- `int64`: `0`
- `bool`: `False`

```
In [21]: s = pd.Series([1, np.nan, np.nan])

In [22]: s
Out[22]:
0    1.0
1    NaN
2    NaN
dtype: float64

In [23]: s.to_sparse()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:
0    1.0
1    NaN
2    NaN
dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [24]: s = pd.Series([1, 0, 0])

In [25]: s
Out[25]:
0    1
1    0
2    0
dtype: int64

In [26]: s.to_sparse()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]:
0    1
1    0
2    0
dtype: Sparse[int64, 0]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [27]: s = pd.Series([True, False, True])
```

(continues on next page)

(continued from previous page)

```
In [28]: s
Out[28]:
0     True
1    False
2     True
dtype: bool

In [29]: s.to_sparse()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[29]:
0     True
1    False
2     True
dtype: Sparse[bool, False]
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)
```

You can change the dtype using `.astype()`, the result is also sparse. Note that `.astype()` also affects to the `fill_value` to keep its dense representation.

```
In [30]: s = pd.Series([1, 0, 0, 0, 0])

In [31]: s
Out[31]:
0    1
1    0
2    0
3    0
4    0
dtype: int64

In [32]: ss = s.to_sparse()

In [33]: ss
Out[33]:
0    1
1    0
2    0
3    0
4    0
dtype: Sparse[int64, 0]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [34]: ss.astype(np.float64)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1.0
1    0.0
2    0.0
3    0.0
4    0.0
dtype: Sparse[float64, 0.0]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)
```

It raises if any value cannot be coerced to specified dtype.

```
In [1]: ss = pd.Series([1, np.nan, np.nan]).to_sparse()
Out[1]:
0    1.0
1    NaN
2    NaN
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [2]: ss.astype(np.int64)
Out[2]:
ValueError: unable to coerce current fill_value nan to int64 dtype
```

4.18.5 Sparse Calculation

You can apply NumPy *ufuncs* to `SparseArray` and get a `SparseArray` as a result.

```
In [35]: arr = pd.SparseArray([1., np.nan, np.nan, -2., np.nan])

In [36]: np.abs(arr)
Out[36]:
[1.0, nan, nan, 2.0, nan]
Fill: nan
IntIndex
Indices: array([0, 3], dtype=int32)
```

The *ufunc* is also applied to `fill_value`. This is needed to get the correct dense result.

```
In [37]: arr = pd.SparseArray([1., -1, -1, -2., -1], fill_value=-1)

In [38]: np.abs(arr)
Out[38]:
[1.0, 1, 1, 2.0, 1]
Fill: 1
IntIndex
Indices: array([0, 3], dtype=int32)

In [39]: np.abs(arr).to_dense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[39]:␣
↪array([ 1.,  1.,  1.,  2.,  1.])
```

4.18.6 Interaction with scipy.sparse

SparseDataFrame

New in version 0.20.0.

Pandas supports creating sparse dataframes directly from `scipy.sparse` matrices.

```
In [40]: from scipy.sparse import csr_matrix
```

(continues on next page)

(continued from previous page)

```
In [41]: arr = np.random.random(size=(1000, 5))

In [42]: arr[arr < .9] = 0

In [43]: sp_arr = csr_matrix(arr)

In [44]: sp_arr
Out[44]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 517 stored elements in Compressed Sparse Row format>

In [45]: sdf = pd.SparseDataFrame(sp_arr)

In [46]: sdf
Out[46]:
            0   1        2         3   4
0    0.956380 NaN      NaN       NaN NaN
1         NaN NaN      NaN       NaN NaN
2         NaN NaN      NaN       NaN NaN
3         NaN NaN      NaN       NaN NaN
4    0.999552 NaN      NaN  0.956153 NaN
5         NaN NaN      NaN       NaN NaN
6    0.913638 NaN      NaN       NaN NaN
..        ...  ..      ...       ...  ..
993       NaN NaN      NaN       NaN NaN
994       NaN NaN      NaN       NaN NaN
995       NaN NaN      NaN  0.998834 NaN
996       NaN NaN      NaN       NaN NaN
997       NaN NaN      NaN       NaN NaN
998       NaN NaN  0.95659       NaN NaN
999       NaN NaN      NaN       NaN NaN

[1000 rows x 5 columns]
```

All sparse formats are supported, but matrices that are not in `COOrdinate` format will be converted, copying data as needed. To convert a `SparseDataFrame` back to sparse SciPy matrix in COO format, you can use the *SparseDataFrame.to_coo()* method:

```
In [47]: sdf.to_coo()
Out[47]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 517 stored elements in COOrdinate format>
```

SparseSeries

A *SparseSeries.to_coo()* method is implemented for transforming a `SparseSeries` indexed by a `MultiIndex` to a `scipy.sparse.coo_matrix`.

The method requires a `MultiIndex` with two or more levels.

```
In [48]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [49]: s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
   ....:                                      (1, 2, 'a', 1),
   ....:                                      (1, 1, 'b', 0),
```

(continues on next page)

(continued from previous page)

```
   ....:                                      (1, 1, 'b', 1),
   ....:                                      (2, 1, 'b', 0),
   ....:                                      (2, 1, 'b', 1)],
   ....:                                     names=['A', 'B', 'C', 'D'])
   ....:

In [50]: s
Out[50]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
dtype: float64

# SparseSeries
In [51]: ss = s.to_sparse()

In [52]: ss
Out[52]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 2], dtype=int32)
```

In the example below, we transform the `SparseSeries` to a sparse representation of a 2-d array by specifying that the first and second `MultiIndex` levels define labels for the rows and the third and fourth levels define labels for the columns. We also specify that the column and row labels should be sorted in the final sparse representation.

```
In [53]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
   ....:                              column_levels=['C', 'D'],
   ....:                              sort_labels=True)
   ....:

In [54]: A
Out[54]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [55]: A.todense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
matrix([[ 0.,  0.,  1.,  3.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])

In [56]: rows
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[(1, 1), (1, 2), (2, 1)]

In [57]: columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

```
In [58]: A, rows, columns = ss.to_coo(row_levels=['A', 'B', 'C'],
   ....:                              column_levels=['D'],
   ....:                              sort_labels=False)
   ....:

In [59]: A
Out[59]:
<3x2 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [60]: A.todense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
matrix([[ 3.,  0.],
        [ 1.,  3.],
        [ 0.,  0.]])

In [61]: rows
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[(1, 2, 'a'), (1, 1, 'b'), (2, 1, 'b')]

In [62]: columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[0, 1]
```

A convenience method *SparseSeries.from_coo()* is implemented for creating a `SparseSeries` from a `scipy.sparse.coo_matrix`.

```
In [63]: from scipy import sparse

In [64]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
   ....:                       shape=(3, 4))
   ....:

In [65]: A
Out[65]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [66]: A.todense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
```

The default behaviour (with `dense_index=False`) simply returns a `SparseSeries` containing only the non-null entries.

```
In [67]: ss = pd.SparseSeries.from_coo(A)

In [68]: ss
Out[68]:
0  2    1.0
   3    2.0
1  0    3.0
dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

Specifying `dense_index=True` will result in an index that is the Cartesian product of the row and columns coordinates of the matrix. Note that this will consume a significant amount of memory (relative to `dense_index=False`) if the sparse matrix is large (and sparse) enough.

```
In [69]: ss_dense = pd.SparseSeries.from_coo(A, dense_index=True)

In [70]: ss_dense
Out[70]:
0  0    NaN
   1    NaN
   2    1.0
   3    2.0
1  0    3.0
   1    NaN
   2    NaN
   3    NaN
2  0    NaN
   1    NaN
   2    NaN
   3    NaN
dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([2], dtype=int32)
Block lengths: array([3], dtype=int32)
```

4.19 Frequently Asked Questions (FAQ)

4.19.1 DataFrame memory usage

The memory usage of a `DataFrame` (including the index) is shown when calling the *info()*. A configuration option, `display.memory_usage` (see *the list of options*), specifies if the `DataFrame`'s memory usage will be displayed when invoking the `df.info()` method.

For example, the memory usage of the `DataFrame` below is shown when calling *info()*:

```
In [1]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
   ...:           'complex128', 'object', 'bool']
   ...:

In [2]: n = 5000

In [3]: data = {t: np.random.randint(100, size=n).astype(t) for t in dtypes}
```

(continues on next page)

(continued from previous page)

```
In [4]: df = pd.DataFrame(data)

In [5]: df['categorical'] = df['object'].astype('category')

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

The + symbol indicates that the true memory usage could be higher, because pandas does not count the memory used by values in columns with `dtype=object`.

Passing `memory_usage='deep'` will enable a more accurate memory usage report, accounting for the full usage of the contained objects. This is optional as it can be expensive to do this deeper introspection.

```
In [7]: df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
int64              5000 non-null int64
float64            5000 non-null float64
datetime64[ns]     5000 non-null datetime64[ns]
timedelta64[ns]    5000 non-null timedelta64[ns]
complex128         5000 non-null complex128
object             5000 non-null object
bool               5000 non-null bool
categorical        5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↪object(1), timedelta64[ns](1)
memory usage: 425.6 KB
```

By default the display option is set to `True` but can be explicitly overridden by passing the `memory_usage` argument when invoking `df.info()`.

The memory usage of each column can be found by calling the *memory_usage()* method. This returns a `Series` with an index represented by column names and memory usage of each column shown in bytes. For the `DataFrame` above, the memory usage of each column and the total memory usage can be found with the `memory_usage` method:

```
In [8]: df.memory_usage()
Out[8]:
Index                 80
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
```

(continues on next page)

(continued from previous page)

```
complex128         80000
object             40000
bool                5000
categorical        10920
dtype: int64

# total memory usage of dataframe
In [9]: df.memory_usage().sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪296000
```

By default the memory usage of the `DataFrame`'s index is shown in the returned `Series`, the memory usage of the index can be suppressed by passing the `index=False` argument:

```
In [10]: df.memory_usage(index=False)
Out[10]:
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
complex128         80000
object             40000
bool                5000
categorical        10920
dtype: int64
```

The memory usage displayed by the *info()* method utilizes the *memory_usage()* method to determine the memory usage of a `DataFrame` while also formatting the output in human-readable units (base-2 representation; i.e. 1KB = 1024 bytes).

See also *Categorical Memory Usage*.

4.19.2 Using If/Truth Statements with pandas

pandas follows the NumPy convention of raising an error when you try to convert something to a `bool`. This happens in an `if`-statement or when using the boolean operations: `and`, `or`, and `not`. It is not clear what the result of the following code should be:

```
>>> if pd.Series([False, True, False]):
...     pass
```

Should it be `True` because it's not zero-length, or `False` because there are `False` values? It is unclear, so instead, pandas raises a `ValueError`:

```
>>> if pd.Series([False, True, False]):
...     print("I was true")
Traceback
    ...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

You need to explicitly choose what you want to do with the `DataFrame`, e.g. use *any()*, *all()* or *empty()*. Alternatively, you might want to compare if the pandas object is `None`:

```
>>> if pd.Series([False, True, False]) is not None:
...     print("I was not None")
I was not None
```

Below is how to check if any of the values are `True`:

```
>>> if pd.Series([False, True, False]).any():
...     print("I am any")
I am any
```

To evaluate single-element pandas objects in a boolean context, use the method *bool()*:

```
In [11]: pd.Series([True]).bool()
Out[11]: True

In [12]: pd.Series([False]).bool()
\\\\\\\\\\\\\\Out[12]: False

In [13]: pd.DataFrame([[True]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]: True

In [14]: pd.DataFrame([[False]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: False
```

Bitwise boolean

Bitwise boolean operators like == and != return a boolean `Series`, which is almost always what you want anyways.

```
>>> s = pd.Series(range(5))
>>> s == 4
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

See *boolean comparisons* for more examples.

Using the `in` operator

Using the Python `in` operator on a `Series` tests for membership in the index, not membership among the values.

```
In [15]: s = pd.Series(range(5), index=list('abcde'))

In [16]: 2 in s
Out[16]: False

In [17]: 'b' in s
\\\\\\\\\\\\\\\Out[17]: True
```

If this behavior is surprising, keep in mind that using `in` on a Python dictionary tests keys, not values, and `Series` are dict-like. To test for membership in the values, use the method *isin()*:

```
In [18]: s.isin([2])
Out[18]:
a    False
b    False
c     True
d    False
e    False
dtype: bool

In [19]: s.isin([2]).any()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:␣
↪True
```

For DataFrames, likewise, in applies to the column axis, testing for membership in the list of column names.

4.19.3 NaN, Integer NA values and NA type promotions

Choice of NA representation

For lack of NA (missing) support from the ground up in NumPy and Python in general, we were given the difficult choice between either:

- A *masked array* solution: an array of data and an array of boolean values indicating whether a value is there or is missing.
- Using a special sentinel value, bit pattern, or set of sentinel values to denote NA across the dtypes.

For many reasons we chose the latter. After years of production use it has proven, at least in my opinion, to be the best decision given the state of affairs in NumPy and Python in general. The special value NaN (Not-A-Number) is used everywhere as the NA value, and there are API functions isna and notna which can be used across the dtypes to detect NA values.

However, it comes with it a couple of trade-offs which I most certainly have not ignored.

Support for integer NA

In the absence of high performance NA support being built into NumPy from the ground up, the primary casualty is the ability to represent NAs in integer arrays. For example:

```
In [20]: s = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))

In [21]: s
Out[21]:
a    1
b    2
c    3
d    4
e    5
dtype: int64

In [22]: s.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]: dtype('int64')

In [23]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])
```

(continues on next page)

(continued from previous page)

```
In [24]: s2
Out[24]:
a    1.0
b    2.0
c    3.0
f    NaN
u    NaN
dtype: float64

In [25]: s2.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]: dtype(
↪'float64')
```

This trade-off is made largely for memory and performance reasons, and also so that the resulting `Series` continues to be "numeric".

If you need to represent integers with possibly missing values, use one of the nullable-integer extension dtypes provided by pandas

- *Int8Dtype*
- *Int16Dtype*
- *Int32Dtype*
- *Int64Dtype*

```
In [26]: s_int = pd.Series([1, 2, 3, 4, 5], index=list('abcde'),
   ....:                   dtype=pd.Int64Dtype())
   ....:

In [27]: s_int
Out[27]:
a    1
b    2
c    3
d    4
e    5
dtype: Int64

In [28]: s_int.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]: Int64Dtype()

In [29]: s2_int = s_int.reindex(['a', 'b', 'c', 'f', 'u'])

In [30]: s2_int
Out[30]:
a      1
b      2
c      3
f    NaN
u    NaN
dtype: Int64

In [31]: s2_int.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[31]:␣
↪Int64Dtype()
```

See *Nullable Integer Data Type* for more.

NA type promotions

When introducing NAs into an existing Series or DataFrame via *reindex()* or some other means, boolean and integer types will be promoted to a different dtype in order to store the NAs. The promotions are summarized in this table:

Typeclass	Promotion dtype for storing NAs
floating	no change
object	no change
integer	cast to float64
boolean	cast to object

While this may seem like a heavy trade-off, I have found very few cases where this is an issue in practice i.e. storing values greater than 2**53. Some explanation for the motivation is in the next section.

Why not make NumPy like R?

Many people have suggested that NumPy should simply emulate the NA support present in the more domain-specific statistical programming language R. Part of the reason is the NumPy type hierarchy:

Typeclass	Dtypes
numpy.floating	float16, float32, float64, float128
numpy.integer	int8, int16, int32, int64
numpy.unsignedinteger	uint8, uint16, uint32, uint64
numpy.object_	object_
numpy.bool_	bool_
numpy.character	string_, unicode_

The R language, by contrast, only has a handful of built-in data types: integer, numeric (floating-point), character, and boolean. NA types are implemented by reserving special bit patterns for each type to be used as the missing value. While doing this with the full NumPy type hierarchy would be possible, it would be a more substantial trade-off (especially for the 8- and 16-bit data types) and implementation undertaking.

An alternate approach is that of using masked arrays. A masked array is an array of data with an associated boolean *mask* denoting whether each value should be considered NA or not. I am personally not in love with this approach as I feel that overall it places a fairly heavy burden on the user and the library implementer. Additionally, it exacts a fairly high performance cost when working with numerical data compared with the simple approach of using NaN. Thus, I have chosen the Pythonic "practicality beats purity" approach and traded integer NA capability for a much simpler approach of using a special value in float and object arrays to denote NA, and promoting integer arrays to floating when NAs must be introduced.

4.19.4 Differences with NumPy

For Series and DataFrame objects, *var()* normalizes by N-1 to produce unbiased estimates of the sample variance, while NumPy's var normalizes by N, which measures the variance of the sample. Note that *cov()* normalizes by N-1 in both pandas and NumPy.

4.19.5 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the *copy()* method. If you are doing a lot of copying of DataFrame objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See this link for more information.

4.19.6 Byte-Ordering Issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. A common symptom of this issue is an error like::

```
Traceback
    ...
ValueError: Big-endian buffer not supported on little-endian compiler
```

To deal with this issue you should convert the underlying NumPy array to the native system byte order *before* passing it to `Series` or `DataFrame` constructors using something similar to the following:

```
In [32]: x = np.array(list(range(10)), '>i4')   # big endian

In [33]: newx = x.byteswap().newbyteorder()   # force native byteorder

In [34]: s = pd.Series(newx)
```

See the NumPy documentation on byte order for more details.

4.20 Cookbook

This is a repository for *short and sweet* examples and links for useful pandas recipes. We encourage users to add to this documentation.

Adding interesting links and/or inline examples to this section is a great *First Pull Request*.

Simplified, condensed, new-user friendly, in-line examples have been inserted where possible to augment the Stack-Overflow and GitHub links. Many of the links contain expanded information, above what the in-line examples offer.

Pandas (pd) and Numpy (np) are the only two abbreviated imported modules. The rest are kept explicitly imported for newer users.

These examples are written for Python 3. Minor tweaks might be necessary for earlier python versions.

4.20.1 Idioms

These are some neat pandas `idioms`

if-then/if-then-else on one column, and assignment to another one or more columns:

```
In [1]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ...:                    'BBB': [10, 20, 30, 40],
   ...:                    'CCC': [100, 50, -30, -50]})
   ...:

In [2]: df
Out[2]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
```

(continues on next page)

(continued from previous page)

```
2     6   30   -30
3     7   40   -50
```

if-then...

An if-then on one column

```
In [3]: df.loc[df.AAA >= 5, 'BBB'] = -1

In [4]: df
Out[4]:
   AAA  BBB  CCC
0    4   10  100
1    5   -1   50
2    6   -1  -30
3    7   -1  -50
```

An if-then with assignment to 2 columns:

```
In [5]: df.loc[df.AAA >= 5, ['BBB', 'CCC']] = 555

In [6]: df
Out[6]:
   AAA  BBB  CCC
0    4   10  100
1    5  555  555
2    6  555  555
3    7  555  555
```

Add another line with different logic, to do the -else

```
In [7]: df.loc[df.AAA < 5, ['BBB', 'CCC']] = 2000

In [8]: df
Out[8]:
   AAA   BBB   CCC
0    4  2000  2000
1    5   555   555
2    6   555   555
3    7   555   555
```

Or use pandas where after you've set up a mask

```
In [9]: df_mask = pd.DataFrame({'AAA': [True] * 4,
   ...:                         'BBB': [False] * 4,
   ...:                         'CCC': [True, False] * 2})
   ...:

In [10]: df.where(df_mask, -1000)
Out[10]:
   AAA   BBB   CCC
0    4 -1000  2000
1    5 -1000 -1000
2    6 -1000   555
3    7 -1000 -1000
```

if-then-else using numpy's where()

```
In [11]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [12]: df
Out[12]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [13]: df['logic'] = np.where(df['AAA'] > 5, 'high', 'low')

In [14]: df
Out[14]:
   AAA  BBB  CCC logic
0    4   10  100   low
1    5   20   50   low
2    6   30  -30  high
3    7   40  -50  high
```

Splitting

Split a frame with a boolean criterion

```
In [15]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [16]: df
Out[16]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [17]: df[df.AAA <= 5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪
   AAA  BBB  CCC
0    4   10  100
1    5   20   50

In [18]: df[df.AAA > 5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   AAA  BBB  CCC
2    6   30  -30
3    7   40  -50
```

Building Criteria

Select with multi-column criteria

```
In [19]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [20]: df
Out[20]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
```

...and (without assignment returns a Series)

```
In [21]: df.loc[(df['BBB'] < 25) & (df['CCC'] >= -40), 'AAA']
Out[21]:
0    4
1    5
Name: AAA, dtype: int64
```

...or (without assignment returns a Series)

```
In [22]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= -40), 'AAA']
Out[22]:
0    4
1    5
2    6
3    7
Name: AAA, dtype: int64
```

...or (with assignment modifies the DataFrame.)

```
In [23]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= 75), 'AAA'] = 0.1

In [24]: df
Out[24]:
   AAA  BBB  CCC
0  0.1   10  100
1  5.0   20   50
2  0.1   30  -30
3  0.1   40  -50
```

Select rows with data closest to certain value using argsort

```
In [25]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [26]: df
Out[26]:
   AAA  BBB  CCC
0    4   10  100
```

(continues on next page)

(continued from previous page)

```
1    5   20   50
2    6   30  -30
3    7   40  -50

In [27]: aValue = 43.0

In [28]: df.loc[(df.CCC - aValue).abs().argsort()]
Out[28]:
   AAA  BBB  CCC
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

Dynamically reduce a list of criteria using a binary operators

```
In [29]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [30]: df
Out[30]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [31]: Crit1 = df.AAA <= 5.5

In [32]: Crit2 = df.BBB == 10.0

In [33]: Crit3 = df.CCC > -40.0
```

One could hard code:

```
In [34]: AllCrit = Crit1 & Crit2 & Crit3
```

. . . Or it can be done with a list of dynamically built criteria

```
In [35]: import functools

In [36]: CritList = [Crit1, Crit2, Crit3]

In [37]: AllCrit = functools.reduce(lambda x, y: x & y, CritList)

In [38]: df[AllCrit]
Out[38]:
   AAA  BBB  CCC
0    4   10  100
```

4.20.2 Selection

DataFrames

The *indexing* docs.

Using both row labels and value conditionals

```
In [39]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [40]: df
Out[40]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [41]: df[(df.AAA <= 6) & (df.index.isin([0, 2, 4]))]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4
↪
   AAA  BBB  CCC
0    4   10  100
2    6   30  -30
```

Use loc for label-oriented slicing and iloc positional slicing

```
In [42]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]},
   ....:                   index=['foo', 'bar', 'boo', 'kar'])
   ....:
```

There are 2 explicit slicing methods, with a third general case

1. Positional-oriented (Python slicing style : exclusive of end)
2. Label-oriented (Non-Python slicing style : inclusive of end)
3. General (Either slicing style : depends on if the slice contains labels or positions)

```
In [43]: df.loc['bar':'kar']  # Label
Out[43]:
     AAA  BBB  CCC
bar    5   20   50
boo    6   30  -30
kar    7   40  -50

# Generic
In [44]: df.iloc[0:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[44]:␣
↪
     AAA  BBB  CCC
foo    4   10  100
bar    5   20   50
boo    6   30  -30

In [45]: df.loc['bar':'kar']
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     AAA  BBB  CCC
bar    5   20   50
boo    6   30  -30
kar    7   40  -50
```

Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.

```
In [46]: data = {'AAA': [4, 5, 6, 7],
   ....:         'BBB': [10, 20, 30, 40],
   ....:         'CCC': [100, 50, -30, -50]}
   ....:

In [47]: df2 = pd.DataFrame(data=data, index=[1, 2, 3, 4])  # Note index starts at 1.

In [48]: df2.iloc[1:3]  # Position-oriented
Out[48]:
   AAA  BBB  CCC
2    5   20   50
3    6   30  -30

In [49]: df2.loc[1:3]  # Label-oriented
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[49]:
   AAA  BBB  CCC
1    4   10  100
2    5   20   50
3    6   30  -30
```

Using inverse operator (~) to take the complement of a mask

```
In [50]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [51]: df
Out[51]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [52]: df[~((df.AAA <= 6) & (df.index.isin([0, 2, 4])))]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[52
↪
   AAA  BBB  CCC
1    5   20   50
3    7   40  -50
```

Panels

Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions

```
In [53]: rng = pd.date_range('1/1/2013', periods=100, freq='D')

In [54]: data = np.random.randn(100, 4)

In [55]: cols = ['A', 'B', 'C', 'D']

In [56]: df1 = pd.DataFrame(data, rng, cols)

In [57]: df2 = pd.DataFrame(data, rng, cols)

In [58]: df3 = pd.DataFrame(data, rng, cols)

In [59]: pf = pd.Panel({'df1': df1, 'df2': df2, 'df3': df3})

In [60]: pf
Out[60]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 4 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to D

In [61]: pf.loc[:, :, 'F'] = pd.DataFrame(data, rng, cols)

In [62]: pf
Out[62]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 5 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to F
```

Mask a panel by using np.where and then reconstructing the panel with the new masked values

New Columns

Efficiently and dynamically creating new columns using applymap

```
In [63]: df = pd.DataFrame({'AAA': [1, 2, 1, 3],
   ....:                    'BBB': [1, 1, 2, 2],
   ....:                    'CCC': [2, 1, 3, 1]})
   ....:

In [64]: df
Out[64]:
   AAA  BBB  CCC
0    1    1    2
1    2    1    1
2    1    2    3
3    3    2    1

In [65]: source_cols = df.columns   # Or some subset would work too

In [66]: new_cols = [str(x) + "_cat" for x in source_cols]

In [67]: categories = {1: 'Alpha', 2: 'Beta', 3: 'Charlie'}
```

(continues on next page)

(continued from previous page)

```
In [68]: df[new_cols] = df[source_cols].applymap(categories.get)

In [69]: df
Out[69]:
   AAA  BBB  CCC  AAA_cat BBB_cat  CCC_cat
0    1    1    2    Alpha   Alpha     Beta
1    2    1    1     Beta   Alpha    Alpha
2    1    2    3    Alpha    Beta  Charlie
3    3    2    1  Charlie    Beta    Alpha
```

Keep other columns when using min() with groupby

```
In [70]: df = pd.DataFrame({'AAA': [1, 1, 1, 2, 2, 2, 3, 3],
   ....:                    'BBB': [2, 1, 3, 4, 5, 1, 2, 3]})
   ....:

In [71]: df
Out[71]:
   AAA  BBB
0    1    2
1    1    1
2    1    3
3    2    4
4    2    5
5    2    1
6    3    2
7    3    3
```

Method 1 : idxmin() to get the index of the minimums

```
In [72]: df.loc[df.groupby("AAA")["BBB"].idxmin()]
Out[72]:
   AAA  BBB
1    1    1
5    2    1
6    3    2
```

Method 2 : sort then take first of each

```
In [73]: df.sort_values(by="BBB").groupby("AAA", as_index=False).first()
Out[73]:
   AAA  BBB
0    1    1
1    2    1
2    3    2
```

Notice the same results, with the exception of the index.

4.20.3 MultiIndexing

The *multindexing* docs.

Creating a MultiIndex from a labeled frame

```
In [74]: df = pd.DataFrame({'row': [0, 1, 2],
   ....:                    'One_X': [1.1, 1.1, 1.1],
   ....:                    'One_Y': [1.2, 1.2, 1.2],
   ....:                    'Two_X': [1.11, 1.11, 1.11],
   ....:                    'Two_Y': [1.22, 1.22, 1.22]})
   ....:

In [75]: df
Out[75]:
   row  One_X  One_Y  Two_X  Two_Y
0    0    1.1    1.2   1.11   1.22
1    1    1.1    1.2   1.11   1.22
2    2    1.1    1.2   1.11   1.22

# As Labelled Index
In [76]: df = df.set_index('row')

In [77]: df
Out[77]:
     One_X  One_Y  Two_X  Two_Y
row
0      1.1    1.2   1.11   1.22
1      1.1    1.2   1.11   1.22
2      1.1    1.2   1.11   1.22

# With Hierarchical Columns
In [78]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split('_'))
   ....:                                          for c in df.columns])
   ....:

In [79]: df
Out[79]:
     One        Two
       X    Y     X     Y
row
0    1.1  1.2  1.11  1.22
1    1.1  1.2  1.11  1.22
2    1.1  1.2  1.11  1.22

# Now stack & Reset
In [80]: df = df.stack(0).reset_index(1)

In [81]: df
Out[81]:
    level_1     X     Y
row
0       One  1.10  1.20
0       Two  1.11  1.22
1       One  1.10  1.20
1       Two  1.11  1.22
2       One  1.10  1.20
2       Two  1.11  1.22

# And fix the labels (Notice the label 'level_1' got added automatically)
In [82]: df.columns = ['Sample', 'All_X', 'All_Y']

In [83]: df
```

(continues on next page)

(continued from previous page)

```
Out[83]:
    Sample  All_X  All_Y
row
0      One   1.10   1.20
0      Two   1.11   1.22
1      One   1.10   1.20
1      Two   1.11   1.22
2      One   1.10   1.20
2      Two   1.11   1.22
```

Arithmetic

Performing arithmetic with a MultiIndex that needs broadcasting

```
In [84]: cols = pd.MultiIndex.from_tuples([(x, y) for x in ['A', 'B', 'C']
   ....:                                   for y in ['O', 'I']])
   ....:

In [85]: df = pd.DataFrame(np.random.randn(2, 6), index=['n', 'm'], columns=cols)

In [86]: df
Out[86]:
          A                   B                   C
          O         I         O         I         O         I
n  1.920906 -0.388231 -2.314394  0.665508  0.402562  0.399555
m -1.765956  0.850423  0.388054  0.992312  0.744086 -0.739776

In [87]: df = df.div(df['C'], level=1)

In [88]: df
Out[88]:
          A                   B              C
          O         I         O         I    O    I
n  4.771702 -0.971660 -5.749162  1.665625  1.0  1.0
m -2.373321 -1.149568  0.521518 -1.341367  1.0  1.0
```

Slicing

Slicing a MultiIndex with xs

```
In [89]: coords = [('AA', 'one'), ('AA', 'six'), ('BB', 'one'), ('BB', 'two'),
   ....:           ('BB', 'six')]
   ....:

In [90]: index = pd.MultiIndex.from_tuples(coords)

In [91]: df = pd.DataFrame([11, 22, 33, 44, 55], index, ['MyData'])

In [92]: df
Out[92]:
        MyData
AA one      11
   six      22
BB one      33
```

(continues on next page)

(continued from previous page)

```
   two       44
   six       55
```

To take the cross section of the 1st level and 1st axis the index:

```
# Note : level and axis are optional, and default to zero
In [93]: df.xs('BB', level=0, axis=0)
Out[93]:
     MyData
one      33
two      44
six      55
```

. . . and now the 2nd level of the 1st axis.

```
In [94]: df.xs('six', level=1, axis=0)
Out[94]:
    MyData
AA      22
BB      55
```

Slicing a MultiIndex with xs, method #2

```
In [95]: import itertools

In [96]: index = list(itertools.product(['Ada', 'Quinn', 'Violet'],
   ....:                                ['Comp', 'Math', 'Sci']))
   ....:

In [97]: headr = list(itertools.product(['Exams', 'Labs'], ['I', 'II']))

In [98]: indx = pd.MultiIndex.from_tuples(index, names=['Student', 'Course'])

In [99]: cols = pd.MultiIndex.from_tuples(headr)   # Notice these are un-named

In [100]: data = [[70 + x + y + (x * y) % 3 for x in range(4)] for y in range(9)]

In [101]: df = pd.DataFrame(data, indx, cols)

In [102]: df
Out[102]:
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Comp      70  71   72  73
        Math      71  73   75  74
        Sci       72  75   75  75
Quinn   Comp      73  74   75  76
        Math      74  76   78  77
        Sci       75  78   78  78
Violet  Comp      76  77   78  79
        Math      77  79   81  80
        Sci       78  81   81  81

In [103]: All = slice(None)

In [104]: df.loc['Violet']
```

(continues on next page)

(continued from previous page)

```
Out[104]:
       Exams     Labs
           I  II    I  II
Course
Comp      76  77   78  79
Math      77  79   81  80
Sci       78  81   81  81

In [105]: df.loc[(All, 'Math'), All]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Math      71  73   75  74
Quinn   Math      74  76   78  77
Violet  Math      77  79   81  80

In [106]: df.loc[(slice('Ada', 'Quinn'), 'Math'), All]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Math      71  73   75  74
Quinn   Math      74  76   78  77

In [107]: df.loc[(All, 'Math'), ('Exams')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                 I  II
Student Course
Ada     Math    71  73
Quinn   Math    74  76
Violet  Math    77  79

In [108]: df.loc[(All, 'Math'), (All, 'II')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               Exams Labs
                  II   II
Student Course
Ada     Math      73   74
Quinn   Math      76   77
Violet  Math      79   80
```

Setting portions of a MultiIndex with xs

Sorting

Sort by specific column or an ordered list of columns, with a MultiIndex

```
In [109]: df.sort_values(by=('Labs', 'II'), ascending=False)
Out[109]:
               Exams     Labs
```

(continues on next page)

(continued from previous page)

```
                   I   II     I   II
Student  Course
Violet   Sci       78   81    81   81
         Math      77   79    81   80
         Comp      76   77    78   79
Quinn    Sci       75   78    78   78
         Math      74   76    78   77
         Comp      73   74    75   76
Ada      Sci       72   75    75   75
         Math      71   73    75   74
         Comp      70   71    72   73
```

Partial Selection, the need for sortedness;

Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

4.20.4 Missing Data

The *missing data* docs.

Fill forward a reversed timeseries

```
In [110]: df = pd.DataFrame(np.random.randn(6, 1),
   .....:                   index=pd.date_range('2013-08-01', periods=6, freq='B'),
   .....:                   columns=list('A'))
   .....:

In [111]: df.loc[df.index[3], 'A'] = np.nan

In [112]: df
Out[112]:
                   A
2013-08-01 -1.054874
2013-08-02 -0.179642
2013-08-05  0.639589
2013-08-06       NaN
2013-08-07  1.906684
2013-08-08  0.104050

In [113]: df.reindex(df.index[::-1]).ffill()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A
2013-08-08  0.104050
2013-08-07  1.906684
2013-08-06  1.906684
2013-08-05  0.639589
2013-08-02 -0.179642
2013-08-01 -1.054874
```

cumsum reset at NaN values

Replace

Using replace with backrefs

4.20.5 Grouping

The *grouping* docs.

Basic grouping with apply

Unlike agg, apply's callable is passed a sub-DataFrame which gives you access to all the columns

```
In [114]: df = pd.DataFrame({'animal': 'cat dog cat fish dog cat cat'.split(),
   .....:                    'size': list('SSMMMLL'),
   .....:                    'weight': [8, 10, 11, 1, 20, 12, 12],
   .....:                    'adult': [False] * 5 + [True] * 2})
   .....:

In [115]: df
Out[115]:
  animal size  weight  adult
0    cat    S       8  False
1    dog    S      10  False
2    cat    M      11  False
3   fish    M       1  False
4    dog    M      20  False
5    cat    L      12   True
6    cat    L      12   True

# List the size of the animals with the highest weight.
In [116]: df.groupby('animal').apply(lambda subf: subf['size'][subf['weight'].
↪idxmax()])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
animal
cat     L
dog     M
fish    M
dtype: object
```

Using get_group

```
In [117]: gb = df.groupby(['animal'])

In [118]: gb.get_group('cat')
Out[118]:
  animal size  weight  adult
0    cat    S       8  False
2    cat    M      11  False
5    cat    L      12   True
6    cat    L      12   True
```

Apply to different items in a group

```
In [119]: def GrowUp(x):
   .....:     avg_weight = sum(x[x['size'] == 'S'].weight * 1.5)
   .....:     avg_weight += sum(x[x['size'] == 'M'].weight * 1.25)
```

(continues on next page)

(continued from previous page)

```
   .....:     avg_weight += sum(x[x['size'] == 'L'].weight)
   .....:     avg_weight /= len(x)
   .....:     return pd.Series(['L', avg_weight, True],
   .....:                      index=['size', 'weight', 'adult'])
   .....:

In [120]: expected_df = gb.apply(GrowUp)

In [121]: expected_df
Out[121]:
       size   weight  adult
animal
cat       L  12.4375   True
dog       L  20.0000   True
fish      L   1.2500   True
```

Expanding Apply

```
In [122]: S = pd.Series([i / 100.0 for i in range(1, 11)])

In [123]: def cum_ret(x, y):
   .....:     return x * (1 + y)
   .....:

In [124]: def red(x):
   .....:     return functools.reduce(cum_ret, x, 1.0)
   .....:

In [125]: S.expanding().apply(red, raw=True)
Out[125]:
0    1.010000
1    1.030200
2    1.061106
3    1.103550
4    1.158728
5    1.228251
6    1.314229
7    1.419367
8    1.547110
9    1.701821
dtype: float64
```

Replacing some values with mean of the rest of a group

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

In [127]: gb = df.groupby('A')

In [128]: def replace(g):
   .....:     mask = g < 0
   .....:     return g.where(mask, g[~mask].mean())
   .....:

In [129]: gb.transform(replace)
Out[129]:
     B
0  1.0
```

(continues on next page)

(continued from previous page)

```
1 -1.0
2  1.5
3  1.5
```

Sort groups by aggregated data

```
In [130]: df = pd.DataFrame({'code': ['foo', 'bar', 'baz'] * 2,
   .....:                    'data': [0.16, -0.21, 0.33, 0.45, -0.59, 0.62],
   .....:                    'flag': [False, True] * 3})
   .....:

In [131]: code_groups = df.groupby('code')

In [132]: agg_n_sort_order = code_groups[['data']].transform(sum).sort_values(by='data
↪')

In [133]: sorted_df = df.loc[agg_n_sort_order.index]

In [134]: sorted_df
Out[134]:
  code  data   flag
1  bar -0.21   True
4  bar -0.59  False
0  foo  0.16  False
3  foo  0.45   True
2  baz  0.33  False
5  baz  0.62   True
```

Create multiple aggregated columns

```
In [135]: rng = pd.date_range(start="2014-10-07", periods=10, freq='2min')

In [136]: ts = pd.Series(data=list(range(10)), index=rng)

In [137]: def MyCust(x):
   .....:     if len(x) > 2:
   .....:         return x[1] * 1.234
   .....:     return pd.NaT
   .....:

In [138]: mhc = {'Mean': np.mean, 'Max': np.max, 'Custom': MyCust}

In [139]: ts.resample("5min").apply(mhc)
Out[139]:
Custom  2014-10-07 00:00:00    1.234
        2014-10-07 00:05:00      NaT
        2014-10-07 00:10:00    7.404
        2014-10-07 00:15:00      NaT
Max     2014-10-07 00:00:00        2
        2014-10-07 00:05:00        4
        2014-10-07 00:10:00        7
        2014-10-07 00:15:00        9
Mean    2014-10-07 00:00:00        1
        2014-10-07 00:05:00      3.5
        2014-10-07 00:10:00        6
        2014-10-07 00:15:00      8.5
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [140]: ts
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2014-10-07 00:00:00    0
2014-10-07 00:02:00    1
2014-10-07 00:04:00    2
2014-10-07 00:06:00    3
2014-10-07 00:08:00    4
2014-10-07 00:10:00    5
2014-10-07 00:12:00    6
2014-10-07 00:14:00    7
2014-10-07 00:16:00    8
2014-10-07 00:18:00    9
Freq: 2T, dtype: int64
```

Create a value counts column and reassign back to the DataFrame

```
In [141]: df = pd.DataFrame({'Color': 'Red Red Red Blue'.split(),
   .....:                     'Value': [100, 150, 50, 50]})
   .....:

In [142]: df
Out[142]:
  Color  Value
0   Red    100
1   Red    150
2   Red     50
3  Blue     50

In [143]: df['Counts'] = df.groupby(['Color']).transform(len)

In [144]: df
Out[144]:
  Color  Value  Counts
0   Red    100       3
1   Red    150       3
2   Red     50       3
3  Blue     50       1
```

Shift groups of the values in a column based on the index

```
In [145]: df = pd.DataFrame({'line_race': [10, 10, 8, 10, 10, 8],
   .....:                    'beyer': [99, 102, 103, 103, 88, 100]},
   .....:                   index=['Last Gunfighter', 'Last Gunfighter',
   .....:                          'Last Gunfighter', 'Paynter', 'Paynter',
   .....:                          'Paynter'])
   .....:

In [146]: df
Out[146]:
                 line_race  beyer
Last Gunfighter         10     99
Last Gunfighter         10    102
Last Gunfighter          8    103
Paynter                 10    103
Paynter                 10     88
```

(continues on next page)

(continued from previous page)

```
Paynter                   8    100

In [147]: df['beyer_shifted'] = df.groupby(level=0)['beyer'].shift(1)

In [148]: df
Out[148]:
                 line_race  beyer  beyer_shifted
Last Gunfighter         10     99            NaN
Last Gunfighter         10    102           99.0
Last Gunfighter          8    103          102.0
Paynter                 10    103            NaN
Paynter                 10     88          103.0
Paynter                  8    100           88.0
```

Select row with maximum value from each group

```
In [149]: df = pd.DataFrame({'host': ['other', 'other', 'that', 'this', 'this'],
   .....:                    'service': ['mail', 'web', 'mail', 'mail', 'web'],
   .....:                    'no': [1, 2, 1, 2, 1]}).set_index(['host', 'service'])
   .....:

In [150]: mask = df.groupby(level=0).agg('idxmax')

In [151]: df_count = df.loc[mask['no']].reset_index()

In [152]: df_count
Out[152]:
    host service  no
0  other     web   2
1   that    mail   1
2   this    mail   2
```

Grouping like Python's itertools.groupby

```
In [153]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=['A'])

In [154]: df.A.groupby((df.A != df.A.shift()).cumsum()).groups
Out[154]:
{1: Int64Index([0], dtype='int64'),
 2: Int64Index([1], dtype='int64'),
 3: Int64Index([2], dtype='int64'),
 4: Int64Index([3, 4, 5], dtype='int64'),
 5: Int64Index([6], dtype='int64'),
 6: Int64Index([7, 8], dtype='int64')}

In [155]: df.A.groupby((df.A != df.A.shift()).cumsum()).cumsum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    0
1    1
2    0
3    1
4    2
5    3
6    0
7    1
8    2
```

(continues on next page)

(continued from previous page)

```
Name: A, dtype: int64
```

Expanding Data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

Splitting

Splitting a frame

Create a list of dataframes, split using a delineation based on logic included in rows.

```
In [156]: df = pd.DataFrame(data={'Case': ['A', 'A', 'A', 'B', 'A', 'A', 'B', 'A',
   .....:                                  'A'],
   .....:                         'Data': np.random.randn(9)})
   .....:

In [157]: dfs = list(zip(*df.groupby((1 * (df['Case'] == 'B')).cumsum()
   .....:                .rolling(window=3, min_periods=1).median())))[-1]
   .....:

In [158]: dfs[0]
Out[158]:
  Case      Data
0    A  0.174068
1    A -0.439461
2    A -0.741343
3    B -0.079673

In [159]: dfs[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪
  Case      Data
4    A -0.922875
5    A  0.303638
6    B -0.917368

In [160]: dfs[2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
  Case      Data
7    A -1.624062
8    A -0.758514
```

Pivot

The *Pivot* docs.

Partial sums and subtotals

```
In [161]: df = pd.DataFrame(data={'Province': ['ON', 'QC', 'BC', 'AL', 'AL', 'MN', 'ON
↪'],
   .....:                         'City': ['Toronto', 'Montreal', 'Vancouver',
   .....:                                  'Calgary', 'Edmonton', 'Winnipeg',
   .....:                                  'Windsor'],
   .....:                         'Sales': [13, 6, 16, 8, 4, 3, 1]})
   .....:

In [162]: table = pd.pivot_table(df, values=['Sales'], index=['Province'],
   .....:                        columns=['City'], aggfunc=np.sum, margins=True)
   .....:

In [163]: table.stack('City')
Out[163]:
                    Sales
Province City
AL       All         12.0
         Calgary      8.0
         Edmonton     4.0
BC       All         16.0
         Vancouver   16.0
MN       All          3.0
         Winnipeg     3.0
...                   ...
All      Calgary      8.0
         Edmonton     4.0
         Montreal     6.0
         Toronto     13.0
         Vancouver   16.0
         Windsor      1.0
         Winnipeg     3.0

[20 rows x 1 columns]
```

Frequency table like plyr in R

```
In [164]: grades = [48, 99, 75, 80, 42, 80, 72, 68, 36, 78]

In [165]: df = pd.DataFrame({'ID': ["x%d" % r for r in range(10)],
   .....:                    'Gender': ['F', 'M', 'F', 'M', 'F',
   .....:                               'M', 'F', 'M', 'M', 'M'],
   .....:                    'ExamYear': ['2007', '2007', '2007', '2008', '2008',
   .....:                                 '2008', '2008', '2009', '2009', '2009'],
   .....:                    'Class': ['algebra', 'stats', 'bio', 'algebra',
   .....:                              'algebra', 'stats', 'stats', 'algebra',
   .....:                              'bio', 'bio'],
   .....:                    'Participated': ['yes', 'yes', 'yes', 'yes', 'no',
   .....:                                     'yes', 'yes', 'yes', 'yes', 'yes'],
   .....:                    'Passed': ['yes' if x > 50 else 'no' for x in grades],
   .....:                    'Employed': [True, True, True, False,
   .....:                                 False, False, False, True, True, False],
   .....:                    'Grade': grades})
   .....:

In [166]: df.groupby('ExamYear').agg({'Participated': lambda x: x.value_counts()['yes
↪'],
   .....:                     'Passed': lambda x: sum(x == 'yes'),
```

(continues on next page)

(continued from previous page)

```
   .....:                              'Employed': lambda x: sum(x),
   .....:                              'Grade': lambda x: sum(x) / len(x)})
   .....:
Out[166]:
          Participated  Passed  Employed      Grade
ExamYear
2007                 3       2         3  74.000000
2008                 3       3         0  68.500000
2009                 3       2         2  60.666667
```

Plot pandas DataFrame with year over year data

To create year and month cross tabulation:

```
In [167]: df = pd.DataFrame({'value': np.random.randn(36)},
   .....:                   index=pd.date_range('2011-01-01', freq='M', periods=36))
   .....:

In [168]: pd.pivot_table(df, index=df.index.month, columns=df.index.year,
   .....:                values='value', aggfunc='sum')
   .....:
Out[168]:
        2011      2012      2013
1  -0.560859  0.120930  0.516870
2  -0.589005 -0.210518  0.343125
3  -1.070678 -0.931184  2.137827
4  -1.681101  0.240647  0.452429
5   0.403776 -0.027462  0.483103
6   0.609862  0.033113  0.061495
7   0.387936 -0.658418  0.240767
8   1.815066  0.324102  0.782413
9   0.705200 -1.403048  0.628462
10 -0.668049 -0.581967 -0.880627
11  0.242501 -1.233862  0.777575
12  0.313421 -3.520876 -0.779367
```

Apply

Rolling Apply to Organize - Turning embedded lists into a MultiIndex frame

```
In [169]: df = pd.DataFrame(data={'A': [[2, 4, 8, 16], [100, 200], [10, 20, 30]],
   .....:                         'B': [['a', 'b', 'c'], ['jj', 'kk'], ['ccc']]},
   .....:                   index=['I', 'II', 'III'])
   .....:

In [170]: def SeriesFromSubList(aList):
   .....:     return pd.Series(aList)
   .....:

In [171]: df_orgz = pd.concat({ind: row.apply(SeriesFromSubList)
   .....:                      for ind, row in df.iterrows()})
   .....:

In [172]: df_orgz
Out[172]:
```

(continues on next page)

(continued from previous page)

```
          0    1    2     3
I   A     2    4    8  16.0
    B     a    b    c   NaN
II  A   100  200  NaN   NaN
    B    jj   kk  NaN   NaN
III A    10   20   30   NaN
    B   ccc  NaN  NaN   NaN
```

Rolling Apply with a DataFrame returning a Series

Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

```
In [173]: df = pd.DataFrame(data=np.random.randn(2000, 2) / 10000,
   .....:                   index=pd.date_range('2001-01-01', periods=2000),
   .....:                   columns=['A', 'B'])
   .....:

In [174]: df
Out[174]:
                   A         B
2001-01-01  0.000032 -0.000004
2001-01-02 -0.000001  0.000207
2001-01-03  0.000120 -0.000220
2001-01-04 -0.000083 -0.000165
2001-01-05 -0.000047  0.000156
2001-01-06  0.000027  0.000104
2001-01-07  0.000041 -0.000101
...              ...       ...
2006-06-17 -0.000034  0.000034
2006-06-18  0.000002  0.000166
2006-06-19  0.000023 -0.000081
2006-06-20 -0.000061  0.000012
2006-06-21 -0.000111  0.000027
2006-06-22 -0.000061 -0.000009
2006-06-23  0.000074 -0.000138

[2000 rows x 2 columns]

In [175]: def gm(df, const):
   .....:     v = ((((df.A + df.B) + 1).cumprod()) - 1) * const
   .....:     return v.iloc[-1]
   .....:

In [176]: s = pd.Series({df.index[i]: gm(df.iloc[i:min(i + 51, len(df) - 1)], 5)
   .....:                for i in range(len(df) - 50)})
   .....:

In [177]: s
Out[177]:
2001-01-01   -0.001373
2001-01-02   -0.001705
2001-01-03   -0.002885
2001-01-04   -0.002987
2001-01-05   -0.002384
2001-01-06   -0.004700
2001-01-07   -0.005500
                ...
```

(continues on next page)

(continued from previous page)

```
2006-04-28   -0.002682
2006-04-29   -0.002436
2006-04-30   -0.002602
2006-05-01   -0.001785
2006-05-02   -0.001799
2006-05-03   -0.000605
2006-05-04   -0.000541
Length: 1950, dtype: float64
```

Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)

```
In [178]: rng = pd.date_range(start='2014-01-01', periods=100)

In [179]: df = pd.DataFrame({'Open': np.random.randn(len(rng)),
   .....:                    'Close': np.random.randn(len(rng)),
   .....:                    'Volume': np.random.randint(100, 2000, len(rng))},
   .....:                   index=rng)
   .....:

In [180]: df
Out[180]:
                Open     Close  Volume
2014-01-01  0.011174 -0.653039    1581
2014-01-02  0.214258  1.314205    1707
2014-01-03 -1.046922 -0.341915    1768
2014-01-04 -0.752902 -1.303586     836
2014-01-05 -0.410793  0.396288     694
2014-01-06  0.648401 -0.548006     796
2014-01-07  0.737320  0.481380     265
...              ...       ...     ...
2014-04-04  0.120378 -2.548128     564
2014-04-05  0.231661  0.223346    1908
2014-04-06  0.952664  1.228841    1090
2014-04-07 -0.176090  0.552784    1813
2014-04-08  1.781318 -0.795389    1103
2014-04-09 -0.753493 -0.018815    1456
2014-04-10 -1.047997  1.138197    1193

[100 rows x 3 columns]

In [181]: def vwap(bars):
   .....:     return ((bars.Close * bars.Volume).sum() / bars.Volume.sum())
   .....:

In [182]: window = 5

In [183]: s = pd.concat([(pd.Series(vwap(df.iloc[i:i + window]),
   .....:                 index=[df.index[i + window]]))
   .....:                for i in range(len(df) - window)])
   .....:

In [184]: s.round(2)
Out[184]:
2014-01-06   -0.03
2014-01-07    0.07
```

(continues on next page)

(continued from previous page)

```
2014-01-08   -0.40
2014-01-09   -0.81
2014-01-10   -0.63
2014-01-11   -0.86
2014-01-12   -0.36
              ...
2014-04-04   -1.27
2014-04-05   -1.36
2014-04-06   -0.73
2014-04-07    0.04
2014-04-08    0.21
2014-04-09    0.07
2014-04-10    0.25
Length: 95, dtype: float64
```

4.20.6 Timeseries

Between times

Using indexer between time

Constructing a datetime range that excludes weekends and includes only certain times

Vectorized Lookup

Aggregation and plotting time series

Turn a matrix with hours in columns and days in rows into a continuous row sequence in the form of a time series. How to rearrange a Python pandas DataFrame?

Dealing with duplicates when reindexing a timeseries to a specified frequency

Calculate the first day of the month for each entry in a DatetimeIndex

```
In [185]: dates = pd.date_range('2000-01-01', periods=5)

In [186]: dates.to_period(freq='M').to_timestamp()
Out[186]:
DatetimeIndex(['2000-01-01', '2000-01-01', '2000-01-01', '2000-01-01',
               '2000-01-01'],
              dtype='datetime64[ns]', freq=None)
```

Resampling

The *Resample* docs.

Using Grouper instead of TimeGrouper for time grouping of values

Time grouping with some missing values

Valid frequency arguments to Grouper

Grouping using a MultiIndex

Using TimeGrouper and another grouping to create subgroups, then apply a custom function

Resampling with custom periods

Resample intraday frame without adding new days

Resample minute data

Resample with groupby

4.20.7 Merge

The *Concat* docs. The *Join* docs.

Append two dataframes with overlapping index (emulate R rbind)

```
In [187]: rng = pd.date_range('2000-01-01', periods=6)

In [188]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=['A', 'B', 'C
↪'])

In [189]: df2 = df1.copy()
```

Depending on df construction, `ignore_index` may be needed

```
In [190]: df = df1.append(df2, ignore_index=True)

In [191]: df
Out[191]: 
           A         B         C
0  -0.480676 -1.305282 -0.212846
1   1.979901  0.363112 -0.275732
2  -1.433852  0.580237 -0.013672
3   1.776623 -0.803467  0.521517
4  -0.302508 -0.442948 -0.395768
5  -0.249024 -0.031510  2.413751
6  -0.480676 -1.305282 -0.212846
7   1.979901  0.363112 -0.275732
8  -1.433852  0.580237 -0.013672
9   1.776623 -0.803467  0.521517
10 -0.302508 -0.442948 -0.395768
11 -0.249024 -0.031510  2.413751
```

Self Join of a DataFrame

```
In [192]: df = pd.DataFrame(data={'Area': ['A'] * 5 + ['C'] * 2,
   .....:                        'Bins': [110] * 2 + [160] * 3 + [40] * 2,
   .....:                        'Test_0': [0, 1, 0, 1, 2, 0, 1],
   .....:                        'Data': np.random.randn(7)})
   .....: 

In [193]: df
Out[193]: 
  Area  Bins  Test_0      Data
0    A   110       0 -0.378914
1    A   110       1 -1.032527
2    A   160       0 -1.402816
3    A   160       1  0.715333
4    A   160       2 -0.091438
5    C    40       0  1.608418
6    C    40       1  0.753207

In [194]: df['Test_1'] = df['Test_0'] - 1
```

(continues on next page)

(continued from previous page)

```
In [195]: pd.merge(df, df, left_on=['Bins', 'Area', 'Test_0'],
   .....:          right_on=['Bins', 'Area', 'Test_1'],
   .....:          suffixes=('_L', '_R'))
   .....:
Out[195]:
  Area  Bins  Test_0_L    Data_L  Test_1_L  Test_0_R    Data_R  Test_1_R
0    A   110         0 -0.378914        -1         1 -1.032527         0
1    A   160         0 -1.402816        -1         1  0.715333         0
2    A   160         1  0.715333         0         2 -0.091438         1
3    C    40         0  1.608418        -1         1  0.753207         0
```

How to set the index and join

KDB like asof join

Join with a criteria based on the values

Using searchsorted to merge based on values inside a range

4.20.8 Plotting

The *Plotting* docs.

Make Matplotlib look like R

Setting x-axis major and minor labels

Plotting multiple charts in an ipython notebook

Creating a multi-line plot

Plotting a heatmap

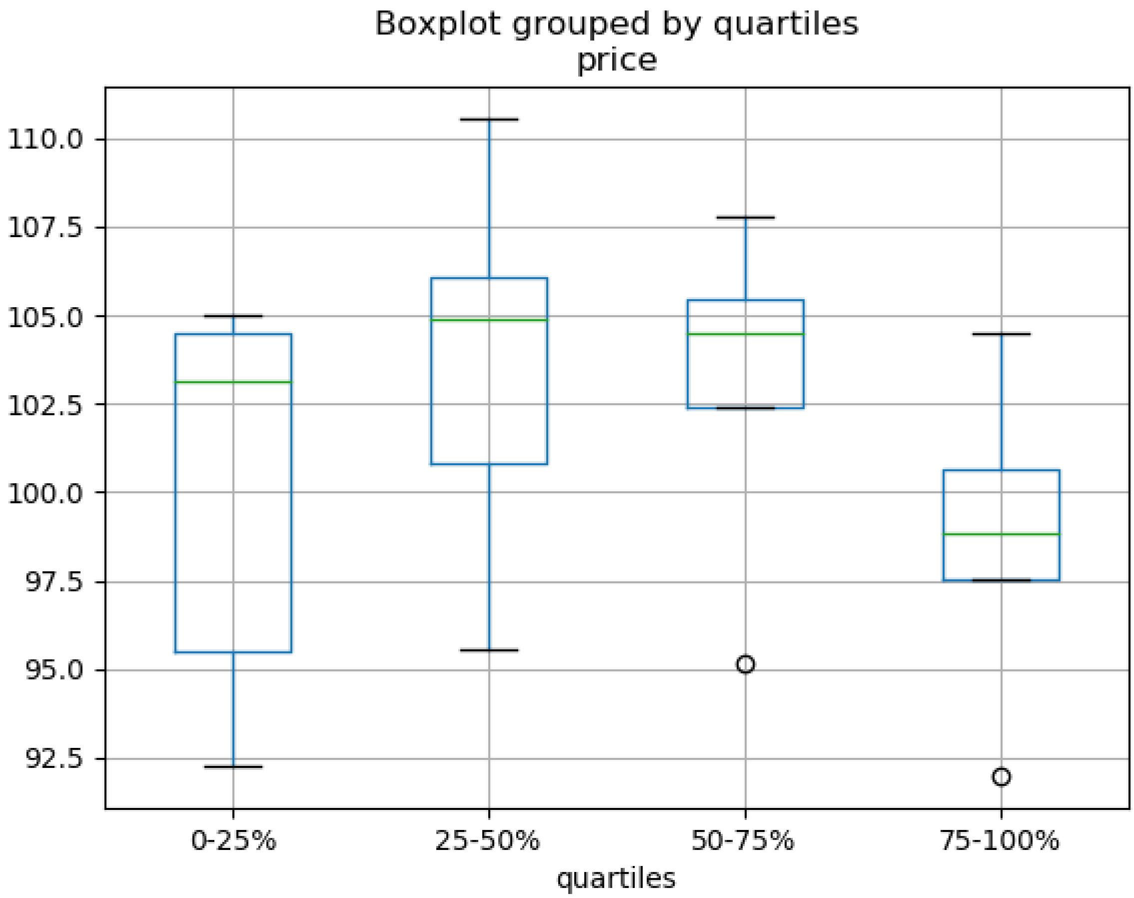
Annotate a time-series plot

Annotate a time-series plot #2

Generate Embedded plots in excel files using Pandas, Vincent and xlsxwriter

Boxplot for each quartile of a stratifying variable

```
In [196]: df = pd.DataFrame(
   .....:     {'stratifying_var': np.random.uniform(0, 100, 20),
   .....:      'price': np.random.normal(100, 5, 20)})
   .....:

In [197]: df['quartiles'] = pd.qcut(
   .....:     df['stratifying_var'],
   .....:     4,
   .....:     labels=['0-25%', '25-50%', '50-75%', '75-100%'])
   .....:

In [198]: df.boxplot(column='price', by='quartiles')
Out[198]: <matplotlib.axes._subplots.AxesSubplot at 0x7f380fe70e48>
```



4.20.9 Data In/Out

Performance comparison of SQL vs HDF5

CSV

The *CSV* docs

read_csv in action

appending to a csv

Reading a csv chunk-by-chunk

Reading only certain rows of a csv chunk-by-chunk

Reading the first few lines of a frame

Reading a file that is compressed but not by `gzip/bz2` (the native compressed formats which `read_csv` understands). This example shows a `WinZipped` file, but is a general application of opening the file within a context manager and using that handle to read. See here

Inferring dtypes from a file

Dealing with bad lines

Dealing with bad lines II

Reading CSV with Unix timestamps and converting to local timezone

Write a multi-row index CSV without writing duplicates

Reading multiple files to create a single DataFrame

The best way to combine multiple files into a single DataFrame is to read the individual frames one by one, put all of the individual frames into a list, and then combine the frames in the list using `pd.concat()`:

```
In [199]: for i in range(3):
   .....:     data = pd.DataFrame(np.random.randn(10, 4))
   .....:     data.to_csv('file_{}.csv'.format(i))
   .....:

In [200]: files = ['file_0.csv', 'file_1.csv', 'file_2.csv']

In [201]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

You can use the same approach to read all files matching a pattern. Here is an example using `glob`:

```
In [202]: import glob

In [203]: import os

In [204]: files = glob.glob('file_*.csv')

In [205]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

Finally, this strategy will work with the other `pd.read_*(...)` functions described in the *io docs*.

Parsing date components in multi-columns

Parsing date components in multi-columns is faster with a format

```
In [206]: i = pd.date_range('20000101', periods=10000)

In [207]: df = pd.DataFrame({'year': i.year, 'month': i.month, 'day': i.day})

In [208]: df.head()
Out[208]:
   year  month  day
0  2000      1    1
1  2000      1    2
2  2000      1    3
3  2000      1    4
4  2000      1    5

In [209]: %timeit pd.to_datetime(df.year * 10000 + df.month * 100 + df.day, format='%Y
↪%m%d')
   .....: ds = df.apply(lambda x: "%04d%02d%02d" % (x['year'],
   .....:                                           x['month'], x['day']), axis=1)
   .....: ds.head()
   .....: %timeit pd.to_datetime(ds)
   .....:
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪63 ms +- 140 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
1.76 ms +- 119 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

Skip row between header and data

```
In [210]: data = """;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....: ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....: ;;;;
   .....: date;Param1;Param2;Param4;Param5
   .....:     ;m²;°C;m²;m
   .....: ;;;;
   .....: 01.01.1990 00:00;1;1;2;3
   .....: 01.01.1990 01:00;5;3;4;5
   .....: 01.01.1990 02:00;9;5;6;7
   .....: 01.01.1990 03:00;13;7;8;9
   .....: 01.01.1990 04:00;17;9;10;11
   .....: 01.01.1990 05:00;21;11;12;13
   .....: """
   .....:
```

Option 1: pass rows explicitly to skip rows

```
In [211]: from pandas.compat import StringIO

In [212]: pd.read_csv(StringIO(data), sep=';', skiprows=[11, 12],
   .....:             index_col=0, parse_dates=True, header=10)
   .....:
Out[212]:
                     Param1  Param2  Param4  Param5
date
1990-01-01 00:00:00       1       1       2       3
1990-01-01 01:00:00       5       3       4       5
1990-01-01 02:00:00       9       5       6       7
1990-01-01 03:00:00      13       7       8       9
1990-01-01 04:00:00      17       9      10      11
1990-01-01 05:00:00      21      11      12      13
```

Option 2: read column names and then data

```
In [213]: pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
Out[213]: Index(['date', 'Param1', 'Param2', 'Param4', 'Param5'], dtype='object')
```

(continues on next page)

(continued from previous page)

```
In [214]: columns = pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns

In [215]: pd.read_csv(StringIO(data), sep=';', index_col=0,
   .....:             header=12, parse_dates=True, names=columns)
   .....:
Out[215]:
                     Param1  Param2  Param4  Param5
date
1990-01-01 00:00:00       1       1       2       3
1990-01-01 01:00:00       5       3       4       5
1990-01-01 02:00:00       9       5       6       7
1990-01-01 03:00:00      13       7       8       9
1990-01-01 04:00:00      17       9      10      11
1990-01-01 05:00:00      21      11      12      13
```

SQL

The *SQL* docs

Reading from databases with SQL

Excel

The *Excel* docs

Reading from a filelike handle

Modifying formatting in XlsxWriter output

HTML

Reading HTML tables from a server that cannot handle the default request header

HDFStore

The *HDFStores* docs

Simple Queries with a Timestamp Index

Managing heterogeneous data using a linked multiple table hierarchy

Merging on-disk tables with millions of rows

Avoiding inconsistencies when writing to a store from multiple processes/threads

De-duplicating a large store by chunks, essentially a recursive reduction operation. Shows a function for taking in data from csv file and creating a store by chunks, with date parsing as well. See here

Creating a store chunk-by-chunk from a csv file

Appending to a store, while creating a unique index

Large Data work flows

Reading in a sequence of files, then providing a global unique index to a store while appending

Groupby on a HDFStore with low group density

Groupby on a HDFStore with high group density

Hierarchical queries on a HDFStore

Counting with a HDFStore

Troubleshoot HDFStore exceptions

Setting min_itemsize with strings

Using ptrepack to create a completely-sorted-index on a store

Storing Attributes to a group node

```
In [216]: df = pd.DataFrame(np.random.randn(8, 3))

In [217]: store = pd.HDFStore('test.h5')

In [218]: store.put('df', df)

# you can store an arbitrary Python object via pickle
In [219]: store.get_storer('df').attrs.my_attribute = {'A': 10}

In [220]: store.get_storer('df').attrs.my_attribute
Out[220]: {'A': 10}
```

Binary Files

pandas readily accepts NumPy record arrays, if you need to read in a binary file consisting of an array of C structs. For example, given this C program in a file called `main.c` compiled with `gcc main.c -std=gnu99` on a 64-bit machine,

```
#include <stdio.h>
#include <stdint.h>

typedef struct _Data
{
    int32_t count;
    double avg;
    float scale;
} Data;

int main(int argc, const char *argv[])
{
    size_t n = 10;
    Data d[n];

    for (int i = 0; i < n; ++i)
    {
        d[i].count = i;
        d[i].avg = i + 1.0;
        d[i].scale = (float) i + 2.0f;
    }

    FILE *file = fopen("binary.dat", "wb");
    fwrite(&d, sizeof(Data), n, file);
    fclose(file);

```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

the following Python code will read the binary file `'binary.dat'` into a pandas `DataFrame`, where each element of the struct corresponds to a column in the frame:

```
names = 'count', 'avg', 'scale'

# note that the offsets are larger than the size of the type because of
# struct padding
offsets = 0, 8, 16
formats = 'i4', 'f8', 'f4'
dt = np.dtype({'names': names, 'offsets': offsets, 'formats': formats},
              align=True)
df = pd.DataFrame(np.fromfile('binary.dat', dt))
```

Note: The offsets of the structure elements may be different depending on the architecture of the machine on which the file was created. Using a raw binary file format like this for general data storage is not recommended, as it is not cross platform. We recommended either HDF5 or msgpack, both of which are supported by pandas' IO facilities.

4.20.10 Computation

Numerical integration (sample-based) of a time series

Correlation

Often it's useful to obtain the lower (or upper) triangular form of a correlation matrix calculated from *DataFrame.corr()*. This can be achieved by passing a boolean mask to `where` as follows:

```
In [221]: df = pd.DataFrame(np.random.random(size=(100, 5)))

In [222]: corr_mat = df.corr()

In [223]: mask = np.tril(np.ones_like(corr_mat, dtype=np.bool), k=-1)

In [224]: corr_mat.where(mask)
Out[224]:
          0         1         2         3   4
0       NaN       NaN       NaN       NaN NaN
1  0.100443       NaN       NaN       NaN NaN
2  0.012441 -0.068965       NaN       NaN NaN
3  0.009641  0.078722 -0.067531       NaN NaN
4 -0.065089 -0.156980 -0.004463  0.075126 NaN
```

The *method* argument within *DataFrame.corr* can accept a callable in addition to the named correlation types. Here we compute the distance correlation matrix for a *DataFrame* object.

```
In [225]: def distcorr(x, y):
   .....:     n = len(x)
   .....:     a = np.zeros(shape=(n, n))
   .....:     b = np.zeros(shape=(n, n))
```

(continues on next page)

(continued from previous page)

```
   .....:

In [226]: df = pd.DataFrame(np.random.normal(size=(100, 3)))

In [227]: df.corr(method=distcorr)
Out[227]:
     0    1    2
0  1.0  NaN  NaN
1  NaN  1.0  NaN
2  NaN  NaN  1.0
```

4.20.11 Timedeltas

The *Timedeltas* docs.

Using timedeltas

```
In [228]: import datetime

In [229]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [230]: s - s.max()
Out[230]:
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [231]: s.max() - s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[231]:
0   2 days
1   1 days
2   0 days
dtype: timedelta64[ns]

In [232]: s - datetime.datetime(2011, 1, 1, 3, 5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   364 days 20:55:00
1   365 days 20:55:00
2   366 days 20:55:00
dtype: timedelta64[ns]

In [233]: s + datetime.timedelta(minutes=5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]

In [234]: datetime.datetime(2011, 1, 1, 3, 5) - s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   -365 days +03:05:00
```

(continues on next page)

(continued from previous page)

```
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]

In [235]: datetime.timedelta(minutes=5) + s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

Adding and subtracting deltas and dates

```
In [236]: deltas = pd.Series([datetime.timedelta(days=i) for i in range(3)])

In [237]: df = pd.DataFrame({'A': s, 'B': deltas})

In [238]: df
Out[238]:
           A      B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [239]: df['New Dates'] = df['A'] + df['B']

In [240]: df['Delta'] = df['A'] - df['New Dates']

In [241]: df
Out[241]:
           A      B  New Dates   Delta
0 2012-01-01 0 days 2012-01-01  0 days
1 2012-01-02 1 days 2012-01-03 -1 days
2 2012-01-03 2 days 2012-01-05 -2 days

In [242]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A             datetime64[ns]
B            timedelta64[ns]
New Dates     datetime64[ns]
Delta        timedelta64[ns]
dtype: object
```

Another example

Values can be set to NaT using np.nan, similar to datetime

```
In [243]: y = s - s.shift()

In [244]: y
Out[244]:
0      NaT
1   1 days
2   1 days
dtype: timedelta64[ns]
```

(continues on next page)

(continued from previous page)

```
In [245]: y[1] = np.nan

In [246]: y
Out[246]:
0       NaT
1       NaT
2    1 days
dtype: timedelta64[ns]
```

4.20.12 Aliasing Axis Names

To globally provide aliases for axis names, one can define these 2 functions:

```
In [247]: def set_axis_alias(cls, axis, alias):
   .....:     if axis not in cls._AXIS_NUMBERS:
   .....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   .....:     cls._AXIS_ALIASES[alias] = axis
   .....:
```

```
In [248]: def clear_axis_alias(cls, axis, alias):
   .....:     if axis not in cls._AXIS_NUMBERS:
   .....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   .....:     cls._AXIS_ALIASES.pop(alias, None)
   .....:
```

```
In [249]: set_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

In [250]: df2 = pd.DataFrame(np.random.randn(3, 2), columns=['c1', 'c2'],
   .....:                    index=['i1', 'i2', 'i3'])
   .....:

In [251]: df2.sum(axis='myaxis2')
Out[251]:
i1   -0.842809
i2   -2.136732
i3   -0.596719
dtype: float64

In [252]: clear_axis_alias(pd.DataFrame, 'columns', 'myaxis2')
```

4.20.13 Creating Example Data

To create a dataframe from every combination of some given values, like R's `expand.grid()` function, we can create a dict where the keys are column names and the values are lists of the data values:

```
In [253]: def expand_grid(data_dict):
   .....:     rows = itertools.product(*data_dict.values())
   .....:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
   .....:

In [254]: df = expand_grid({'height': [60, 70],
```

(continues on next page)

(continued from previous page)

```
   .....:                      'weight': [100, 140, 180],
   .....:                      'sex': ['Male', 'Female']})
   .....:

In [255]: df
Out[255]: 
    height  weight     sex
0       60     100    Male
1       60     100  Female
2       60     140    Male
3       60     140  Female
4       60     180    Male
5       60     180  Female
6       70     100    Male
7       70     100  Female
8       70     140    Male
9       70     140  Female
10      70     180    Male
11      70     180  Female
```

CHAPTER

FIVE

PANDAS ECOSYSTEM

Increasingly, packages are being built on top of pandas to address specific needs in data preparation, analysis and visualization. This is encouraging because it means pandas is not only helping users to handle their data tasks but also that it provides a better starting point for developers to build powerful and more focused data tools. The creation of libraries that complement pandas' functionality also allows pandas development to remain focused around it's original requirements.

This is an inexhaustive list of projects that build on pandas in order to provide tools in the PyData space. For a list of projects that depend on pandas, see the libraries.io usage page for pandas or search pypi for pandas.

We'd like to make it easier for users to find these projects, if you know of other substantial projects that you feel should be on this list, please let us know.

5.1 Statistics and Machine Learning

5.1.1 Statsmodels

Statsmodels is the prominent Python "statistics and econometrics library" and it has a long-standing special relationship with pandas. Statsmodels provides powerful statistics, econometrics, analysis and modeling functionality that is out of pandas' scope. Statsmodels leverages pandas objects as the underlying data container for computation.

5.1.2 sklearn-pandas

Use pandas DataFrames in your scikit-learn ML pipeline.

5.1.3 Featuretools

Featuretools is a Python library for automated feature engineering built on top of pandas. It excels at transforming temporal and relational datasets into feature matrices for machine learning using reusable feature engineering "primitives". Users can contribute their own primitives in Python and share them with the rest of the community.

5.2 Visualization

5.2.1 Altair

Altair is a declarative statistical visualization library for Python. With Altair, you can spend more time understanding your data and its meaning. Altair's API is simple, friendly and consistent and built on top of the powerful Vega-Lite

JSON specification. This elegant simplicity produces beautiful and effective visualizations with a minimal amount of code. Altair works with Pandas DataFrames.

5.2.2 Bokeh

Bokeh is a Python interactive visualization library for large datasets that natively uses the latest web technologies. Its goal is to provide elegant, concise construction of novel graphics in the style of Protovis/D3, while delivering high-performance interactivity over large data to thin clients.

5.2.3 seaborn

Seaborn is a Python visualization library based on matplotlib. It provides a high-level, dataset-oriented interface for creating attractive statistical graphics. The plotting functions in seaborn understand pandas objects and leverage pandas grouping operations internally to support concise specification of complex visualizations. Seaborn also goes beyond matplotlib and pandas with the option to perform statistical estimation while plotting, aggregating across observations and visualizing the fit of statistical models to emphasize patterns in a dataset.

5.2.4 yhat/ggpy

Hadley Wickham's ggplot2 is a foundational exploratory visualization package for the R language. Based on "The Grammar of Graphics" it provides a powerful, declarative and extremely general way to generate bespoke plots of any kind of data. It's really quite incredible. Various implementations to other languages are available, but a faithful implementation for Python users has long been missing. Although still young (as of Jan-2014), the yhat/ggpy project has been progressing quickly in that direction.

5.2.5 IPython Vega

IPython Vega leverages Vega to create plots within Jupyter Notebook.

5.2.6 Plotly

Plotly's Python API enables interactive figures and web shareability. Maps, 2D, 3D, and live-streaming graphs are rendered with WebGL and D3.js. The library supports plotting directly from a pandas DataFrame and cloud-based collaboration. Users of matplotlib, ggplot for Python, and Seaborn can convert figures into interactive web-based plots. Plots can be drawn in IPython Notebooks , edited with R or MATLAB, modified in a GUI, or embedded in apps and dashboards. Plotly is free for unlimited sharing, and has cloud, offline, or on-premise accounts for private use.

5.2.7 QtPandas

Spun off from the main pandas library, the qtpandas library enables DataFrame visualization and manipulation in PyQt4 and PySide applications.

5.3 IDE

5.3.1 IPython

IPython is an interactive command shell and distributed computing environment. IPython tab completion works with Pandas methods and also attributes like DataFrame columns.

5.3.2 Jupyter Notebook / Jupyter Lab

Jupyter Notebook is a web application for creating Jupyter notebooks. A Jupyter notebook is a JSON document containing an ordered list of input/output cells which can contain code, text, mathematics, plots and rich media. Jupyter notebooks can be converted to a number of open standard output formats (HTML, HTML presentation slides, LaTeX, PDF, ReStructuredText, Markdown, Python) through 'Download As' in the web interface and `jupyter convert` in a shell.

Pandas DataFrames implement `_repr_html_``and ``_repr_latex` methods which are utilized by Jupyter Notebook for displaying (abbreviated) HTML or LaTeX tables. LaTeX output is properly escaped. (Note: HTML tables may or may not be compatible with non-HTML Jupyter output formats.)

See *Options and Settings* and *Available Options* for pandas `display.` settings.

5.3.3 quantopian/qgrid

qgrid is "an interactive grid for sorting and filtering DataFrames in IPython Notebook" built with SlickGrid.

5.3.4 Spyder

Spyder is a cross-platform PyQt-based IDE combining the editing, analysis, debugging and profiling functionality of a software development tool with the data exploration, interactive execution, deep inspection and rich visualization capabilities of a scientific environment like MATLAB or Rstudio.

Its Variable Explorer allows users to view, manipulate and edit pandas `Index`, `Series`, and `DataFrame` objects like a "spreadsheet", including copying and modifying values, sorting, displaying a "heatmap", converting data types and more. Pandas objects can also be renamed, duplicated, new columns added, copyed/pasted to/from the clipboard (as TSV), and saved/loaded to/from a file. Spyder can also import data from a variety of plain text and binary files or the clipboard into a new pandas DataFrame via a sophisticated import wizard.

Most pandas classes, methods and data attributes can be autocompleted in Spyder's Editor and IPython Console, and Spyder's Help pane can retrieve and render Numpydoc documentation on pandas objects in rich text with Sphinx both automatically and on-demand.

5.4 API

5.4.1 pandas-datareader

`pandas-datareader` is a remote data access library for pandas (PyPI:`pandas-datareader`). It is based on functionality that was located in `pandas.io.data` and `pandas.io.wb` but was split off in v0.19. See more in the pandas-datareader docs:

The following data feeds are available:

- Google Finance
- Tiingo
- Morningstar
- IEX
- Robinhood
- Enigma
- Quandl
- FRED
- Fama/French
- World Bank
- OECD
- Eurostat
- TSP Fund Data
- Nasdaq Trader Symbol Definitions
- Stooq Index Data
- MOEX Data

5.4.2 quandl/Python

Quandl API for Python wraps the Quandl REST API to return Pandas DataFrames with timeseries indexes.

5.4.3 pydatastream

PyDatastream is a Python interface to the Thomson Dataworks Enterprise (DWE/Datastream) SOAP API to return indexed Pandas DataFrames or Panels with financial data. This package requires valid credentials for this API (non free).

5.4.4 pandaSDMX

pandaSDMX is a library to retrieve and acquire statistical data and metadata disseminated in SDMX 2.1, an ISO-standard widely used by institutions such as statistics offices, central banks, and international organisations. pandaSDMX can expose datasets and related structural metadata including data flows, code-lists, and data structure definitions as pandas Series or MultiIndexed DataFrames.

5.4.5 fredapi

fredapi is a Python interface to the Federal Reserve Economic Data (FRED) provided by the Federal Reserve Bank of St. Louis. It works with both the FRED database and ALFRED database that contains point-in-time data (i.e. historic data revisions). fredapi provides a wrapper in Python to the FRED HTTP API, and also provides several convenient methods for parsing and analyzing point-in-time data from ALFRED. fredapi makes use of pandas and returns data in a Series or DataFrame. This module requires a FRED API key that you can obtain for free on the FRED website.

5.5 Domain Specific

5.5.1 Geopandas

Geopandas extends pandas data objects to include geographic information which support geometric operations. If your work entails maps and geographical coordinates, and you love pandas, you should take a close look at Geopandas.

5.5.2 xarray

xarray brings the labeled data power of pandas to the physical sciences by providing N-dimensional variants of the core pandas data structures. It aims to provide a pandas-like and pandas-compatible toolkit for analytics on multi-dimensional arrays, rather than the tabular data for which pandas excels.

5.6 Out-of-core

5.6.1 Blaze

Blaze provides a standard API for doing computations with various in-memory and on-disk backends: NumPy, Pandas, SQLAlchemy, MongoDB, PyTables, PySpark.

5.6.2 Dask

Dask is a flexible parallel computing library for analytics. Dask provides a familiar `DataFrame` interface for out-of-core, parallel and distributed computing.

5.6.3 Dask-ML

Dask-ML enables parallel and distributed machine learning using Dask alongside existing machine learning libraries like Scikit-Learn, XGBoost, and TensorFlow.

5.6.4 Odo

Odo provides a uniform API for moving data between different formats. It uses pandas own `read_csv` for CSV IO and leverages many existing packages such as PyTables, h5py, and pymongo to move data between non pandas formats. Its graph based approach is also extensible by end users for custom formats that may be too specific for the core of odo.

5.6.5 Ray

Pandas on Ray is an early stage DataFrame library that wraps Pandas and transparently distributes the data and computation. The user does not need to know how many cores their system has, nor do they need to specify how to distribute the data. In fact, users can continue using their previous Pandas notebooks while experiencing a considerable speedup from Pandas on Ray, even on a single machine. Only a modification of the import statement is needed, as we demonstrate below. Once you've changed your import statement, you're ready to use Pandas on Ray just like you would Pandas.

```
# import pandas as pd
import ray.dataframe as pd
```

5.6.6 Vaex

Increasingly, packages are being built on top of pandas to address specific needs in data preparation, analysis and visualization. Vaex is a python library for Out-of-Core DataFrames (similar to Pandas), to visualize and explore big tabular datasets. It can calculate statistics such as mean, sum, count, standard deviation etc, on an N-dimensional grid up to a billion (10^9) objects/rows per second. Visualization is done using histograms, density plots and 3d volume rendering, allowing interactive exploration of big data. Vaex uses memory mapping, zero memory copy policy and lazy computations for best performance (no memory wasted).

- vaex.from_pandas
- vaex.to_pandas_df

5.7 Data validation

5.7.1 Engarde

Engarde is a lightweight library used to explicitly state your assumptions about your datasets and check that they're *actually* true.

5.8 Extension Data Types

Pandas provides an interface for defining *extension types* to extend NumPy's type system. The following libraries implement that interface to provide types not found in NumPy or pandas, which work well with pandas' data containers.

5.8.1 cyberpandas

Cyberpandas provides an extension type for storing arrays of IP Addresses. These arrays can be stored inside pandas' Series and DataFrame.

5.9 Accessors

A directory of projects providing *extension accessors*. This is for users to discover new accessors and for library authors to coordinate on the namespace.

Library	Accessor	Classes
cyberpandas	`ip`	`Series`
pdvega	`vgplot`	`Series, DataFrame`

CHAPTER

SIX

API REFERENCE

This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in `pandas.*` namespace are public.

Some subpackages are public which include `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Public functions in `pandas.io` and `pandas.tseries` submodules are mentioned in the documentation. `pandas.api.types` subpackage holds some public functions related to data types in pandas.

> **Warning:** The `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

6.1 Input/Output

6.1.1 Pickling

read_pickle(path[, compression])	Load pickled pandas object (or any object) from file.

pandas.read_pickle

`pandas.`**`read_pickle`**(*path*, *compression='infer'*)

Load pickled pandas object (or any object) from file.

> **Warning:** Loading pickled data received from untrusted sources can be unsafe. See here.

Parameters

path [str] File path where the pickled object will be loaded.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, xz or zip if path ends in '.gz', '.bz2', '.xz', or '.zip' respectively, and no decompression otherwise. Set to None for no decompression.

New in version 0.20.0.

Returns

unpickled [same type as object stored in file]

See also:

DataFrame.to_pickle Pickle (serialize) DataFrame object to file.

Series.to_pickle Pickle (serialize) Series object to file.

read_hdf Read HDF5 file into a DataFrame.

read_sql Read SQL query or database table into a DataFrame.

read_parquet Load a parquet object, returning a DataFrame.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> pd.to_pickle(original_df, "./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

6.1.2 Flat File

read_table(filepath_or_buffer[, sep, …])	(DEPRECATED) Read general delimited file into DataFrame.
read_csv(filepath_or_buffer[, sep, …])	Read a comma-separated values (csv) file into DataFrame.
read_fwf(filepath_or_buffer[, colspecs, …])	Read a table of fixed-width formatted lines into DataFrame.
read_msgpack(path_or_buf[, encoding, iterator])	Load msgpack pandas object from the specified file path

pandas.read_table

`pandas.read_table`(*filepath_or_buffer, sep=False, delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*)

Read general delimited file into DataFrame.

Deprecated since version 0.24.0.

Use `pandas.read_csv()` instead, passing `sep='\t'` if necessary.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object, or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts either `pathlib.Path` or `py._path.local.LocalPath`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

sep [str, default '\t' (tab-stop)] Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

delimiter [str, default `None`] Alias for sep.

header [int, list of int, default 'infer'] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

names [array-like, optional] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

index_col [int, sequence or bool, optional] Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to not use the first column as the index (row names).

usecols [list-like or callable, optional] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

prefix [str, optional] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

mangle_dupe_cols [bool, default True] Duplicate columns will be specified as 'X', 'X.1', …'X.N', rather than 'X'…'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

dtype [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use *str* or *object* together with suitable *na_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [{'c', 'python'}, optional] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

converters [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, optional] Values to consider as True.

false_values [list, optional] Values to consider as False.

skipinitialspace [bool, default False] Skip spaces after delimiter.

skiprows [list-like, int or callable, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

skipfooter [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

nrows [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

na_values [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is True, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is True, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is False, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is False, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as False, the *keep_default_na* and *na_values* parameters will be ignored.

na_filter [bool, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

parse_dates [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format [bool, default False] If True and *parse_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_col [bool, default False] If True and *parse_dates* specifies combining multiple columns then keep the original columns.

date_parser [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse_dates* into a single array and pass that; and 3) call *date_parser* once for each row using one or more strings (corresponding to the columns defined by *parse_dates*) as arguments.

dayfirst [bool, default False] DD/MM format dates, international and European format.

iterator [bool, default False] Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

chunksize [int, optional] Return TextFileReader object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer' and *filepath_or_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

thousands [str, optional] Thousands separator.

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

lineterminator [str (length 1), optional] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per `csv.QUOTE_*` constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequote [bool, default `True`] When quotechar is specified and quoting is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive quotechar elements INSIDE a field as a single `quotechar` element.

escapechar [str (length 1), optional] One-character string used to escape other characters.

comment [str, optional] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in 'a,b,c' being treated as the header.

encoding [str, optional] Encoding to use for UTF when reading/writing (ex. 'utf-8'). List of Python standard encodings .

dialect [str or csv.Dialect, optional] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

tupleize_cols [bool, default False] Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns).

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

error_bad_lines [bool, default True] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these "bad lines" will dropped from the DataFrame that is returned.

warn_bad_lines [bool, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

delim_whitespace [bool, default False] Specifies whether or not whitespace (e.g. `' '` or `'    '`) will be used as the sep. Equivalent to setting `sep='\s+'`. If this option is set to True, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

low_memory [bool, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the *dtype* parameter. Note that the entire file is read into a single DataFrame regardless, use the *chunksize* or *iterator* parameter to return the data in chunks. (Only valid with C parser).

memory_map [bool, default False] If a filepath is provided for *filepath_or_buffer*, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

float_precision [str, optional] Specifies which converter the C engine should use for floating-point values. The options are *None* for the ordinary converter, *high* for the high-precision converter, and *round_trip* for the round-trip converter.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

`read_fwf` Read a table of fixed-width formatted lines into DataFrame.

Examples

```
>>> pd.read_table('data.csv')  # doctest: +SKIP
```

pandas.read_csv

`pandas.`**`read_csv`**(*filepath_or_buffer*, *sep=', '*, *delimiter=None*, *header='infer'*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *prefix=None*, *mangle_dupe_cols=True*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skipinitialspace=False*, *skiprows=None*, *skipfooter=0*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *na_filter=True*, *verbose=False*, *skip_blank_lines=True*, *parse_dates=False*, *infer_datetime_format=False*, *keep_date_col=False*, *date_parser=None*, *dayfirst=False*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal=b'.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *doublequote=True*, *escapechar=None*, *comment=None*, *encoding=None*, *dialect=None*, *tupleize_cols=None*, *error_bad_lines=True*, *warn_bad_lines=True*, *delim_whitespace=False*, *low_memory=True*, *memory_map=False*, *float_precision=None*)

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object, or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts either `pathlib.Path` or `py._path.local.LocalPath`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

sep [str, default ','] Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

delimiter [str, default `None`] Alias for sep.

header [int, list of int, default 'infer'] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

names [array-like, optional] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

index_col [int, sequence or bool, optional] Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to not use the first column as the index (row names).

usecols [list-like or callable, optional] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

prefix [str, optional] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

mangle_dupe_cols [bool, default True] Duplicate columns will be specified as 'X', 'X.1', …'X.N', rather than 'X'…'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

dtype [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use *str* or *object* together with suitable *na_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [{'c', 'python'}, optional] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

converters [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, optional] Values to consider as True.

false_values [list, optional] Values to consider as False.

skipinitialspace [bool, default False] Skip spaces after delimiter.

skiprows [list-like, int or callable, optional] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

skipfooter [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

nrows [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

na_values [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is True, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is True, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is False, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is False, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as False, the *keep_default_na* and *na_values* parameters will be ignored.

na_filter [bool, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

parse_dates [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format [bool, default False] If True and *parse_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_col [bool, default False] If True and *parse_dates* specifies combining multiple columns then keep the original columns.

date_parser [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse_dates* into a single array and pass that; and 3) call *date_parser* once for each row using one or more strings (corresponding to the columns defined by *parse_dates*) as arguments.

dayfirst [bool, default False] DD/MM format dates, international and European format.

iterator [bool, default False] Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

chunksize [int, optional] Return TextFileReader object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer' and *filepath_or_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

thousands [str, optional] Thousands separator.

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

lineterminator [str (length 1), optional] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int or csv.QUOTE_* instance, default 0] Control field quoting behavior per `csv.QUOTE_*` constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequote [bool, default `True`] When quotechar is specified and quoting is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive quotechar elements INSIDE a field as a single `quotechar` element.

escapechar [str (length 1), optional] One-character string used to escape other characters.

comment [str, optional] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in 'a,b,c' being treated as the header.

encoding [str, optional] Encoding to use for UTF when reading/writing (ex. 'utf-8'). List of Python standard encodings .

dialect [str or csv.Dialect, optional] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

tupleize_cols [bool, default False] Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns).

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

error_bad_lines [bool, default True] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these "bad lines" will dropped from the DataFrame that is returned.

warn_bad_lines [bool, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

delim_whitespace [bool, default False] Specifies whether or not whitespace (e.g. `' '` or `'    '`) will be used as the sep. Equivalent to setting `sep='\s+'`. If this option is set to True, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

low_memory [bool, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the *dtype* parameter. Note that the entire file is read into a single DataFrame regardless, use the *chunksize* or *iterator* parameter to return the data in chunks. (Only valid with C parser).

memory_map [bool, default False] If a filepath is provided for *filepath_or_buffer*, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

float_precision [str, optional] Specifies which converter the C engine should use for floating-point values. The options are *None* for the ordinary converter, *high* for the high-precision converter, and *round_trip* for the round-trip converter.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

read_csv Read a comma-separated values (csv) file into DataFrame.

read_fwf Read a table of fixed-width formatted lines into DataFrame.

Examples

```
>>> pd.read_csv('data.csv')  # doctest: +SKIP
```

pandas.read_fwf

pandas.**read_fwf**(*filepath_or_buffer*, *colspecs='infer'*, *widths=None*, *infer_nrows=100*, ***kwds*)

Read a table of fixed-width formatted lines into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object, or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts either `pathlib.Path` or `py._path.local.LocalPath`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

colspecs [list of tuple (int, int) or 'infer'. optional] A list of tuples giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data which are not being skipped via skiprows (default='infer').

widths [list of int, optional] A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.

infer_nrows [int, default 100] The number of rows to consider when letting the parser determine the *colspecs*.

New in version 0.24.0.

****kwds** [optional] Optional keyword arguments can be passed to `TextFileReader`.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

to_csv Write DataFrame to a comma-separated values (csv) file.

read_csv Read a comma-separated values (csv) file into DataFrame.

Examples

```
>>> pd.read_fwf('data.csv')  # doctest: +SKIP
```

pandas.read_msgpack

`pandas.read_msgpack(path_or_buf, encoding='utf-8', iterator=False, **kwargs)`

Load msgpack pandas object from the specified file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path_or_buf [string File path, BytesIO like or string]

encoding [Encoding for decoding msgpack str type]

iterator [boolean, if True, return an iterator to the unpacker] (default is False)

Returns

obj [same type as object stored in file]

6.1.3 Clipboard

read_clipboard([sep])	Read text from clipboard and pass to read_csv.

pandas.read_clipboard

`pandas.read_clipboard(sep='\\s+', **kwargs)`

Read text from clipboard and pass to read_csv. See read_csv for the full argument list

Parameters

sep [str, default '\s+'] A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.

Returns

parsed [DataFrame]

6.1.4 Excel

read_excel(io[, sheet_name, header, names, . . .])	Read an Excel file into a pandas DataFrame.
ExcelFile.parse([sheet_name, header, names, . . .])	Parse specified sheet(s) into a DataFrame

pandas.read_excel

`pandas.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, parse_cols=None, usecols=None, squeeze=False, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, keep_default_na=True, verbose=False, parse_dates=False, date_parser=None, thousands=None, comment=None, skip_footer=0, skipfooter=0, convert_float=True, mangle_dupe_cols=True, **kwds)`

Read an Excel file into a pandas DataFrame.

Support both *xls* and *xlsx* file extensions from a local filesystem or URL. Support an option to read a single sheet or a list of sheets.

Parameters

io [str, file descriptor, pathlib.Path, ExcelFile or xlrd.Book] The string could be a URL. Valid URL schemes include http, ftp, s3, gcs, and file. For file URLs, a host is expected. For instance, a local file could be /path/to/workbook.xlsx.

sheet_name [str, int, list, or None, default 0] Strings are used for sheet names. Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets.

Available cases:

- Defaults to `0`: 1st sheet as a *DataFrame*
- `1`: 2nd sheet as a *DataFrame*
- `"Sheet1"`: Load sheet with name "Sheet1"
- `[0, 1, "Sheet5"]`: Load first, second and sheet named "Sheet5" as a dict of *DataFrame*
- None: All sheets.

header [int, list of int, default 0] Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a `MultiIndex`. Use None if there is no header.

names [array-like, default None] List of column names to use. If file contains no header row, then you should explicitly pass header=None.

index_col [int, list of int, default None] Column (0-indexed) to use as the row labels of the DataFrame. Pass None if there is no such column. If a list is passed, those columns will be combined into a `MultiIndex`. If a subset of data is selected with `usecols`, index_col is based on the subset.

parse_cols [int or list, default None] Alias of *usecols*.

Deprecated since version 0.21.0: Use *usecols* instead.

usecols [int, str, list-like, or callable default None] Return a subset of the columns. * If None, then parse all columns. * If int, then indicates last column to be parsed.

Deprecated since version 0.24.0: Pass in a list of int instead from 0 to *usecols* inclusive.

- If str, then indicates comma separated list of Excel column letters and column ranges (e.g. "A:E" or "A,C,E:F"). Ranges are inclusive of both sides.
- If list of int, then indicates list of column numbers to be parsed.
- If list of string, then indicates list of column names to be parsed.

New in version 0.24.0.

- If callable, then evaluate each column name against it and parse the column if the callable returns `True`.

New in version 0.24.0.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

dtype [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use *object* to preserve data as stored in Excel and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

New in version 0.20.0.

engine [str, default None] If io is not a buffer or path, this must be set to identify io. Acceptable values are None or xlrd.

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the Excel cell content, and return the transformed content.

true_values [list, default None] Values to consider as True.

New in version 0.19.0.

false_values [list, default None] Values to consider as False.

New in version 0.19.0.

skiprows [list-like] Rows to skip at the beginning (0-indexed).

nrows [int, default None] Number of rows to parse.

New in version 0.23.0.

na_values [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, default True] If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

parse_dates [bool, list-like, or dict, default False] The behavior is as follows:

- bool. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {{'foo' : [1, 3]}} -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

date_parser [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse_dates* into a single array and pass that; and 3) call *date_parser* once for each row using one or more strings (corresponding to the columns defined by *parse_dates*) as arguments.

thousands [str, default None] Thousands separator for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.

comment [str, default None] Comments out remainder of line. Pass a character or characters to this argument to indicate comments in the input file. Any data between the comment string and the end of the current line is ignored.

skip_footer [int, default 0] Alias of *skipfooter*.

Deprecated since version 0.23.0: Use *skipfooter* instead.

skipfooter [int, default 0] Rows at the end to skip (0-indexed).

convert_float [bool, default True] Convert integral floats to int (i.e., 1.0 –> 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally.

mangle_dupe_cols [bool, default True] Duplicate columns will be specified as 'X', 'X.1', …'X.N', rather than 'X'…'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

****kwds** [optional] Optional keyword arguments can be passed to `TextFileReader`.

Returns

DataFrame or dict of DataFrames DataFrame from the passed in Excel file. See notes in sheet_name argument for more information on when a dict of DataFrames is returned.

See also:

`to_excel` Write DataFrame to an Excel file.

`to_csv` Write DataFrame to a comma-separated values (csv) file.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

`read_fwf` Read a table of fixed-width formatted lines into DataFrame.

Examples

The file can be read using the file name as string or an open file object:

```
>>> pd.read_excel('tmp.xlsx', index_col=0)  # doctest: +SKIP
       Name  Value
0   string1      1
1   string2      2
2  #Comment      3
```

```
>>> pd.read_excel(open('tmp.xlsx', 'rb'),
...               sheet_name='Sheet3')  # doctest: +SKIP
   Unnamed: 0      Name  Value
0           0   string1      1
1           1   string2      2
2           2  #Comment      3
```

Index and header can be specified via the *index_col* and *header* arguments

```
>>> pd.read_excel('tmp.xlsx', index_col=None, header=None)  # doctest: +SKIP
     0         1      2
0  NaN      Name  Value
1  0.0   string1      1
2  1.0   string2      2
3  2.0  #Comment      3
```

Column types are inferred but can be explicitly specified

```
>>> pd.read_excel('tmp.xlsx', index_col=0,
...               dtype={'Name': str, 'Value': float})  # doctest: +SKIP
       Name  Value
0   string1    1.0
1   string2    2.0
2  #Comment    3.0
```

True, False, and NA values, and thousands separators have defaults, but can be explicitly specified, too. Supply the values you would like as strings or lists of strings!

```
>>> pd.read_excel('tmp.xlsx', index_col=0,
...               na_values=['string1', 'string2'])  # doctest: +SKIP
       Name  Value
0       NaN      1
1       NaN      2
2  #Comment      3
```

Comment lines in the excel input file can be skipped using the *comment* kwarg

```
>>> pd.read_excel('tmp.xlsx', index_col=0, comment='#')  # doctest: +SKIP
      Name  Value
0  string1    1.0
1  string2    2.0
2     None    NaN
```

pandas.ExcelFile.parse

`ExcelFile.`**`parse`**(*sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=False, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, convert_float=True, mangle_dupe_cols=True, **kwds*)

Parse specified sheet(s) into a DataFrame

Equivalent to read_excel(ExcelFile, …) See the read_excel docstring for more info on accepted parameters

ExcelWriter(path[, engine, date_format, …])	Class for writing DataFrame objects into excel sheets, default is to use xlwt for xls, openpyxl for xlsx.

pandas.ExcelWriter

class pandas.**ExcelWriter**(*path, engine=None, date_format=None, datetime_format=None, mode='w', **engine_kwargs*)

Class for writing DataFrame objects into excel sheets, default is to use xlwt for xls, openpyxl for xlsx. See DataFrame.to_excel for typical usage.

Parameters

path [string] Path to xls or xlsx file.

engine [string (optional)] Engine to use for writing. If None, defaults to `io.excel.<extension>.writer`. NOTE: can only be passed as a keyword argument.

date_format [string, default None] Format string for dates written into Excel files (e.g. 'YYYY-MM-DD')

datetime_format [string, default None] Format string for datetime objects written into Excel files (e.g. 'YYYY-MM-DD HH:MM:SS')

mode [{'w' or 'a'}, default 'w'] File mode to use (write or append).

.. versionadded:: 0.24.0

Notes

None of the methods and properties are considered public.

For compatibility with CSV writers, ExcelWriter serializes lists and dicts to strings before writing.

Examples

Default usage:

```
>>> with ExcelWriter('path_to_file.xlsx') as writer:
...     df.to_excel(writer)
```

To write to separate sheets in a single file:

```
>>> with ExcelWriter('path_to_file.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet1')
...     df2.to_excel(writer, sheet_name='Sheet2')
```

You can set the date format or datetime format:

```
>>> with ExcelWriter('path_to_file.xlsx',
                      date_format='YYYY-MM-DD',
                      datetime_format='YYYY-MM-DD HH:MM:SS') as writer:
...     df.to_excel(writer)
```

You can also append to an existing Excel file:

```
>>> with ExcelWriter('path_to_file.xlsx', mode='a') as writer:
...     df.to_excel(writer, sheet_name='Sheet3')
```

Attributes

None	

Methods

None	

6.1.5 JSON

read_json([path_or_buf, orient, typ, dtype, ...])	Convert a JSON string to pandas object.

pandas.read_json

`pandas.read_json(path_or_buf=None, orient=None, typ='frame', dtype=True, convert_axes=True, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, lines=False, chunksize=None, compression='infer')`

Convert a JSON string to pandas object.

Parameters

path_or_buf [a valid JSON string or file-like, default: None] The string could be a URL. Valid URL schemes include http, ftp, s3, gcs, and file. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.json`

orient [string,] Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:

- `'split'` : dict like `{index -> [index], columns -> [columns], data -> [values]}`
- `'records'` : list like `[{column -> value}, ... , {column -> value}]`
- `'index'` : dict like `{index -> {column -> value}}`
- `'columns'` : dict like `{column -> {index -> value}}`
- `'values'` : just the values array

The allowed and default values depend on the value of the *typ* parameter.

- when `typ == 'series'`,
  - allowed orients are `{'split','records','index'}`
  - default is `'index'`
  - The Series index must be unique for orient `'index'`.
- when `typ == 'frame'`,
  - allowed orients are `{'split','records','index', 'columns', 'values', 'table'}`
  - default is `'columns'`
  - The DataFrame index must be unique for orients `'index'` and `'columns'`.
  - The DataFrame columns must be unique for orients `'index'`, `'columns'`, and `'records'`.

New in version 0.23.0: 'table' as an allowed value for the `orient` argument

typ [type of object to recover (series or frame), default 'frame']

dtype [boolean or dict, default True] If True, infer dtypes, if a dict of column to dtype, then use those, if False, then don't infer dtypes at all, applies only to the data.

convert_axes [boolean, default True] Try to convert the axes to the proper dtypes.

convert_dates [boolean, default True] List of columns to parse for dates; If True, then try to parse datelike columns default is True; a column label is datelike if

- it ends with `'_at'`,
- it ends with `'_time'`,

- it begins with `'timestamp'`,
- it is `'modified'`, or
- it is `'date'`

keep_default_dates [boolean, default True] If parsing dates, then parse the default datelike columns

numpy [boolean, default False] Direct decoding to numpy arrays. Supports numeric data only, but non-numeric column and index labels are supported. Note also that the JSON ordering MUST be the same for each term if numpy=True.

precise_float [boolean, default False] Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (False) is to use fast but less precise builtin functionality

date_unit [string, default None] The timestamp unit to detect if converting dates. The default behaviour is to try and detect the correct precision, but if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force parsing only seconds, milliseconds, microseconds or nanoseconds respectively.

encoding [str, default is 'utf-8'] The encoding to use to decode py3 bytes.

New in version 0.19.0.

lines [boolean, default False] Read the file as a json object per line.

New in version 0.19.0.

chunksize [integer, default None] Return JsonReader object for iteration. See the line-delimted json docs for more information on `chunksize`. This can only be passed if *lines=True*. If this is None, the file will be read into memory all at once.

New in version 0.21.0.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip or xz if path_or_buf is a string ending in '.gz', '.bz2', '.zip', or 'xz', respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.21.0.

Returns

result [Series or DataFrame, depending on the value of *typ*.]

See also:

DataFrame.to_json

Notes

Specific to `orient='table'`, if a *DataFrame* with a literal *Index* name of *index* gets written with `to_json()`, the subsequent read operation will incorrectly set the *Index* name to `None`. This is because *index* is also used by *DataFrame.to_json()* to denote a missing *Index* name, and the subsequent *read_json()* operation cannot distinguish between the two. The same limitation is encountered with a *MultiIndex* and any names beginning with `'level_'`.

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
```

Encoding/decoding a Dataframe using `'split'` formatted JSON:

```
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
>>> pd.read_json(_, orient='split')
      col 1 col 2
row 1     a     b
row 2     c     d
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
>>> pd.read_json(_, orient='index')
      col 1 col 2
row 1     a     b
row 2     c     d
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
>>> pd.read_json(_, orient='records')
  col 1 col 2
0     a     b
1     c     d
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
                "primaryKey": "index",
                "pandas_version": "0.20.0"},
    "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
            {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

json_normalize(data[, record_path, meta, ...])	Normalize semi-structured JSON data into a flat table.
build_table_schema(data[, index, ...])	Create a Table schema from `data`.

pandas.io.json.json_normalize

`pandas.io.json.`**`json_normalize`**(*data*, *record_path=None*, *meta=None*, *meta_prefix=None*, *record_prefix=None*, *errors='raise'*, *sep='.'*)

Normalize semi-structured JSON data into a flat table.

Parameters

data [dict or list of dicts] Unserialized JSON objects

record_path [string or list of strings, default None] Path in each object to list of records. If not passed, data will be assumed to be an array of records

meta [list of paths (string or list of strings), default None] Fields to use as metadata for each record in resulting table

meta_prefix [string, default None]

record_prefix [string, default None] If True, prefix records with dotted (?) path, e.g. foo.bar.field if path to records is ['foo', 'bar']

errors [{'raise', 'ignore'}, default 'raise']

- 'ignore' : will ignore KeyError if keys listed in meta are not always present
- 'raise' : will raise KeyError if keys listed in meta are not always present

New in version 0.20.0.

sep [string, default '.'] Nested records will generate names separated by sep, e.g., for sep='.', { 'foo' : { 'bar' : 0 } } -> foo.bar

New in version 0.20.0.

Returns

frame [DataFrame]

Examples

```
>>> from pandas.io.json import json_normalize
>>> data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
...         {'name': {'given': 'Mose', 'family': 'Regner'}},
...         {'id': 2, 'name': 'Faye Raker'}]
>>> json_normalize(data)
    id        name name.family name.first name.given name.last
0  1.0         NaN         NaN     Coleen        NaN      Volk
1  NaN         NaN      Regner        NaN       Mose       NaN
2  2.0  Faye Raker         NaN        NaN        NaN       NaN
```

```
>>> data = [{'state': 'Florida',
...          'shortname': 'FL',
...          'info': {
...               'governor': 'Rick Scott'
...          },
...          'counties': [{'name': 'Dade', 'population': 12345},
...                      {'name': 'Broward', 'population': 40000},
...                      {'name': 'Palm Beach', 'population': 60000}]},
...         {'state': 'Ohio',
...          'shortname': 'OH',
...          'info': {
...               'governor': 'John Kasich'
...          },
...          'counties': [{'name': 'Summit', 'population': 1234},
...                       {'name': 'Cuyahoga', 'population': 1337}]}]
>>> result = json_normalize(data, 'counties', ['state', 'shortname',
```

(continues on next page)

(continued from previous page)

```
...                                              ['info', 'governor']])
>>> result
         name  population info.governor    state  shortname
0        Dade       12345    Rick Scott  Florida         FL
1     Broward       40000    Rick Scott  Florida         FL
2  Palm Beach       60000    Rick Scott  Florida         FL
3      Summit        1234   John Kasich     Ohio         OH
4    Cuyahoga        1337   John Kasich     Ohio         OH
```

```
>>> data = {'A': [1, 2]}
>>> json_normalize(data, 'A', record_prefix='Prefix.')
    Prefix.0
0          1
1          2
```

pandas.io.json.build_table_schema

pandas.io.json.**build_table_schema**(*data*, *index=True*, *primary_key=None*, *version=True*)

Create a Table schema from `data`.

Parameters

data [Series, DataFrame]

index [bool, default True] Whether to include `data.index` in the schema.

primary_key [bool or None, default True] column names to designate as the primary key. The default *None* will set *'primaryKey'* to the index level or levels if the index is unique.

version [bool, default True] Whether to include a field *pandas_version* with the version of pandas that generated the schema.

Returns

schema [dict]

Notes

See *_as_json_table_type* for conversion types. Timedeltas as converted to ISO8601 duration format with 9 decimal places after the seconds field for nanosecond precision.

Categoricals are converted to the *any* dtype, and use the *enum* field constraint to list the allowed values. The *ordered* attribute is included in an *ordered* field.

Examples

```
>>> df = pd.DataFrame(
...     {'A': [1, 2, 3],
...      'B': ['a', 'b', 'c'],
...      'C': pd.date_range('2016-01-01', freq='d', periods=3),
...     }, index=pd.Index(range(3), name='idx'))
>>> build_table_schema(df)
{'fields': [{'name': 'idx', 'type': 'integer'},
{'name': 'A', 'type': 'integer'},
```

(continues on next page)

(continued from previous page)

```
{'name': 'B', 'type': 'string'},
{'name': 'C', 'type': 'datetime'}],
'pandas_version': '0.20.0',
'primaryKey': ['idx']}
```

6.1.6 HTML

read_html(io[, match, flavor, header, . . .])	Read HTML tables into a `list` of `DataFrame` objects.

pandas.read_html

pandas.**read_html**(*io*, *match='.+'*, *flavor=None*, *header=None*, *index_col=None*, *skiprows=None*, *attrs=None*, *parse_dates=False*, *tupleize_cols=None*, *thousands=', '*, *encoding=None*, *decimal='.'*, *converters=None*, *na_values=None*, *keep_default_na=True*, *displayed_only=True*)

Read HTML tables into a `list` of `DataFrame` objects.

Parameters

io [str or file-like] A URL, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with `'https'` you might try removing the `'s'`.

match [str or compiled regular expression, optional] The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to '.+' (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

flavor [str or None, container of strings] The parsing engine to use. 'bs4' and 'html5lib' are synonymous with each other, they are both there for backwards compatibility. The default of `None` tries to use `lxml` to parse and if that fails it falls back on `bs4` + `html5lib`.

header [int or list-like or None, optional] The row (or list of rows for a *MultiIndex*) to use to make the columns headers.

index_col [int or list-like or None, optional] The column (or list of columns) to use to create the index.

skiprows [int or list-like or slice or None, optional] 0-based. Number of rows to skip after parsing the column integer. If a sequence of integers or a slice is given, will skip the rows indexed by that sequence. Note that a single element sequence means 'skip the nth row' whereas an integer means 'skip n rows'.

attrs [dict or None, optional] This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to lxml or Beautiful Soup. However, these attributes must be valid HTML table attributes to work correctly. For example,

```
attrs = {'id': 'table'}
```

is a valid attribute dictionary because the 'id' HTML tag attribute is a valid HTML attribute for *any* HTML tag as per this document.

```
attrs = {'asdf': 'table'}
```

is *not* a valid attribute dictionary because 'asdf' is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found here. A working draft of the HTML 5 spec can be found here. It contains the latest information on table attributes for the modern web.

parse_dates [bool, optional] See `read_csv()` for more details.

tupleize_cols [bool, optional] If `False` try to parse multiple header rows into a `MultiIndex`, otherwise return raw tuples. Defaults to `False`.

Deprecated since version 0.21.0: This argument will be removed and will always convert to MultiIndex

thousands [str, optional] Separator to use to parse thousands. Defaults to `','`.

encoding [str or None, optional] The encoding used to decode the web page. Defaults to `None`.''None'' preserves the previous encoding behavior, which depends on the underlying parser library (e.g., the parser library will try to use the encoding provided by the document).

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

New in version 0.19.0.

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the cell (not column) content, and return the transformed content.

New in version 0.19.0.

na_values [iterable, default None] Custom NA values

New in version 0.19.0.

keep_default_na [bool, default True] If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to

New in version 0.19.0.

displayed_only [bool, default True] Whether elements with "display: none" should be parsed

New in version 0.23.0.

Returns

dfs [list of DataFrames]

See also:

pandas.read_csv

Notes

Before using this function you should read the *gotchas about the HTML parsing libraries*.

Expect to do some cleanup after you call this function. For example, you might need to manually assign column names if the column names are converted to NaN when you pass the *header=0* argument. We try to assume as little as possible about the structure of the table and push the idiosyncrasies of the HTML contained in the table to the user.

This function searches for `<table>` elements and only for `<tr>` and `<th>` rows and `<td>` elements within each `<tr>` or `<th>` element in the table. `<td>` stands for "table data". This function attempts to properly handle `colspan` and `rowspan` attributes. If the function has a `<thead>` argument, it is used to construct the header, otherwise the function attempts to find the header within the body (by putting rows with only `<th>` elements into the header).

New in version 0.21.0.

Similar to `read_csv()` the *header* argument is applied **after** *skiprows* is applied.

This function will *always* return a list of `DataFrame` *or* it will fail, e.g., it will *not* return an empty list.

Examples

See the *read_html documentation in the IO section of the docs* for some examples of reading in HTML tables.

6.1.7 HDFStore: PyTables (HDF5)

`read_hdf`(path_or_buf[, key, mode])	Read from the store, close it if we opened it.
`HDFStore.put`(key, value[, format, append])	Store object in HDFStore
`HDFStore.append`(key, value[, format, ...])	Append to Table in file.
`HDFStore.get`(key)	Retrieve pandas object stored in file
`HDFStore.select`(key[, where, start, stop, ...])	Retrieve pandas object stored in file, optionally based on where criteria
`HDFStore.info`()	Print detailed information on the store.
`HDFStore.keys`()	Return a (potentially unordered) list of the keys corresponding to the objects stored in the HDFStore.
`HDFStore.groups`()	return a list of all the top-level nodes (that are not themselves a pandas storage object)
`HDFStore.walk`([where])	Walk the pytables group hierarchy for pandas objects

pandas.read_hdf

`pandas.`**`read_hdf`**(*path_or_buf*, *key=None*, *mode='r'*, ***kwargs*)

Read from the store, close it if we opened it.

Retrieve pandas object stored in file, optionally based on where criteria

Parameters

path_or_buf [string, buffer or path object] Path to the file to open, or an open `pandas.HDFStore` object. Supports any object implementing the `__fspath__` protocol. This includes `pathlib.Path` and py._path.local.LocalPath objects.

New in version 0.19.0: support for pathlib, py.path.

New in version 0.21.0: support for __fspath__ protocol.

key [object, optional] The group identifier in the store. Can be omitted if the HDF file contains a single pandas object.

mode [{'r', 'r+', 'a'}, optional] Mode to use when opening the file. Ignored if path_or_buf is a `pandas.HDFStore`. Default is 'r'.

where [list, optional] A list of Term (or convertible) objects.

start [int, optional] Row number to start selection.

stop [int, optional] Row number to stop selection.

columns [list, optional] A list of columns names to return.

iterator [bool, optional] Return an iterator object.

chunksize [int, optional] Number of rows to include in an iteration when using an iterator.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

****kwargs** Additional keyword arguments passed to HDFStore.

Returns

item [object] The selected object. Return type depends on the object stored.

See also:

pandas.DataFrame.to_hdf Write a HDF file from a DataFrame.

pandas.HDFStore Low-level access to HDF files.

Examples

```
>>> df = pd.DataFrame([[1, 1.0, 'a']], columns=['x', 'y', 'z'])
>>> df.to_hdf('./store.h5', 'data')
>>> reread = pd.read_hdf('./store.h5')
```

pandas.HDFStore.put

`HDFStore.put`(*key*, *value*, *format=None*, *append=False*, ***kwargs*)
Store object in HDFStore

Parameters

key [object]

value [{Series, DataFrame, Panel}]

format ['fixed(f)|table(t)', default is 'fixed']

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

append [boolean, default False] This will force Table format, append the input data to the existing.

data_columns [list of columns to create as data columns, or True to] use all columns. See here # noqa

encoding [default None, provide an encoding for strings]

dropna [boolean, default False, do not write an ALL nan row to] the store settable by the option 'io.hdf.dropna_table'

pandas.HDFStore.append

`HDFStore.append(key, value, format=None, append=True, columns=None, dropna=None, **kwargs)`

Append to Table in file. Node must already exist and be Table format.

Parameters

key [object]

value [{Series, DataFrame, Panel}]

format ['table' is the default]

table(t) [table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

append [boolean, default True, append the input data to the] existing

data_columns [list of columns, or True, default None] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See here.

min_itemsize [dict of columns that specify minimum string sizes]

nan_rep [string to use as string nan represenation]

chunksize [size to chunk the writing]

expectedrows [expected TOTAL row size of this table]

encoding [default None, provide an encoding for strings]

dropna [boolean, default False, do not write an ALL nan row to] the store settable by the option 'io.hdf.dropna_table'

Notes

Does *not* check if data being appended overlaps with existing data in the table, so be careful

pandas.HDFStore.get

`HDFStore.get(key)`

Retrieve pandas object stored in file

Parameters

key [object]

Returns

obj [same type as object stored in file]

pandas.HDFStore.select

`HDFStore.select(key, where=None, start=None, stop=None, columns=None, iterator=False, chunksize=None, auto_close=False, **kwargs)`

Retrieve pandas object stored in file, optionally based on where criteria

Parameters

key [object]

where [list of Term (or convertible) objects, optional]

start [integer (defaults to None), row number to start selection]

stop [integer (defaults to None), row number to stop selection]

columns [a list of columns that if not None, will limit the return] columns

iterator [boolean, return an iterator, default False]

chunksize [nrows to include in iteration, return an iterator]

auto_close [boolean, should automatically close the store when] finished, default is False

Returns

The selected object

pandas.HDFStore.info

`HDFStore.info()`

Print detailed information on the store.

New in version 0.21.0.

pandas.HDFStore.keys

`HDFStore.keys()`

Return a (potentially unordered) list of the keys corresponding to the objects stored in the HDFStore. These are ABSOLUTE path-names (e.g. have the leading '/'

pandas.HDFStore.groups

`HDFStore.groups()`

return a list of all the top-level nodes (that are not themselves a pandas storage object)

pandas.HDFStore.walk

`HDFStore.walk(where='/')`

Walk the pytables group hierarchy for pandas objects

This generator will yield the group path, subgroups and pandas object names for each group. Any non-pandas PyTables objects that are not a group will be ignored.

The *where* group itself is listed first (preorder), then each of its child groups (following an alphanumerical order) is also traversed, following the same procedure.

New in version 0.24.0.

Parameters

where [str, optional] Group where to start walking. If not supplied, the root group is used.

Yields

path [str] Full path to a group (without trailing '/')

groups [list of str] names of the groups contained in *path*

leaves [list of str] names of the pandas objects contained in *path*

6.1.8 Feather

read_feather(path[, columns, use_threads])	Load a feather-format object from the file path

pandas.read_feather

pandas.**read_feather**(*path*, *columns=None*, *use_threads=True*)

Load a feather-format object from the file path

Parameters

path [string file path, or file-like object]

columns [sequence, default None] If not provided, all columns are read

nthreads [int, default 1]

Number of CPU threads to use when reading to pandas.DataFrame

use_threads [bool, default True]

Whether to parallelize reading using multiple threads

Returns

type of object stored in file

6.1.9 Parquet

read_parquet(path[, engine, columns])	Load a parquet object from the file path, returning a DataFrame.

pandas.read_parquet

pandas.**read_parquet**(*path*, *engine='auto'*, *columns=None*, ***kwargs*)

Load a parquet object from the file path, returning a DataFrame.

Parameters

path [string] File path

columns [list, default=None] If not None, only these columns will be read from the file.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

kwargs are passed to the engine

Returns

DataFrame

6.1.10 SAS

read_sas(filepath_or_buffer[, format, . . .])	Read SAS files stored as either XPORT or SAS7BDAT format files.

pandas.read_sas

pandas.**read_sas**(*filepath_or_buffer*, *format=None*, *index=None*, *encoding=None*, *chunksize=None*, *iterator=False*)

Read SAS files stored as either XPORT or SAS7BDAT format files.

Parameters

filepath_or_buffer [string or file-like object] Path to the SAS file.

format [string {'xport', 'sas7bdat'} or None] If None, file format is inferred from file extension. If 'xport' or 'sas7bdat', uses the corresponding format.

index [identifier of index column, defaults to None] Identifier of column that should be used as index of the DataFrame.

encoding [string, default is None] Encoding for text data. If None, text data are stored as raw bytes.

chunksize [int] Read file *chunksize* lines at a time, returns iterator.

iterator [bool, defaults to False] If True, returns an iterator for reading the file incrementally.

Returns

DataFrame if iterator=False and chunksize=None, else SAS7BDATReader

or XportReader

6.1.11 SQL

read_sql_table(table_name, con[, schema, . . .])	Read SQL database table into a DataFrame.
read_sql_query(sql, con[, index_col, . . .])	Read SQL query into a DataFrame.
read_sql(sql, con[, index_col, . . .])	Read SQL query or database table into a DataFrame.

6.1.12 Google BigQuery

read_gbq(query[, project_id, index_col, . . .])	Load data from Google BigQuery.

pandas.read_gbq

pandas.**read_gbq**(*query*, *project_id=None*, *index_col=None*, *col_order=None*, *reauth=False*, *auth_local_webserver=False*, *dialect=None*, *location=None*, *configuration=None*, *credentials=None*, *private_key=None*, *verbose=None*)

Load data from Google BigQuery.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

query [str] SQL-Like Query to return data values.

project_id [str, optional] Google BigQuery Account project ID. Optional when available from the environment.

index_col [str, optional] Name of result column to use for index in results DataFrame.

col_order [list(str), optional] List of BigQuery column names in the desired order for results DataFrame.

reauth [boolean, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

auth_local_webserver [boolean, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

dialect [str, default 'legacy'] Note: The default value is changing to 'standard' in a future verion.

SQL syntax dialect to use. Value can be one of:

`'legacy'` Use BigQuery's legacy SQL dialect. For more information see BigQuery Legacy SQL Reference.

`'standard'` Use BigQuery's standard SQL, which is compliant with the SQL 2011 standard. For more information see BigQuery Standard SQL Reference.

Changed in version 0.24.0.

location [str, optional] Location where the query job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of any datasets used in the query.

New in version 0.5.0 of pandas-gbq.

configuration [dict, optional] Query config parameters for job processing. For example:

configuration = {'query': {'useQueryCache': False}}

For more information see BigQuery REST API Reference.

credentials [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

New in version 0.24.0.

private_key [str, deprecated] Deprecated in pandas-gbq version 0.8.0. Use the `credentials` parameter and `google.oauth2.service_account.Credentials.from_service_account_info()` or `google.oauth2.service_account.Credentials.from_service_account_file()` instead.

Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

verbose [None, deprecated] Deprecated in pandas-gbq version 0.4.0. Use the logging module to adjust verbosity instead.

Returns

df: DataFrame DataFrame representing results of query.

See also:

`pandas_gbq.read_gbq` This function in the pandas-gbq library.

`pandas.DataFrame.to_gbq` Write a DataFrame to Google BigQuery.

6.1.13 STATA

`read_stata`(filepath_or_buffer[, ...])	Read Stata file into DataFrame.

pandas.read_stata

`pandas.`**`read_stata`**(*filepath_or_buffer*, *convert_dates=True*, *convert_categoricals=True*, *encoding=None*, *index_col=None*, *convert_missing=False*, *preserve_dtypes=True*, *columns=None*, *order_categoricals=True*, *chunksize=None*, *iterator=False*)

Read Stata file into DataFrame.

Parameters

filepath_or_buffer [string or file-like object] Path to .dta file or object implementing a binary read() functions.

convert_dates [boolean, defaults to True] Convert date variables to DataFrame time values.

convert_categoricals [boolean, defaults to True] Read value labels and convert columns to Categorical/Factor variables.

encoding [string, None or encoding] Encoding used to parse the files. None defaults to latin-1.

index_col [string, optional, default: None] Column to set as index.

convert_missing [boolean, defaults to False] Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissingValue objects.

preserve_dtypes [boolean, defaults to True] Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).

columns [list or None] Columns to retain. Columns will be returned in the given order. None returns all columns.

order_categoricals [boolean, defaults to True] Flag indicating whether converted categorical data are ordered.

chunksize [int, default None] Return StataReader object for iterations, returns chunks with given number of lines.

iterator [boolean, default False] Return StataReader object.

Returns

DataFrame or StataReader

See also:

`pandas.io.stata.StataReader` Low-level reader for Stata data files.

`pandas.DataFrame.to_stata` Export Stata data files.

Examples

Read a Stata dta file:

```
>>> df = pd.read_stata('filename.dta')
```

Read a Stata dta file in 10,000 line chunks:

```
>>> itr = pd.read_stata('filename.dta', chunksize=10000)
>>> for chunk in itr:
...     do_something(chunk)
```

StataReader.data(**kwargs)	(DEPRECATED) Reads observations from Stata file, converting them into a dataframe
StataReader.data_label()	Returns data label of Stata file
StataReader.value_labels()	Returns a dict, associating each variable name a dict, associating each value its corresponding label
StataReader.variable_labels()	Returns variable labels as a dict, associating each variable name with corresponding label
StataWriter.write_file()	

pandas.io.stata.StataReader.data

StataReader.**data**(***kwargs*)

Reads observations from Stata file, converting them into a dataframe

Deprecated since version This: is a legacy method. Use *read* in new code.

Parameters

convert_dates [boolean, defaults to True] Convert date variables to DataFrame time values.

convert_categoricals [boolean, defaults to True] Read value labels and convert columns to Categorical/Factor variables.

index_col [string, optional, default: None] Column to set as index.

convert_missing [boolean, defaults to False] Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissingValue objects.

preserve_dtypes [boolean, defaults to True] Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).

columns [list or None] Columns to retain. Columns will be returned in the given order. None returns all columns.

order_categoricals [boolean, defaults to True] Flag indicating whether converted categorical data are ordered.

Returns

DataFrame

pandas.io.stata.StataReader.data_label

`StataReader.data_label()`

Returns data label of Stata file

pandas.io.stata.StataReader.value_labels

`StataReader.value_labels()`

Returns a dict, associating each variable name a dict, associating each value its corresponding label

pandas.io.stata.StataReader.variable_labels

`StataReader.variable_labels()`

Returns variable labels as a dict, associating each variable name with corresponding label

pandas.io.stata.StataWriter.write_file

`StataWriter.write_file()`

6.2 General functions

6.2.1 Data manipulations

melt(frame[, id_vars, value_vars, var_name, …])	Unpivots a DataFrame from wide format to long format, optionally leaving identifier variables set.
pivot(data[, index, columns, values])	Return reshaped DataFrame organized by given index / column values.
pivot_table(data[, values, index, columns, …])	Create a spreadsheet-style pivot table as a DataFrame.
crosstab(index, columns[, values, rownames, …])	Compute a simple cross-tabulation of two (or more) factors.
cut(x, bins[, right, labels, retbins, …])	Bin values into discrete intervals.
qcut(x, q[, labels, retbins, precision, …])	Quantile-based discretization function.
merge(left, right[, how, on, left_on, …])	Merge DataFrame or named Series objects with a database-style join.
merge_ordered(left, right[, on, left_on, …])	Perform merge with optional filling/interpolation designed for ordered data like time series data.
merge_asof(left, right[, on, left_on, …])	Perform an asof merge.
concat(objs[, axis, join, join_axes, …])	Concatenate pandas objects along a particular axis with optional set logic along the other axes.
get_dummies(data[, prefix, prefix_sep, …])	Convert categorical variable into dummy/indicator variables
factorize(values[, sort, order, …])	Encode the object as an enumerated type or categorical variable.
unique(values)	Hash table-based unique.
wide_to_long(df, stubnames, i, j[, sep, suffix])	Wide panel to long format.

pandas.melt

`pandas.melt`(*frame*, *id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

Unpivots a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

Parameters

frame [DataFrame]

id_vars [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

See also:

DataFrame.melt, *pivot_table*, *DataFrame.pivot*

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> pd.melt(df, col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> pd.melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

pandas.pivot

pandas.**pivot**(*data*, *index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the *User Guide* for more on reshaping.

Parameters

data [DataFrame]

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

Returns

DataFrame Returns reshaped DataFrame.

Raises

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

DataFrame.pivot_table Generalization of pivot that can handle duplicate values for one index/column pair.

DataFrame.unstack Pivot based on the index values instead of a column.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
```

(continues on next page)

(continued from previous page)

```
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pandas.pivot_table

pandas.**pivot_table**(*data*, *values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

data [DataFrame]

values [column to aggregate, optional]

index [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

Returns

table [DataFrame]

See also:

DataFrame.pivot Pivot without aggregation that can handle non-numeric data.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two    NaN      6
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum, fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': np.mean})
>>> table
                  D         E
               mean      mean
A   C
bar large  5.500000  7.500000
```

(continues on next page)

(continued from previous page)

```
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max      mean min
A   C
bar large  5.500000  9   7.500000   6
    small  5.500000  9   8.500000   8
foo large  2.000000  5   4.500000   4
    small  2.333333  6   4.333333   2
```

pandas.crosstab

`pandas.crosstab`(*index*, *columns*, *values=None*, *rownames=None*, *colnames=None*, *aggfunc=None*, *margins=False*, *margins_name='All'*, *dropna=True*, *normalize=False*)

Compute a simple cross-tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed

Parameters

index [array-like, Series, or list of arrays/Series] Values to group by in the rows

columns [array-like, Series, or list of arrays/Series] Values to group by in the columns

values [array-like, optional] Array of values to aggregate according to the factors. Requires *aggfunc* be specified.

rownames [sequence, default None] If passed, must match number of row arrays passed

colnames [sequence, default None] If passed, must match number of column arrays passed

aggfunc [function, optional] If specified, requires *values* be specified as well

margins [boolean, default False] Add row/column margins (subtotals)

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

New in version 0.21.0.

dropna [boolean, default True] Do not include columns whose entries are all NaN

normalize [boolean, {'all', 'index', 'columns'}, or {0,1}, default False] Normalize by dividing all values by the sum of values.

- If passed 'all' or *True*, will normalize over all values.
- If passed 'index' will normalize over each row.
- If passed 'columns' will normalize over each column.
- If margins is *True*, will also normalize margin values.

New in version 0.18.1.

Returns

crosstab [DataFrame]

Notes

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

Any input passed containing Categorical data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

In the event that there aren't overlapping indexes an empty DataFrame will be returned.

Examples

```
>>> a = np.array(["foo", "foo", "foo", "foo", "bar", "bar",
...               "bar", "bar", "foo", "foo", "foo"], dtype=object)
>>> b = np.array(["one", "one", "one", "two", "one", "one",
...               "one", "two", "two", "two", "one"], dtype=object)
>>> c = np.array(["dull", "dull", "shiny", "dull", "dull", "shiny",
...               "shiny", "dull", "shiny", "shiny", "shiny"],
...               dtype=object)
```

```
>>> pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
... # doctest: +NORMALIZE_WHITESPACE
b   one        two
c   dull shiny dull shiny
a
bar    1     2    1     0
foo    2     2    1     2
```

```
>>> foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
>>> bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
>>> crosstab(foo, bar)  # 'c' and 'f' are not represented in the data,
                        # and will not be shown in the output because
                        # dropna is True by default. Set 'dropna=False'
                        # to preserve categories with no data
... # doctest: +SKIP
col_0  d  e
row_0
a      1  0
b      0  1
```

```
>>> crosstab(foo, bar, dropna=False)  # 'c' and 'f' are not represented
                        # in the data, but they still will be counted
                        # and shown in the output
... # doctest: +SKIP
col_0  d  e  f
row_0
a      1  0  0
b      0  1  0
c      0  0  0
```

pandas.cut

`pandas.cut`(*x*, *bins*, *right=True*, *labels=None*, *retbins=False*, *precision=3*, *include_lowest=False*, *duplicates='raise'*)

Bin values into discrete intervals.

Use *cut* when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable. For example, *cut* could convert ages to groups of age ranges. Supports binning into an equal number of bins, or a pre-specified array of bins.

Parameters

x [array-like] The input array to be binned. Must be 1-dimensional.

bins [int, sequence of scalars, or pandas.IntervalIndex] The criteria to bin by.

- int : Defines the number of equal-width bins in the range of *x*. The range of *x* is extended by .1% on each side to include the minimum and maximum values of *x*.
- sequence of scalars : Defines the bin edges allowing for non-uniform width. No extension of the range of *x* is done.
- IntervalIndex : Defines the exact bins to be used. Note that IntervalIndex for *bins* must be non-overlapping.

right [bool, default True] Indicates whether *bins* includes the rightmost edge or not. If `right == True` (the default), then the *bins* `[1, 2, 3, 4]` indicate (1,2], (2,3], (3,4]. This argument is ignored when *bins* is an IntervalIndex.

labels [array or bool, optional] Specifies the labels for the returned bins. Must be the same length as the resulting bins. If False, returns only integer indicators of the bins. This affects the type of the output container (see below). This argument is ignored when *bins* is an IntervalIndex.

retbins [bool, default False] Whether to return the bins or not. Useful when bins is provided as a scalar.

precision [int, default 3] The precision at which to store and display the bins labels.

include_lowest [bool, default False] Whether the first interval should be left-inclusive or not.

duplicates [{default 'raise', 'drop'}, optional] If bin edges are not unique, raise ValueError or drop non-uniques.

New in version 0.23.0.

Returns

out [pandas.Categorical, Series, or ndarray] An array-like object representing the respective bin for each value of *x*. The type depends on the value of *labels*.

- True (default) : returns a Series for Series *x* or a pandas.Categorical for all other inputs. The values stored within are Interval dtype.
- sequence of scalars : returns a Series for Series *x* or a pandas.Categorical for all other inputs. The values stored within are whatever the type in the sequence is.
- False : returns an ndarray of integers.

bins [numpy.ndarray or IntervalIndex.] The computed or specified bins. Only returned when *retbins=True*. For scalar or sequence *bins*, this is an ndarray with the computed bins. If set *duplicates=drop*, *bins* will drop non-unique bin. For an IntervalIndex *bins*, this is equal to *bins*.

See also:

qcut Discretize variable into equal-sized buckets based on rank or based on sample quantiles.

pandas.Categorical Array type for storing data that come from a fixed set of values.

Series One-dimensional array with axis labels (including time series).

pandas.IntervalIndex Immutable Index implementing an ordered, sliceable set.

Notes

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Series or pandas.Categorical object.

Examples

Discretize into three equal-sized bins.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
... # doctest: +ELLIPSIS
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
```

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3, retbins=True)
... # doctest: +ELLIPSIS
([(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
array([0.994, 3.   , 5.   , 7.   ]))
```

Discovers the same bins, but assign them specific labels. Notice that the returned Categorical's categories are *labels* and is ordered.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]),
...        3, labels=["bad", "medium", "good"])
[bad, good, medium, medium, good, bad]
Categories (3, object): [bad < medium < good]
```

`labels=False` implies you just want the bins back.

```
>>> pd.cut([0, 1, 1, 2], bins=4, labels=False)
array([0, 1, 1, 3])
```

Passing a Series as an input returns a Series with categorical dtype:

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, 3)
... # doctest: +ELLIPSIS
a    (1.992, 4.667]
b    (1.992, 4.667]
c    (4.667, 7.333]
d     (7.333, 10.0]
e     (7.333, 10.0]
dtype: category
Categories (3, interval[float64]): [(1.992, 4.667] < (4.667, ...
```

Passing a Series as an input returns a Series with mapping value. It is used to map numerically to intervals based on bins.

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, [0, 2, 4, 6, 8, 10], labels=False, retbins=True, right=False)
... # doctest: +ELLIPSIS
(a    0.0
 b    1.0
 c    2.0
 d    3.0
 e    4.0
 dtype: float64, array([0, 2, 4, 6, 8]))
```

Use *drop* optional when bins is not unique

```
>>> pd.cut(s, [0, 2, 4, 6, 10, 10], labels=False, retbins=True,
...     right=False, duplicates='drop')
... # doctest: +ELLIPSIS
(a    0.0
 b    1.0
 c    2.0
 d    3.0
 e    3.0
 dtype: float64, array([0, 2, 4, 6, 8]))
```

Passing an IntervalIndex for *bins* results in those categories exactly. Notice that values not covered by the IntervalIndex are set to NaN. 0 is to the left of the first bin (which is closed on the right), and 1.5 falls between two bins.

```
>>> bins = pd.IntervalIndex.from_tuples([(0, 1), (2, 3), (4, 5)])
>>> pd.cut([0, 0.5, 1.5, 2.5, 4.5], bins)
[NaN, (0, 1], NaN, (2, 3], (4, 5]]
Categories (3, interval[int64]): [(0, 1] < (2, 3] < (4, 5]]
```

pandas.qcut

`pandas.qcut` (*x*, *q*, *labels=None*, *retbins=False*, *precision=3*, *duplicates='raise'*)

Quantile-based discretization function. Discretize variable into equal-sized buckets based on rank or based on sample quantiles. For example 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data point.

Parameters

x [1d ndarray or Series]

q [integer or array of quantiles] Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately array of quantiles, e.g. [0, .25, .5, .75, 1.] for quartiles

labels [array or boolean, default None] Used as labels for the resulting bins. Must be of the same length as the resulting bins. If False, return only integer indicators of the bins.

retbins [bool, optional] Whether to return the (bins, labels) or not. Can be useful if bins is given as a scalar.

precision [int, optional] The precision at which to store and display the bins labels

duplicates [{default 'raise', 'drop'}, optional] If bin edges are not unique, raise ValueError or drop non-uniques.

New in version 0.20.0.

Returns

out [Categorical or Series or array of integers if labels is False] The return type (Categorical or Series) depends on the input: a Series of type category if input is a Series else Categorical. Bins are represented as categories when categorical data is returned.

bins [ndarray of floats] Returned only if *retbins* is True.

Notes

Out of bounds values will be NA in the resulting Categorical object

Examples

```
>>> pd.qcut(range(5), 4)
... # doctest: +ELLIPSIS
[(-0.001, 1.0], (-0.001, 1.0], (1.0, 2.0], (2.0, 3.0], (3.0, 4.0]]
Categories (4, interval[float64]): [(-0.001, 1.0] < (1.0, 2.0] ...
```

```
>>> pd.qcut(range(5), 3, labels=["good", "medium", "bad"])
... # doctest: +SKIP
[good, good, medium, bad, bad]
Categories (3, object): [good < medium < bad]
```

```
>>> pd.qcut(range(5), 4, labels=False)
array([0, 0, 1, 2, 3])
```

pandas.merge

`pandas.merge`(*left*, *right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

left [DataFrame]

right [DataFrame or named Series] Object to merge with.

how [{'left', 'right', 'outer', 'inner'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes [tuple of (str, str), default ('_x', '_y')] Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Returns

DataFrame A DataFrame of the two merged objects.

See also:

merge_ordered Merge with optional filling/interpolation.

merge_asof Merge on nearest keys.

DataFrame.join Similar method using indices.

Notes

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0 Support for merging named Series objects was added in version 0.24.0

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo            5
1  foo           1  foo            8
2  foo           5  foo            5
3  foo           5  foo            8
4  bar           2  bar            6
5  baz           3  baz            7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

pandas.merge_ordered

`pandas.merge_ordered`(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *left_by=None*, *right_by=None*, *fill_method=None*, *suffixes=('_x', '_y')*, *how='outer'*)

Perform merge with optional filling/interpolation designed for ordered data like time series data. Optionally perform group-wise merge (see examples)

Parameters

left [DataFrame]

right [DataFrame]

on [label or list] Field names to join on. Must be found in both DataFrames.

left_on [label or list, or array-like] Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

right_on [label or list, or array-like] Field names to join on in right DataFrame or vector/list of vectors per left_on docs

left_by [column name or list of column names] Group left DataFrame by group columns and merge piece by piece with right DataFrame

right_by [column name or list of column names] Group right DataFrame by group columns and merge piece by piece with left DataFrame

fill_method [{'ffill', None}, default None] Interpolation method for data

suffixes [2-length sequence (tuple, list, …)] Suffix to apply to overlapping column names in the left and right side, respectively

how [{'left', 'right', 'outer', 'inner'}, default 'outer']

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

New in version 0.19.0.

Returns

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

See also:

merge, *merge_asof*

Examples

```
>>> A                          >>> B
   key  lvalue group              key  rvalue
0   a       1     a            0     b       1
1   c       2     a            1     c       2
2   e       3     a            2     d       3
3   a       1     b
4   c       2     b
5   e       3     b
```

```
>>> merge_ordered(A, B, fill_method='ffill', left_by='group')
  group key  lvalue  rvalue
0     a   a       1     NaN
1     a   b       1     1.0
2     a   c       2     2.0
3     a   d       2     3.0
4     a   e       3     3.0
5     b   a       1     NaN
6     b   b       1     1.0
7     b   c       2     2.0
8     b   d       2     3.0
9     b   e       3     3.0
```

pandas.merge_asof

pandas.**merge_asof**(*left, right, on=None, left_on=None, right_on=None, left_index=False, right_index=False, by=None, left_by=None, right_by=None, suffixes=('_x', '_y'), tolerance=None, allow_exact_matches=True, direction='backward'*)

Perform an asof merge. This is similar to a left-join except that we match on nearest key rather than equal keys.

Both DataFrames must be sorted by the key.

For each row in the left DataFrame:

- A "backward" search selects the last row in the right DataFrame whose 'on' key is less than or equal to the left's key.
- A "forward" search selects the first row in the right DataFrame whose 'on' key is greater than or equal to the left's key.
- A "nearest" search selects the row in the right DataFrame whose 'on' key is closest in absolute distance to the left's key.

The default is "backward" and is compatible in versions below 0.20.0. The direction parameter was added in version 0.20.0 and introduces "forward" and "nearest".

Optionally match on equivalent keys with 'by' before searching with 'on'.

New in version 0.19.0.

Parameters

left [DataFrame]

right [DataFrame]

on [label] Field name to join on. Must be found in both DataFrames. The data MUST be ordered. Furthermore this must be a numeric column, such as datetimelike, integer, or float. On or left_on/right_on must be given.

left_on [label] Field name to join on in left DataFrame.

right_on [label] Field name to join on in right DataFrame.

left_index [boolean] Use the index of the left DataFrame as the join key.

New in version 0.19.2.

right_index [boolean] Use the index of the right DataFrame as the join key.

New in version 0.19.2.

by [column name or list of column names] Match on these columns before performing merge operation.

left_by [column name] Field names to match on in the left DataFrame.

New in version 0.19.2.

right_by [column name] Field names to match on in the right DataFrame.

New in version 0.19.2.

suffixes [2-length sequence (tuple, list, …)] Suffix to apply to overlapping column names in the left and right side, respectively.

tolerance [integer or Timedelta, optional, default None] Select asof tolerance within this range; must be compatible with the merge index.

allow_exact_matches [boolean, default True]

- If True, allow matching with the same 'on' value (i.e. less-than-or-equal-to / greater-than-or-equal-to)
- If False, don't match the same 'on' value (i.e., strictly less-than / strictly greater-than)

direction ['backward' (default), 'forward', or 'nearest'] Whether to search for prior, subsequent, or closest matches.

New in version 0.20.0.

Returns

merged [DataFrame]

See also:

merge, *merge_ordered*

Examples

```
>>> left = pd.DataFrame({'a': [1, 5, 10], 'left_val': ['a', 'b', 'c']})
>>> left
    a left_val
0   1        a
1   5        b
2  10        c
```

```
>>> right = pd.DataFrame({'a': [1, 2, 3, 6, 7],
...                       'right_val': [1, 2, 3, 6, 7]})
>>> right
   a  right_val
0  1          1
1  2          2
2  3          3
3  6          6
4  7          7
```

```
>>> pd.merge_asof(left, right, on='a')
    a left_val  right_val
0   1        a          1
1   5        b          3
2  10        c          7
```

```
>>> pd.merge_asof(left, right, on='a', allow_exact_matches=False)
    a left_val  right_val
0   1        a        NaN
1   5        b        3.0
2  10        c        7.0
```

```
>>> pd.merge_asof(left, right, on='a', direction='forward')
    a left_val  right_val
0   1        a        1.0
1   5        b        6.0
2  10        c        NaN
```

```
>>> pd.merge_asof(left, right, on='a', direction='nearest')
    a left_val  right_val
0   1        a          1
1   5        b          6
2  10        c          7
```

We can use indexed DataFrames as well.

```
>>> left = pd.DataFrame({'left_val': ['a', 'b', 'c']}, index=[1, 5, 10])
>>> left
   left_val
1         a
5         b
10        c
```

```
>>> right = pd.DataFrame({'right_val': [1, 2, 3, 6, 7]},
...                      index=[1, 2, 3, 6, 7])
>>> right
   right_val
1          1
2          2
3          3
6          6
7          7
```

```
>>> pd.merge_asof(left, right, left_index=True, right_index=True)
   left_val  right_val
1         a          1
5         b          3
10        c          7
```

Here is a real-world times-series example

```
>>> quotes
                     time ticker     bid     ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT   51.95   51.96
2 2016-05-25 13:30:00.030   MSFT   51.97   51.98
3 2016-05-25 13:30:00.041   MSFT   51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL   97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT   52.01   52.03
```

```
>>> trades
                     time ticker   price  quantity
0 2016-05-25 13:30:00.023   MSFT   51.95        75
1 2016-05-25 13:30:00.038   MSFT   51.95       155
2 2016-05-25 13:30:00.048   GOOG  720.77       100
3 2016-05-25 13:30:00.048   GOOG  720.92       100
4 2016-05-25 13:30:00.048   AAPL   98.00       100
```

By default we are taking the asof of the quotes

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker')
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within 2ms between the quote time and the trade time

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker',
...                       tolerance=pd.Timedelta('2ms'))
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155     NaN     NaN
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. However *prior* data will propagate forward

```
>>> pd.merge_asof(trades, quotes,
...                       on='time',
...                       by='ticker',
...                       tolerance=pd.Timedelta('10ms'),
...                       allow_exact_matches=False)
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75     NaN     NaN
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100     NaN     NaN
3 2016-05-25 13:30:00.048   GOOG  720.92       100     NaN     NaN
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN
```

pandas.concat

`pandas.concat`(*objs*, *axis=0*, *join='outer'*, *join_axes=None*, *ignore_index=False*, *keys=None*, *levels=None*, *names=None*, *verify_integrity=False*, *sort=None*, *copy=True*)

Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

Parameters

objs [a sequence or mapping of Series, DataFrame, or Panel objects] If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised

axis [{0/'index', 1/'columns'}, default 0] The axis to concatenate along

join [{'inner', 'outer'}, default 'outer'] How to handle indexes on other axis(es)

join_axes [list of Index objects] Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic

ignore_index [boolean, default False] If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, . . . , n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.

keys [sequence, default None] If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level

levels [list of sequences, default None] Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys

names [list, default None] Names for the levels in the resulting hierarchical index

verify_integrity [boolean, default False] Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation

sort [boolean, default None] Sort non-concatenation axis if it is not already aligned when *join* is 'outer'. The current default of sorting is deprecated and will change to not-sorting in a future version of pandas.

Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

This has no effect when `join='inner'`, which already preserves the order of the non-concatenation axis.

New in version 0.23.0.

copy [boolean, default True] If False, do not copy data unnecessarily

Returns

concatenated [object, type of objs] When concatenating all `Series` along the index (axis=0), a `Series` is returned. When `objs` contains at least one `DataFrame`, a `DataFrame` is returned. When concatenating along the columns (axis=1), a `DataFrame` is returned.

See also:

Series.append, *DataFrame.append*, *DataFrame.join*, *DataFrame.merge*

Notes

The keys, levels, and names arguments are all optional.

A walkthrough of how this method fits in with other tools for combining pandas objects can be found here.

Examples

Combine two `Series`.

```
>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```

Clear the existing index and reset it in the result by setting the `ignore_index` option to `True`.

```
>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object
```

Add a hierarchical index at the outermost level of the data with the `keys` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2',])
s1  0    a
    1    b
s2  0    c
    1    d
dtype: object
```

Label the index keys you create with the `names` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2'],
...           names=['Series name', 'Row ID'])
Series name  Row ID
s1           0         a
             1         b
s2           0         c
             1         d
dtype: object
```

Combine two `DataFrame` objects with identical columns.

```
>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                    columns=['letter', 'number'])
>>> df1
  letter  number
0      a       1
1      b       2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                    columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
>>> pd.concat([df1, df2])
  letter  number
```

(continues on next page)

(continued from previous page)

```
0      a       1
1      b       2
0      c       3
1      d       4
```

Combine DataFrame objects with overlapping columns and return everything. Columns outside the intersection will be filled with NaN values.

```
>>> df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                    columns=['letter', 'number', 'animal'])
>>> df3
  letter  number animal
0      c       3    cat
1      d       4    dog
>>> pd.concat([df1, df3], sort=False)
  letter  number animal
0      a       1    NaN
1      b       2    NaN
0      c       3    cat
1      d       4    dog
```

Combine DataFrame objects with overlapping columns and return only those that are shared by passing inner to the join keyword argument.

```
>>> pd.concat([df1, df3], join="inner")
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4
```

Combine DataFrame objects horizontally along the x axis by passing in axis=1.

```
>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                    columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
  letter  number  animal    name
0      a       1    bird   polly
1      b       2  monkey  george
```

Prevent the result from including duplicate index values with the verify_integrity option.

```
>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
   0
a  1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
   0
a  2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
    ...
ValueError: Indexes have overlapping values: ['a']
```

pandas.get_dummies

`pandas.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)`

Convert categorical variable into dummy/indicator variables

Parameters

data [array-like, Series, or DataFrame]

prefix [string, list of strings, or dict of strings, default None] String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

prefix_sep [string, default '_'] If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

dummy_na [bool, default False] Add a column to indicate NaNs, if False NaNs are ignored.

columns [list-like, default None] Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

sparse [bool, default False] Whether the dummy-encoded columns should be be backed by a `SparseArray` (True) or a regular NumPy array (False).

drop_first [bool, default False] Whether to get k-1 dummies out of k categorical levels by removing the first level.

New in version 0.18.0.

dtype [dtype, default np.uint8] Data type for new columns. Only a single dtype is allowed.

New in version 0.23.0.

Returns

dummies [DataFrame]

See also:

`Series.str.get_dummies`

Examples

```
>>> s = pd.Series(list('abca'))
```

```
>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

```
>>> s1 = ['a', 'b', np.nan]
```

```
>>> pd.get_dummies(s1)
   a  b
0  1  0
```

(continues on next page)

(continued from previous page)

```
1  0  1
2  0  0
```

```
>>> pd.get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0    0
1  0  1    0
2  0  0    1
```

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
...                     'C': [1, 2, 3]})
```

```
>>> pd.get_dummies(df, prefix=['col1', 'col2'])
   C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1       1       0       0       1       0
1  2       0       1       1       0       0
2  3       1       0       0       0       1
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')))
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abc')), dtype=float)
     a    b    c
0  1.0  0.0  0.0
1  0.0  1.0  0.0
2  0.0  0.0  1.0
```

pandas.factorize

pandas.**factorize**(*values*, *sort=False*, *order=None*, *na_sentinel=-1*, *size_hint=None*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

Parameters

values [sequence] A 1-D sequence. Sequences that aren't pandas objects are coerced to ndarrays before factorization.

sort [bool, default False] Sort *uniques* and shuffle *labels* to maintain the relationship.

order Deprecated since version 0.23.0: This parameter has no effect and is deprecated.

na_sentinel [int, default -1] Value to mark "not found".

size_hint [int, optional] Hint to the hashtable sizer.

Returns

labels [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

uniques [ndarray, Index, or Categorical] The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

See also:

cut Discretize continuous-valued array.

unique Find the unique value in an array.

Examples

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like *Series.factorize()*.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

pandas.unique

`pandas.unique(values)`

Hash table-based unique. Uniques are returned in order of appearance. This does NOT sort.

Significantly faster than numpy.unique. Includes NA values.

Parameters

values [1d array-like]

Returns

unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

See also:

pandas.Index.unique, *pandas.Series.unique*

Examples

```
>>> pd.unique(pd.Series([2, 1, 3, 3]))
array([2, 1, 3])
```

```
>>> pd.unique(pd.Series([2] + [1] * 5))
array([2, 1])
```

```
>>> pd.unique(pd.Series([pd.Timestamp('20160101'),
...                      pd.Timestamp('20160101')]))
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
...                      pd.Timestamp('20160101', tz='US/Eastern')]))
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

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

```
>>> pd.unique(list('baabc'))
array(['b', 'a', 'c'], dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'),
...                                    categories=list('abc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.unique(pd.Series(pd.Categorical(list('baabc'),
...                                    categories=list('abc'),
...                                    ordered=True)))
[b, a, c]
Categories (3, object): [a < b < c]
```

An array of tuples

```
>>> pd.unique([('a', 'b'), ('b', 'a'), ('a', 'c'), ('b', 'a')])
array([('a', 'b'), ('b', 'a'), ('a', 'c')], dtype=object)
```

pandas.wide_to_long

pandas.**wide_to_long**(*df*, *stubnames*, *i*, *j*, *sep=''*, *suffix='\\d+'*)

Wide panel to long format. Less flexible but more user-friendly than melt.

With stubnames ['A', 'B'], this function expects to find one or more group of columns with format A-suffix1, A-suffix2,…, B-suffix1, B-suffix2,… You specify what you want to call this suffix in the resulting long format with *j* (for example *j='year'*)

Each row of these wide variables are assumed to be uniquely identified by *i* (can be a single column name or a list of column names)

All remaining variables in the data frame are left intact.

Parameters

df [DataFrame] The wide-format DataFrame

stubnames [str or list-like] The stub name(s). The wide format variables are assumed to start with the stub names.

i [str or list-like] Column(s) to use as id variable(s)

j [str] The name of the sub-observation variable. What you wish to name your suffix in the long format.

sep [str, default ""] A character indicating the separation of the variable names in the wide format, to be stripped from the names in the long format. For example, if your column names are A-suffix1, A-suffix2, you can strip the hyphen by specifying *sep='-'*

New in version 0.20.0.

suffix [str, default '\d+'] A regular expression capturing the wanted suffixes. '\d+' captures numeric suffixes. Suffixes with no numbers could be specified with the negated character class '\D+'. You can also further disambiguate suffixes, for example, if your wide variables are of the form A-one, B-two,.., and you have an unrelated column A-rating, you can ignore the last one by specifying *suffix='(!?one|two)'*

New in version 0.20.0.

Changed in version 0.23.0: When all suffixes are numeric, they are cast to int64/float64.

Returns

DataFrame A DataFrame that contains each stub name as a variable, with new index (i, j)

Notes

All extra variables are left untouched. This simply uses *pandas.melt* under the hood, but is hard-coded to "do the right thing" in a typical case.

Examples

```
>>> np.random.seed(123)
>>> df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
...                    "A1980" : {0 : "d", 1 : "e", 2 : "f"},
...                    "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
...                    "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
...                    "X"     : dict(zip(range(3), np.random.randn(3)))
...                   })
>>> df["id"] = df.index
>>> df
  A1970 A1980  B1970  B1980         X  id
0     a     d    2.5    3.2 -1.085631   0
1     b     e    1.2    1.3  0.997345   1
2     c     f    0.7    0.1  0.282978   2
>>> pd.wide_to_long(df, ["A", "B"], i="id", j="year")
... # doctest: +NORMALIZE_WHITESPACE
                X  A    B
id year
0  1970 -1.085631  a  2.5
1  1970  0.997345  b  1.2
2  1970  0.282978  c  0.7
0  1980 -1.085631  d  3.2
1  1980  0.997345  e  1.3
2  1980  0.282978  f  0.1
```

With multiple id columns

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht1': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   birth  famid  ht1  ht2
0      1      1  2.8  3.4
1      2      1  2.9  3.8
2      3      1  2.2  2.9
3      1      2  2.0  3.2
4      2      2  1.8  2.8
5      3      2  1.9  2.4
6      1      3  2.2  3.3
7      2      3  2.3  3.4
8      3      3  2.1  2.9
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age')
>>> l
... # doctest: +NORMALIZE_WHITESPACE
                  ht
famid birth age
1     1     1    2.8
            2    3.4
      2     1    2.9
            2    3.8
      3     1    2.2
            2    2.9
2     1     1    2.0
            2    3.2
      2     1    1.8
            2    2.8
      3     1    1.9
            2    2.4
3     1     1    2.2
            2    3.3
      2     1    2.3
            2    3.4
      3     1    2.1
            2    2.9
```

Going from long back to wide just takes some creative use of *unstack*

```
>>> w = l.unstack()
>>> w.columns = w.columns.map('{0[0]}{0[1]}'.format)
>>> w.reset_index()
   famid  birth  ht1  ht2
0      1      1  2.8  3.4
1      1      2  2.9  3.8
2      1      3  2.2  2.9
3      2      1  2.0  3.2
4      2      2  1.8  2.8
5      2      3  1.9  2.4
6      3      1  2.2  3.3
7      3      2  2.3  3.4
8      3      3  2.1  2.9
```

Less wieldy column names are also handled

```
>>> np.random.seed(0)
>>> df = pd.DataFrame({'A(quarterly)-2010': np.random.rand(3),
...                    'A(quarterly)-2011': np.random.rand(3),
...                    'B(quarterly)-2010': np.random.rand(3),
...                    'B(quarterly)-2011': np.random.rand(3),
...                    'X' : np.random.randint(3, size=3)})
>>> df['id'] = df.index
>>> df # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
   A(quarterly)-2010  A(quarterly)-2011  B(quarterly)-2010  ...
0           0.548814           0.544883           0.437587  ...
1           0.715189           0.423655           0.891773  ...
2           0.602763           0.645894           0.963663  ...
   X  id
0  0   0
1  1   1
2  1   2
```

```
>>> pd.wide_to_long(df, ['A(quarterly)', 'B(quarterly)'], i='id',
...                 j='year', sep='-')
... # doctest: +NORMALIZE_WHITESPACE
         X  A(quarterly)  B(quarterly)
id year
0  2010  0      0.548814     0.437587
1  2010  1      0.715189     0.891773
2  2010  1      0.602763     0.963663
0  2011  0      0.544883     0.383442
1  2011  1      0.423655     0.791725
2  2011  1      0.645894     0.528895
```

If we have many columns, we could also use a regex to find our stubnames and pass that list on to wide_to_long

```
>>> stubnames = sorted(
...     set([match[0] for match in df.columns.str.findall(
...         r'[A-B]\(.*\)').values if match != [] ])
... )
>>> list(stubnames)
['A(quarterly)', 'B(quarterly)']
```

All of the above examples have integers as suffixes. It is possible to have non-integers as suffixes.

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht_one': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht_two': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   birth  famid  ht_one  ht_two
0      1      1     2.8     3.4
1      2      1     2.9     3.8
2      3      1     2.2     2.9
3      1      2     2.0     3.2
4      2      2     1.8     2.8
5      3      2     1.9     2.4
6      1      3     2.2     3.3
7      2      3     2.3     3.4
8      3      3     2.1     2.9
```

```
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age',
                        sep='_', suffix='\w')
>>> l
... # doctest: +NORMALIZE_WHITESPACE
                  ht
famid birth age
1     1     one   2.8
            two   3.4
      2     one   2.9
            two   3.8
      3     one   2.2
            two   2.9
2     1     one   2.0
            two   3.2
      2     one   1.8
            two   2.8
      3     one   1.9
            two   2.4
3     1     one   2.2
            two   3.3
      2     one   2.3
            two   3.4
      3     one   2.1
            two   2.9
```

6.2.2 Top-level missing data

isna(obj)	Detect missing values for an array-like object.
isnull(obj)	Detect missing values for an array-like object.
notna(obj)	Detect non-missing values for an array-like object.
notnull(obj)	Detect non-missing values for an array-like object.

pandas.isna

pandas.**isna**(*obj*)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (`NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [scalar or array-like] Object to check for null or missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

notna Boolean inverse of pandas.isna.

Series.isna Detect missing values in a Series.

DataFrame.isna Detect missing values in a DataFrame.

Index.isna Detect missing values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

pandas.isnull

`pandas.`**`isnull`**(*obj*)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (`NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [scalar or array-like] Object to check for null or missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

notna Boolean inverse of pandas.isna.

Series.isna Detect missing values in a Series.

DataFrame.isna Detect missing values in a DataFrame.

Index.isna Detect missing values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
```

(continues on next page)

(continued from previous page)

```
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

pandas.notna

pandas.**notna**(*obj*)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is `NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [array-like or object value] Object to check for *not* null or *non*-missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

See also:

isna Boolean inverse of pandas.notna.

Series.notna Detect valid values in a Series.

DataFrame.notna Detect valid values in a DataFrame.

Index.notna Detect valid values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

pandas.notnull

pandas.**notnull**(*obj*)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is `NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters

obj [array-like or object value] Object to check for *not* null or *non*-missing values.

Returns

bool or array-like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

See also:

isna Boolean inverse of pandas.notna.

Series.notna Detect valid values in a Series.

DataFrame.notna Detect valid values in a DataFrame.

Index.notna Detect valid values in an Index.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

6.2.3 Top-level conversions

to_numeric(arg[, errors, downcast])	Convert argument to a numeric type.

pandas.to_numeric

pandas.**to_numeric**(*arg*, *errors='raise'*, *downcast=None*)

Convert argument to a numeric type.

The default return dtype is *float64* or *int64* depending on the data supplied. Use the *downcast* parameter to obtain other dtypes.

Parameters

arg [list, tuple, 1-d array, or Series]

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaN
- If 'ignore', then invalid parsing will return the input

downcast [{'integer', 'signed', 'unsigned', 'float'} , default None] If not None, and if the data has been successfully cast to a numerical dtype (or if the data was numeric to begin with), downcast that resulting data to the smallest numerical dtype possible according to the following rules:

- 'integer' or 'signed': smallest signed int dtype (min.: np.int8)
- 'unsigned': smallest unsigned int dtype (min.: np.uint8)
- 'float': smallest float dtype (min.: np.float32)

As this behaviour is separate from the core conversion to numeric values, any errors raised during the downcasting will be surfaced regardless of the value of the 'errors' input.

In addition, downcasting will only occur if the size of the resulting data's dtype is strictly larger than the dtype it is to be cast to, so if none of the dtypes checked satisfy that specification, no downcasting will be performed on the data.

New in version 0.19.0.

Returns

ret [numeric if parsing succeeded.] Return type depends on input. Series if Series, otherwise ndarray

See also:

pandas.DataFrame.astype Cast argument to a specified dtype.

pandas.to_datetime Convert argument to datetime.

pandas.to_timedelta Convert argument to timedelta.

numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

Take separate series and convert to numeric, coercing when told to

```
>>> s = pd.Series(['1.0', '2', -3])
>>> pd.to_numeric(s)
0    1.0
1    2.0
2   -3.0
dtype: float64
>>> pd.to_numeric(s, downcast='float')
0    1.0
1    2.0
2   -3.0
dtype: float32
>>> pd.to_numeric(s, downcast='signed')
0    1
1    2
2   -3
```

(continues on next page)

(continued from previous page)

```
dtype: int8
>>> s = pd.Series(['apple', '1.0', '2', -3])
>>> pd.to_numeric(s, errors='ignore')
0    apple
1      1.0
2        2
3       -3
dtype: object
>>> pd.to_numeric(s, errors='coerce')
0    NaN
1    1.0
2    2.0
3   -3.0
dtype: float64
```

6.2.4 Top-level dealing with datetimelike

to_datetime(arg[, errors, dayfirst, . . .])	Convert argument to datetime.
to_timedelta(arg[, unit, box, errors])	Convert argument to timedelta.
date_range([start, end, periods, freq, tz, . . .])	Return a fixed frequency DatetimeIndex.
bdate_range([start, end, periods, freq, tz, . . .])	Return a fixed frequency DatetimeIndex, with business day as the default frequency
period_range([start, end, periods, freq, name])	Return a fixed frequency PeriodIndex, with day (calendar) as the default frequency
timedelta_range([start, end, periods, freq, . . .])	Return a fixed frequency TimedeltaIndex, with day as the default frequency
infer_freq(index[, warn])	Infer the most likely frequency given the input index.

pandas.to_datetime

pandas.**to_datetime**(*arg*, *errors='raise'*, *dayfirst=False*, *yearfirst=False*, *utc=None*, *box=True*, *format=None*, *exact=True*, *unit=None*, *infer_datetime_format=False*, *origin='unix'*, *cache=False*)

Convert argument to datetime.

Parameters

arg [integer, float, string, datetime, list, tuple, 1-d array, Series] New in version 0.18.1: or DataFrame/dict-like

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaT
- If 'ignore', then invalid parsing will return the input

dayfirst [boolean, default False] Specify a date parse order if *arg* is str or its list-likes. If True, parses dates with the day first, eg 10/11/12 is parsed as 2012-11-10. Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug, based on dateutil behavior).

yearfirst [boolean, default False] Specify a date parse order if *arg* is str or its list-likes.

- If True parses dates with the year first, eg 10/11/12 is parsed as 2010-11-12.

- If both dayfirst and yearfirst are True, yearfirst is preceded (same as dateutil).

Warning: yearfirst=True is not strict, but will prefer to parse with year first (this is a known bug, based on dateutil behavior).

New in version 0.16.1.

utc [boolean, default None] Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well).

box [boolean, default True]

- If True returns a DatetimeIndex or Index-like object
- If False returns ndarray of values.

format [string, default None] strftime to parse time, eg "%d/%m/%Y", note that "%f" will parse all the way up to nanoseconds.

exact [boolean, True by default]

- If True, require an exact format match.
- If False, allow the format to match anywhere in the target string.

unit [string, default 'ns'] unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer or float number. This will be based off the origin. Example, with unit='ms' and origin='unix' (the default), this would calculate the number of milliseconds to the unix epoch start.

infer_datetime_format [boolean, default False] If True and no *format* is given, attempt to infer the format of the datetime strings, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

origin [scalar, default is 'unix'] Define the reference date. The numeric values would be parsed as number of units (defined by *unit*) since this reference date.

- If 'unix' (or POSIX) time; origin is set to 1970-01-01.
- If 'julian', unit must be 'D', and origin is set to beginning of Julian Calendar. Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.
- If Timestamp convertible, origin is set to Timestamp identified by origin.

New in version 0.20.0.

cache [boolean, default False] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.23.0.

Returns

ret [datetime if parsing succeeded.] Return type depends on input:

- list-like: DatetimeIndex
- Series: Series of datetime64 dtype
- scalar: Timestamp

In case when it is not possible to return designated types (e.g. when any element of input is before Timestamp.min or after Timestamp.max) return will have datetime.datetime type (or corresponding array/Series).

See also:

pandas.DataFrame.astype Cast argument to a specified dtype.

pandas.to_timedelta Convert argument to timedelta.

Examples

Assembling a datetime from multiple columns of a DataFrame. The keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same

```
>>> df = pd.DataFrame({'year': [2015, 2016],
                       'month': [2, 3],
                       'day': [4, 5]})
>>> pd.to_datetime(df)
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

If a date does not meet the timestamp limitations, passing errors='ignore' will return the original input instead of raising any exception.

Passing errors='coerce' will force an out-of-bounds date to NaT, in addition to forcing non-dates (or non-parseable dates) to NaT.

```
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='ignore')
datetime.datetime(1300, 1, 1, 0, 0)
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='coerce')
NaT
```

Passing infer_datetime_format=True can often-times speedup a parsing if its not an ISO8601 format exactly, but in a regular format.

```
>>> s = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000']*1000)
```

```
>>> s.head()
0    3/11/2000
1    3/12/2000
2    3/13/2000
3    3/11/2000
4    3/12/2000
dtype: object
```

```
>>> %timeit pd.to_datetime(s,infer_datetime_format=True)
100 loops, best of 3: 10.4 ms per loop
```

```
>>> %timeit pd.to_datetime(s,infer_datetime_format=False)
1 loop, best of 3: 471 ms per loop
```

Using a unix epoch time

```
>>> pd.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
>>> pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

> **Warning:** For float arg, precision rounding might happen. To prevent unexpected behavior use a fixed-width exact type.

Using a non-unix epoch origin

```
>>> pd.to_datetime([1, 2, 3], unit='D',
                   origin=pd.Timestamp('1960-01-01'))
0    1960-01-02
1    1960-01-03
2    1960-01-04
```

pandas.to_timedelta

`pandas.to_timedelta`(*arg*, *unit='ns'*, *box=True*, *errors='raise'*)

Convert argument to timedelta.

Timedeltas are absolute differences in times, expressed in difference units (e.g. days, hours, minutes, seconds). This method converts an argument from a recognized timedelta format / value into a Timedelta type.

Parameters

arg [str, timedelta, list-like or Series] The data to be converted to timedelta.

unit [str, default 'ns'] Denotes the unit of the arg. Possible values: ('Y', 'M', 'W', 'D', 'days', 'day', 'hours', hour', 'hr', 'h', 'm', 'minute', 'min', 'minutes', 'T', 'S', 'seconds', 'sec', 'second', 'ms', 'milliseconds', 'millisecond', 'milli', 'millis', 'L', 'us', 'microseconds', 'microsecond', 'micro', 'micros', 'U', 'ns', 'nanoseconds', 'nano', 'nanos', 'nanosecond', 'N').

box [bool, default True]

- If True returns a Timedelta/TimedeltaIndex of the results.
- If False returns a numpy.timedelta64 or numpy.darray of values of dtype timedelta64[ns].

errors [{'ignore', 'raise', 'coerce'}, default 'raise']

- If 'raise', then invalid parsing will raise an exception.
- If 'coerce', then invalid parsing will be set as NaT.
- If 'ignore', then invalid parsing will return the input.

Returns

timedelta64 or numpy.array of timedelta64 Output type returned if parsing succeeded.

See also:

DataFrame.astype Cast argument to a specified dtype.

to_datetime Convert argument to datetime.

Examples

Parsing a single string to a Timedelta:

```
>>> pd.to_timedelta('1 days 06:05:01.00003')
Timedelta('1 days 06:05:01.000030')
>>> pd.to_timedelta('15.5us')
Timedelta('0 days 00:00:00.000015')
```

Parsing a list or array of strings:

```
>>> pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT],
               dtype='timedelta64[ns]', freq=None)
```

Converting numbers by specifying the *unit* keyword argument:

```
>>> pd.to_timedelta(np.arange(5), unit='s')
TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02',
                '00:00:03', '00:00:04'],
               dtype='timedelta64[ns]', freq=None)
>>> pd.to_timedelta(np.arange(5), unit='d')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

Returning an ndarray by using the 'box' keyword argument:

```
>>> pd.to_timedelta(np.arange(5), box=False)
array([0, 1, 2, 3, 4], dtype='timedelta64[ns]')
```

pandas.date_range

pandas.**date_range**(*start=None*, *end=None*, *periods=None*, *freq=None*, *tz=None*, *normalize=False*, *name=None*, *closed=None*, ***kwargs*)

Return a fixed frequency DatetimeIndex.

Parameters

start [str or datetime-like, optional] Left bound for generating dates.

end [str or datetime-like, optional] Right bound for generating dates.

periods [integer, optional] Number of periods to generate.

freq [str or DateOffset, default 'D'] Frequency strings can have multiples, e.g. '5H'. See *here* for a list of frequency aliases.

tz [str or tzinfo, optional] Time zone name for returning localized DatetimeIndex, for example 'Asia/Hong_Kong'. By default, the resulting DatetimeIndex is timezone-naive.

normalize [bool, default False] Normalize start/end dates to midnight before generating date range.

name [str, default None] Name of the resulting DatetimeIndex.

closed [{None, 'left', 'right'}, optional] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None, the default).

****kwargs** For compatibility. Has no effect on the result.

Returns

rng [DatetimeIndex]

See also:

pandas.DatetimeIndex An immutable container for datetimes.

pandas.timedelta_range Return a fixed frequency TimedeltaIndex.

pandas.period_range Return a fixed frequency PeriodIndex.

pandas.interval_range Return a fixed frequency IntervalIndex.

Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `DatetimeIndex` will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see this link.

Examples

Specifying the values

The next four examples generate the same *DatetimeIndex*, but vary the combination of *start*, *end* and *periods*.

Specify *start* and *end*, with the default daily frequency.

```
>>> pd.date_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify *start* and *periods*, the number of periods (days).

```
>>> pd.date_range(start='1/1/2018', periods=8)
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify *end* and *periods*, the number of periods (days).

```
>>> pd.date_range(end='1/1/2018', periods=8)
DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
               '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

Specify *start*, *end*, and *periods*; the frequency is generated automatically (linearly spaced).

```
>>> pd.date_range(start='2018-04-24', end='2018-04-27', periods=3)
DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
               '2018-04-27 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Other Parameters

Changed the *freq* (frequency) to `'M'` (month end frequency).

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='M')
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31'],
              dtype='datetime64[ns]', freq='M')
```

Multiples are allowed

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='3M')
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
               '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

freq can also be specified as an Offset object.

```
>>> pd.date_range(start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3))
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
               '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

Specify *tz* to set the timezone.

```
>>> pd.date_range(start='1/1/2018', periods=5, tz='Asia/Tokyo')
DatetimeIndex(['2018-01-01 00:00:00+09:00', '2018-01-02 00:00:00+09:00',
               '2018-01-03 00:00:00+09:00', '2018-01-04 00:00:00+09:00',
               '2018-01-05 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq='D')
```

closed controls whether to include *start* and *end* that are on the boundary. The default includes boundary points on either end.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed=None)
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

Use `closed='left'` to exclude *end* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='left')
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'],
              dtype='datetime64[ns]', freq='D')
```

Use `closed='right'` to exclude *start* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='right')
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

pandas.bdate_range

`pandas.bdate_range`(*start=None*, *end=None*, *periods=None*, *freq='B'*, *tz=None*, *normalize=True*, *name=None*, *weekmask=None*, *holidays=None*, *closed=None*, ***kwargs*)

Return a fixed frequency DatetimeIndex, with business day as the default frequency

Parameters

start [string or datetime-like, default None] Left bound for generating dates.

end [string or datetime-like, default None] Right bound for generating dates.

periods [integer, default None] Number of periods to generate.

freq [string or DateOffset, default 'B' (business daily)] Frequency strings can have multiples, e.g. '5H'.

tz [string or None] Time zone name for returning localized DatetimeIndex, for example Asia/Beijing.

normalize [bool, default False] Normalize start/end dates to midnight before generating date range.

name [string, default None] Name of the resulting DatetimeIndex.

weekmask [string or None, default None] Weekmask of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed. The default value None is equivalent to 'Mon Tue Wed Thu Fri'.

New in version 0.21.0.

holidays [list-like or None, default None] Dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed.

New in version 0.21.0.

closed [string, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None).

****kwargs** For compatibility. Has no effect on the result.

Returns

DatetimeIndex

Notes

Of the four parameters: `start`, `end`, `periods`, and `freq`, exactly three must be specified. Specifying `freq` is a requirement for `bdate_range`. Use `date_range` if specifying `freq` is not desired.

To learn more about the frequency strings, please see this link.

Examples

Note how the two weekend days are skipped in the result.

```
>>> pd.bdate_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
           '2018-01-05', '2018-01-08'],
          dtype='datetime64[ns]', freq='B')
```

pandas.period_range

`pandas.period_range(start=None, end=None, periods=None, freq=None, name=None)`

Return a fixed frequency PeriodIndex, with day (calendar) as the default frequency

Parameters

start [string or period-like, default None] Left bound for generating periods

end [string or period-like, default None] Right bound for generating periods

periods [integer, default None] Number of periods to generate

freq [string or DateOffset, optional] Frequency alias. By default the freq is taken from *start* or *end* if those are Period objects. Otherwise, the default is `"D"` for daily frequency.

name [string, default None] Name of the resulting PeriodIndex

Returns

prng [PeriodIndex]

Notes

Of the three parameters: `start`, `end`, and `periods`, exactly two must be specified.

To learn more about the frequency strings, please see this link.

Examples

```
>>> pd.period_range(start='2017-01-01', end='2018-01-01', freq='M')
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', '2017-05',
             '2017-06', '2017-06', '2017-07', '2017-08', '2017-09',
             '2017-10', '2017-11', '2017-12', '2018-01'],
            dtype='period[M]', freq='M')
```

If `start` or `end` are `Period` objects, they will be used as anchor endpoints for a `PeriodIndex` with frequency matching that of the `period_range` constructor.

```
>>> pd.period_range(start=pd.Period('2017Q1', freq='Q'),
...                 end=pd.Period('2017Q2', freq='Q'), freq='M')
PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'],
            dtype='period[M]', freq='M')
```

pandas.timedelta_range

`pandas.`**`timedelta_range`**(*start=None*, *end=None*, *periods=None*, *freq=None*, *name=None*, *closed=None*)

Return a fixed frequency TimedeltaIndex, with day as the default frequency

Parameters

start [string or timedelta-like, default None] Left bound for generating timedeltas

end [string or timedelta-like, default None] Right bound for generating timedeltas

periods [integer, default None] Number of periods to generate

freq [string or DateOffset, default 'D'] Frequency strings can have multiples, e.g. '5H'

name [string, default None] Name of the resulting TimedeltaIndex

closed [string, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

Returns

rng [TimedeltaIndex]

Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `TimedeltaIndex` will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see this link.

Examples

```
>>> pd.timedelta_range(start='1 day', periods=4)
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The closed parameter specifies which endpoint is included. The default behavior is to include both endpoints.

```
>>> pd.timedelta_range(start='1 day', periods=4, closed='right')
TimedeltaIndex(['2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq='D')
```

The freq parameter specifies the frequency of the TimedeltaIndex. Only fixed frequencies can be passed, non-fixed frequencies such as 'M' (month end) will raise.

```
>>> pd.timedelta_range(start='1 day', end='2 days', freq='6H')
TimedeltaIndex(['1 days 00:00:00', '1 days 06:00:00', '1 days 12:00:00',
                '1 days 18:00:00', '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='6H')
```

Specify start, end, and periods; the frequency is generated automatically (linearly spaced).

```
>>> pd.timedelta_range(start='1 day', end='5 days', periods=4)
TimedeltaIndex(['1 days 00:00:00', '2 days 08:00:00', '3 days 16:00:00',
            '5 days 00:00:00'],
           dtype='timedelta64[ns]', freq=None)
```

pandas.infer_freq

pandas.**infer_freq**(*index*, *warn=True*)

Infer the most likely frequency given the input index. If the frequency is uncertain, a warning will be printed.

Parameters

index [DatetimeIndex or TimedeltaIndex] if passed a Series will use the values of the series (NOT THE INDEX)

warn [boolean, default True]

Returns

freq [string or None] None if no discernible frequency TypeError if the index is not datetime-like ValueError if there are less than three values.

6.2.5 Top-level dealing with intervals

interval_range([start, end, periods, freq, …])	Return a fixed frequency IntervalIndex

pandas.interval_range

pandas.**interval_range**(*start=None, end=None, periods=None, freq=None, name=None, closed='right'*)

Return a fixed frequency IntervalIndex

Parameters

start [numeric or datetime-like, default None] Left bound for generating intervals

end [numeric or datetime-like, default None] Right bound for generating intervals

periods [integer, default None] Number of periods to generate

freq [numeric, string, or DateOffset, default None] The length of each interval. Must be consistent with the type of start and end, e.g. 2 for numeric, or '5H' for datetime-like. Default is 1 for numeric and 'D' for datetime-like.

name [string, default None] Name of the resulting IntervalIndex

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

rng [IntervalIndex]

See also:

IntervalIndex An Index of intervals that are all closed on the same side.

Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `IntervalIndex` will have `periods` linearly spaced elements between `start` and `end`, inclusively.

To learn more about datetime-like frequency strings, please see this link.

Examples

Numeric `start` and `end` is supported.

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right', dtype='interval[int64]')
```

Additionally, datetime-like input is also supported.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
...                   end=pd.Timestamp('2017-01-04'))
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03],
               (2017-01-03, 2017-01-04]],
              closed='right', dtype='interval[datetime64[ns]]')
```

The `freq` parameter specifies the frequency between the left and right. endpoints of the individual intervals within the `IntervalIndex`. For numeric `start` and `end`, the frequency must also be numeric.

```
>>> pd.interval_range(start=0, periods=4, freq=1.5)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
              closed='right', dtype='interval[float64]')
```

Similarly, for datetime-like `start` and `end`, the frequency must be convertible to a DateOffset.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
...                   periods=3, freq='MS')
IntervalIndex([(2017-01-01, 2017-02-01], (2017-02-01, 2017-03-01],
               (2017-03-01, 2017-04-01]],
              closed='right', dtype='interval[datetime64[ns]]')
```

Specify `start`, `end`, and `periods`; the frequency is generated automatically (linearly spaced).

```
>>> pd.interval_range(start=0, end=6, periods=4)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]],
          closed='right',
          dtype='interval[float64]')
```

The `closed` parameter specifies which endpoints of the individual intervals within the `IntervalIndex` are closed.

```
>>> pd.interval_range(end=5, periods=4, closed='both')
IntervalIndex([[1, 2], [2, 3], [3, 4], [4, 5]],
              closed='both', dtype='interval[int64]')
```

6.2.6 Top-level evaluation

eval(expr[, parser, engine, truediv, . . .])	Evaluate a Python expression as a string using various backends.

pandas.eval

`pandas.eval`(*expr*, *parser='pandas'*, *engine=None*, *truediv=True*, *local_dict=None*, *global_dict=None*, *resolvers=()*, *level=0*, *target=None*, *inplace=False*)

Evaluate a Python expression as a string using various backends.

The following arithmetic operations are supported: `+`, `-`, `*`, `/`, `**`, `%`, `//` (python engine only) along with the following boolean operations: `|` (or), `&` (and), and `~` (not). Additionally, the `'pandas'` parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators. *Series* and *DataFrame* objects are supported and behave as they would with plain ol' Python evaluation.

Parameters

expr [str or unicode] The expression to evaluate. This string cannot contain any Python statements, only Python expressions.

parser [string, default 'pandas', {'pandas', 'python'}] The parser to use to construct the syntax tree from the expression. The default of `'pandas'` parses code slightly different than standard Python. Alternatively, you can parse an expression using the `'python'` parser to retain strict Python semantics. See the *enhancing performance* documentation for more details.

engine [string or None, default 'numexpr', {'python', 'numexpr'}] The engine used to evaluate the expression. Supported engines are

- None : tries to use `numexpr`, falls back to `python`
- **`'numexpr'`: This default engine evaluates pandas objects using** numexpr for large speed ups in complex expressions with large frames.
- **`'python'`: Performs operations as if you had `eval`'d in top** level python. This engine is generally not that useful.

More backends may be available in the future.

truediv [bool, optional] Whether to use true division, like in Python >= 3

local_dict [dict or None, optional] A dictionary of local variables, taken from locals() by default.

global_dict [dict or None, optional] A dictionary of global variables, taken from globals() by default.

resolvers [list of dict-like or None, optional] A list of objects implementing the `__getitem__` special method that you can use to inject an additional collection of namespaces to use for variable lookup. For example, this is used in the *`query()`* method to inject the `DataFrame.index` and `DataFrame.columns` variables that refer to their respective *`DataFrame`* instance attributes.

level [int, optional] The number of prior stack frames to traverse and add to the current scope. Most users will **not** need to change this parameter.

target [object, optional, default None] This is the target object for assignment. It is used when there is variable assignment in the expression. If so, then *target* must support item assignment with string keys, and if a copy is being returned, it must also support *.copy()*.

inplace [bool, default False] If *target* is provided, and the expression mutates *target*, whether to modify *target* inplace. Otherwise, return a copy of *target* with the mutation.

Returns

ndarray, numeric scalar, DataFrame, Series

Raises

ValueError There are many instances where such an error can be raised:

- *target=None*, but the expression is multiline.
- The expression is multiline, but not all them have item assignment. An example of such an arrangement is this:

  a = b + 1 a + 2

  Here, there are expressions on different lines, making it multiline, but the last line has no variable assigned to the output of *a + 2*.
- *inplace=True*, but the expression is missing item assignment.
- Item assignment is provided, but the *target* does not support string item assignment.
- Item assignment is provided and *inplace=False*, but the *target* does not support the *.copy()* method

See also:

`pandas.DataFrame.query`, *`pandas.DataFrame.eval`*

Notes

The `dtype` of any objects involved in an arithmetic `%` operation are recursively cast to `float64`.

See the *enhancing performance* documentation for more details.

6.2.7 Hashing

util.hash_array(vals[, encoding, hash_key, . . .])	Given a 1d array, return an array of deterministic integers.
util.hash_pandas_object(obj[, index, . . .])	Return a data hash of the Index/Series/DataFrame

pandas.util.hash_array

pandas.util.**hash_array**(*vals*, *encoding='utf8'*, *hash_key=None*, *categorize=True*)

Given a 1d array, return an array of deterministic integers.

New in version 0.19.2.

Parameters

vals [ndarray, Categorical]

encoding [string, default 'utf8'] encoding for data & key when strings

hash_key [string key to encode, default to _default_hash_key]

categorize [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

New in version 0.20.0.

Returns

1d uint64 numpy array of hash values, same length as the vals

pandas.util.hash_pandas_object

pandas.util.**hash_pandas_object**(*obj*, *index=True*, *encoding='utf8'*, *hash_key=None*, *categorize=True*)

Return a data hash of the Index/Series/DataFrame

New in version 0.19.2.

Parameters

index [boolean, default True] include the index in the hash (if Series/DataFrame)

encoding [string, default 'utf8'] encoding for data & key when strings

hash_key [string key to encode, default to _default_hash_key]

categorize [bool, default True] Whether to first categorize object arrays before hashing. This is more efficient when the array contains duplicate values.

New in version 0.20.0.

Returns

Series of uint64, same length as the object

6.2.8 Testing

test([extra_args])	

pandas.test

pandas.**test**(*extra_args=None*)

6.3 Series

6.3.1 Constructor

Series([data, index, dtype, name, copy, . . .])	One-dimensional ndarray with axis labels (including time series).

pandas.Series

class pandas.**Series**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*)

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, *, *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

Parameters

data [array-like, Iterable, dict, or scalar value] Contains data stored in Series.

Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index [array-like or Index (1d)] Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, . . . , n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

dtype [str, numpy.dtype, or ExtensionDtype, optional] dtype for the output Series. If not specified, this will be inferred from *data*. See the *user guide* for more usages.

copy [bool, default False] Copy input data.

Attributes

T	Return the transpose, which is by definition self.
array	The ExtensionArray of the data backing this Series or Index.
asobject	Return object Series which contains boxed values.

Continued on next page

Table 26 – continued from previous page

at	Access a single value for a row/column label pair.
axes	Return a list of the row axis labels.
base	Return the base object if the memory of the underlying data is shared.
blocks	(DEPRECATED) Internal property, property synonym for as_blocks().
data	Return the data pointer of the underlying data.
dtype	Return the dtype object of the underlying data.
dtypes	Return the dtype object of the underlying data.
flags	
ftype	Return if the data is sparseldense.
ftypes	Return if the data is sparseldense.
hasnans	Return if I have any nans; enables various perf speedups.
iat	Access a single value for a row/column pair by integer position.
iloc	Purely integer-location based indexing for selection by position.
imag	Return imag value of vector.
index	The index (axis labels) of the Series.
is_copy	Return the copy.
is_monotonic	Return boolean if values in the object are monotonic_increasing.
is_monotonic_decreasing	Return boolean if values in the object are monotonic_decreasing.
is_monotonic_increasing	Return boolean if values in the object are monotonic_increasing.
is_unique	Return boolean if values in the object are unique.
itemsize	Return the size of the dtype of the item of the underlying data.
ix	A primarily label-location based indexer, with integer position fallback.
loc	Access a group of rows and columns by label(s) or a boolean array.
name	Return name of the Series.
nbytes	Return the number of bytes in the underlying data.
ndim	Number of dimensions of the underlying data, by definition 1.
real	Return the real value of vector.
shape	Return a tuple of the shape of the underlying data.
size	Return the number of elements in the underlying data.
strides	Return the strides of the underlying data.
values	Return Series as ndarray or ndarray-like depending on the dtype.

pandas.Series.T

`Series.`**`T`**

Return the transpose, which is by definition self.

pandas.Series.array

`Series.array`

The ExtensionArray of the data backing this Series or Index.

New in version 0.24.0.

Returns

array [ExtensionArray] An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around `numpy.ndarray`.

`.array` differs `.values` which may require converting the data to a different form.

See also:

Index.to_numpy Similar method that always returns a NumPy array.

Series.to_numpy Similar method that always returns a NumPy array.

Notes

This table lays out the different array types for each extension dtype within pandas.

dtype	array type
category	Categorical
period	PeriodArray
interval	IntervalArray
IntegerNA	IntegerArray
datetime64[ns, tz]	DatetimeArray

For any 3rd-party extension types, the array type will be an ExtensionArray.

For all remaining dtypes `.array` will be a `arrays.NumpyExtensionArray` wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use *`Series.to_numpy()`* instead.

Examples

For regular NumPy types like int, and float, a PandasArray is returned.

```
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
[a, b, a]
Categories (2, object): [a, b]
```

pandas.Series.asobject

`Series.`**`asobject`**

Return object Series which contains boxed values.

Deprecated since version 0.23.0: Use `astype(object)` instead.

this is an internal non-public method

pandas.Series.at

`Series.`**`at`**

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

`DataFrame.iat` Access a single value for a row/column pair by integer position.

`DataFrame.loc` Access a group of rows and columns by label(s).

`Series.at` Access a single value using a label.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

pandas.Series.axes

`Series.axes`
Return a list of the row axis labels.

pandas.Series.base

`Series.base`
Return the base object if the memory of the underlying data is shared.

pandas.Series.blocks

`Series.blocks`
Internal property, property synonym for as_blocks().

Deprecated since version 0.21.0.

pandas.Series.data

`Series.data`
Return the data pointer of the underlying data.

pandas.Series.dtype

`Series.dtype`
Return the dtype object of the underlying data.

pandas.Series.dtypes

`Series.dtypes`
Return the dtype object of the underlying data.

pandas.Series.flags

`Series.flags`

pandas.Series.ftype

`Series.ftype`
Return if the data is sparseldense.

pandas.Series.ftypes

`Series.ftypes`
Return if the data is sparseldense.

pandas.Series.hasnans

Series.**hasnans**
Return if I have any nans; enables various perf speedups.

pandas.Series.iat

Series.**iat**
Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.loc Access a group of rows and columns by label(s).

DataFrame.iloc Access a group of rows and columns by integer position(s).

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

pandas.Series.iloc

Series.**iloc**
Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at ref:*Selection by Position <indexing.integer>*.

See also:

`DataFrame.iat` Fast integer location scalar accessor.

`DataFrame.loc` Purely label-location based indexer for selection by label.

`Series.iloc` Purely integer-location based indexing for selection by position.

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
```

(continues on next page)

(continued from previous page)

```
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a *slice* object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

pandas.Series.imag

Series.**imag**
Return imag value of vector.

pandas.Series.index

Series.**index**
The index (axis labels) of the Series.

pandas.Series.is_copy

Series.**is_copy**
Return the copy.

pandas.Series.is_monotonic

Series.**is_monotonic**
Return boolean if values in the object are monotonic_increasing.

New in version 0.19.0.

Returns

is_monotonic [boolean]

pandas.Series.is_monotonic_decreasing

Series.**is_monotonic_decreasing**
Return boolean if values in the object are monotonic_decreasing.

New in version 0.19.0.

Returns

is_monotonic_decreasing [boolean]

pandas.Series.is_monotonic_increasing

`Series.is_monotonic_increasing`

Return boolean if values in the object are monotonic_increasing.

New in version 0.19.0.

Returns

is_monotonic [boolean]

pandas.Series.is_unique

`Series.is_unique`

Return boolean if values in the object are unique.

Returns

is_unique [boolean]

pandas.Series.itemsize

`Series.itemsize`

Return the size of the dtype of the item of the underlying data.

pandas.Series.ix

`Series.ix`

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at *Advanced Indexing*.

pandas.Series.loc

`Series.loc`

Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.

- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at *Selection by Label*

Raises

KeyError: when any items are not found

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.iloc Access group of rows and columns by integer position(s).

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels.

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=['cobra', 'viper', 'sidewinder'],
...      columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
```

(continues on next page)

(continued from previous page)

```
viper                30       50
sidewinder           30       50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...    ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...         [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

pandas.Series.name

`Series.name`
Return name of the Series.

pandas.Series.nbytes

`Series.nbytes`
Return the number of bytes in the underlying data.

pandas.Series.ndim

`Series.ndim`
Number of dimensions of the underlying data, by definition 1.

pandas.Series.real

`Series.real`
Return the real value of vector.

pandas.Series.shape

`Series.shape`
Return a tuple of the shape of the underlying data.

pandas.Series.size

`Series.size`
Return the number of elements in the underlying data.

pandas.Series.strides

`Series.strides`
Return the strides of the underlying data.

pandas.Series.values

`Series.values`
Return Series as ndarray or ndarray-like depending on the dtype.

> **Warning:** We recommend using *Series.array* or *Series.to_numpy()*, depending on whether you need a reference to the underlying data or a NumPy array.

Returns

arr [numpy.ndarray or ndarray-like]

See also:

Series.array Reference to the underlying data.

Series.to_numpy A NumPy array representing the underlying data.

Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
[a, a, b, c]
Categories (3, object): [a, b, c]
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
...                         tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

empty	
timetuple	

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	Suffix labels with string *suffix*.
agg(func[, axis])	Aggregate using one or more operations over the specified axis.
aggregate(func[, axis])	Aggregate using one or more operations over the specified axis.
align(other[, join, axis, level, copy, . . .])	Align two objects on their axes with the specified join method for each axis Index.
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
append(to_append[, ignore_index, . . .])	Concatenate two or more Series.
apply(func[, convert_dtype, args])	Invoke function on values of Series.
argmax([axis, skipna])	(DEPRECATED) Return the row label of the maximum value.
argmin([axis, skipna])	(DEPRECATED) Return the row label of the minimum value.
argsort([axis, kind, order])	Overrides ndarray.argsort.

Continued on next page

Table 27 – continued from previous page

as_blocks([copy])	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	(DEPRECATED) Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, . . .])	Convert TimeSeries to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g.
autocorr([lag])	Compute the lag-N autocorrelation.
between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
between_time(start_time, end_time[, . . .])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for *DataFrame.fillna()* with `method='bfill'`.
bool()	Return the bool of a single element PandasObject.
cat	alias of `pandas.core.arrays.categorical.CategoricalAccessor`
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.
clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.
combine(other, func[, fill_value])	Combine the Series with a Series or scalar according to *func*.
combine_first(other)	Combine Series values, choosing the calling Series's values first.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis.
compress(condition, *args, **kwargs)	(DEPRECATED) Return selected slices of an array along given axis as a Series.
convert_objects([convert_dates, . . .])	(DEPRECATED) Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.
corr(other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values.
count([level])	Return number of non-NA/null observations in the Series.
cov(other[, min_periods])	Compute covariance with Series, excluding missing values.
cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.

Continued on next page

Table 27 – continued from previous page

describe([percentiles, include, exclude])	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
diff([periods])	First discrete difference of element.
div(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divide(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divmod(other[, level, fill_value, axis])	Integer division and modulo of series and other, element-wise (binary operator *divmod*).
dot(other)	Compute the dot product between the Series and the columns of other.
drop([labels, axis, index, columns, level, ...])	Return Series with specified index labels removed.
drop_duplicates([keep, inplace])	Return Series with duplicate values removed.
droplevel(level[, axis])	Return DataFrame with requested index / column level(s) removed.
dropna([axis, inplace])	Return a new Series with missing values removed.
dt	alias of pandas.core.indexes.accessors.CombinedDatetimelikeProperties
duplicated([keep])	Indicate duplicate Series values.
eq(other[, level, fill_value, axis])	Equal to of series and other, element-wise (binary operator *eq*).
equals(other)	Test whether two objects contain the same elements.
ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions.
expanding([min_periods, center, axis])	Provides expanding transformations.
factorize([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
ffill([axis, inplace, limit, downcast])	Synonym for *DataFrame.fillna()* with method='ffill'.
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
from_array(arr[, index, name, dtype, copy, ...])	Construct Series from array.
from_csv(path[, sep, parse_dates, header, ...])	(DEPRECATED) Read CSV file.
ge(other[, level, fill_value, axis])	Greater than or equal to of series and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	(DEPRECATED) Return counts of unique ftypes in this object.
get_value(label[, takeable])	(DEPRECATED) Quickly retrieve single value at passed index label.
get_values()	Same as values (but handles sparseness conversions); is a view.

Continued on next page

Table 27 – continued from previous page

groupby([by, axis, level, as_index, sort, . . .])	Group DataFrame or Series using a mapper or by a Series of columns.
gt(other[, level, fill_value, axis])	Greater than of series and other, element-wise (binary operator *gt*).
head([n])	Return the first *n* rows.
hist([by, ax, grid, xlabelsize, xrot, . . .])	Draw histogram of the input series using matplotlib.
idxmax([axis, skipna])	Return the row label of the maximum value.
idxmin([axis, skipna])	Return the row label of the minimum value.
infer_objects()	Attempt to infer better dtypes for object columns.
interpolate([method, axis, limit, inplace, . . .])	Interpolate values according to different methods.
isin(values)	Check whether *values* are contained in Series.
isna()	Detect missing values.
isnull()	Detect missing values.
item()	Return the first element of the underlying data as a python scalar.
items()	Lazily iterate over (index, value) tuples.
iteritems()	Lazily iterate over (index, value) tuples.
keys()	Alias for index.
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, level, fill_value, axis])	Less than or equal to of series and other, element-wise (binary operator *le*).
lt(other[, level, fill_value, axis])	Less than of series and other, element-wise (binary operator *lt*).
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
map(arg[, na_action])	Map values of Series according to input correspondence.
mask(cond[, other, inplace, axis, level, . . .])	Replace values where the condition is True.
max([axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis.
memory_usage([index, deep])	Return the memory usage of the Series.
min([axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
mod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mode([dropna])	Return the mode(s) of the dataset.
mul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).

Continued on next page

Table 27 – continued from previous page

ne(other[, level, fill_value, axis])	Not equal to of series and other, element-wise (binary operator *ne*).
nlargest([n, keep])	Return the largest *n* elements.
nonzero()	(DEPRECATED) Return the *integer* indices of the elements that are non-zero.
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
nsmallest([n, keep])	Return the smallest *n* elements.
nunique([dropna])	Return number of unique elements in the object.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs).
plot	alias of `pandas.plotting._core.SeriesPlotMethods`
pop(item)	Return item and drop from frame.
pow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, . . .])	Return the product of the values for the requested axis.
product([axis, skipna, level, numeric_only, . . .])	Return the product of the values for the requested axis.
ptp([axis, skipna, level, numeric_only])	(DEPRECATED) Returns the difference between the maximum value and the
put(*args, **kwargs)	Applies the *put* method to its *values* attribute if it has one.
quantile([q, interpolation])	Return value at the given quantile.
radd(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, . . .])	Compute numerical data ranks (1 through n) along axis.
ravel([order])	Return the flattened underlying data as an ndarray.
rdiv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
rdivmod(other[, level, fill_value, axis])	Integer division and modulo of series and other, element-wise (binary operator *rdivmod*).
reindex([index])	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis])	(DEPRECATED) Conform Series to new index with optional filling logic.
reindex_like(other[, method, copy, limit, . . .])	Return an object with matching indices as other object.
rename([index])	Alter Series index labels or name.
rename_axis([mapper, index, columns, axis, . . .])	Set the name of the axis for the index or columns.
reorder_levels(order)	Rearrange index levels using input order.
repeat(repeats[, axis])	Repeat elements of a Series.
replace([to_replace, value, inplace, limit, . . .])	Replace values given in *to_replace* with *value*.
resample(rule[, how, axis, fill_method, . . .])	Resample time-series data.
reset_index([level, drop, name, inplace])	Generate a new DataFrame or Series with the index reset.

Continued on next page

Table 27 – continued from previous page

rfloordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, center, . . .])	Provides rolling window calculations.
round([decimals])	Round each value in a Series to the given number of decimals.
rpow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, . . .])	Return a random sample of items from an axis of object.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.
set_value(label, value[, takeable])	(DEPRECATED) Quickly set single value at passed label.
shift([periods, freq, axis, fill_value])	Shift index by desired number of periods with an optional time *freq*.
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1.
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, . . .])	Sort Series by index labels.
sort_values([axis, ascending, inplace, . . .])	Sort by the values.
`sparse`	alias of `pandas.core.arrays.sparse.SparseAccessor`
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
str	alias of `pandas.core.strings.StringMethods`
sub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, . . .])	Return the sum of the values for the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, copy])	Swap levels i and j in a MultiIndex.
tail([n])	Return the last *n* rows.
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.

Continued on next page

Table 27 – continued from previous page

to_csv(*args, **kwargs)	Write object to a comma-separated values (csv) file.
to_dense()	Return dense representation of NDFrame (as opposed to sparse).
to_dict([into])	Convert Series to {label -> value} dict or dict-like object.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write object to an Excel sheet.
to_frame([name])	Convert Series to DataFrame.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, ...])	Render an object to a LaTeX tabular environment table.
to_list()	Return a list of the values.
to_msgpack([path_or_buf, encoding])	Serialize object to input file path using msgpack format.
to_numpy([dtype, copy])	A NumPy ndarray representing the values in this Series or Index.
to_period([freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_sparse([kind, fill_value])	Convert Series to SparseSeries.
to_sql(name, con[, schema, if_exists, ...])	Write records stored in a DataFrame to a SQL database.
to_string([buf, na_rep, float_format, ...])	Render a string representation of the Series.
to_timestamp([freq, how, copy])	Cast to datetimeindex of timestamps, at *beginning* of period.
to_xarray()	Return an xarray object from the pandas object.
tolist()	Return a list of the values.
transform(func[, axis])	Call `func` on self producing a Series with transformed values and that has the same axis length as self.
transpose(*args, **kwargs)	Return the transpose, which is by definition self.
truediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
unique()	Return unique values of Series object.
unstack([level, fill_value])	Unstack, a.k.a.
update(other)	Modify Series in place using non-NA values from passed Series.
valid([inplace])	(DEPRECATED) Return Series without null values.
value_counts([normalize, sort, ascending, ...])	Return a Series containing counts of unique values.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
view([dtype])	Create a new view of the Series.

Continued on next page

Table 27 – continued from previous page

where(cond[, other, inplace, axis, level, . . .])	Replace values where the condition is False.
xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

pandas.Series.abs

`Series.abs()`

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
   a   b    c
0  4  10  100
1  5  20   50
2  6  30  -30
3  7  40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
   a   b    c
1  5  20   50
0  4  10  100
2  6  30  -30
3  7  40  -50
```

pandas.Series.add

`Series.add`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Addition of series and other, element-wise (binary operator *add*).

Equivalent to `series + other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.radd

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.add_prefix

`Series.add_prefix(prefix)`

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix [str] The string to add before each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_suffix Suffix row labels with string *suffix*.

DataFrame.add_suffix Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0        1      3
1        2      4
2        3      5
3        4      6
```

pandas.Series.add_suffix

Series.**add_suffix**(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix [str] The string to add after each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_prefix Prefix row labels with string *prefix*.

DataFrame.add_prefix Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

pandas.Series.agg

`Series.agg`(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index'}] Parameter needed for compatibility with DataFrame.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

Series.apply Invoke function on a Series.

Series.transform Transform function producing a Series with like indexes.

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max'])
min    1
max    4
dtype: int64
```

pandas.Series.aggregate

`Series.aggregate`(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index'}] Parameter needed for compatibility with DataFrame.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

Series.apply Invoke function on a Series.

Series.transform Transform function producing a Series with like indexes.

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max'])
min   1
max   4
dtype: int64
```

pandas.Series.align

`Series.align`(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index.

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

axis [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

fill_axis [{0 or 'index'}, default 0] Filling axis, method and limit

broadcast_axis [{0 or 'index'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

Returns

(left, right) [(Series, type of other)] Aligned objects

pandas.Series.all

`Series.all`(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

scalar or Series If level is specified, then, Series is returned; otherwise, scalar is returned.

See also:

Series.all Return True if all elements are True.

DataFrame.any Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([]).all()
True
```

(continues on next page)

(continued from previous page)

```
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

pandas.Series.any

`Series.any`(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether any element is True, potentially over an axis.

Returns False unless there at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

scalar or Series If level is specified, then, Series is returned; otherwise, scalar is returned.

See also:

`numpy.any` Numpy version of this method.

Series.any Return whether any element is True.

Series.all Return whether all elements are True.

DataFrame.any Return whether any element is True over requested axis.

DataFrame.all Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

pandas.Series.append

`Series.`**append**(*to_append*, *ignore_index=False*, *verify_integrity=False*)

Concatenate two or more Series.

Parameters

to_append [Series or list/tuple of Series]

ignore_index [boolean, default False] If True, do not use the index labels.

New in version 0.19.0.

verify_integrity [boolean, default False] If True, raise Exception on creating index with duplicates

Returns

appended [Series]

See also:

concat General function to concatenate DataFrame, Series or Panel objects.

Notes

Iteratively appending to a Series can be more computationally intensive than a single concatenate. A better solution is to append values to a list and then concatenate the list with the original Series all at once.

Examples

```
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *ignore_index* set to True:

```
>>> s1.append(s2, ignore_index=True)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *verify_integrity* set to True:

```
>>> s1.append(s2, verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: [0, 1, 2]
```

pandas.Series.apply

`Series.`**`apply`** (*func*, *convert_dtype=True*, *args=()*, ***kwds*)
Invoke function on values of Series.

Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values.

Parameters

func [function] Python function or NumPy ufunc to apply.

convert_dtype [bool, default True] Try to find better dtype for elementwise function results. If False, leave as dtype=object.

args [tuple] Positional arguments passed to func after the series value.

****kwds** Additional keyword arguments passed to func.

Returns

Series or DataFrame If func returns a Series object the result will be a DataFrame.

See also:

Series.map For element-wise operations.

Series.agg Only perform aggregating type operations.

Series.transform Only perform transforming type operations.

Examples

Create a series with typical summer temperatures for each city.

```
>>> s = pd.Series([20, 21, 12],
...               index=['London', 'New York', 'Helsinki'])
>>> s
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x):
...     return x ** 2
>>> s.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> s.apply(lambda x: x ** 2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x - custom_value
```

```
>>> s.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x += kwargs[month]
...     return x
```

```
>>> s.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> s.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

pandas.Series.argmax

`Series.`**`argmax`**(*axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the maximum value.

Deprecated since version 0.21.0.

The current behaviour of 'Series.argmax' is deprecated, use 'idxmax' instead. The behavior of 'argmax' will be corrected to return the positional maximum in the future. For now, use 'series.values.argmax' or 'np.argmax(np.array(values))' to get the position of the maximum row.

If multiple values equal the maximum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

idxmax [Index of maximum of values.]

Raises

ValueError If the Series is empty.

See also:

numpy.argmax Return indices of the maximum values along the given axis.

DataFrame.idxmax Return index of first occurrence of maximum over requested axis.

Series.idxmin Return index *label* of the first occurrence of minimum of values.

Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

pandas.Series.argmin

Series.**argmin**(*axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the minimum value.

Deprecated since version 0.21.0.

The current behaviour of 'Series.argmin' is deprecated, use 'idxmin' instead. The behavior of 'argmin' will be corrected to return the positional minimum in the future. For now, use 'series.values.argmin' or 'np.argmin(np.array(values))' to get the position of the minimum row.

If multiple values equal the minimum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

idxmin [Index of minimum of values.]

Raises

ValueError If the Series is empty.

See also:

`numpy.argmin` Return indices of the minimum values along the given axis.

DataFrame.idxmin Return index of first occurrence of minimum over requested axis.

Series.idxmax Return index *label* of the first occurrence of maximum of values.

Notes

This method is the Series version of `ndarray.argmin`. This method returns the label of the minimum, while `ndarray.argmin` returns the position. To get the position, use `series.values.argmin()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...               index=['A' ,'B' ,'C' ,'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmin(skipna=False)
nan
```

pandas.Series.argsort

`Series.`**`argsort`**(*axis=0*, *kind='quicksort'*, *order=None*)

Overrides ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values.

Parameters

axis [int] Has no effect but is accepted for compatibility with numpy.

kind [{'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See np.sort for more information. 'mergesort' is the only stable algorithm

order [None] Has no effect but is accepted for compatibility with numpy.

Returns

argsorted [Series, with -1 indicated where nan values are present]

See also:

`numpy.ndarray.argsort`

pandas.Series.as_blocks

`Series.as_blocks(copy=True)`

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

pandas.Series.as_matrix

`Series.as_matrix(columns=None)`

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

Parameters

columns [list, optional, default:None] If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

`DataFrame.values`

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.Series.asfreq

Series.**asfreq**(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, or string]

method [{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

Returns

converted [same type as caller]

See also:

reindex

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

pandas.Series.asof

`Series.`**`asof`**(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a *`DataFrame`*, the last row without NaN considering only the subset of columns (if not *None*)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where [date or array-like of dates] Date(s) before which the last row(s) are returned.

subset [str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame

- scalar : when *self* is a Series and *where* is a scalar

- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                      a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
                       a   b
2018-02-27 09:03:30   30.0 NaN
2018-02-27 09:04:30   40.0 NaN
```

pandas.Series.astype

Series.**astype**(*dtype*, *copy=True*, *errors='raise'*, ***kwargs*)
Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, …}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [same type as caller]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> cat_dtype = pd.api.types.CategoricalDtype(
...                     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.Series.at_time

`Series.at_time`(*time*, *asof=False*, *axis=None*)

Select values at particular time of day (e.g. 9:30AM).

Parameters

time [datetime.time or string]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_at_time [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

between_time Select values between particular times of the day.

first Select initial periods of time series based on a date offset.

last Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

pandas.Series.autocorr

Series.**autocorr**(*lag=1*)

Compute the lag-N autocorrelation.

This method computes the Pearson correlation between the Series and its shifted self.

Parameters

lag [int, default 1] Number of lags to apply before performing autocorrelation.

Returns

float The Pearson correlation between self and self.shift(lag).

See also:

Series.corr Compute the correlation between two Series.

Series.shift Shift index by desired number of periods.

DataFrame.corr Compute pairwise correlation of columns.

DataFrame.corrwith Compute pairwise correlation between rows or columns of two DataFrame objects.

Notes

If the Pearson correlation is not well defined return 'NaN'.

Examples

```
>>> s = pd.Series([0.25, 0.5, 0.2, -0.05])
>>> s.autocorr()  # doctest: +ELLIPSIS
0.10355...
>>> s.autocorr(lag=2)  # doctest: +ELLIPSIS
-0.99999...
```

If the Pearson correlation is not well defined, then 'NaN' is returned.

```
>>> s = pd.Series([1, 0, 0, 0])
>>> s.autocorr()
nan
```

pandas.Series.between

`Series.between(left, right, inclusive=True)`

Return boolean Series equivalent to left <= series <= right.

This function returns a boolean vector containing *True* wherever the corresponding Series element is between the boundary values *left* and *right*. NA values are treated as *False*.

Parameters

left [scalar] Left boundary.

right [scalar] Right boundary.

inclusive [bool, default True] Include boundaries.

Returns

Series Each element will be a boolean.

See also:

Series.gt Greater than of series and other.

Series.lt Less than of series and other.

Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

Examples

```
>>> s = pd.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

With *inclusive* set to `False` boundary values are excluded:

```
>>> s.between(1, 4, inclusive=False)
0     True
1    False
2    False
3    False
```

(continues on next page)

(continued from previous page)

```
4    False
dtype: bool
```

left and *right* can be any scalar value:

```
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0    False
1     True
2     True
3    False
dtype: bool
```

pandas.Series.between_time

`Series.`**`between_time`**(*start_time*, *end_time*, *include_start=True*, *include_end=True*, *axis=None*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time [datetime.time or string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_between_time [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

`at_time` Select values at a particular time of the day.

`first` Select initial periods of time series based on a date offset.

`last` Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_between_time` Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
```

(continues on next page)

(continued from previous page)

```
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

pandas.Series.bfill

`Series.bfill(axis=None, inplace=False, limit=None, downcast=None)`
Synonym for `DataFrame.fillna()` with `method='bfill'`.

pandas.Series.bool

`Series.bool()`
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

pandas.Series.cat

`Series.cat()`
Accessor object for categorical properties of the Series values.

Be aware that assigning to *categories* is a inplace operation, while all methods return new categorical data per default (but can be called with *inplace=True*).

Parameters

data [Series or CategoricalIndex]

Examples

```
>>> s.cat.categories
>>> s.cat.categories = list('abc')
>>> s.cat.rename_categories(list('cab'))
>>> s.cat.reorder_categories(list('cab'))
>>> s.cat.add_categories(['d','e'])
>>> s.cat.remove_categories(['d'])
>>> s.cat.remove_unused_categories()
```

(continues on next page)

(continued from previous page)

```
>>> s.cat.set_categories(list('abcde'))
>>> s.cat.as_ordered()
>>> s.cat.as_unordered()
```

pandas.Series.clip

`Series.clip`(*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

pandas.Series.clip_lower

`Series.`**`clip_lower`**(*threshold*, *axis=None*, *inplace=False*)

Trim values below a given threshold.

Deprecated since version 0.24.0: Use clip(lower=threshold) instead.

Elements below the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

`Series.clip` General purpose method to trim Series values to given threshold(s).

`DataFrame.clip` General purpose method to trim DataFrame values to given threshold(s).

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip(lower=8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip(lower=elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip(lower=3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip(lower=[3, 3, 5], axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip(lower=[4, 5], axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

pandas.Series.clip_upper

Series.**clip_upper**(*threshold*, *axis=None*, *inplace=False*)

Trim values above a given threshold.

Deprecated since version 0.24.0: Use clip(upper=threshold) instead.

Elements above the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Maximum value allowed. All values above threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align object with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

Series.clip General purpose method to trim Series values to given threshold(s).

DataFrame.clip General purpose method to trim DataFrame values to given threshold(s).

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.clip(upper=3)
0    1
1    2
2    3
3    3
4    3
dtype: int64
```

```
>>> elemwise_thresholds = [5, 4, 3, 2, 1]
>>> elemwise_thresholds
[5, 4, 3, 2, 1]
```

```
>>> s.clip(upper=elemwise_thresholds)
0    1
1    2
2    3
3    2
4    1
dtype: int64
```

pandas.Series.combine

Series.**combine**(*other*, *func*, *fill_value=None*)

Combine the Series with a Series or scalar according to *func*.

Combine the Series and *other* using *func* to perform elementwise selection for combined Series. *fill_value* is assumed when value is missing at some index from one of the two objects being combined.

Parameters

other [Series or scalar] The value(s) to be combined with the *Series*.

func [function] Function that takes two scalars as inputs and returns an element.

fill_value [scalar, optional] The value to assume when an index is missing from one Series or the other. The default specifies to use the appropriate NaN value for the underlying dtype of the Series.

Returns

Series The result of combining the Series with the other object.

See also:

Series.combine_first Combine Series values, choosing the calling Series' values first.

Examples

Consider 2 Datasets s1 and s2 containing highest clocked speeds of different birds.

```
>>> s1 = pd.Series({'falcon': 330.0, 'eagle': 160.0})
>>> s1
falcon    330.0
eagle     160.0
dtype: float64
>>> s2 = pd.Series({'falcon': 345.0, 'eagle': 200.0, 'duck': 30.0})
```

(continues on next page)

(continued from previous page)

```
>>> s2
falcon    345.0
eagle     200.0
duck       30.0
dtype: float64
```

Now, to combine the two datasets and view the highest speeds of the birds across the two datasets

```
>>> s1.combine(s2, max)
duck        NaN
eagle     200.0
falcon    345.0
dtype: float64
```

In the previous example, the resulting value for duck is missing, because the maximum of a NaN and a float is a NaN. So, in the example, we set `fill_value=0`, so the maximum value returned will be the value from some dataset.

```
>>> s1.combine(s2, max, fill_value=0)
duck       30.0
eagle     200.0
falcon    345.0
dtype: float64
```

pandas.Series.combine_first

Series.**combine_first**(*other*)

Combine Series values, choosing the calling Series's values first.

Parameters

other [Series] The value(s) to be combined with the *Series*.

Returns

Series The result of combining the Series with the other object.

See also:

Series.combine Perform elementwise operation on two Series using a given function.

Notes

Result index will be the union of the two indexes.

Examples

```
>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4])
>>> s1.combine_first(s2)
0    1.0
1    4.0
dtype: float64
```

pandas.Series.compound

`Series.compound`(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

compounded [scalar or Series (if level specified)]

pandas.Series.compress

`Series.compress`(*condition*, **args*, ***kwargs*)

Return selected slices of an array along given axis as a Series.

Deprecated since version 0.24.0.

See also:

`numpy.ndarray.compress`

pandas.Series.convert_objects

`Series.convert_objects`(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

pandas.Series.copy

`Series.copy(deep=True)`
Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

copy [Series, DataFrame or Panel] Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

pandas.Series.corr

`Series.corr(other, method='pearson', min_periods=None)`

Compute correlation with *other* Series, excluding missing values.

Parameters

other [Series]

method [{'pearson', 'kendall', 'spearman'} or callable]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarray** and returning a float .. versionadded:: 0.24.0

min_periods [int, optional] Minimum number of observations needed to have a valid result

Returns

correlation [float]

Examples

```
>>> histogram_intersection = lambda a, b: np.minimum(a, b
... ).sum().round(decimals=1)
>>> s1 = pd.Series([.2, .0, .6, .2])
>>> s2 = pd.Series([.3, .6, .0, .1])
>>> s1.corr(s2, method=histogram_intersection)
0.3
```

pandas.Series.count

`Series.count(level=None)`

Return number of non-NA/null observations in the Series.

Parameters

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

Returns

nobs [int or Series (if level specified)]

pandas.Series.cov

`Series.cov(other, min_periods=None)`

Compute covariance with Series, excluding missing values.

Parameters

other [Series]

min_periods [int, optional] Minimum number of observations needed to have a valid result

Returns

covariance [float]

Normalized by N-1 (unbiased estimator).

pandas.Series.cummax

`Series.cummax(axis=None, skipna=True, *args, **kwargs)`

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummax [scalar or Series]

See also:

core.window.Expanding.max Similar functionality but ignores `NaN` values.

Series.max Return the maximum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.Series.cummin

`Series.`**`cummin`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummin [scalar or Series]

See also:

core.window.Expanding.min Similar functionality but ignores `NaN` values.

Series.min Return the minimum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.Series.cumprod

`Series.`**`cumprod`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumprod [scalar or Series]

See also:

`core.window.Expanding.prod` Similar functionality but ignores `NaN` values.

`Series.prod` Return the product over Series axis.

`Series.cummax` Return cumulative maximum over Series axis.

`Series.cummin` Return cumulative minimum over Series axis.

`Series.cumsum` Return cumulative sum over Series axis.

`Series.cumprod` Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.Series.cumsum

`Series.cumsum(axis=None, skipna=True, *args, **kwargs)`

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumsum [scalar or Series]

See also:

core.window.Expanding.sum Similar functionality but ignores `NaN` values.

Series.sum Return the sum over Series axis.

Series.cummax Return cumulative maximum over Series axis.

Series.cummin Return cumulative minimum over Series axis.

Series.cumsum Return cumulative sum over Series axis.

Series.cumprod Return cumulative product over Series axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.Series.describe

`Series.`**`describe`**(*percentiles=None*, *include=None*, *exclude=None*)

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.

- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

DataFrame.count Count number of non-NA/null observations.

DataFrame.max Maximum of the values in the object.

DataFrame.min Minimum of the values in the object.

DataFrame.mean Mean of the values.

DataFrame.std Standard deviation of the obersvations.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
```

(continues on next page)

(continued from previous page)

```
top              f
freq             1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

pandas.Series.diff

Series.**diff**(*periods=1*)

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the Series (default is element in previous row).

Parameters

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

Returns

diffed [Series]

See also:

Series.pct_change Percent change over given number of periods.

Series.shift Shift index by desired number of periods with an optional time freq.

DataFrame.diff First discrete difference of object.

Examples

Difference with previous row

```
>>> s = pd.Series([1, 1, 2, 3, 5, 8])
>>> s.diff()
0    NaN
1    0.0
2    1.0
3    1.0
4    2.0
5    3.0
dtype: float64
```

Difference with 3rd previous row

```
>>> s.diff(periods=3)
0    NaN
1    NaN
2    NaN
3    2.0
4    4.0
5    6.0
dtype: float64
```

Difference with following row

```
>>> s.diff(periods=-1)
0    0.0
1   -1.0
2   -1.0
3   -2.0
4   -3.0
5    NaN
dtype: float64
```

pandas.Series.div

`Series.`**`div`**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

`Series.rtruediv`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.divide

`Series.divide`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rtruediv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
```

(continues on next page)

(continued from previous page)

```
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.divmod

`Series.`**`divmod`**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division and modulo of series and other, element-wise (binary operator *divmod*).

Equivalent to `series divmod other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

`Series.rdivmod`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.dot

Series.**dot**(*other*)

Compute the dot product between the Series and the columns of other.

This method computes the dot product between the Series and another one, or the Series and each columns of a DataFrame, or the Series and each columns of an array.

It can also be called using *self @ other* in Python >= 3.5.

Parameters

other [Series, DataFrame or array-like] The other object to compute the dot product with its columns.

Returns

scalar, Series or numpy.ndarray Return the dot product of the Series and other if other is a Series, the Series of the dot product of Series and each rows of other if other is a DataFrame or a numpy.ndarray between the Series and each columns of the numpy array.

See also:

DataFrame.dot Compute the matrix product with the DataFrame.

Series.mul Multiplication of series and other, element-wise.

Notes

The Series and other has to share the same index if other is a Series or a DataFrame.

Examples

```
>>> s = pd.Series([0, 1, 2, 3])
>>> other = pd.Series([-1, 2, -3, 4])
>>> s.dot(other)
8
>>> s @ other
```

(continues on next page)

(continued from previous page)

```
8
>>> df = pd.DataFrame([[0 ,1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(df)
0    24
1    14
dtype: int64
>>> arr = np.array([[0, 1], [-2, 3], [4, -5], [6, 7]])
>>> s.dot(arr)
array([24, 14])
```

pandas.Series.drop

Series.**drop**(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index labels to drop.

axis [0, default 0] Redundant for application on Series.

index, columns [None] Redundant for application on Series, but index can be used instead of labels.

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level for which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

dropped [pandas.Series]

Raises

KeyError If none of the labels are found in the index.

See also:

Series.reindex Return only specified index labels of Series.

Series.dropna Return series without null values.

Series.drop_duplicates Return Series with duplicate values removed.

DataFrame.drop Drop specified labels from rows or columns.

Examples

```
>>> s = pd.Series(data=np.arange(3), index=['A','B','C'])
>>> s
A   0
B   1
C   2
dtype: int64
```

Drop labels B en C

```
>>> s.drop(labels=['B','C'])
A   0
dtype: int64
```

Drop 2nd level label in MultiIndex Series

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = pd.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
lama    speed      45.0
        weight    200.0
        length      1.2
cow     speed      30.0
        weight    250.0
        length      1.5
falcon  speed     320.0
        weight      1.0
        length      0.3
dtype: float64
```

```
>>> s.drop(labels='weight', level=1)
lama    speed      45.0
        length      1.2
cow     speed      30.0
        length      1.5
falcon  speed     320.0
        length      0.3
dtype: float64
```

pandas.Series.drop_duplicates

`Series.drop_duplicates(keep='first', inplace=False)`
Return Series with duplicate values removed.

Parameters

keep [{'first', 'last', `False`}, default 'first']

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

inplace [boolean, default `False`] If `True`, performs operation inplace and returns None.

Returns

deduplicated [Series]

See also:

Index.drop_duplicates Equivalent method on Index.

DataFrame.drop_duplicates Equivalent method on DataFrame.

Series.duplicated Related method on Series, indicating duplicate Series values.

Examples

Generate an Series with duplicated entries.

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
>>> s
0      lama
1       cow
2      lama
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

With the 'keep' parameter, the selection behaviour of duplicated values can be changed. The value 'first' keeps the first occurrence for each set of duplicated entries. The default value of keep is 'first'.

```
>>> s.drop_duplicates()
0      lama
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

The value 'last' for parameter 'keep' keeps the last occurrence for each set of duplicated entries.

```
>>> s.drop_duplicates(keep='last')
1       cow
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

The value `False` for parameter 'keep' discards all sets of duplicated entries. Setting the value of 'inplace' to `True` performs the operation inplace and returns `None`.

```
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

pandas.Series.droplevel

Series.**droplevel**(*level*, *axis=0*)

Return DataFrame with requested index / column level(s) removed.

New in version 0.24.0.

Parameters

level [int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis [{0 or 'index', 1 or 'columns'}, default 0]

Returns

DataFrame.droplevel()

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...    ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

pandas.Series.dropna

`Series.dropna(axis=0, inplace=False, **kwargs)`

Return a new Series with missing values removed.

See the *User Guide* for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or 'index'}, default 0] There is only one axis to drop values from.

inplace [bool, default False] If True, do operation inplace and return None.

****kwargs** Not in use.

Returns

Series Series with NA entries dropped from it.

See also:

Series.isna Indicate missing values.

Series.notna Indicate existing (non-missing) values.

Series.fillna Replace missing values.

DataFrame.dropna Drop rows or columns which contain NA values.

Index.dropna Drop missing indices.

Examples

```
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2    NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Keep the Series with valid entries in the same variable.

```
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```

Empty strings are not considered NA values. `None` is considered an NA value.

```
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0       NaN
```

(continues on next page)

(continued from previous page)

```
1          2
2        NaT
3
4       None
5     I stay
dtype: object
>>> ser.dropna()
1          2
3
5     I stay
dtype: object
```

pandas.Series.dt

`Series.dt()`

Accessor object for datetimelike properties of the Series values.

Examples

```
>>> s.dt.hour
>>> s.dt.second
>>> s.dt.quarter
```

Returns a Series indexed like the original Series. Raises TypeError if the Series does not contain datetimelike values.

pandas.Series.duplicated

`Series.duplicated(keep='first')`

Indicate duplicate Series values.

Duplicated values are indicated as `True` values in the resulting Series. Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.

Parameters

keep [{'first', 'last', False}, default 'first']

- 'first' : Mark duplicates as `True` except for the first occurrence.
- 'last' : Mark duplicates as `True` except for the last occurrence.
- `False` : Mark all duplicates as `True`.

Returns

pandas.core.series.Series

See also:

Index.duplicated Equivalent method on pandas.Index.

DataFrame.duplicated Equivalent method on pandas.DataFrame.

Series.drop_duplicates Remove duplicate values from Series.

Examples

By default, for each set of duplicated values, the first occurrence is set on False and all others on True:

```
>>> animals = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> animals.duplicated()
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

which is equivalent to

```
>>> animals.duplicated(keep='first')
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```
>>> animals.duplicated(keep='last')
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

By setting keep on `False`, all duplicates are True:

```
>>> animals.duplicated(keep=False)
0     True
1    False
2     True
3    False
4     True
dtype: bool
```

pandas.Series.eq

`Series.`**`eq`**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Equal to of series and other, element-wise (binary operator *eq*).

Equivalent to `series == other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before

computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.equals

`Series.equals(other)`

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal. The column headers do not need to have the same type, but the elements within the columns must be the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

assert_series_equal Return True if left and right Series are equal, False otherwise.

assert_frame_equal Return True if left and right DataFrames are equal, False otherwise.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Notes

This function requires that the elements have the same dtype as their respective elements in the other Series or DataFrame. However, the column labels do not need to have the same type, as long as they are still considered equal.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

pandas.Series.ewm

`Series.ewm`(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)

Provides exponential weighted functions.

New in version 0.18.0.

Parameters

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [bool, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations.

expanding Provides expanding transformations.

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), …, 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

pandas.Series.expanding

`Series.expanding`(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [bool, default False] Set the labels at the center of the window.

axis [int or str, default 0]

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations.

ewm Provides exponential weighted functions.

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

pandas.Series.factorize

`Series.factorize`(*sort=False*, *na_sentinel=-1*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

Parameters

sort [boolean, default False] Sort *uniques* and shuffle *labels* to maintain the relationship.

na_sentinel [int, default -1] Value to mark "not found".

Returns

labels [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

uniques [ndarray, Index, or Categorical] The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

See also:

cut Discretize continuous-valued array.

unique Find the unique value in an array.

Examples

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
```

(continues on next page)

(continued from previous page)

```
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

pandas.Series.ffill

`Series.`**`ffill`**(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for *`DataFrame.fillna()`* with `method='ffill'`.

pandas.Series.fillna

`Series.`**`fillna`**(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)

Fill NA/NaN values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis [{0 or 'index'}]

inplace [boolean, default False] If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns

filled [Series]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A    B    C    D
0   NaN  2.0  NaN  0
1   3.0  4.0  NaN  1
2   3.0  4.0  NaN  5
3   3.0  3.0  NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A    B    C    D
0   0.0  2.0  2.0  0
1   3.0  4.0  2.0  1
2   0.0  1.0  2.0  5
3   0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A    B    C    D
0   0.0  2.0  2.0  0
1   3.0  4.0  NaN  1
2   NaN  1.0  NaN  5
3   NaN  3.0  NaN  4
```

pandas.Series.filter

`Series.`**`filter`**(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items [list-like] List of axis to restrict to (must not all be present).

like [string] Keep axis where "arg in col == True".

regex [string (regular expression)] Keep axis with re.search(regex, col) == True.

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

same type as input object

See also:

`DataFrame.loc`

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1,2,3], [4,5,6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

pandas.Series.first

`Series.first(offset)`

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

last Select final periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
```

(continues on next page)

(continued from previous page)

```
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

pandas.Series.first_valid_index

Series.**first_valid_index**()

Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.Series.floordiv

Series.**floordiv**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *floordiv*).

Equivalent to `series // other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rfloordiv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.from_array

classmethod `Series.`**`from_array`**(*arr*, *index=None*, *name=None*, *dtype=None*, *copy=False*, *fastpath=False*)

Construct Series from array.

Deprecated since version 0.23.0: Use pd.Series(..) constructor instead.

pandas.Series.from_csv

classmethod `Series.`**`from_csv`**(*path*, *sep=', '*, *parse_dates=True*, *header=None*, *index_col=0*, *encoding=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use *`pandas.read_csv()`* instead.

It is preferable to use the more powerful *`pandas.read_csv()`* for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a time Series.

This method only differs from *`pandas.read_csv()`* in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *header* is `None` instead of `0` (the first row is not used as the column names)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

With *`pandas.read_csv()`*, the option `squeeze=True` can be used to return a Series like `from_csv`.

Parameters

path [string file path or file handle / StringIO]

sep [string, default ','] Field delimiter

parse_dates [boolean, default True] Parse dates. Different default from read_table

header [int, default None] Row to use as header (skip prior rows)

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

encoding [string, optional] a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

infer_datetime_format [boolean, default False] If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns

y [Series]

See also:

read_csv

pandas.Series.ge

`Series.ge(other, level=None, fill_value=None, axis=0)`

Greater than or equal to of series and other, element-wise (binary operator *ge*).

Equivalent to `series >= other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.get

`Series.get(key, default=None)`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

pandas.Series.get_dtype_counts

`Series.get_dtype_counts()`

Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64     1
int64       1
object      1
dtype: int64
```

pandas.Series.get_ftype_counts

`Series.get_ftype_counts()`

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()  # doctest: +SKIP
float64:dense     1
int64:dense       1
object:dense      1
dtype: int64
```

pandas.Series.get_value

`Series.get_value(label, takeable=False)`

Quickly retrieve single value at passed index label.

Deprecated since version 0.21.0: Please use .at[] or .iat[] accessors.

Parameters

label [object]

takeable [interpret the index as indexers, default False]

Returns

value [scalar value]

pandas.Series.get_values

`Series.get_values()`

Same as values (but handles sparseness conversions); is a view.

pandas.Series.groupby

`Series.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False, **kwargs)`

Group DataFrame or Series using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis [{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1).

level [int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

as_index [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output.

sort [bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

group_keys [bool, default True] When calling apply, add group keys to index to identify pieces.

squeeze [bool, default False] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

observed [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

****kwargs** Optional, only accepts keyword argument 'mutated' and is passed to groupby.

Returns

DataFrameGroupBy or SeriesGroupBy Depends on the calling object and returns groupby object that contains information about the groups.

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

```
>>> df = pd.DataFrame({'Animal' : ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed' : [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Capitve', 'Wild', 'Capitve', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed' : [390., 350., 30., 20.]},
...                    index=index)
>>> df
                Max Speed
Animal Type
Falcon Capitve      390.0
       Wild         350.0
Parrot Capitve       30.0
       Wild          20.0
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level=1).mean()
         Max Speed
Type
Capitve      210.0
Wild         185.0
```

pandas.Series.gt

`Series.gt` (*other*, *level=None*, *fill_value=None*, *axis=0*)

Greater than of series and other, element-wise (binary operator *gt*).

Equivalent to `series > other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.head

`Series.head(n=5)`

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [same type as caller] The first *n* rows of the caller object.

See also:

DataFrame.tail Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

pandas.Series.hist

`Series.`**`hist`**(*by=None*, *ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *figsize=None*, *bins=10*, ***kwds*)

Draw histogram of the input series using matplotlib.

Parameters

by [object, optional] If passed, then used to form histograms for separate groups

ax [matplotlib axis object] If not passed, uses gca()

grid [boolean, default True] Whether to show axis grid lines

xlabelsize [int, default None] If specified changes the x-axis label size

xrot [float, default None] rotation of x axis labels

ylabelsize [int, default None] If specified changes the y-axis label size

yrot [float, default None] rotation of y axis labels

figsize [tuple, default None] figure size in inches by default

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

bins [integer, default 10] Number of histogram bins to be used

'kwds'** [keywords] To be passed to the actual plotting function

See also:

`matplotlib.axes.Axes.hist` Plot a histogram using matplotlib.

pandas.Series.idxmax

`Series.idxmax`(*axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmax. Redundant for application on Series.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

idxmax [Index of maximum of values.]

Raises

ValueError If the Series is empty.

See also:

`numpy.argmax` Return indices of the maximum values along the given axis.

`DataFrame.idxmax` Return index of first occurrence of maximum over requested axis.

`Series.idxmin` Return index *label* of the first occurrence of minimum of values.

Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

pandas.Series.idxmin

`Series.idxmin`(*axis=0*, *skipna=True*, **args*, ***kwargs*)

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

Parameters

skipna [boolean, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

axis [int, default 0] For compatibility with DataFrame.idxmin. Redundant for application on Series.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

idxmin [Index of minimum of values.]

Raises

ValueError If the Series is empty.

See also:

`numpy.argmin` Return indices of the minimum values along the given axis.

`DataFrame.idxmin` Return index of first occurrence of minimum over requested axis.

`Series.idxmax` Return index *label* of the first occurrence of maximum of values.

Notes

This method is the Series version of `ndarray.argmin`. This method returns the label of the minimum, while `ndarray.argmin` returns the position. To get the position, use `series.values.argmin()`.

Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...               index=['A' ,'B' ,'C' ,'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmin(skipna=False)
nan
```

pandas.Series.infer_objects

`Series.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

pandas.Series.interpolate

`Series.interpolate`(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

New in version 0.18.1: Added support for the 'akima' method. Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in SciPy 0.18; backwards-compatible with SciPy < 0.18

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.21.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all NaN values

See also:

fillna Fill missing values using different methods.

scipy.interpolate.Akima1DInterpolator Piecewise cubic polynomials (Akima interpolator).

scipy.interpolate.BPoly.from_derivatives Piecewise polynomial in the Bernstein basis.

scipy.interpolate.interp1d Interpolate a 1-D function.

scipy.interpolate.KroghInterpolator Interpolate polynomial (Krogh interpolator).

scipy.interpolate.PchipInterpolator PCHIP 1-d monotonic cubic interpolation.

scipy.interpolate.CubicSpline Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in NaN in a *Series* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in NaN in a Series by padding, but filling at most two consecutive NaN at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
```

(continues on next page)

(continued from previous page)

```
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in `NaN` in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an `order` (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains `NaN`, because there is no entry befofe it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0,  np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

pandas.Series.isin

`Series.isin(values)`

Check whether *values* are contained in Series.

Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

Parameters

values [set or list-like] The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a list of one element.

New in version 0.18.1: Support for values as a set.

Returns

isin [Series (bool dtype)]

Raises

TypeError

- If *values* is a string

See also:

DataFrame.isin Equivalent method on DataFrame.

Examples

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...                'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0     True
1     True
2     True
3    False
4     True
5    False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['lama'])
0     True
1    False
2     True
3    False
4     True
```

(continues on next page)

(continued from previous page)

```
5    False
Name: animal, dtype: bool
```

pandas.Series.isna

`Series.isna()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull Alias of isna.

Series.notna Boolean inverse of isna.

Series.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
```

(continues on next page)

(continued from previous page)

```
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Series.isnull

`Series.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.isnull Alias of isna.

Series.notna Boolean inverse of isna.

Series.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Series.item

`Series.item()`

Return the first element of the underlying data as a python scalar.

pandas.Series.items

`Series.items()`

Lazily iterate over (index, value) tuples.

pandas.Series.iteritems

`Series.iteritems()`

Lazily iterate over (index, value) tuples.

pandas.Series.keys

`Series.keys()`

Alias for index.

pandas.Series.kurt

`Series.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [scalar or Series (if level specified)]

pandas.Series.kurtosis

`Series.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [scalar or Series (if level specified)]

pandas.Series.last

`Series.last(offset)`

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

pandas.Series.last_valid_index

`Series.`**`last_valid_index`**`()`

Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.Series.le

`Series.`**`le`**`(`*other*, *level=None*, *fill_value=None*, *axis=0*`)`

Less than or equal to of series and other, element-wise (binary operator *le*).

Equivalent to `series <= other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.None

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.lt

Series.**lt**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Less than of series and other, element-wise (binary operator *lt*).

Equivalent to `series < other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.None

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.mad

`Series.mad`(*axis=None*, *skipna=None*, *level=None*)

Return the mean absolute deviation of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mad [scalar or Series (if level specified)]

pandas.Series.map

`Series.map`(*arg*, *na_action=None*)

Map values of Series according to input correspondence.

Used for substituting each value in a Series with another value, that may be derived from a function, a `dict` or a *Series*.

Parameters

arg [function, dict, or Series] Mapping correspondence.

na_action [{None, 'ignore'}, default None] If 'ignore', propagate NaN values, without passing them to the mapping correspondence.

Returns

Series Same index as caller.

See also:

Series.apply For applying more complex functions on a Series.

DataFrame.apply Apply a function row-/column-wise.

DataFrame.applymap Apply a function elementwise on a whole DataFrame.

Notes

When `arg` is a dictionary, values in Series that are not in the dictionary (as keys) are converted to `NaN`. However, if the dictionary is a `dict` subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than `NaN`.

Examples

```
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])
>>> s
0      cat
1      dog
2      NaN
3   rabbit
dtype: object
```

`map` accepts a `dict` or a `Series`. Values that are not found in the `dict` are converted to `NaN`, unless the dict has a default value (e.g. `defaultdict`):

```
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0   kitten
1    puppy
2      NaN
3      NaN
dtype: object
```

It also accepts a function:

```
>>> s.map('I am a {}'.format)
0       I am a cat
1       I am a dog
2       I am a nan
3    I am a rabbit
dtype: object
```

To avoid applying the function to missing values (and keep them as `NaN`) `na_action='ignore'` can be used:

```
>>> s.map('I am a {}'.format, na_action='ignore')
0     I am a cat
1     I am a dog
2            NaN
3  I am a rabbit
dtype: object
```

pandas.Series.mask

`Series.mask`(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Replace values where the condition is True.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

DataFrame.where() Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `mask` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
```

(continues on next page)

(continued from previous page)

```
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.Series.max

Series.**max**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the maximum of the values for the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

max [scalar or Series (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

Max using level names, as well as indices.

```
>>> s.max(level='blooded')
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

```
>>> s.max(level=0)
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

pandas.Series.mean

Series.**mean**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mean [scalar or Series (if level specified)]

pandas.Series.median

Series.**median**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

median [scalar or Series (if level specified)]

pandas.Series.memory_usage

Series.**memory_usage**(*index=True*, *deep=False*)

Return the memory usage of the Series.

The memory usage can optionally include the contribution of the index and of elements of *object* dtype.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the Series index.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned value.

Returns

int Bytes of memory consumed.

See also:

`numpy.ndarray.nbytes` Total bytes consumed by the elements of the array.

`DataFrame.memory_usage` Bytes consumed by a DataFrame.

Examples

```
>>> s = pd.Series(range(3))
>>> s.memory_usage()
104
```

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```
>>> s.memory_usage(index=False)
24
```

The memory footprint of *object* values is ignored by default:

```
>>> s = pd.Series(["a", "b"])
>>> s.values
array(['a', 'b'], dtype=object)
>>> s.memory_usage()
96
>>> s.memory_usage(deep=True)
212
```

pandas.Series.min

`Series.min`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the minimum of the values for the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

min [scalar or Series (if level specified)]

See also:

`Series.sum` Return the sum.

`Series.min` Return the minimum.

`Series.max` Return the maximum.

`Series.idxmin` Return the index of the minimum.

`Series.idxmax` Return the index of the maximum.

`DataFrame.min` Return the sum over the requested axis.

`DataFrame.min` Return the minimum over the requested axis.

`DataFrame.max` Return the maximum over the requested axis.

`DataFrame.idxmin` Return the index of the minimum over the requested axis.

`DataFrame.idxmax` Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

Min using level names, as well as indices.

```
>>> s.min(level='blooded')
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

```
>>> s.min(level=0)
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

pandas.Series.mod

`Series.mod`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Modulo of series and other, element-wise (binary operator *mod*).

Equivalent to `series % other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rmod

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.mode

`Series.`**`mode`**(*dropna=True*)

Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

Parameters

dropna [boolean, default True] Don't consider counts of NaN/NaT.

New in version 0.24.0.

Returns

modes [Series (sorted)]

pandas.Series.mul

`Series.`**`mul`**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before

computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rmul

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.multiply

`Series.`**`multiply`**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rmul

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.ne

`Series.ne`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Not equal to of series and other, element-wise (binary operator *ne*).

Equivalent to `series != other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

`Series.None`

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.nlargest

`Series.`**`nlargest`**(*n=5*, *keep='first'*)

Return the largest *n* elements.

Parameters

n [int, default 5] Return this many descending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- `first` : take the first occurrences based on the index order
- `last` : take the last occurrences based on the index order
- **`all`** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* largest values in the Series, sorted in decreasing order.

See also:

`Series.nsmallest` Get the *n* smallest elements.

`Series.sort_values` Sort Series by values.

`Series.head` Return the first *n* rows.

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Malta": 434000, "Maldives": 434000,
...                         "Brunei": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Malta         434000
Maldives      434000
Brunei        434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Monserat        5200
dtype: int64
```

The *n* largest elements where `n=5` by default.

```
>>> s.nlargest()
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3`. Default *keep* value is 'first' so Malta will be kept.

```
>>> s.nlargest(3)
France    65000000
Italy     59000000
Malta       434000
dtype: int64
```

The *n* largest elements where `n=3` and keeping the last duplicates. Brunei will be kept since it is the last with value 434000 based on the index order.

```
>>> s.nlargest(3, keep='last')
France      65000000
Italy       59000000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3` with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```
>>> s.nlargest(3, keep='all')
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

pandas.Series.nonzero

`Series.nonzero()`

Return the *integer* indices of the elements that are non-zero.

Deprecated since version 0.24.0: Please use .to_numpy().nonzero() as a replacement.

This method is equivalent to calling *numpy.nonzero* on the series data. For compatibility with NumPy, the return value is the same (a tuple with an array of indices for each dimension), but it will always be a one-item tuple because series only have one dimension.

See also:

`numpy.nonzero`

Examples

```
>>> s = pd.Series([0, 3, 0, 4])
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
1    3
3    4
dtype: int64
```

```
>>> s = pd.Series([0, 3, 0, 4], index=['a', 'b', 'c', 'd'])
# same return although index of s is different
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
b    3
d    4
dtype: int64
```

pandas.Series.notna

`Series.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

Series.notnull Alias of notna.

Series.isna Boolean inverse of notna.

Series.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Series.notnull

`Series.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

`Series.notnull` Alias of notna.

`Series.isna` Boolean inverse of notna.

Series.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Series.nsmallest

`Series.`**`nsmallest`**(*n=5*, *keep='first'*)

Return the smallest *n* elements.

Parameters

n [int, default 5] Return this many ascending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- `first` : take the first occurrences based on the index order
- `last` : take the last occurrences based on the index order
- **`all`** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* smallest values in the Series, sorted in increasing order.

See also:

Series.nlargest Get the *n* largest elements.

Series.sort_values Sort Series by values.

Series.head Return the first *n* rows.

Notes

Faster than `.sort_values().head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Brunei": 434000, "Malta": 434000,
...                         "Maldives": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Brunei        434000
Malta         434000
Maldives      434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Monserat        5200
dtype: int64
```

The *n* largest elements where `n=5` by default.

```
>>> s.nsmallest()
Monserat      5200
Nauru        11300
Tuvalu       11300
Anguilla     11300
Iceland     337000
dtype: int64
```

The *n* smallest elements where `n=3`. Default *keep* value is 'first' so Nauru and Tuvalu will be kept.

```
>>> s.nsmallest(3)
Monserat     5200
Nauru       11300
Tuvalu      11300
dtype: int64
```

The *n* smallest elements where `n=3` and keeping the last duplicates. Anguilla and Tuvalu will be kept since they are the last with value 11300 based on the index order.

```
>>> s.nsmallest(3, keep='last')
Monserat      5200
Anguilla     11300
Tuvalu       11300
dtype: int64
```

The *n* smallest elements where n=3 with all duplicates kept. Note that the returned Series has four elements due to the three duplicates.

```
>>> s.nsmallest(3, keep='all')
Monserat      5200
Nauru        11300
Tuvalu       11300
Anguilla     11300
dtype: int64
```

pandas.Series.nunique

Series.**nunique**(*dropna=True*)

Return number of unique elements in the object.

Excludes NA values by default.

Parameters

dropna [boolean, default True] Don't include NaN in the count.

Returns

nunique [int]

pandas.Series.pct_change

Series.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

chg [Series or DataFrame] The same type as the calling object.

See also:

Series.diff Compute the difference of two elements in a Series.

DataFrame.diff Compute the difference of two elements in a DataFrame.

Series.shift Shift the index by some number of periods.

DataFrame.shift Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
```

(continues on next page)

(continued from previous page)

```
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pandas.Series.pipe

`Series.pipe`(*func*, **args*, ***kwargs*)
Apply func(self, *args, **kwargs).

Parameters

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

DataFrame.apply, *DataFrame.applymap*, *Series.map*

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.Series.plot

`Series.plot`(*kind='line'*, *ax=None*, *figsize=None*, *use_index=True*, *title=None*, *grid=None*, *legend=False*, *style=None*, *logx=False*, *logy=False*, *loglog=False*, *xticks=None*, *yticks=None*, *xlim=None*, *ylim=None*, *rot=None*, *fontsize=None*, *colormap=None*, *table=False*, *yerr=None*, *xerr=None*, *label=None*, *secondary_y=False*, ***kwds*)

Make plots of Series using matplotlib / pylab.

New in version 0.17.0: Each plot kind has a corresponding method on the `Series.plot` accessor: `s.plot(kind='line')` is equivalent to `s.plot.line()`.

Parameters

data [Series]

kind [str]

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot

ax [matplotlib axes object] If not passed, uses gca()

figsize [a tuple (width, height) in inches]

use_index [boolean, default True] Use index as ticks for x axis

title [string or list] Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and *subplots* is True, print each item in the list above the corresponding subplot.

grid [boolean, default None (matlab style default)] Axis grid lines

legend [False/True/'reverse'] Place legend on axis subplots

style [list or dict] matplotlib line style per column

logx [boolean, default False] Use log scaling on x axis

logy [boolean, default False] Use log scaling on y axis

loglog [boolean, default False] Use log scaling on both x and y axes

xticks [sequence] Values to use for the xticks

yticks [sequence] Values to use for the yticks

xlim [2-tuple/list]

ylim [2-tuple/list]

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots)

fontsize [int, default None] Font size for xticks and yticks

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

colorbar [boolean, optional] If True, plot colorbar (only relevant for 'scatter' and 'hexbin' plots)

position [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

table [boolean, Series or DataFrame, default False] If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib's default layout. If a Series or DataFrame is passed, use passed data to draw a table.

yerr [DataFrame, Series, array-like, dict and str] See *Plotting with Error Bars* for detail.

xerr [same types as yerr.]

label [label argument to provide to plot]

secondary_y [boolean or sequence of ints, default False] If True then y-axis will be on the right

mark_right [boolean, default True] When using a secondary_y axis, automatically mark the column labels with "(right)" in the legend

'kwds'** [keywords] Options to pass to matplotlib plotting method

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them]

Notes

- See matplotlib documentation online for more on this subject
- If *kind* = 'bar' or 'barh', you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

pandas.Series.pop

`Series.pop(item)`

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [Series]

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pandas.Series.pow

`Series.pow(`*other*, *level=None*, *fill_value=None*, *axis=0*`)`

Exponential power of series and other, element-wise (binary operator *pow*).

Equivalent to `series ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rpow

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.prod

`Series.prod`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [scalar or Series (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.Series.product

`Series.product`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [scalar or Series (if level specified)]

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the min_count parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the skipna parameter, min_count handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.Series.ptp

Series.**ptp**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Returns the difference between the maximum value and the minimum value in the object. This is the equivalent of the numpy.ndarray method ptp.

Deprecated since version 0.24.0: Use numpy.ptp instead

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

ptp [scalar or Series (if level specified)]

pandas.Series.put

Series.**put**(**args*, ***kwargs*)

Applies the *put* method to its *values* attribute if it has one.

See also:

numpy.ndarray.put

pandas.Series.quantile

`Series.quantile(q=0.5, interpolation='linear')`

Return value at the given quantile.

Parameters

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

Returns

quantile [float or Series] if `q` is an array, a Series will be returned where the index is `q` and the values are the quantiles.

See also:

core.window.Rolling.quantile, `numpy.percentile`

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25    1.75
0.50    2.50
0.75    3.25
dtype: float64
```

pandas.Series.radd

`Series.radd(other, level=None, fill_value=None, axis=0)`

Addition of series and other, element-wise (binary operator *radd*).

Equivalent to `other + series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before

computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.add

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rank

Series.**rank**(*axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

pandas.Series.ravel

`Series.ravel(order='C')`

Return the flattened underlying data as an ndarray.

See also:

`numpy.ndarray.ravel`

pandas.Series.rdiv

`Series.rdiv(other, level=None, fill_value=None, axis=0)`

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.truediv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rdivmod

`Series.rdivmod`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division and modulo of series and other, element-wise (binary operator *rdivmod*).

Equivalent to `other divmod series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.divmod

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
```

(continues on next page)

(continued from previous page)

```
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.reindex

`Series.reindex`(*index=None*, ***kwargs*)

Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

index [array-like, optional] New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit [int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

Series with changed index.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...      'http_status': [200,200,404,404,301],
...      'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...       index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
```

(continues on next page)

(continued from previous page)

```
2009-12-30      NaN
2009-12-31      NaN
2010-01-01    100.0
2010-01-02    101.0
2010-01-03      NaN
2010-01-04    100.0
2010-01-05     89.0
2010-01-06     88.0
2010-01-07      NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the *user guide* for more.

pandas.Series.reindex_axis

`Series.`**`reindex_axis`**(*labels*, *axis=0*, ***kwargs*)

Conform Series to new index with optional filling logic.

Deprecated since version 0.21.0: Use `Series.reindex` instead.

pandas.Series.reindex_like

`Series.`**`reindex_like`**(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other [Object of the same data type] Its row and column indices are used to define the new indices of this object.

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

Series or DataFrame Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...     columns=['temp_celsius', 'temp_fahrenheit', 'windspeed'],
...     index=pd.date_range(start='2014-02-12',
...                         end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...     columns=['temp_celsius', 'windspeed'],
...     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                             '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

pandas.Series.rename

`Series.rename`(*index=None*, ***kwargs*)

Alter Series index labels or name.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Alternatively, change `Series.name` with a scalar value.

See the *user guide* for more.

Parameters

index [scalar, hashable sequence, dict-like or function, optional] dict-like or functions are transformations to apply to the index. Scalar or hashable sequence-like will alter the `Series.name` attribute.

copy [bool, default True] Also copy underlying data

inplace [bool, default False] Whether to return a new Series. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

Returns

renamed [Series (new object)]

See also:

Series.rename_axis

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
```

pandas.Series.rename_axis

Series.**rename_axis**(*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*)

Set the name of the axis for the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set the axis name attribute.

index, columns [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

Changed in version 0.24.0.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to rename.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None The same type as the caller or None if *inplace* is True.

See also:

Series.rename Alter Series index labels or name.

DataFrame.rename Alter DataFrame index labels or name.

Index.rename Set new names on index.

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
>>> df
limbs   num_legs  num_arms
animal
```

(continues on next page)

(continued from previous page)

```
dog              4         0
cat              4         0
monkey           2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

pandas.Series.reorder_levels

Series.**reorder_levels**(*order*)

Rearrange index levels using input order.

May not drop or duplicate levels.

Parameters

order [list of int representing new level order] (reference level by number or key)

Returns

type of caller (new object)

pandas.Series.repeat

Series.**repeat**(*repeats*, *axis=None*)

Repeat elements of a Series.

Returns a new Series where each element of the current Series is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Series.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_series [Series] Newly created Series with repeated elements.

See also:

Index.repeat Equivalent function for Index.

numpy.repeat Similar method for `numpy.ndarray`.

Examples

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
dtype: object
>>> s.repeat(2)
0    a
0    a
1    b
1    b
2    c
2    c
dtype: object
>>> s.repeat([1, 2, 3])
0    a
1    b
1    b
2    c
2    c
2    c
dtype: object
```

pandas.Series.replace

`Series.`**`replace`**(*to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the Series are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*

  - regex: regexs matching *to_replace* will be replaced with *value*

- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.

- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [bool, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace* *must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

Series Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

`Series.fillna` Fill NA values.

`Series.where` Replace values based on boolean condition.

`Series.str.replace` Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like 'to_replace'

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like 'to_replace'

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression ‘to_replace‘

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

pandas.Series.resample

`Series.resample`(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or pass datetime-like values to the *on* or *level* keyword.

Parameters

rule [str] The offset string or object representing target conversion.

how [str] Method for down/re-sampling, default to 'mean' for downsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).mean()`, or `.resample(...).apply(<func>)`

axis [{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this will default to 0, i.e. along the rows. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

fill_method [str, default None] Filling method for upsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).<func>()`, e.g. `.resample(...).pad()`

closed [{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind [{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

loffset [timedelta, default None] Adjust the resampled time labels.

limit [int, default None] Maximum size gap when reindexing with *fill_method*.

Deprecated since version 0.18.0.

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0.

on [str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

Series.resample Resample a Series.

DataFrame.resample Resample a DataFrame.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
2018Q1    1
2018Q2    2
2018Q3    3
```

(continues on next page)

(continued from previous page)

```
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...           'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df2 = pd.DataFrame(d2,
...                    index=pd.MultiIndex.from_product([days,
...                                                      ['morning',
...                                                       'afternoon']]
...                                                      ))
>>> df2
                      price  volume
2000-01-01 morning       10      50
           afternoon     11      60
2000-01-02 morning        9      40
```

(continues on next page)

(continued from previous page)

```
            afternoon     13     100
2000-01-03 morning       14      50
            afternoon     18     100
2000-01-04 morning       17      40
            afternoon     19      50
>>> df2.resample('D', level=0).sum()
             price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

pandas.Series.reset_index

Series.**reset_index**(*level=None*, *drop=False*, *name=None*, *inplace=False*)

Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

Parameters

level [int, str, tuple, or list, default optional] For a Series with a MultiIndex, only remove the specified levels from the index. Removes all levels by default.

drop [bool, default False] Just reset the index, without inserting it as a column in the new DataFrame.

name [object, optional] The name to use for the column containing the original Series values. Uses `self.name` by default. This argument is ignored when *drop* is True.

inplace [bool, default False] Modify the Series in place (do not create a new object).

Returns

Series or DataFrame When *drop* is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When *drop* is True, a *Series* is returned. In either case, if `inplace=True`, no value is returned.

See also:

DataFrame.reset_index Analogous function for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4], name='foo',
...               index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
>>> s.reset_index()
  idx  foo
0   a    1
1   b    2
```

(continues on next page)

(continued from previous page)

```
2    c     3
3    d     4
```

To specify the name of the new column use *name*.

```
>>> s.reset_index(name='values')
  idx  values
0   a       1
1   b       2
2   c       3
3   d       4
```

To generate a new Series with the default set *drop* to True.

```
>>> s.reset_index(drop=True)
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

To update the Series in place, without generating a new one set *inplace* to True. Note that it also requires `drop=True`.

```
>>> s.reset_index(inplace=True, drop=True)
>>> s
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

The *level* parameter is interesting for Series with a multi-level index.

```
>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
...           np.array(['one', 'two', 'one', 'two'])]
>>> s2 = pd.Series(
...     range(4), name='foo',
...     index=pd.MultiIndex.from_arrays(arrays,
...                                     names=['a', 'b']))
```

To remove a specific level from the Index, use *level*.

```
>>> s2.reset_index(level='a')
       a  foo
b
one  bar    0
two  bar    1
one  baz    2
two  baz    3
```

If *level* is not set, all levels are removed from the Index.

```
>>> s2.reset_index()
     a    b  foo
0  bar  one    0
```

(continues on next page)

(continued from previous page)

```
1  bar  two    1
2  baz  one    2
3  baz  two    3
```

pandas.Series.rfloordiv

`Series.rfloordiv`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.floordiv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rmod

`Series.rmod`(*other*, *level=None*, *fill_value=None*, *axis=0*)
Modulo of series and other, element-wise (binary operator *rmod*).

Equivalent to `other % series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.mod

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rmul

`Series.rmul`(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *rmul*).

Equivalent to `other * series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.mul

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rolling

`Series.rolling`(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

Parameters

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, *min_periods* will default to 1. Otherwise, *min_periods* will default to the size of the window.

center [bool, default False] Set the labels at the center of the window.

win_type [str, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [str, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

axis [int or str, default 0]

closed [str, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

Returns

a Window or Rolling sub-classed for the particular operation

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions.

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)

- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                       B
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
```

pandas.Series.round

Series.**round**(*decimals=0*, **args*, ***kwargs*)

Round each value in a Series to the given number of decimals.

Parameters

decimals [int] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Series object

See also:

`numpy.around`, *DataFrame.round*

pandas.Series.rpow

Series.**rpow**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *rpow*).

Equivalent to `other ** series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.pow

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rsub

`Series.rsub`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *rsub*).

Equivalent to `other - series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.sub

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.rtruediv

Series.**rtruediv**(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.truediv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
```

(continues on next page)

(continued from previous page)

```
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.sample

Series.**sample**(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [bool, default False] Sample with or without replacement.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

Series or DataFrame A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`numpy.random.choice` Generates a random sample from a given 1-D numpy array.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

pandas.Series.searchsorted

`Series.`**`searchsorted`**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Series *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

int or array of int A scalar or array of insertion points with the same shape as *value*.

Changed in version 0.24.0: If *value* is a scalar, an int is now always returned. Previously, scalar inputs returned an 1-item array for `Series` and `Categorical`.

See also:

`numpy.searchsorted`

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
3
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',
                        'cheese', 'milk'], ordered=True)
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
1
```

```
>>> x.searchsorted(['bread'], side='right')
array([3])
```

pandas.Series.select

`Series.`**`select`**(*crit, axis=0*)

Return data corresponding to axis labels matching criteria.

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [same type as caller]

pandas.Series.sem

`Series.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [scalar or Series (if level specified)]

pandas.Series.set_axis

`Series.set_axis(labels, axis=0, inplace=None)`

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

Parameters

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

DataFrame.rename_axis Alter the name of the index or columns.

Examples

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

pandas.Series.set_value

`Series.set_value`(*label*, *value*, *takeable=False*)

Quickly set single value at passed label.

Deprecated since version 0.21.0: Please use .at[] or .iat[] accessors.

If label is not contained, a new object is created with the label placed at the end of the result index.

Parameters

label [object] Partial indexing with MultiIndex not allowed

value [object] Scalar value

takeable [interpret the index as indexers, default False]

Returns

series [Series] If label is contained, will be reference to calling Series, otherwise a new object

pandas.Series.shift

`Series.shift`(*periods=1*, *freq=None*, *axis=0*, *fill_value=None*)

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*.

Parameters

periods [int] Number of periods to shift. Can be positive or negative.

freq [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data.

axis [{0 or 'index', 1 or 'columns', None}, default None] Shift direction.

fill_value [object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of *self*. For numeric data, `np.nan` is used. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

Changed in version 0.24.0.

Returns

Series Copy of input object, shifted.

See also:

Index.shift Shift values of Index.

DatetimeIndex.shift Shift values of DatetimeIndex.

PeriodIndex.shift Shift values of PeriodIndex.

tshift Shift the time index, using the index's frequency if available.

Examples

```
>>> df = pd.DataFrame({'Col1': [10, 20, 15, 30, 45],
...                    'Col2': [13, 23, 18, 33, 48],
...                    'Col3': [17, 27, 22, 37, 52]})
```

```
>>> df.shift(periods=3)
   Col1  Col2  Col3
0   NaN   NaN   NaN
1   NaN   NaN   NaN
2   NaN   NaN   NaN
3  10.0  13.0  17.0
4  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis='columns')
   Col1  Col2  Col3
0   NaN  10.0  13.0
1   NaN  20.0  23.0
2   NaN  15.0  18.0
3   NaN  30.0  33.0
4   NaN  45.0  48.0
```

```
>>> df.shift(periods=3, fill_value=0)
   Col1  Col2  Col3
0     0     0     0
1     0     0     0
2     0     0     0
3    10    13    17
4    20    23    27
```

pandas.Series.skew

Series.**skew**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

skew [scalar or Series (if level specified)]

pandas.Series.slice_shift

`Series.slice_shift(periods=1, axis=0)`

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

pandas.Series.sort_index

`Series.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True)`

Sort Series by index labels.

Returns a new Series sorted by label if *inplace* argument is `False`, otherwise updates the original series and returns None.

Parameters

axis [int, default 0] Axis to direct sorting. This can only be 0 for Series.

level [int, optional] If not None, sort on values in specified index level(s).

ascending [bool, default true] Sort ascending vs. descending.

inplace [bool, default False] If True, perform operation in-place.

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] If 'first' puts NaNs at the beginning, 'last' puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] If true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

Returns

pandas.Series The original Series sorted by the labels

See also:

DataFrame.sort_index Sort DataFrame by the index.

DataFrame.sort_values Sort DataFrame by the value.

Series.sort_values Sort Series by the value.

Examples

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, 4])
>>> s.sort_index()
1    c
2    b
3    a
4    d
dtype: object
```

Sort Descending

```
>>> s.sort_index(ascending=False)
4    d
3    a
2    b
1    c
dtype: object
```

Sort Inplace

```
>>> s.sort_index(inplace=True)
>>> s
1    c
2    b
3    a
4    d
dtype: object
```

By default NaNs are put at the end, but use *na_position* to place them at the beginning

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, np.nan])
>>> s.sort_index(na_position='first')
NaN     d
 1.0    c
 2.0    b
 3.0    a
dtype: object
```

Specify index level to sort

```
>>> arrays = [np.array(['qux', 'qux', 'foo', 'foo',
...                     'baz', 'baz', 'bar', 'bar']),
...           np.array(['two', 'one', 'two', 'one',
...                     'two', 'one', 'two', 'one'])]
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
>>> s.sort_index(level=1)
bar  one    8
baz  one    6
foo  one    4
qux  one    2
```

(continues on next page)

(continued from previous page)

```
bar  two    7
baz  two    5
foo  two    3
qux  two    1
dtype: int64
```

Does not sort by remaining levels when sorting by levels

```
>>> s.sort_index(level=1, sort_remaining=False)
qux  one    2
foo  one    4
baz  one    6
bar  one    8
qux  two    1
foo  two    3
baz  two    5
bar  two    7
dtype: int64
```

pandas.Series.sort_values

`Series.sort_values`(*axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last'*)

Sort by the values.

Sort a Series in ascending or descending order by some criterion.

Parameters

axis [{0 or 'index'}, default 0] Axis to direct sorting. The value 'index' is accepted for compatibility with DataFrame.sort_values.

ascending [bool, default True] If True, sort values in ascending order, otherwise descending.

inplace [bool, default False] If True, perform operation in-place.

kind [{'quicksort', 'mergesort' or 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm.

na_position [{'first' or 'last'}, default 'last'] Argument 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.

Returns

Series Series ordered by values.

See also:

Series.sort_index Sort by the Series indices.

DataFrame.sort_values Sort DataFrame by the values along either axis.

DataFrame.sort_index Sort DataFrame by indices.

Examples

```
>>> s = pd.Series([np.nan, 1, 3, 10, 5])
>>> s
0     NaN
1     1.0
2     3.0
3     10.0
4     5.0
dtype: float64
```

Sort values ascending order (default behaviour)

```
>>> s.sort_values(ascending=True)
1     1.0
2     3.0
4     5.0
3    10.0
0     NaN
dtype: float64
```

Sort values descending order

```
>>> s.sort_values(ascending=False)
3    10.0
4     5.0
2     3.0
1     1.0
0     NaN
dtype: float64
```

Sort values inplace

```
>>> s.sort_values(ascending=False, inplace=True)
>>> s
3    10.0
4     5.0
2     3.0
1     1.0
0     NaN
dtype: float64
```

Sort values putting NAs first

```
>>> s.sort_values(na_position='first')
0     NaN
1     1.0
2     3.0
4     5.0
3    10.0
dtype: float64
```

Sort a series of strings

```
>>> s = pd.Series(['z', 'b', 'd', 'a', 'c'])
>>> s
0    z
```

(continues on next page)

(continued from previous page)

```
1    b
2    d
3    a
4    c
dtype: object
```

```
>>> s.sort_values()
3    a
1    b
4    c
2    d
0    z
dtype: object
```

pandas.Series.squeeze

`Series.squeeze(axis=None)`
Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

New in version 0.20.0.

Returns

DataFrame, Series, or scalar The projection after squeezing *axis* or all the axes.

See also:

Series.iloc Integer-location based indexing for selecting scalars.

DataFrame.iloc Integer-location based indexing for selecting Series.

Series.to_frame Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes wil project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

pandas.Series.std

`Series.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [scalar or Series (if level specified)]

pandas.Series.str

`Series.str()`
Vectorized string functions for Series and Index. NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

Examples

```
>>> s.str.split('_')
>>> s.str.replace('_', '')
```

pandas.Series.sub

`Series.sub(other, level=None, fill_value=None, axis=0)`
Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.rsub

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.subtract

`Series.subtract`(*other*, *level=None*, *fill_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rsub

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.sum

`Series.`**sum**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis [{index (0)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

sum [scalar or Series (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

Sum using level names, as well as indices.

```
>>> s.sum(level='blooded')
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

```
>>> s.sum(level=0)
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

pandas.Series.swapaxes

`Series.`**`swapaxes`**(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately.

Returns

y [same as input]

pandas.Series.swaplevel

`Series.`**`swaplevel`**(*i=-2*, *j=-1*, *copy=True*)

Swap levels i and j in a MultiIndex.

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [Series]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

pandas.Series.tail

`Series.`**`tail`**(*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

DataFrame.head The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

pandas.Series.take

Series.**take**(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [same type as caller] An array-like containing the elements taken from the object.

See also:

DataFrame.loc Select a subset of a DataFrame by labels.

DataFrame.iloc Select a subset of a DataFrame by positions.

numpy.take Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
```

(continues on next page)

(continued from previous page)

```
2     bird        24.0
3   mammal        80.5
1   mammal         NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

pandas.Series.to_clipboard

Series.**to_clipboard**(*excel=True*, *sep=None*, ***kwargs*)
Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

See also:

DataFrame.to_csv Write a DataFrame to a comma-separated values (csv) file.

read_clipboard Read text from clipboard and pass to read_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
```

(continues on next page)

(continued from previous page)

```
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

pandas.Series.to_csv

`Series.to_csv(*args, **kwargs)`

Write object to a comma-separated values (csv) file.

Changed in version 0.24.0: The order of arguments for Series was changed.

Parameters

path_or_buf [str or file handle, default None] File path or object, if None is provided the result is returned as a string.

Changed in version 0.24.0: Was previously named "path" for Series.

sep [str, default ','] String of length 1. Field delimiter for the output file.

na_rep [str, default ''] Missing data representation.

float_format [str, default None] Format string for floating point numbers.

columns [sequence, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

Changed in version 0.24.0: Previously defaulted to False for Series.

index [bool, default True] Write row names (index).

index_label [str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode [str] Python write mode, default 'w'.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [str, default 'infer'] Compression mode among the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If 'infer' and *path_or_buf* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression).

Changed in version 0.24.0: 'infer' option added and set to default.

quoting [optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar [str, default '"'] String of length 1. Character used to quote fields.

line_terminator [string, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('n' for linux, 'rn' for Windows, i.e.).

Changed in version 0.24.0.

chunksize [int or None] Rows to write at a time.

tupleize_cols [bool, default False] Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

date_format [str, default None] Format string for datetime objects.

doublequote [bool, default True] Control quoting of *quotechar* inside a field.

escapechar [str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal [str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

Returns

None or str If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

read_csv Load a CSV file into a DataFrame.

to_excel Load an Excel file into a DataFrame.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

pandas.Series.to_dense

Series.**to_dense**()

Return dense representation of NDFrame (as opposed to sparse).

pandas.Series.to_dict

Series.**to_dict**(*into=<class 'dict'>*)

Convert Series to {label -> value} dict or dict-like object.

Parameters

into [class, default dict] The collections.Mapping subclass to use as the return object. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

Returns

value_dict [collections.Mapping]

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_dict()
{0: 1, 1: 2, 2: 3, 3: 4}
>>> from collections import OrderedDict, defaultdict
>>> s.to_dict(OrderedDict)
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> dd = defaultdict(list)
>>> s.to_dict(dd)
defaultdict(<type 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})
```

pandas.Series.to_excel

Series.**to_excel**(*excel_writer*, *sheet_name='Sheet1'*, *na_rep="*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer [str or ExcelWriter object] File path or existing ExcelWriter.

sheet_name [str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep [str, default ''] Missing data representation.

float_format [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns [sequence or list of str, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

index_label [str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow [int, default 0] Upper left cell row to dump data frame.

startcol [int, default 0] Upper left cell column to dump data frame.

engine [str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

verbose [bool, default True] Display more information in the error logs.

freeze_panes [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

New in version 0.20.0..

See also:

to_csv Write DataFrame to a comma-separated values (csv) file.

ExcelWriter Class for writing DataFrame objects into excel sheets.

read_excel Read an Excel file into a pandas DataFrame.

read_csv Read a comma-separated values (csv) file into DataFrame.

Notes

For compatibility with *to_csv()*, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")  # doctest: +SKIP
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')  # doctest: +SKIP
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:  # doctest: +SKIP
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')  # doctest: +SKIP
```

pandas.Series.to_frame

`Series.to_frame(name=None)`
Convert Series to DataFrame.

Parameters

name [object, default None] The passed name should substitute for the series name (if it has one).

Returns

data_frame [DataFrame]

pandas.Series.to_hdf

`Series.to_hdf(path_or_buf, key, **kwargs)`
Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the *user guide*.

Parameters

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.

- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See *Query via Data Columns*. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

`DataFrame.read_hdf` Read from HDF file.

`DataFrame.to_parquet` Write a DataFrame to the binary parquet format.

`DataFrame.to_sql` Write to a sql table.

`DataFrame.to_feather` Write out feather-format for DataFrames.

`DataFrame.to_csv` Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

pandas.Series.to_json

Series.**to_json**(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression='infer'*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index','table'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values','table'}
- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [bool, default True] Force encoded string to be ASCII.

date_unit [string, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [bool, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}] A string representing the compression to use in the output file, only used when the first argument is a filename. By default, the compression is inferred from the filename.

New in version 0.21.0.

Changed in version 0.24.0: 'infer' option added and set to default

index [bool, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

read_json

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

pandas.Series.to_latex

Series.**to_latex**(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a LaTeX tabular environment table.

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

Parameters

buf [file descriptor or None] Buffer to write to. If None, the output is returned as a string.

columns [list of label, optional] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

na_rep [str, default 'NaN'] Missing data representation.

formatters [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [str, optional] Format string for floating point numbers.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names [bool, default True] Prints the names of the indexes.

bold_rows [bool, default False] Make the row labels bold in the output.

column_format [str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a usepackage{longtable} to your LaTeX preamble.

escape [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe. .. versionadded:: 0.18.0

multicolumn [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module. .. versionadded:: 0.20.0

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module. .. versionadded:: 0.20.0

multirow [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module. .. versionadded:: 0.20.0

Returns

str or None If buf is None, returns the resulting LateX format as a string. Otherwise returns None.

See also:

DataFrame.to_string Render a DataFrame to a console-friendly tabular output.

DataFrame.to_html Render a DataFrame as an HTML table.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_latex(index=False) # doctest: +NORMALIZE_WHITESPACE
'\\begin{tabular}{lll}\n\\toprule\n      name &     mask &     weapon
\\\\\n\\midrule\n   Raphael &      red &        sai \\\\\n Donatello &
 purple &  bo staff \\\\\n\\bottomrule\n\\end{tabular}\n'
```

pandas.Series.to_list

`Series.to_list()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

See also:

`numpy.ndarray.tolist`

pandas.Series.to_msgpack

`Series.to_msgpack(path_or_buf=None, encoding='utf-8', **kwargs)`

Serialize object to input file path using msgpack format.

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [bool whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

pandas.Series.to_numpy

`Series.to_numpy(dtype=None, copy=False)`

A NumPy ndarray representing the values in this Series or Index.

New in version 0.24.0.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`

copy [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

Returns

numpy.ndarray

See also:

Series.array Get the actual data stored within.

Index.array Get the actual data stored within.

DataFrame.to_numpy Similar method for DataFrame.

Notes

The returned array will be the same up to equality (values equal in *self* will be equal in the returned array; likewise for values that are not equal). When *self* contains an ExtensionArray, the dtype may be different. For example, for a category-dtype Series, `to_numpy()` will return a NumPy array and the categorical dtype will be lost.

For NumPy dtypes, this will be a reference to the actual data stored in this Series or Index (assuming `copy=False`). Modifying the result in place will modify the data stored in the Series or Index (not that we recommend doing that).

For extension types, `to_numpy()` *may* require copying data and coercing the result to a NumPy type (possibly object), which may be expensive. When you need a no-copy reference to the underlying data, `Series.array` should be used instead.

This table lays out the different dtypes and default return types of `to_numpy()` for various dtypes within pandas.

dtype	array type
category[T]	ndarray[T] (same dtype as input)
period	ndarray[object] (Periods)
interval	ndarray[object] (Intervals)
IntegerNA	ndarray[object]
datetime64[ns]	datetime64[ns]
datetime64[ns, tz]	ndarray[object] (Timestamps)

Examples

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

Specify the *dtype* to control how datetime-aware data is represented. Use `dtype=object` to return an ndarray of pandas *Timestamp* objects, each with the correct `tz`.

```
>>> ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> ser.to_numpy(dtype=object)
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or `dtype='datetime64[ns]'` to return an ndarray of native datetime64 values. The values are converted to UTC and the timezone info is dropped.

```
>>> ser.to_numpy(dtype="datetime64[ns]")
... # doctest: +ELLIPSIS
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00...'],
      dtype='datetime64[ns]')
```

pandas.Series.to_period

`Series.to_period`(*freq=None*, *copy=True*)

Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

freq [string, default]

Returns

ts [Series with PeriodIndex]

pandas.Series.to_pickle

`Series.to_pickle`(*path*, *compression='infer'*, *protocol=4*)

Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [**?**] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

See also:

read_pickle Load pickled pandas object (or any object) from file.

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_sql Write DataFrame to a SQL database.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

pandas.Series.to_sparse

Series.**to_sparse**(*kind='block'*, *fill_value=None*)

Convert Series to SparseSeries.

Parameters

kind [{'block', 'integer'}]

fill_value [float, defaults to NaN (missing)]

Returns

sp [SparseSeries]

pandas.Series.to_sql

`Series.to_sql`(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [**?**] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

method [{None, 'multi', callable}, default None] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section *insert method*.

New in version 0.24.0.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

New in version 0.24.0.

References

[?], [?]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

pandas.Series.to_string

`Series.to_string`(*buf=None*, *na_rep='NaN'*, *float_format=None*, *header=True*, *index=True*, *length=False*, *dtype=False*, *name=False*, *max_rows=None*)

Render a string representation of the Series.

Parameters

buf [StringIO-like, optional] buffer to write to

na_rep [string, optional] string representation of NAN to use, default 'NaN'

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats default None

header [boolean, default True] Add the Series header (index name)

index [bool, optional] Add index (row) labels, default True

length [boolean, default False] Add the Series length

dtype [boolean, default False] Add the Series dtype

name [boolean, default False] Add the Series name if not None

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

Returns

formatted [string (if not buffer passed)]

pandas.Series.to_timestamp

`Series.to_timestamp`(*freq=None*, *how='start'*, *copy=True*)

Cast to datetimeindex of timestamps, at *beginning* of period.

Parameters

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

Returns

ts [Series with DatetimeIndex]

pandas.Series.to_xarray

`Series.to_xarray()`

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',  389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion',   'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                    columns=['name', 'class', 'max_speed',
...                             'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                    'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]}).set_index(['date',
...                                                  'animal'])
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
  * date      (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal    (animal) object 'falcon' 'parrot'
Data variables:
    speed     (date, animal) int64 350 18 361 15
```

pandas.Series.tolist

`Series.tolist()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

See also:

`numpy.ndarray.tolist`

pandas.Series.transform

`Series.transform(func, axis=0, *args, **kwargs)`

Call `func` on self producing a Series with transformed values and that has the same axis length as self.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for transforming the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.exp. 'sqrt']`
- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index'}] Parameter needed for compatibility with DataFrame.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

Series A Series that must have the same length as self.

Raises

ValueError [If the returned Series has a different length than self.]

See also:

Series.agg Only perform aggregating type operations.

Series.apply Invoke function on a Series.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting Series must have the same length as the input Series, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

pandas.Series.transpose

Series.**transpose**(**args*, ***kwargs*)
Return the transpose, which is by definition self.

pandas.Series.truediv

`Series.truediv(other, level=None, fill_value=None, axis=0)`

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters

other [Series or scalar value]

fill_value [None or float value, default None (NaN)] Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

Returns

result [Series]

See also:

Series.rtruediv

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

pandas.Series.truncate

`Series.truncate(before=None, after=None, axis=None, copy=True)`

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

Returns

type of caller The truncated Series or DataFrame.

See also:

DataFrame.loc Select a subset of a DataFrame by label.

DataFrame.iloc Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

pandas.Series.tshift

`Series.tshift`(*periods=1, freq=None, axis=0*)

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

pandas.Series.tz_convert

`Series.tz_convert(tz, axis=0, level=None, copy=True)`

Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

pandas.Series.tz_localize

`Series.tz_localize(tz, axis=0, level=None, copy=True, ambiguous='raise', nonexistent='raise')`

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent [str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid valuse are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

Series or DataFrame Same type as the input.

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series([1],
... index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7), index=pd.DatetimeIndex([
... '2018-10-28 01:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 03:00:00',
... '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3), index=pd.DatetimeIndex([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.Series(range(2), index=pd.DatetimeIndex([... '2015-03-29 02:30:00', ... '2015-03-29 03:30:00'])) >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 2015-03-29 03:00:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 2015-03-29 01:59:59.999999999+01:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 2015-03-29 03:30:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64

pandas.Series.unique

`Series.unique()`

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

Returns

ndarray or ExtensionArray The unique values returned as a NumPy array. In case of an extension-array backed Series, a new *ExtensionArray* of that type with just the unique values is returned. This includes

- Categorical
- Period
- Datetime with Timezone
- Interval
- Sparse
- IntegerNA

See also:

unique Top-level unique method for any 1-d array-like object.

Index.unique Return Index with unique values from an Index object.

Examples

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...            for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                          ordered=True)).unique()
[b, a, c]
Categories (3, object): [a < b < c]
```

pandas.Series.unstack

`Series.unstack`(*level=-1*, *fill_value=None*)

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

Parameters

level [int, string, or list of these, default last level] Level(s) to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

Returns

unstacked [DataFrame]

Examples

```
>>> s = pd.Series([1, 2, 3, 4],
...     index=pd.MultiIndex.from_product([['one', 'two'], ['a', 'b']]))
>>> s
one  a    1
     b    2
two  a    3
     b    4
dtype: int64
```

```
>>> s.unstack(level=-1)
     a  b
one  1  2
two  3  4
```

```
>>> s.unstack(level=0)
   one  two
a    1    3
b    2    4
```

pandas.Series.update

`Series.update(other)`

Modify Series in place using non-NA values from passed Series. Aligns on index.

Parameters

other [Series]

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6]))
>>> s
0    4
1    5
2    6
dtype: int64
```

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s.update(pd.Series(['d', 'e'], index=[0, 2]))
>>> s
0    d
1    b
2    e
dtype: object
```

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6, 7, 8]))
>>> s
0    4
1    5
2    6
dtype: int64
```

If `other` contains NaNs the corresponding values are not updated in the original Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, np.nan, 6]))
>>> s
0    4
1    2
2    6
dtype: int64
```

pandas.Series.valid

`Series.valid(inplace=False, **kwargs)`

Return Series without null values.

Deprecated since version 0.23.0: Use `Series.dropna()` instead.

pandas.Series.value_counts

`Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)`

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters

normalize [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.

sort [boolean, default True] Sort by values.

ascending [boolean, default False] Sort in ascending order.

bins [integer, optional] Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data.

dropna [boolean, default True] Don't include counts of NaN.

Returns

counts [Series]

See also:

`Series.count` Number of non-NA elements in a Series.

`DataFrame.count` Number of non-NA elements in a DataFrame.

Examples

```
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0    2
4.0    1
2.0    1
1.0    1
dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
>>> s.value_counts(normalize=True)
3.0    0.4
4.0    0.2
2.0    0.2
1.0    0.2
dtype: float64
```

bins

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

```
>>> s.value_counts(bins=3)
(2.0, 3.0]      2
(0.996, 2.0]    2
(3.0, 4.0]      1
dtype: int64
```

dropna

With *dropna* set to *False* we can also see NaN index values.

```
>>> s.value_counts(dropna=False)
3.0    2
NaN    1
4.0    1
2.0    1
1.0    1
dtype: int64
```

pandas.Series.var

`Series.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [scalar or Series (if level specified)]

pandas.Series.view

`Series.view(dtype=None)`

Create a new view of the Series.

This function will return a new Series with a view of the same underlying values in memory, optionally reinterpreted with a new data type. The new data type must preserve the same size in bytes as to not cause index misalignment.

Parameters

dtype [data type] Data type object or one of their string representations.

Returns

Series A new Series object as a view of the same data in memory.

See also:

numpy.ndarray.view Equivalent numpy function to create a new view of the same data in memory.

Notes

Series are instantiated with `dtype=float64` by default. While `numpy.ndarray.view()` will return a view with the same data type as the original array, `Series.view()` (without specified dtype) will try using `float64` and may fail if the original data type size in bytes is not the same.

Examples

```
>>> s = pd.Series([-2, -1, 0, 1, 2], dtype='int8')
>>> s
0   -2
1   -1
2    0
3    1
4    2
dtype: int8
```

The 8 bit signed integer representation of *-1* is *0b11111111*, but the same bytes represent 255 if read as an 8 bit unsigned integer:

```
>>> us = s.view('uint8')
>>> us
0    254
1    255
2      0
3      1
4      2
dtype: uint8
```

The views share the same underlying values:

```
>>> us[0] = 128
>>> s
0   -128
1     -1
2      0
3      1
4      2
dtype: int8
```

pandas.Series.where

`Series.where`(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Replace values where the condition is False.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

DataFrame.mask() Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `where` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.Series.xs

`Series.xs` (*key*, *axis=0*, *level=None*, *drop_level=True*)

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

DataFrame.loc Access a group of rows and columns by label(s) or a boolean array.

DataFrame.iloc Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see *MultiIndex Slicers*.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
                           num_legs  num_wings
class  animal  locomotion
mammal cat     walks              4          0
       dog     walks              4          0
       bat     flies              2          2
bird   penguin walks              2          2
```

Get values at specified index

```
>>> df.xs('mammal')
                   num_legs  num_wings
animal locomotion
cat    walks              4          0
dog    walks              4          0
bat    flies              2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog'))
            num_legs  num_wings
locomotion
walks              4          0
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
                   num_legs  num_wings
class  locomotion
mammal walks              4          0
```

Get values at several indexes and levels

```
>>> df.xs(('bird', 'walks'),
...       level=[0, 'locomotion'])
         num_legs  num_wings
animal
penguin         2          2
```

Get values at specified column and axis

```
>>> df.xs('num_wings', axis=1)
class   animal   locomotion
mammal  cat      walks         0
        dog      walks         0
        bat      flies         2
bird    penguin  walks         2
Name: num_wings, dtype: int64
```

6.3.2 Attributes

Axes

Series.index	The index (axis labels) of the Series.

Series.array	The ExtensionArray of the data backing this Series or Index.
Series.values	Return Series as ndarray or ndarray-like depending on the dtype.
Series.dtype	Return the dtype object of the underlying data.
Series.ftype	Return if the data is sparseldense.
Series.shape	Return a tuple of the shape of the underlying data.
Series.nbytes	Return the number of bytes in the underlying data.
Series.ndim	Number of dimensions of the underlying data, by definition 1.
Series.size	Return the number of elements in the underlying data.
Series.strides	Return the strides of the underlying data.
Series.itemsize	Return the size of the dtype of the item of the underlying data.
Series.base	Return the base object if the memory of the underlying data is shared.

Continued on next page

Table 29 – continued from previous page

Series.T	Return the transpose, which is by definition self.
Series.memory_usage([index, deep])	Return the memory usage of the Series.
Series.hasnans	Return if I have any nans; enables various perf speedups.
Series.flags	
Series.empty	
Series.dtypes	Return the dtype object of the underlying data.
Series.ftypes	Return if the data is sparse|dense.
Series.data	Return the data pointer of the underlying data.
Series.is_copy	Return the copy.
Series.name	Return name of the Series.
Series.put(*args, **kwargs)	Applies the *put* method to its *values* attribute if it has one.

pandas.Series.empty

`Series.empty`

6.3.3 Conversion

Series.astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
Series.infer_objects()	Attempt to infer better dtypes for object columns.
Series.convert_objects([convert_dates, . . .])	(DEPRECATED) Attempt to infer better dtype for object columns.
Series.copy([deep])	Make a copy of this object's indices and data.
Series.bool()	Return the bool of a single element PandasObject.
Series.to_numpy([dtype, copy])	A NumPy ndarray representing the values in this Series or Index.
Series.to_period([freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
Series.to_timestamp([freq, how, copy])	Cast to datetimeindex of timestamps, at *beginning* of period.
Series.to_list()	Return a list of the values.
Series.get_values()	Same as values (but handles sparseness conversions); is a view.
Series.__array__([dtype])	Return the values as a NumPy array.

pandas.Series.__array__

`Series.__array__(dtype=None)`

Return the values as a NumPy array.

Users should not call this directly. Rather, it is invoked by `numpy.array()` and `numpy.asarray()`.

Parameters

dtype [str or numpy.dtype, optional] The dtype to use for the resulting NumPy array. By default, the dtype is inferred from the data.

Returns

numpy.ndarray The values in the series converted to a `numpy.ndarary` with the specified *dtype*.

See also:

pandas.array Create a new array from data.

Series.array Zero-copy view to the array backing the Series.

Series.to_numpy Series method for similar behavior.

Examples

```
>>> ser = pd.Series([1, 2, 3])
>>> np.asarray(ser)
array([1, 2, 3])
```

For timezone-aware data, the timezones may be retained with `dtype='object'`

```
>>> tzser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> np.asarray(tzser, dtype="object")
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or the values may be localized to UTC and the tzinfo discared with `dtype='datetime64[ns]'`

```
>>> np.asarray(tzser, dtype="datetime64[ns]")  # doctest: +ELLIPSIS
array(['1999-12-31T23:00:00.000000000', ...],
      dtype='datetime64[ns]')
```

6.3.4 Indexing, iteration

`Series.get(key[, default])`	Get item from object for given key (DataFrame column, Panel slice, etc.).
`Series.at`	Access a single value for a row/column label pair.
`Series.iat`	Access a single value for a row/column pair by integer position.
`Series.loc`	Access a group of rows and columns by label(s) or a boolean array.
`Series.iloc`	Purely integer-location based indexing for selection by position.
`Series.__iter__()`	Return an iterator of the values.
`Series.iteritems()`	Lazily iterate over (index, value) tuples.
`Series.items()`	Lazily iterate over (index, value) tuples.
`Series.keys()`	Alias for index.
`Series.pop(item)`	Return item and drop from frame.
`Series.item()`	Return the first element of the underlying data as a python scalar.
`Series.xs(key[, axis, level, drop_level])`	Return cross-section from the Series/DataFrame.

pandas.Series.__iter__

`Series.__iter__()`

Return an iterator of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the *indexing documentation*.

6.3.5 Binary operator functions

Series.add(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *add*).
Series.sub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
Series.mul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
Series.div(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
Series.truediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
Series.floordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
Series.mod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *mod*).
Series.pow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
Series.radd(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *radd*).
Series.rsub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
Series.rmul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
Series.rdiv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
Series.rtruediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
Series.rfloordiv(other[, level, fill_value, …])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
Series.rmod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
Series.rpow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
Series.combine(other, func[, fill_value])	Combine the Series with a Series or scalar according to *func*.
Series.combine_first(other)	Combine Series values, choosing the calling Series's values first.
Series.round([decimals])	Round each value in a Series to the given number of decimals.
Series.lt(other[, level, fill_value, axis])	Less than of series and other, element-wise (binary operator *lt*).

Continued on next page

Table 32 – continued from previous page

Series.gt(other[, level, fill_value, axis])	Greater than of series and other, element-wise (binary operator *gt*).
Series.le(other[, level, fill_value, axis])	Less than or equal to of series and other, element-wise (binary operator *le*).
Series.ge(other[, level, fill_value, axis])	Greater than or equal to of series and other, element-wise (binary operator *ge*).
Series.ne(other[, level, fill_value, axis])	Not equal to of series and other, element-wise (binary operator *ne*).
Series.eq(other[, level, fill_value, axis])	Equal to of series and other, element-wise (binary operator *eq*).
Series.product([axis, skipna, level, ...])	Return the product of the values for the requested axis.
Series.dot(other)	Compute the dot product between the Series and the columns of other.

6.3.6 Function application, GroupBy & Window

Series.apply(func[, convert_dtype, args])	Invoke function on values of Series.
Series.agg(func[, axis])	Aggregate using one or more operations over the specified axis.
Series.aggregate(func[, axis])	Aggregate using one or more operations over the specified axis.
Series.transform(func[, axis])	Call `func` on self producing a Series with transformed values and that has the same axis length as self.
Series.map(arg[, na_action])	Map values of Series according to input correspondence.
Series.groupby([by, axis, level, as_index, ...])	Group DataFrame or Series using a mapper or by a Series of columns.
Series.rolling(window[, min_periods, ...])	Provides rolling window calculations.
Series.expanding([min_periods, center, axis])	Provides expanding transformations.
Series.ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions.
Series.pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs).

6.3.7 Computations / Descriptive Stats

Series.abs()	Return a Series/DataFrame with absolute numeric value of each element.
Series.all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
Series.any([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
Series.autocorr([lag])	Compute the lag-N autocorrelation.
Series.between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
Series.clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
Series.clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.
Series.clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.
Series.corr(other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values.

Continued on next page

Table 34 – continued from previous page

Series.count([level])	Return number of non-NA/null observations in the Series.
Series.cov(other[, min_periods])	Compute covariance with Series, excluding missing values.
Series.cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
Series.cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
Series.cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
Series.cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
Series.describe([percentiles, include, exclude])	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
Series.diff([periods])	First discrete difference of element.
Series.factorize([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
Series.kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
Series.mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
Series.max([axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
Series.mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
Series.median([axis, skipna, level, . . .])	Return the median of the values for the requested axis.
Series.min([axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
Series.mode([dropna])	Return the mode(s) of the dataset.
Series.nlargest([n, keep])	Return the largest *n* elements.
Series.nsmallest([n, keep])	Return the smallest *n* elements.
Series.pct_change([periods, fill_method, . . .])	Percentage change between the current and a prior element.
Series.prod([axis, skipna, level, . . .])	Return the product of the values for the requested axis.
Series.quantile([q, interpolation])	Return value at the given quantile.
Series.rank([axis, method, numeric_only, . . .])	Compute numerical data ranks (1 through n) along axis.
Series.sem([axis, skipna, level, ddof, . . .])	Return unbiased standard error of the mean over requested axis.
Series.skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1.
Series.std([axis, skipna, level, ddof, . . .])	Return sample standard deviation over requested axis.
Series.sum([axis, skipna, level, . . .])	Return the sum of the values for the requested axis.
Series.var([axis, skipna, level, ddof, . . .])	Return unbiased variance over requested axis.
Series.kurtosis([axis, skipna, level, . . .])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
Series.unique()	Return unique values of Series object.
Series.nunique([dropna])	Return number of unique elements in the object.
Series.is_unique	Return boolean if values in the object are unique.
Series.is_monotonic	Return boolean if values in the object are monotonic_increasing.

Continued on next page

Table 34 – continued from previous page

Series.is_monotonic_increasing	Return boolean if values in the object are monotonic_increasing.
Series.is_monotonic_decreasing	Return boolean if values in the object are monotonic_decreasing.
Series.value_counts([normalize, sort, …])	Return a Series containing counts of unique values.
Series.compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis.

6.3.8 Reindexing / Selection / Label manipulation

Series.align(other[, join, axis, level, …])	Align two objects on their axes with the specified join method for each axis Index.
Series.drop([labels, axis, index, columns, …])	Return Series with specified index labels removed.
Series.droplevel(level[, axis])	Return DataFrame with requested index / column level(s) removed.
Series.drop_duplicates([keep, inplace])	Return Series with duplicate values removed.
Series.duplicated([keep])	Indicate duplicate Series values.
Series.equals(other)	Test whether two objects contain the same elements.
Series.first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
Series.head([n])	Return the first *n* rows.
Series.idxmax([axis, skipna])	Return the row label of the maximum value.
Series.idxmin([axis, skipna])	Return the row label of the minimum value.
Series.isin(values)	Check whether *values* are contained in Series.
Series.last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
Series.reindex([index])	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
Series.reindex_like(other[, method, copy, …])	Return an object with matching indices as other object.
Series.rename([index])	Alter Series index labels or name.
Series.rename_axis([mapper, index, columns, …])	Set the name of the axis for the index or columns.
Series.reset_index([level, drop, name, inplace])	Generate a new DataFrame or Series with the index reset.
Series.sample([n, frac, replace, weights, …])	Return a random sample of items from an axis of object.
Series.select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
Series.set_axis(labels[, axis, inplace])	Assign desired index to given axis.
Series.take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
Series.tail([n])	Return the last *n* rows.
Series.truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
Series.where(cond[, other, inplace, axis, …])	Replace values where the condition is False.
Series.mask(cond[, other, inplace, axis, …])	Replace values where the condition is True.
Series.add_prefix(prefix)	Prefix labels with string *prefix*.
Series.add_suffix(suffix)	Suffix labels with string *suffix*.
Series.filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.

6.3.9 Missing data handling

Series.isna()	Detect missing values.
Series.notna()	Detect existing (non-missing) values.
Series.dropna([axis, inplace])	Return a new Series with missing values removed.
Series.fillna([value, method, axis, ...])	Fill NA/NaN values using the specified method.
Series.interpolate([method, axis, limit, ...])	Interpolate values according to different methods.

6.3.10 Reshaping, sorting

Series.argsort([axis, kind, order])	Overrides ndarray.argsort.
Series.argmin([axis, skipna])	(DEPRECATED) Return the row label of the minimum value.
Series.argmax([axis, skipna])	(DEPRECATED) Return the row label of the maximum value.
Series.reorder_levels(order)	Rearrange index levels using input order.
Series.sort_values([axis, ascending, ...])	Sort by the values.
Series.sort_index([axis, level, ascending, ...])	Sort Series by index labels.
Series.swaplevel([i, j, copy])	Swap levels i and j in a MultiIndex.
Series.unstack([level, fill_value])	Unstack, a.k.a.
Series.searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
Series.ravel([order])	Return the flattened underlying data as an ndarray.
Series.repeat(repeats[, axis])	Repeat elements of a Series.
Series.squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
Series.view([dtype])	Create a new view of the Series.

6.3.11 Combining / joining / merging

Series.append(to_append[, ignore_index, ...])	Concatenate two or more Series.
Series.replace([to_replace, value, inplace, ...])	Replace values given in *to_replace* with *value*.
Series.update(other)	Modify Series in place using non-NA values from passed Series.

6.3.12 Time series-related

Series.asfreq(freq[, method, how, ...])	Convert TimeSeries to specified frequency.
Series.asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
Series.shift([periods, freq, axis, fill_value])	Shift index by desired number of periods with an optional time *freq*.
Series.first_valid_index()	Return index for first non-NA/null value.
Series.last_valid_index()	Return index for last non-NA/null value.
Series.resample(rule[, how, axis, ...])	Resample time-series data.
Series.tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
Series.tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.
Series.at_time(time[, asof, axis])	Select values at particular time of day (e.g.

Continued on next page

Table 39 – continued from previous page

Series.between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
Series.tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
Series.slice_shift([periods, axis])	Equivalent to *shift* without copying data.

6.3.13 Accessors

Pandas provides dtype-specific methods under various accessors. These are separate namespaces within *Series* that only apply to specific data types.

Data Type	Accessor
Datetime, Timedelta, Period	*dt*
String	*str*
Categorical	*cat*
Sparse	*sparse*

Datetimelike Properties

`Series.dt` can be used to access the values of the series as datetimelike and return several properties. These can be accessed like `Series.dt.<property>`.

Datetime Properties

Series.dt.date	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
Series.dt.time	Returns numpy array of datetime.time.
Series.dt.timetz	Returns numpy array of datetime.time also containing timezone information.
Series.dt.year	The year of the datetime.
Series.dt.month	The month as January=1, December=12.
Series.dt.day	The days of the datetime.
Series.dt.hour	The hours of the datetime.
Series.dt.minute	The minutes of the datetime.
Series.dt.second	The seconds of the datetime.
Series.dt.microsecond	The microseconds of the datetime.
Series.dt.nanosecond	The nanoseconds of the datetime.
Series.dt.week	The week ordinal of the year.
Series.dt.weekofyear	The week ordinal of the year.
Series.dt.dayofweek	The day of the week with Monday=0, Sunday=6.
Series.dt.weekday	The day of the week with Monday=0, Sunday=6.
Series.dt.dayofyear	The ordinal day of the year.
Series.dt.quarter	The quarter of the date.
Series.dt.is_month_start	Indicates whether the date is the first day of the month.
Series.dt.is_month_end	Indicates whether the date is the last day of the month.
Series.dt.is_quarter_start	Indicator for whether the date is the first day of a quarter.
Series.dt.is_quarter_end	Indicator for whether the date is the last day of a quarter.

Continued on next page

Table 40 – continued from previous page

Series.dt.is_year_start	Indicate whether the date is the first day of a year.
Series.dt.is_year_end	Indicate whether the date is the last day of the year.
Series.dt.is_leap_year	Boolean indicator if the date belongs to a leap year.
Series.dt.daysinmonth	The number of days in the month.
Series.dt.days_in_month	The number of days in the month.
Series.dt.tz	Return timezone, if any.
Series.dt.freq	

pandas.Series.dt.date

`Series.dt.date`

Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).

pandas.Series.dt.time

`Series.dt.time`

Returns numpy array of datetime.time. The time part of the Timestamps.

pandas.Series.dt.timetz

`Series.dt.timetz`

Returns numpy array of datetime.time also containing timezone information. The time part of the Timestamps.

pandas.Series.dt.year

`Series.dt.year`

The year of the datetime.

pandas.Series.dt.month

`Series.dt.month`

The month as January=1, December=12.

pandas.Series.dt.day

`Series.dt.day`

The days of the datetime.

pandas.Series.dt.hour

`Series.dt.hour`

The hours of the datetime.

pandas.Series.dt.minute

`Series.dt.minute`
The minutes of the datetime.

pandas.Series.dt.second

`Series.dt.second`
The seconds of the datetime.

pandas.Series.dt.microsecond

`Series.dt.microsecond`
The microseconds of the datetime.

pandas.Series.dt.nanosecond

`Series.dt.nanosecond`
The nanoseconds of the datetime.

pandas.Series.dt.week

`Series.dt.week`
The week ordinal of the year.

pandas.Series.dt.weekofyear

`Series.dt.weekofyear`
The week ordinal of the year.

pandas.Series.dt.dayofweek

`Series.dt.dayofweek`
The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.

Series.dt.weekday Alias.

Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.Series.dt.weekday

`Series.dt.weekday`
The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.

Series.dt.weekday Alias.

Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.Series.dt.dayofyear

`Series.dt.dayofyear`
The ordinal day of the year.

pandas.Series.dt.quarter

`Series.dt.quarter`
The quarter of the date.

pandas.Series.dt.is_month_start

`Series.dt.is_month_start`
Indicates whether the date is the first day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.

is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
dtype: bool
>>> s.dt.is_month_end
0    False
1    True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.Series.dt.is_month_end

`Series.dt.is_month_end`
Indicates whether the date is the last day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.

is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
dtype: bool
>>> s.dt.is_month_end
0    False
1    True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.Series.dt.is_quarter_start

`Series.dt.`**`is_quarter_start`**
Indicator for whether the date is the first day of a quarter.

Returns

is_quarter_start [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.

is_quarter_end Similar property for indicating the quarter start.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                    periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
       dates  quarter  is_quarter_start
0 2017-03-30        1             False
1 2017-03-31        1             False
2 2017-04-01        2              True
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

pandas.Series.dt.is_quarter_end

`Series.dt.`**`is_quarter_end`**

Indicator for whether the date is the last day of a quarter.

Returns

is_quarter_end [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

`quarter` Return the quarter of the date.

`is_quarter_start` Similar property indicating the quarter start.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                     periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
       dates  quarter    is_quarter_end
0 2017-03-30        1             False
1 2017-03-31        1              True
2 2017-04-01        2             False
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

pandas.Series.dt.is_year_start

`Series.dt.is_year_start`
Indicate whether the date is the first day of a year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_end Similar property indicating the last day of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2    True
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

pandas.Series.dt.is_year_end

`Series.dt.is_year_end`
Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

pandas.Series.dt.is_leap_year

`Series.dt.`**`is_leap_year`**

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

Returns

Series or ndarray Booleans indicating if dates belong to a leap year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```
>>> dates = pd.Series(idx)
>>> dates_series
0   2012-12-31
```

(continues on next page)

(continued from previous page)

```
1   2013-12-31
2   2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool
```

pandas.Series.dt.daysinmonth

Series.dt.**daysinmonth**

The number of days in the month.

pandas.Series.dt.days_in_month

Series.dt.**days_in_month**

The number of days in the month.

pandas.Series.dt.tz

Series.dt.**tz**

Return timezone, if any.

Returns

datetime.tzinfo, pytz.tzinfo.BaseTZInfo, dateutil.tz.tz.tzfile, or None Returns None when the array is tz-naive.

pandas.Series.dt.freq

Series.dt.**freq**

Datetime Methods

Series.dt.to_period(*args, **kwargs)	Cast to PeriodArray/Index at a particular frequency.
Series.dt.to_pydatetime()	Return the data as an array of native Python datetime objects.
Series.dt.tz_localize(*args, **kwargs)	Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.
Series.dt.tz_convert(*args, **kwargs)	Convert tz-aware Datetime Array/Index from one time zone to another.
Series.dt.normalize(*args, **kwargs)	Convert times to midnight.
Series.dt.strftime(*args, **kwargs)	Convert to Index using specified date_format.
Series.dt.round(*args, **kwargs)	Perform round operation on the data to the specified *freq*.
Series.dt.floor(*args, **kwargs)	Perform floor operation on the data to the specified *freq*.

Continued on next page

Table 41 – continued from previous page

Series.dt.ceil(*args, **kwargs)	Perform ceil operation on the data to the specified *freq*.
Series.dt.month_name(*args, **kwargs)	Return the month names of the DateTimeIndex with specified locale.
Series.dt.day_name(*args, **kwargs)	Return the day names of the DateTimeIndex with specified locale.

pandas.Series.dt.to_period

Series.dt.**to_period**(**args*, ***kwargs*)

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

Parameters

freq [string or Offset, optional] One of pandas' *offset strings* or an Offset object. Will be inferred by default.

Returns

PeriodArray/Index

Raises

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

See also:

PeriodIndex Immutable ndarray holding ordinal values.

DatetimeIndex.to_pydatetime Return DatetimeIndex as object.

Examples

```
>>> df = pd.DataFrame({"y": [1,2,3]},
...                   index=pd.to_datetime(["2000-03-31 00:00:00",
...                                         "2000-05-31 00:00:00",
...                                         "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

pandas.Series.dt.to_pydatetime

Series.dt.**to_pydatetime**()

Return the data as an array of native Python datetime objects.

Timezone information is retained if present.

Warning: Python's datetime uses microsecond resolution, which is lower than pandas (nanosecond). The values are truncated.

Returns

numpy.ndarray object dtype array containing native Python datetime objects.

See also:

`datetime.datetime` Standard library value for a datetime.

Examples

```
>>> s = pd.Series(pd.date_range('20180310', periods=2))
>>> s
0   2018-03-10
1   2018-03-11
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 11, 0, 0)], dtype=object)
```

pandas' nanosecond precision is truncated to microseconds.

```
>>> s = pd.Series(pd.date_range('20180310', periods=2, freq='ns'))
>>> s
0   2018-03-10 00:00:00.000000000
1   2018-03-10 00:00:00.000000001
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 10, 0, 0)], dtype=object)
```

pandas.Series.dt.tz_localize

`Series.dt.tz_localize(*args, **kwargs)`

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

ambiguous ['infer', 'NaT', bool array, default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

errors [{'raise', 'coerce'}, default None]

- 'raise' will raise a NonExistentTimeError if a timestamp is not valid in the specified time zone (e.g. due to a transition from or to DST time). Use `nonexistent='raise'` instead.
- 'coerce' will return NaT if the timestamp can not be converted to the specified time zone. Use `nonexistent='NaT'` instead.

Deprecated since version 0.24.0.

Returns

result [same type as self] Array/Index converted to the specified time zone.

Raises

TypeError If the Datetime Array/Index is tz-aware and tz is not None.

See also:

`DatetimeIndex.tz_convert` Convert tz-aware DatetimeIndex from one time zone to another.

Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
               '2018-03-02 09:00:00-05:00',
               '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the `tz=None`, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time: >>> s = pd.to_datetime(pd.Series([... '2018-10-28 01:30:00', ... '2018-10-28 02:00:00', ... '2018-10-28 02:30:00', ... '2018-10-28 02:00:00', ... '2018-10-28 02:30:00', ... '2018-10-28 03:00:00', ... '2018-10-28 03:30:00'])) >>> s.dt.tz_localize('CET', ambiguous='infer') 2018-10-28 01:30:00+02:00 0 2018-10-28 02:00:00+02:00 1 2018-10-28 02:30:00+02:00 2 2018-10-28 02:00:00+01:00 3 2018-10-28 02:30:00+01:00 4 2018-10-28 03:00:00+01:00 5 2018-10-28 03:30:00+01:00 6 dtype: int64

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.to_datetime(pd.Series([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.dt.tz_localize('CET', ambiguous=np.array([True, True, False]))
0   2018-10-28 01:20:00+02:00
1   2018-10-28 02:36:00+02:00
2   2018-10-28 03:46:00+01:00
dtype: datetime64[ns, CET]
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.to_datetime(pd.Series([... '2015-03-29 02:30:00', ... '2015-03-29 03:30:00'])) >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 0 2015-03-29 03:00:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 0 2015-03-29 01:59:59.999999999+01:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 0 2015-03-29 03:30:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw']

pandas.Series.dt.tz_convert

`Series.dt.tz_convert(*args, **kwargs)`

Convert tz-aware Datetime Array/Index from one time zone to another.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A *tz* of None will convert to UTC and remove the timezone information.

Returns

normalized [same type as self]

Raises

TypeError If Datetime Array/Index is tz-naive.

See also:

DatetimeIndex.tz A timezone that has a variable offset from UTC.

DatetimeIndex.tz_localize Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.

Examples

With the *tz* parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.date_range(start='2014-08-01 09:00',
...                     freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.date_range(start='2014-08-01 09:00',freq='H',
...                     periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
                dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
                dtype='datetime64[ns]', freq='H')
```

pandas.Series.dt.normalize

`Series.dt.normalize(*args, **kwargs)`
Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on Datetime Array/Index.

Returns

DatetimeArray, DatetimeIndex or Series The same type as the original data. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

floor Floor the datetimes to the specified freq.

ceil Ceil the datetimes to the specified freq.

round Round the datetimes to the specified freq.

Examples

```
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
...                     periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
                dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

pandas.Series.dt.strftime

`Series.dt.strftime(*args, **kwargs)`

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc

Parameters

date_format [str] Date format string (e.g. "%Y-%m-%d").

Returns

Index Index of formatted strings

See also:

to_datetime Convert the given argument to datetime.

DatetimeIndex.normalize Return DatetimeIndex with times to midnight.

DatetimeIndex.round Round the DatetimeIndex to the specified freq.

DatetimeIndex.floor Floor the DatetimeIndex to the specified freq.

Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

pandas.Series.dt.round

`Series.dt.round(*args, **kwargs)`

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.floor

`Series.dt.floor(*args, **kwargs)`

Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.ceil

`Series.dt.ceil(*args, **kwargs)`

Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.month_name

`Series.dt.month_name(*args, **kwargs)`

Return the month names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the month name. Default is English locale.

Returns

Index Index of month names.

Examples

```
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'],
              dtype='datetime64[ns]', freq='M')
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

pandas.Series.dt.day_name

`Series.dt.day_name(*args, **kwargs)`

Return the day names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the day name. Default is English locale.

Returns

Index Index of day names.

Examples

```
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03'],
              dtype='datetime64[ns]', freq='D')
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

Period Properties

Series.dt.qyear
Series.dt.start_time
Series.dt.end_time

pandas.Series.dt.qyear

`Series.dt.qyear`

pandas.Series.dt.start_time

`Series.dt.start_time`

pandas.Series.dt.end_time

`Series.dt.end_time`

Timedelta Properties

Series.dt.days	Number of days for each element.
Series.dt.seconds	Number of seconds (>= 0 and less than 1 day) for each element.
Series.dt.microseconds	Number of microseconds (>= 0 and less than 1 second) for each element.
Series.dt.nanoseconds	Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
Series.dt.components	Return a Dataframe of the components of the Timedeltas.

pandas.Series.dt.days

`Series.dt.days`
Number of days for each element.

pandas.Series.dt.seconds

`Series.dt.seconds`
Number of seconds (>= 0 and less than 1 day) for each element.

pandas.Series.dt.microseconds

`Series.dt.microseconds`
Number of microseconds (>= 0 and less than 1 second) for each element.

pandas.Series.dt.nanoseconds

`Series.dt.nanoseconds`
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

pandas.Series.dt.components

`Series.dt.components`
Return a Dataframe of the components of the Timedeltas.

Returns

DataFrame

Examples

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='s'))
>>> s
0   00:00:00
1   00:00:01
2   00:00:02
3   00:00:03
4   00:00:04
dtype: timedelta64[ns]
>>> s.dt.components
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     0      0        0        0             0             0            0
1     0      0        0        1             0             0            0
2     0      0        0        2             0             0            0
3     0      0        0        3             0             0            0
4     0      0        0        4             0             0            0
```

Timedelta Methods

Series.dt.to_pytimedelta()	Return an array of native *datetime.timedelta* objects.
Series.dt.total_seconds(*args, **kwargs)	Return total duration of each element expressed in seconds.

pandas.Series.dt.to_pytimedelta

Series.dt.**to_pytimedelta**()

Return an array of native *datetime.timedelta* objects.

Python's standard *datetime* library uses a different representation timedelta's. This method converts a Series of pandas Timedeltas to *datetime.timedelta* format with the same length as the original Series.

Returns

a [numpy.ndarray] 1D array containing data with *datetime.timedelta* type.

See also:

datetime.timedelta

Examples

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0   0 days
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.to_pytimedelta()
array([datetime.timedelta(0), datetime.timedelta(1),
```

(continues on next page)

(continued from previous page)

```
       datetime.timedelta(2), datetime.timedelta(3),
       datetime.timedelta(4)], dtype=object)
```

pandas.Series.dt.total_seconds

`Series.dt.total_seconds(*args, **kwargs)`

Return total duration of each element expressed in seconds.

This method is available directly on TimedeltaArray, TimedeltaIndex and on Series containing timedelta values under the `.dt` namespace.

Returns

seconds [[ndarray, Float64Index, Series]] When the calling object is a TimedeltaArray, the return type is ndarray. When the calling object is a TimedeltaIndex, the return type is a Float64Index. When the calling object is a Series, the return type is Series of type *float64* whose index is the same as the original.

See also:

`datetime.timedelta.total_seconds` Standard library version of this method.

`TimedeltaIndex.components` Return a DataFrame with components of each Timedelta.

Examples

Series

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0   0 days
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.total_seconds()
0         0.0
1     86400.0
2    172800.0
3    259200.0
4    345600.0
dtype: float64
```

TimedeltaIndex

```
>>> idx = pd.to_timedelta(np.arange(5), unit='d')
>>> idx
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

```
>>> idx.total_seconds()
Float64Index([0.0, 86400.0, 172800.0, 259200.00000000003, 345600.0],
             dtype='float64')
```

String handling

`Series.str` can be used to access the values of the series as strings and apply several methods to it. These can be accessed like `Series.str.<function/property>`.

Series.str.capitalize()	Convert strings in the Series/Index to be capitalized.
Series.str.cat([others, sep, na_rep, join])	Concatenate strings in the Series/Index with given separator.
Series.str.center(width[, fillchar])	Filling left and right side of strings in the Series/Index with an additional character.
Series.str.contains(pat[, case, flags, na, . . .])	Test if pattern or regex is contained within a string of a Series or Index.
Series.str.count(pat[, flags])	Count occurrences of pattern in each string of the Series/Index.
Series.str.decode(encoding[, errors])	Decode character string in the Series/Index using indicated encoding.
Series.str.encode(encoding[, errors])	Encode character string in the Series/Index using indicated encoding.
Series.str.endswith(pat[, na])	Test if the end of each string element matches a pattern.
Series.str.extract(pat[, flags, expand])	Extract capture groups in the regex *pat* as columns in a DataFrame.
Series.str.extractall(pat[, flags])	For each subject string in the Series, extract groups from all matches of regular expression pat.
Series.str.find(sub[, start, end])	Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].
Series.str.findall(pat[, flags])	Find all occurrences of pattern or regular expression in the Series/Index.
Series.str.get(i)	Extract element from each component at specified position.
Series.str.index(sub[, start, end])	Return lowest indexes in each strings where the substring is fully contained between [start:end].
Series.str.join(sep)	Join lists contained as elements in the Series/Index with passed delimiter.
Series.str.len()	Computes the length of each element in the Series/Index.
Series.str.ljust(width[, fillchar])	Filling right side of strings in the Series/Index with an additional character.
Series.str.lower()	Convert strings in the Series/Index to lowercase.
Series.str.lstrip([to_strip])	Remove leading and trailing characters.
Series.str.match(pat[, case, flags, na])	Determine if each string matches a regular expression.
Series.str.normalize(form)	Return the Unicode normal form for the strings in the Series/Index.
Series.str.pad(width[, side, fillchar])	Pad strings in the Series/Index up to width.
Series.str.partition([sep, expand])	Split the string at the first occurrence of *sep*.
Series.str.repeat(repeats)	Duplicate each string in the Series or Index.
Series.str.replace(pat, repl[, n, case, . . .])	Replace occurrences of pattern/regex in the Series/Index with some other string.
Series.str.rfind(sub[, start, end])	Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].

Continued on next page

Table 45 – continued from previous page

Series.str.rindex(sub[, start, end])	Return highest indexes in each strings where the substring is fully contained between [start:end].
Series.str.rjust(width[, fillchar])	Filling left side of strings in the Series/Index with an additional character.
Series.str.rpartition([sep, expand])	Split the string at the last occurrence of *sep*.
Series.str.rstrip([to_strip])	Remove leading and trailing characters.
Series.str.slice([start, stop, step])	Slice substrings from each element in the Series or Index.
Series.str.slice_replace([start, stop, repl])	Replace a positional slice of a string with another value.
Series.str.split([pat, n, expand])	Split strings around given separator/delimiter.
Series.str.rsplit([pat, n, expand])	Split strings around given separator/delimiter.
Series.str.startswith(pat[, na])	Test if the start of each string element matches a pattern.
Series.str.strip([to_strip])	Remove leading and trailing characters.
Series.str.swapcase()	Convert strings in the Series/Index to be swapcased.
Series.str.title()	Convert strings in the Series/Index to titlecase.
Series.str.translate(table[, deletechars])	Map all characters in the string through the given mapping table.
Series.str.upper()	Convert strings in the Series/Index to uppercase.
Series.str.wrap(width, **kwargs)	Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given width.
Series.str.zfill(width)	Pad strings in the Series/Index by prepending '0' characters.
Series.str.isalnum()	Check whether all characters in each string are alphanumeric.
Series.str.isalpha()	Check whether all characters in each string are alphabetic.
Series.str.isdigit()	Check whether all characters in each string are digits.
Series.str.isspace()	Check whether all characters in each string are whitespace.
Series.str.islower()	Check whether all characters in each string are lowercase.
Series.str.isupper()	Check whether all characters in each string are uppercase.
Series.str.istitle()	Check whether all characters in each string are titlecase.
Series.str.isnumeric()	Check whether all characters in each string are numeric.
Series.str.isdecimal()	Check whether all characters in each string are decimal.
Series.str.get_dummies([sep])	Split each string in the Series by sep and return a frame of dummy/indicator variables.

pandas.Series.str.capitalize

Series.str.**capitalize**()

Convert strings in the Series/Index to be capitalized.

Equivalent to `str.capitalize()`.

Returns

Series/Index of objects

See also:

Series.str.lower Converts all characters to lowercase.

Series.str.upper Converts all characters to uppercase.

Series.str.title Converts first character of each word to uppercase and remaining to lowercase.

Series.str.capitalize Converts first character to uppercase and remaining to lowercase.

Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.cat

`Series.str.cat`(*others=None*, *sep=None*, *na_rep=None*, *join=None*)

Concatenate strings in the Series/Index with given separator.

If *others* is specified, this function concatenates the Series/Index and elements of *others* element-wise. If *others* is not passed, then all values in the Series/Index are concatenated into a single string with a given *sep*.

Parameters

others [Series, Index, DataFrame, np.ndarrary or list-like] Series, Index, DataFrame, np.ndarray (one- or two-dimensional) and other list-likes of strings must have the same length as the calling Series/Index, with the exception of indexed objects (i.e. Series/Index/DataFrame) if *join* is not None.

If others is a list-like that contains a combination of Series, Index or np.ndarray (1-dim), then all elements will be unpacked and must satisfy the above criteria individually.

If others is None, the method returns the concatenation of all strings in the calling Series/Index.

sep [str, default ''] The separator between the different elements/columns. By default the empty string *''* is used.

na_rep [str or None, default None] Representation that is inserted for all missing values:

- If *na_rep* is None, and *others* is None, missing values in the Series/Index are omitted from the result.
- If *na_rep* is None, and *others* is not None, a row containing a missing value in any of the columns (before concatenation) will have a missing value in the result.

join [{'left', 'right', 'outer', 'inner'}, default None] Determines the join-style between the calling Series/Index and any Series/Index/DataFrame in *others* (objects without an index need to match the length of the calling Series/Index). If None, alignment is disabled, but this option will be removed in a future version of pandas and replaced with a default of *'left'*. To disable alignment, use *.values* on any Series/Index/DataFrame in *others*.

New in version 0.23.0.

Returns

concat [str or Series/Index of objects] If *others* is None, *str* is returned, otherwise a *Series/Index* (same type as caller) of objects is returned.

See also:

split Split each string in the Series/Index.

join Join lists contained as elements in the Series/Index.

Examples

When not passing *others*, all values are concatenated into a single string:

```
>>> s = pd.Series(['a', 'b', np.nan, 'd'])
>>> s.str.cat(sep=' ')
'a b d'
```

By default, NA values in the Series are ignored. Using *na_rep*, they can be given a representation:

```
>>> s.str.cat(sep=' ', na_rep='?')
'a b ? d'
```

If *others* is specified, corresponding values are concatenated with the separator. Result will be a Series of strings.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',')
0    a,A
1    b,B
2    NaN
3    d,D
dtype: object
```

Missing values will remain missing in the result, but can again be represented using *na_rep*

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',', na_rep='-')
0    a,A
1    b,B
2    -,C
3    d,D
dtype: object
```

If *sep* is not specified, the values are concatenated without separation.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], na_rep='-')
0    aA
1    bB
2    -C
3    dD
dtype: object
```

Series with different indexes can be aligned before concatenation. The *join*-keyword works as in other methods.

```
>>> t = pd.Series(['d', 'a', 'e', 'c'], index=[3, 0, 4, 2])
>>> s.str.cat(t, join='left', na_rep='-')
0    aa
1    b-
2    -c
3    dd
dtype: object
>>>
>>> s.str.cat(t, join='outer', na_rep='-')
0    aa
1    b-
2    -c
3    dd
4    -e
dtype: object
>>>
>>> s.str.cat(t, join='inner', na_rep='-')
0    aa
2    -c
3    dd
dtype: object
>>>
>>> s.str.cat(t, join='right', na_rep='-')
3    dd
0    aa
```

(continues on next page)

(continued from previous page)

```
4    -e
2    -c
dtype: object
```

For more examples, see *here*.

pandas.Series.str.center

`Series.str.center(width, fillchar=' ')`

Filling left and right side of strings in the Series/Index with an additional character. Equivalent to `str.center()`.

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with `fillchar`

fillchar [str] Additional character for filling, default is whitespace

Returns

filled [Series/Index of objects]

pandas.Series.str.contains

`Series.str.contains(pat, case=True, flags=0, na=nan, regex=True)`

Test if pattern or regex is contained within a string of a Series or Index.

Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

Parameters

pat [str] Character sequence or regular expression.

case [bool, default True] If True, case sensitive.

flags [int, default 0 (no flags)] Flags to pass through to the re module, e.g. re.IGNORECASE.

na [default NaN] Fill value for missing values.

regex [bool, default True] If True, assumes the pat is a regular expression.

If False, treats the pat as a literal string.

Returns

Series or Index of boolean values A Series or Index of boolean values indicating whether the given pattern is contained within the string of each element of the Series or Index.

See also:

match Analogous, but stricter, relying on re.match instead of re.search.

Series.str.startswith Test if the start of each string element matches a pattern.

Series.str.endswith Same as startswith, but tests the end of string.

Examples

Returning a Series of booleans using only a literal pattern.

```
>>> s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s1.str.contains('og', regex=False)
0    False
1     True
2    False
3    False
4      NaN
dtype: object
```

Returning an Index of booleans using only a literal pattern.

```
>>> ind = pd.Index(['Mouse', 'dog', 'house and parrot', '23.0', np.NaN])
>>> ind.str.contains('23', regex=False)
Index([False, False, False, True, nan], dtype='object')
```

Specifying case sensitivity using *case*.

```
>>> s1.str.contains('oG', case=True, regex=True)
0    False
1    False
2    False
3    False
4      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN* replaces NaN values with *False*. If Series or Index does not contain NaN values the resultant dtype will be *bool*, otherwise, an *object* dtype.

```
>>> s1.str.contains('og', na=False, regex=True)
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

Returning 'house' or 'dog' when either expression occurs in a string.

```
>>> s1.str.contains('house|dog', regex=True)
0    False
1     True
2     True
3    False
4      NaN
dtype: object
```

Ignoring case sensitivity using *flags* with regex.

```
>>> import re
>>> s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0    False
1    False
2     True
```

(continues on next page)

(continued from previous page)

```
3    False
4      NaN
dtype: object
```

Returning any digit using regular expression.

```
>>> s1.str.contains('\d', regex=True)
0    False
1    False
2    False
3     True
4      NaN
dtype: object
```

Ensure *pat* is a not a literal pattern when *regex* is set to True. Note in the following example one might expect only *s2[1]* and *s2[3]* to return *True*. However, '.0' as a regex matches any character followed by a 0.

```
>>> s2 = pd.Series(['40','40.0','41','41.0','35'])
>>> s2.str.contains('.0', regex=True)
0     True
1     True
2    False
3     True
4    False
dtype: bool
```

pandas.Series.str.count

`Series.str.count`(*pat*, *flags=0*, ***kwargs*)

Count occurrences of pattern in each string of the Series/Index.

This function is used to count the number of times a particular regex pattern is repeated in each of the string elements of the `Series`.

Parameters

pat [str] Valid regular expression.

flags [int, default 0, meaning no flags] Flags for the *re* module. For a complete list, see here.

****kwargs** For compatibility with other string methods. Not used.

Returns

counts [Series or Index] Same type as the calling object containing the integer counts.

See also:

`re` Standard library module for regular expressions.

`str.count` Standard library version, without regular expression support.

Notes

Some characters need to be escaped when passing in *pat*. eg. `'$'` has a special meaning in regex and must be escaped when finding this literal character.

Examples

```
>>> s = pd.Series(['A', 'B', 'Aaba', 'Baca', np.nan, 'CABA', 'cat'])
>>> s.str.count('a')
0    0.0
1    0.0
2    2.0
3    2.0
4    NaN
5    0.0
6    1.0
dtype: float64
```

Escape `'$'` to find the literal dollar sign.

```
>>> s = pd.Series(['$', 'B', 'Aab$', '$$ca', 'C$B$', 'cat'])
>>> s.str.count('\$')
0    1
1    0
2    1
3    2
4    2
5    0
dtype: int64
```

This is also available on Index

```
>>> pd.Index(['A', 'A', 'Aaba', 'cat']).str.count('a')
Int64Index([0, 0, 2, 1], dtype='int64')
```

pandas.Series.str.decode

`Series.str.decode`(*encoding*, *errors='strict'*)

Decode character string in the Series/Index using indicated encoding. Equivalent to `str.decode()` in python2 and `bytes.decode()` in python3.

Parameters

encoding [str]

errors [str, optional]

Returns

decoded [Series/Index of objects]

pandas.Series.str.encode

`Series.str.encode`(*encoding*, *errors='strict'*)

Encode character string in the Series/Index using indicated encoding. Equivalent to `str.encode()`.

Parameters

encoding [str]

errors [str, optional]

Returns

encoded [Series/Index of objects]

pandas.Series.str.endswith

`Series.str.endswith(pat, na=nan)`

Test if the end of each string element matches a pattern.

Equivalent to `str.endswith()`.

Parameters

pat [str] Character sequence. Regular expressions are not accepted.

na [object, default NaN] Object shown if element tested is not a string.

Returns

Series or Index of bool A Series of booleans indicating whether the given pattern matches the end of each string element.

See also:

`str.endswith` Python standard library string method.

`Series.str.startswith` Same as endswith, but tests the start of string.

`Series.str.contains` Tests if string element contains a pattern.

Examples

```
>>> s = pd.Series(['bat', 'bear', 'caT', np.nan])
>>> s
0     bat
1    bear
2     caT
3     NaN
dtype: object
```

```
>>> s.str.endswith('t')
0     True
1    False
2    False
3      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN*.

```
>>> s.str.endswith('t', na=False)
0     True
1    False
2    False
3    False
dtype: bool
```

pandas.Series.str.extract

`Series.str.extract(pat, flags=0, expand=True)`

Extract capture groups in the regex *pat* as columns in a DataFrame.

For each subject string in the Series, extract groups from the first match of regular expression *pat*.

Parameters

pat [string] Regular expression pattern with capturing groups.

flags [int, default 0 (no flags)] Flags from the `re` module, e.g. `re.IGNORECASE`, that modify regular expression matching for things like case, spaces, etc. For more details, see `re`.

expand [bool, default True] If True, return DataFrame with one column per capture group. If False, return a Series/Index if there is one capture group or DataFrame if there are multiple capture groups.

New in version 0.18.0.

Returns

DataFrame or Series or Index A DataFrame with one row for each subject string, and one column for each group. Any capture group names in regular expression pat will be used for column names; otherwise capture group numbers will be used. The dtype of each result column is always object, even when no match is found. If `expand=False` and pat has only one capture group, then return a Series (if subject is a Series) or Index (if subject is an Index).

See also:

extractall Returns all matches (not just the first match).

Examples

A pattern with two groups will return a DataFrame with two columns. Non-matches will be NaN.

```
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.extract(r'([ab])(\d)')
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

A pattern may contain optional groups.

```
>>> s.str.extract(r'([ab])?(\d)')
     0  1
0    a  1
1    b  2
2  NaN  3
```

Named groups will become column names in the result.

```
>>> s.str.extract(r'(?P<letter>[ab])(?P<digit>\d)')
  letter digit
0      a     1
```

(continues on next page)

(continued from previous page)

```
1      b      2
2    NaN    NaN
```

A pattern with one group will return a DataFrame with one column if expand=True.

```
>>> s.str.extract(r'[ab](\d)', expand=True)
     0
0    1
1    2
2  NaN
```

A pattern with one group will return a Series if expand=False.

```
>>> s.str.extract(r'[ab](\d)', expand=False)
0      1
1      2
2    NaN
dtype: object
```

pandas.Series.str.extractall

`Series.str.extractall(pat, flags=0)`

For each subject string in the Series, extract groups from all matches of regular expression pat. When each subject string in the Series has exactly one match, extractall(pat).xs(0, level='match') is the same as extract(pat).

New in version 0.18.0.

Parameters

pat [str] Regular expression pattern with capturing groups.

flags [int, default 0 (no flags)] A `re` module flag, for example `re.IGNORECASE`. These allow to modify regular expression matching for things like case, spaces, etc. Multiple flags can be combined with the bitwise OR operator, for example `re.IGNORECASE | re.MULTILINE`.

Returns

DataFrame A `DataFrame` with one row for each match, and one column for each group. Its rows have a `MultiIndex` with first levels that come from the subject `Series`. The last level is named 'match' and indexes the matches in each item of the `Series`. Any capture group names in regular expression pat will be used for column names; otherwise capture group numbers will be used.

See also:

extract Returns first match only (not all matches).

Examples

A pattern with one group will return a DataFrame with one column. Indices with no matches will not appear in the result.

```
>>> s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])
>>> s.str.extractall(r"[ab](\d)")
         0
  match
A 0      1
  1      2
B 0      1
```

Capture group names are used for column names of the result.

```
>>> s.str.extractall(r"[ab](?P<digit>\d)")
        digit
  match
A 0         1
  1         2
B 0         1
```

A pattern with two groups will return a DataFrame with two columns.

```
>>> s.str.extractall(r"(?P<letter>[ab])(?P<digit>\d)")
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
```

Optional groups that do not match are NaN in the result.

```
>>> s.str.extractall(r"(?P<letter>[ab])?(?P<digit>\d)")
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
C 0        NaN     1
```

pandas.Series.str.find

`Series.str.find`(*sub*, *start=0*, *end=None*)

Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard `str.find()`.

Parameters

sub [str] Substring being searched

start [int] Left edge index

end [int] Right edge index

Returns

found [Series/Index of integer values]

See also:

rfind Return highest indexes in each strings.

pandas.Series.str.findall

`Series.str.findall(pat, flags=0, **kwargs)`

Find all occurrences of pattern or regular expression in the Series/Index.

Equivalent to applying `re.findall()` to all the elements in the Series/Index.

Parameters

pat [string] Pattern or regular expression.

flags [int, default 0] `re` module flags, e.g. *re.IGNORECASE* (default is 0, which means no flags).

Returns

Series/Index of lists of strings All non-overlapping matches of pattern or regular expression in each string of this Series/Index.

See also:

count Count occurrences of pattern or regular expression in each string of the Series/Index.

extractall For each string in the Series, extract groups from all matches of regular expression and return a DataFrame with one row for each match and one column for each group.

re.findall The equivalent `re` function to all non-overlapping matches of pattern or regular expression in string, as a list of strings.

Examples

```
>>> s = pd.Series(['Lion', 'Monkey', 'Rabbit'])
```

The search for the pattern 'Monkey' returns one match:

```
>>> s.str.findall('Monkey')
0          []
1    [Monkey]
2          []
dtype: object
```

On the other hand, the search for the pattern 'MONKEY' doesn't return any match:

```
>>> s.str.findall('MONKEY')
0    []
1    []
2    []
dtype: object
```

Flags can be added to the pattern or regular expression. For instance, to find the pattern 'MONKEY' ignoring the case:

```
>>> import re
>>> s.str.findall('MONKEY', flags=re.IGNORECASE)
0          []
1    [Monkey]
2          []
dtype: object
```

When the pattern matches more than one string in the Series, all matches are returned:

```
>>> s.str.findall('on')
0    [on]
1    [on]
2      []
dtype: object
```

Regular expressions are supported too. For instance, the search for all the strings ending with the word 'on' is shown next:

```
>>> s.str.findall('on$')
0    [on]
1      []
2      []
dtype: object
```

If the pattern is found more than once in the same string, then a list of multiple strings is returned:

```
>>> s.str.findall('b')
0        []
1        []
2    [b, b]
dtype: object
```

pandas.Series.str.get

`Series.str.get(i)`

Extract element from each component at specified position.

Extract element from lists, tuples, or strings in each element in the Series/Index.

Parameters

i [int] Position of element to extract.

Returns

items [Series/Index of objects]

Examples

```
>>> s = pd.Series(["String",
           (1, 2, 3),
           ["a", "b", "c"],
           123, -456,
           {1:"Hello", "2":"World"}])
>>> s
0                        String
1                     (1, 2, 3)
2                     [a, b, c]
3                           123
4                          -456
5    {1: 'Hello', '2': 'World'}
dtype: object
```

```
>>> s.str.get(1)
0        t
1        2
2        b
3      NaN
4      NaN
5    Hello
dtype: object
```

```
>>> s.str.get(-1)
0      g
1      3
2      c
3    NaN
4    NaN
5    NaN
dtype: object
```

pandas.Series.str.index

`Series.str.`**`index`**(*sub*, *start=0*, *end=None*)

Return lowest indexes in each strings where the substring is fully contained between [start:end]. This is the same as `str.find` except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard `str.index`.

Parameters

sub [str] Substring being searched

start [int] Left edge index

end [int] Right edge index

Returns

found [Series/Index of objects]

See also:

`rindex` Return highest indexes in each strings.

pandas.Series.str.join

`Series.str.`**`join`**(*sep*)

Join lists contained as elements in the Series/Index with passed delimiter.

If the elements of a Series are lists themselves, join the content of these lists using the delimiter passed to the function. This function is an equivalent to `str.join()`.

Parameters

sep [str] Delimiter to use between list entries.

Returns

Series/Index: object The list entries concatenated by intervening occurrences of the delimiter.

Raises

AttributeError If the supplied Series contains neither strings nor lists.

See also:

`str.join` Standard library version of this method.

`Series.str.split` Split strings around given separator/delimiter.

Notes

If any of the list items is not a string object, the result of the join will be *NaN*.

Examples

Example with a list that contains non-string elements.

```
>>> s = pd.Series([['lion', 'elephant', 'zebra'],
...                [1.1, 2.2, 3.3],
...                ['cat', np.nan, 'dog'],
...                ['cow', 4.5, 'goat'],
...                ['duck', ['swan', 'fish'], 'guppy']])
>>> s
0        [lion, elephant, zebra]
1                [1.1, 2.2, 3.3]
2                [cat, nan, dog]
3               [cow, 4.5, goat]
4    [duck, [swan, fish], guppy]
dtype: object
```

Join all lists using a '-'. The lists containing object(s) of types other than str will produce a NaN.

```
>>> s.str.join('-')
0    lion-elephant-zebra
1                    NaN
2                    NaN
3                    NaN
4                    NaN
dtype: object
```

pandas.Series.str.len

`Series.str.len()`

Computes the length of each element in the Series/Index. The element may be a sequence (such as a string, tuple or list) or a collection (such as a dictionary).

Returns

Series or Index of int A Series or Index of integer values indicating the length of each element in the Series or Index.

See also:

`str.len` Python built-in function returning the length of an object.

`Series.size` Returns the length of the Series.

Examples

Returns the length (number of characters) in a string. Returns the number of entries for dictionaries, lists or tuples.

```
>>> s = pd.Series(['dog',
...                 '',
...                 5,
...                 {'foo' : 'bar'},
...                 [2, 3, 5, 7],
...                 ('one', 'two', 'three')])
>>> s
0                  dog
1
2                    5
3       {'foo': 'bar'}
4         [2, 3, 5, 7]
5    (one, two, three)
dtype: object
>>> s.str.len()
0    3.0
1    0.0
2    NaN
3    1.0
4    4.0
5    3.0
dtype: float64
```

pandas.Series.str.ljust

`Series.str.ljust(width, fillchar=' ')`

Filling right side of strings in the Series/Index with an additional character. Equivalent to `str.ljust()`.

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with `fillchar`

fillchar [str] Additional character for filling, default is whitespace

Returns

filled [Series/Index of objects]

pandas.Series.str.lower

`Series.str.lower()`

Convert strings in the Series/Index to lowercase.

Equivalent to `str.lower()`.

Returns

Series/Index of objects

See also:

Series.str.lower Converts all characters to lowercase.

Series.str.upper Converts all characters to uppercase.

Series.str.title Converts first character of each word to uppercase and remaining to lowercase.

Series.str.capitalize Converts first character to uppercase and remaining to lowercase.

Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.lstrip

`Series.str.lstrip(to_strip=None)`

Remove leading and trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left side. Equivalent to `str.lstrip()`.

Parameters

to_strip [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

Returns

Series/Index of objects

See also:

Series.str.strip Remove leading and trailing characters in Series/Index.

Series.str.lstrip Remove leading characters in Series/Index.

Series.str.rstrip Remove trailing characters in Series/Index.

Examples

```
>>> s = pd.Series(['1. Ant.  ', '2. Bee!\n', '3. Cat?\t', np.nan])
>>> s
0    1. Ant.
1    2. Bee!\n
2    3. Cat?\t
3          NaN
dtype: object
```

```
>>> s.str.strip()
0    1. Ant.
1    2. Bee!
2    3. Cat?
3        NaN
dtype: object
```

```
>>> s.str.lstrip('123.')
0    Ant.
1    Bee!\n
2    Cat?\t
3       NaN
dtype: object
```

```
>>> s.str.rstrip('.!? \n\t')
0    1. Ant
1    2. Bee
2    3. Cat
3       NaN
dtype: object
```

```
>>> s.str.strip('123.!? \n\t')
0    Ant
1    Bee
2    Cat
3    NaN
dtype: object
```

pandas.Series.str.match

`Series.str.match(pat, case=True, flags=0, na=nan)`
Determine if each string matches a regular expression.

Parameters

pat [string] Character sequence or regular expression

case [boolean, default True] If True, case sensitive

flags [int, default 0 (no flags)] re module flags, e.g. re.IGNORECASE

na [default NaN, fill value for missing values]

Returns

Series/array of boolean values

See also:

contains Analogous, but less strict, relying on re.search instead of re.match.

extract Extract matched groups.

pandas.Series.str.normalize

`Series.str.normalize(form)`
Return the Unicode normal form for the strings in the Series/Index. For more information on the forms, see the `unicodedata.normalize()`.

Parameters

form [{'NFC', 'NFKC', 'NFD', 'NFKD'}] Unicode form

Returns

normalized [Series/Index of objects]

pandas.Series.str.pad

`Series.str.pad(width, side='left', fillchar=' ')`
Pad strings in the Series/Index up to width.

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with character defined in *fillchar*.

side [{'left', 'right', 'both'}, default 'left'] Side from which to fill resulting string.

fillchar [str, default ' '] Additional character for filling, default is whitespace.

Returns

Series or Index of object Returns Series or Index with minimum number of char in object.

See also:

Series.str.rjust Fills the left side of strings with an arbitrary character. Equivalent to `Series.str.pad(side='left')`.

Series.str.ljust Fills the right side of strings with an arbitrary character. Equivalent to `Series.str.pad(side='right')`.

Series.str.center Fills boths sides of strings with an arbitrary character. Equivalent to `Series.str.pad(side='both')`.

Series.str.zfill Pad strings in the Series/Index by prepending '0' character. Equivalent to `Series.str.pad(side='left', fillchar='0')`.

Examples

```
>>> s = pd.Series(["caribou", "tiger"])
>>> s
0    caribou
1      tiger
dtype: object
```

```
>>> s.str.pad(width=10)
0       caribou
1         tiger
dtype: object
```

```
>>> s.str.pad(width=10, side='right', fillchar='-')
0    caribou---
1    tiger-----
dtype: object
```

```
>>> s.str.pad(width=10, side='both', fillchar='-')
0    -caribou--
1    --tiger---
dtype: object
```

pandas.Series.str.partition

`Series.str.`**`partition`**(*sep=' '*, *expand=True*)

Split the string at the first occurrence of *sep*.

This method splits the string at the first occurrence of *sep*, and returns 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing the string itself, followed by two empty strings.

Parameters

sep [str, default whitespace] String to split on.

pat [str, default whitespace] Deprecated since version 0.24.0: Use `sep` instead

expand [bool, default True] If True, return DataFrame/MultiIndex expanding dimensionality. If False, return Series/Index.

Returns

DataFrame/MultiIndex or Series/Index of objects

See also:

rpartition Split the string at the last occurrence of *sep*.

Series.str.split Split strings around given separators.

str.partition Standard library version.

Examples

```
>>> s = pd.Series(['Linda van der Berg', 'George Pitt-Rivers'])
>>> s
0     Linda van der Berg
1     George Pitt-Rivers
dtype: object
```

```
>>> s.str.partition()
        0  1             2
0   Linda     van der Berg
1  George      Pitt-Rivers
```

To partition by the last space instead of the first one:

```
>>> s.str.rpartition()
               0  1            2
0  Linda van der            Berg
1         George     Pitt-Rivers
```

To partition by something different than a space:

```
>>> s.str.partition('-')
                    0  1       2
0  Linda van der Berg
1         George Pitt  -  Rivers
```

To return a Series containining tuples instead of a DataFrame:

```
>>> s.str.partition('-', expand=False)
0     (Linda van der Berg, , )
1     (George Pitt, -, Rivers)
dtype: object
```

Also available on indices:

```
>>> idx = pd.Index(['X 123', 'Y 999'])
>>> idx
Index(['X 123', 'Y 999'], dtype='object')
```

Which will create a MultiIndex:

```
>>> idx.str.partition()
MultiIndex(levels=[['X', 'Y'], [' '], ['123', '999']],
           codes=[[0, 1], [0, 0], [0, 1]])
```

Or an index with tuples with `expand=False`:

```
>>> idx.str.partition(expand=False)
Index([('X', ' ', '123'), ('Y', ' ', '999')], dtype='object')
```

pandas.Series.str.repeat

`Series.str.repeat(repeats)`

Duplicate each string in the Series or Index.

Parameters

repeats [int or sequence of int] Same value for all (int) or different value per (sequence).

Returns

Series or Index of object Series or Index of repeated string objects specified by input parameter repeats.

Examples

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
```

Single int repeats string in Series

```
>>> s.str.repeat(repeats=2)
0    aa
1    bb
2    cc
```

Sequence of int repeats corresponding string in Series

```
>>> s.str.repeat(repeats=[1, 2, 3])
0      a
1     bb
2    ccc
```

pandas.Series.str.replace

`Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=True)`

Replace occurrences of pattern/regex in the Series/Index with some other string. Equivalent to `str.replace()` or `re.sub()`.

Parameters

pat [string or compiled regex] String can be a character sequence or regular expression.

New in version 0.20.0: *pat* also accepts a compiled regex.

repl [string or callable] Replacement string or a callable. The callable is passed the regex match object and must return a replacement string to be used. See `re.sub()`.

New in version 0.20.0: *repl* also accepts a callable.

n [int, default -1 (all)] Number of replacements to make from start

case [boolean, default None]

- If True, case sensitive (the default if *pat* is a string)
- Set to False for case insensitive
- Cannot be set if *pat* is a compiled regex

flags [int, default 0 (no flags)]

- re module flags, e.g. re.IGNORECASE
- Cannot be set if *pat* is a compiled regex

regex [boolean, default True]

- If True, assumes the passed-in pattern is a regular expression.
- If False, treats the pattern as a literal string
- Cannot be set to False if *pat* is a compiled regex or *repl* is a callable.

New in version 0.23.0.

Returns

Series or Index of object A copy of the object with all matching occurrences of *pat* replaced by *repl*.

Raises

ValueError

- if *regex* is False and *repl* is a callable or *pat* is a compiled regex
- if *pat* is a compiled regex and *case* or *flags* is set

Notes

When *pat* is a compiled regex, all flags should be included in the compiled regex. Use of *case*, *flags*, or *regex=False* with a compiled regex will raise an error.

Examples

When *pat* is a string and *regex* is True (the default), the given *pat* is compiled as a regex. When *repl* is a string, it replaces matching regex patterns as with `re.sub()`. NaN value(s) in the Series are left as is:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f.', 'ba', regex=True)
0    bao
1    baz
2    NaN
dtype: object
```

When *pat* is a string and *regex* is False, every *pat* is replaced with *repl* as with `str.replace()`:

```
>>> pd.Series(['f.o', 'fuz', np.nan]).str.replace('f.', 'ba', regex=False)
0    bao
1    fuz
2    NaN
dtype: object
```

When *repl* is a callable, it is called on every *pat* using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

To get the idea:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f', repr)
0    <_sre.SRE_Match object; span=(0, 1), match='f'>oo
1    <_sre.SRE_Match object; span=(0, 1), match='f'>uz
2                                                  NaN
dtype: object
```

Reverse every lowercase alphabetic word:

```
>>> repl = lambda m: m.group(0)[::-1]
>>> pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(r'[a-z]+', repl)
0    oof 123
1    rab zab
2        NaN
dtype: object
```

Using regex groups (extract second group and swap case):

```
>>> pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
>>> repl = lambda m: m.group('two').swapcase()
>>> pd.Series(['One Two Three', 'Foo Bar Baz']).str.replace(pat, repl)
0    tWO
1    bAR
dtype: object
```

Using a compiled regex with flags

```
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar')
0    foo
1    bar
2    NaN
dtype: object
```

pandas.Series.str.rfind

`Series.str.`**`rfind`**`(`*sub*, *start=0*, *end=None*`)`

Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard `str.rfind()`.

Parameters

sub [str] Substring being searched

start [int] Left edge index

end [int] Right edge index

Returns

found [Series/Index of integer values]

See also:

find Return lowest indexes in each strings.

pandas.Series.str.rindex

`Series.str.rindex(sub, start=0, end=None)`

Return highest indexes in each strings where the substring is fully contained between [start:end]. This is the same as `str.rfind` except instead of returning -1, it raises a ValueError when the substring is not found. Equivalent to standard `str.rindex`.

Parameters

sub [str] Substring being searched

start [int] Left edge index

end [int] Right edge index

Returns

found [Series/Index of objects]

See also:

index Return lowest indexes in each strings.

pandas.Series.str.rjust

`Series.str.rjust(width, fillchar=' ')`

Filling left side of strings in the Series/Index with an additional character. Equivalent to `str.rjust()`.

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with `fillchar`

fillchar [str] Additional character for filling, default is whitespace

Returns

filled [Series/Index of objects]

pandas.Series.str.rpartition

`Series.str.rpartition(sep=' ', expand=True)`

Split the string at the last occurrence of *sep*.

This method splits the string at the last occurrence of *sep*, and returns 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing two empty strings, followed by the string itself.

Parameters

sep [str, default whitespace] String to split on.

pat [str, default whitespace] Deprecated since version 0.24.0: Use `sep` instead

expand [bool, default True] If True, return DataFrame/MultiIndex expanding dimensionality. If False, return Series/Index.

Returns

DataFrame/MultiIndex or Series/Index of objects

See also:

partition Split the string at the first occurrence of *sep*.

Series.str.split Split strings around given separators.

str.partition Standard library version.

Examples

```
>>> s = pd.Series(['Linda van der Berg', 'George Pitt-Rivers'])
>>> s
0    Linda van der Berg
1    George Pitt-Rivers
dtype: object
```

```
>>> s.str.partition()
        0  1             2
0   Linda     van der Berg
1  George      Pitt-Rivers
```

To partition by the last space instead of the first one:

```
>>> s.str.rpartition()
               0  1            2
0  Linda van der            Berg
1         George     Pitt-Rivers
```

To partition by something different than a space:

```
>>> s.str.partition('-')
                    0  1       2
0  Linda van der Berg
1         George Pitt  -  Rivers
```

To return a Series containining tuples instead of a DataFrame:

```
>>> s.str.partition('-', expand=False)
0    (Linda van der Berg, , )
1    (George Pitt, -, Rivers)
dtype: object
```

Also available on indices:

```
>>> idx = pd.Index(['X 123', 'Y 999'])
>>> idx
Index(['X 123', 'Y 999'], dtype='object')
```

Which will create a MultiIndex:

```
>>> idx.str.partition()
MultiIndex(levels=[['X', 'Y'], [' '], ['123', '999']],
           codes=[[0, 1], [0, 0], [0, 1]])
```

Or an index with tuples with `expand=False`:

```
>>> idx.str.partition(expand=False)
Index([('X', ' ', '123'), ('Y', ' ', '999')], dtype='object')
```

pandas.Series.str.rstrip

`Series.str.rstrip(to_strip=None)`

Remove leading and trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from right side. Equivalent to `str.rstrip()`.

Parameters

to_strip [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

Returns

Series/Index of objects

See also:

Series.str.strip Remove leading and trailing characters in Series/Index.

Series.str.lstrip Remove leading characters in Series/Index.

Series.str.rstrip Remove trailing characters in Series/Index.

Examples

```
>>> s = pd.Series(['1. Ant.  ', '2. Bee!\n', '3. Cat?\t', np.nan])
>>> s
0    1. Ant.
1    2. Bee!\n
2    3. Cat?\t
3          NaN
dtype: object
```

```
>>> s.str.strip()
0    1. Ant.
1    2. Bee!
2    3. Cat?
3        NaN
dtype: object
```

```
>>> s.str.lstrip('123.')
0    Ant.
1    Bee!\n
2    Cat?\t
3       NaN
dtype: object
```

```
>>> s.str.rstrip('.!? \n\t')
0    1. Ant
1    2. Bee
2    3. Cat
3       NaN
dtype: object
```

```
>>> s.str.strip('123.!? \n\t')
0    Ant
1    Bee
2    Cat
3    NaN
dtype: object
```

pandas.Series.str.slice

`Series.str.slice`(*start=None*, *stop=None*, *step=None*)

Slice substrings from each element in the Series or Index.

Parameters

start [int, optional] Start position for slice operation.

stop [int, optional] Stop position for slice operation.

step [int, optional] Step size for slice operation.

Returns

Series or Index of object Series or Index from sliced substring from original string object.

See also:

`Series.str.slice_replace` Replace a slice with a string.

`Series.str.get` Return element at position. Equivalent to *Series.str.slice(start=i, stop=i+1)* with *i* being the position.

Examples

```
>>> s = pd.Series(["koala", "fox", "chameleon"])
>>> s
0        koala
1          fox
2    chameleon
dtype: object
```

```
>>> s.str.slice(start=1)
0        oala
1          ox
2    hameleon
dtype: object
```

```
>>> s.str.slice(stop=2)
0    ko
1    fo
2    ch
dtype: object
```

```
>>> s.str.slice(step=2)
0      kaa
1       fx
2    caeen
dtype: object
```

```
>>> s.str.slice(start=0, stop=5, step=3)
0    kl
1     f
2    cm
dtype: object
```

Equivalent behaviour to:

```
>>> s.str[0:5:3]
0    kl
1     f
2    cm
dtype: object
```

pandas.Series.str.slice_replace

`Series.str.`**`slice_replace`**(*start=None*, *stop=None*, *repl=None*)

Replace a positional slice of a string with another value.

Parameters

start [int, optional] Left index position to use for the slice. If not specified (None), the slice is unbounded on the left, i.e. slice from the start of the string.

stop [int, optional] Right index position to use for the slice. If not specified (None), the slice is unbounded on the right, i.e. slice until the end of the string.

repl [str, optional] String for replacement. If not specified (None), the sliced region is replaced with an empty string.

Returns

replaced [Series or Index] Same type as the original object.

See also:

`Series.str.slice` Just slicing without replacement.

Examples

```
>>> s = pd.Series(['a', 'ab', 'abc', 'abdc', 'abcde'])
>>> s
0        a
1       ab
2      abc
3     abdc
4    abcde
dtype: object
```

Specify just *start*, meaning replace *start* until the end of the string with *repl*.

```
>>> s.str.slice_replace(1, repl='X')
0    aX
1    aX
2    aX
3    aX
4    aX
dtype: object
```

Specify just *stop*, meaning the start of the string to *stop* is replaced with *repl*, and the rest of the string is included.

```
>>> s.str.slice_replace(stop=2, repl='X')
0       X
1       X
2      Xc
3     Xdc
4    Xcde
dtype: object
```

Specify *start* and *stop*, meaning the slice from *start* to *stop* is replaced with *repl*. Everything before or after *start* and *stop* is included as is.

```
>>> s.str.slice_replace(start=1, stop=3, repl='X')
0      aX
1      aX
2      aX
3     aXc
4    aXde
dtype: object
```

pandas.Series.str.split

`Series.str.split`(*pat=None*, *n=-1*, *expand=False*)

Split strings around given separator/delimiter.

Splits the string in the Series/Index from the beginning, at the specified delimiter string. Equivalent to `str.split()`.

Parameters

pat [str, optional] String or regular expression to split on. If not specified, split on whitespace.

n [int, default -1 (all)] Limit number of splits in output. `None`, 0 and -1 will be interpreted as return all splits.

expand [bool, default False] Expand the splitted strings into separate columns.

- If `True`, return DataFrame/MultiIndex expanding dimensionality.
- If `False`, return Series/Index, containing lists of strings.

Returns

Series, Index, DataFrame or MultiIndex Type matches caller unless `expand=True` (see Notes).

See also:

Series.str.split Split strings around given separator/delimiter.

Series.str.rsplit Splits string around given separator/delimiter, starting from the right.

Series.str.join Join lists contained as elements in the Series/Index with passed delimiter.

str.split Standard library version for split.

str.rsplit Standard library version for rsplit.

Notes

The handling of the *n* keyword depends on the number of found splits:

- If found splits > *n*, make first *n* splits only
- If found splits <= *n*, make all splits
- If for a certain row the number of found splits < *n*, append *None* for padding up to *n* if `expand=True`

If using `expand=True`, Series and Index callers return DataFrame and MultiIndex objects, respectively.

Examples

```
>>> s = pd.Series(["this is a regular sentence",
"https://docs.python.org/3/tutorial/index.html", np.nan])
```

In the default setting, the string is split by whitespace.

```
>>> s.str.split()
0                   [this, is, a, regular, sentence]
1     [https://docs.python.org/3/tutorial/index.html]
2                                                 NaN
dtype: object
```

Without the *n* parameter, the outputs of *rsplit* and *split* are identical.

```
>>> s.str.rsplit()
0                   [this, is, a, regular, sentence]
1     [https://docs.python.org/3/tutorial/index.html]
2                                                 NaN
dtype: object
```

The *n* parameter can be used to limit the number of splits on the delimiter. The outputs of *split* and *rsplit* are different.

```
>>> s.str.split(n=2)
0                     [this, is, a regular sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

```
>>> s.str.rsplit(n=2)
0                     [this is a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

The *pat* parameter can be used to split by other characters.

```
>>> s.str.split(pat = "/")
0                         [this is a regular sentence]
1    [https:, , docs.python.org, 3, tutorial, index...
2                                                  NaN
dtype: object
```

When using `expand=True`, the split elements will expand out into separate columns. If NaN is present, it is propagated throughout the columns during the split.

```
>>> s.str.split(expand=True)
                                               0     1     2        3
0                                           this    is     a  regular
1  https://docs.python.org/3/tutorial/index.html  None  None     None
2                                            NaN   NaN   NaN      NaN
             4
0     sentence
1         None
2          NaN
```

For slightly more complex use cases like splitting the html document name from a url, a combination of parameter settings can be used.

```
>>> s.str.rsplit("/", n=1, expand=True)
                                    0           1
0          this is a regular sentence        None
1  https://docs.python.org/3/tutorial  index.html
2                                 NaN         NaN
```

pandas.Series.str.rsplit

`Series.str.rsplit(pat=None, n=-1, expand=False)`

Split strings around given separator/delimiter.

Splits the string in the Series/Index from the end, at the specified delimiter string. Equivalent to `str.rsplit()`.

Parameters

pat [str, optional] String or regular expression to split on. If not specified, split on whitespace.

n [int, default -1 (all)] Limit number of splits in output. `None`, 0 and -1 will be interpreted as return all splits.

expand [bool, default False] Expand the splitted strings into separate columns.

- If `True`, return DataFrame/MultiIndex expanding dimensionality.
- If `False`, return Series/Index, containing lists of strings.

Returns

Series, Index, DataFrame or MultiIndex Type matches caller unless `expand=True` (see Notes).

See also:

Series.str.split Split strings around given separator/delimiter.

Series.str.rsplit Splits string around given separator/delimiter, starting from the right.

Series.str.join Join lists contained as elements in the Series/Index with passed delimiter.

str.split Standard library version for split.

str.rsplit Standard library version for rsplit.

Notes

The handling of the *n* keyword depends on the number of found splits:

- If found splits > *n*, make first *n* splits only
- If found splits <= *n*, make all splits
- If for a certain row the number of found splits < *n*, append *None* for padding up to *n* if `expand=True`

If using `expand=True`, Series and Index callers return DataFrame and MultiIndex objects, respectively.

Examples

```
>>> s = pd.Series(["this is a regular sentence",
"https://docs.python.org/3/tutorial/index.html", np.nan])
```

In the default setting, the string is split by whitespace.

```
>>> s.str.split()
0                   [this, is, a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

Without the *n* parameter, the outputs of *rsplit* and *split* are identical.

```
>>> s.str.rsplit()
0                   [this, is, a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

The *n* parameter can be used to limit the number of splits on the delimiter. The outputs of *split* and *rsplit* are different.

```
>>> s.str.split(n=2)
0                     [this, is, a regular sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

```
>>> s.str.rsplit(n=2)
0                     [this is a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                                NaN
dtype: object
```

The *pat* parameter can be used to split by other characters.

```
>>> s.str.split(pat = "/")
0                         [this is a regular sentence]
1    [https:, , docs.python.org, 3, tutorial, index...
2                                                  NaN
dtype: object
```

When using `expand=True`, the split elements will expand out into separate columns. If NaN is present, it is propagated throughout the columns during the split.

```
>>> s.str.split(expand=True)
                                               0     1     2        3
0                                           this    is     a  regular
1  https://docs.python.org/3/tutorial/index.html  None  None     None
2                                            NaN   NaN   NaN      NaN
          4
0  sentence
1      None
2       NaN
```

For slightly more complex use cases like splitting the html document name from a url, a combination of parameter settings can be used.

```
>>> s.str.rsplit("/", n=1, expand=True)
                                    0           1
0          this is a regular sentence        None
1  https://docs.python.org/3/tutorial  index.html
2                                 NaN         NaN
```

pandas.Series.str.startswith

`Series.str.`**`startswith`**(*pat*, *na=nan*)

Test if the start of each string element matches a pattern.

Equivalent to `str.startswith()`.

Parameters

pat [str] Character sequence. Regular expressions are not accepted.

na [object, default NaN] Object shown if element tested is not a string.

Returns

Series or Index of bool A Series of booleans indicating whether the given pattern matches the start of each string element.

See also:

`str.startswith` Python standard library string method.

Series.str.endswith Same as startswith, but tests the end of string.

Series.str.contains Tests if string element contains a pattern.

Examples

```
>>> s = pd.Series(['bat', 'Bear', 'cat', np.nan])
>>> s
0     bat
1    Bear
2     cat
3     NaN
dtype: object
```

```
>>> s.str.startswith('b')
0     True
1    False
2    False
3      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN*.

```
>>> s.str.startswith('b', na=False)
0     True
1    False
2    False
3    False
dtype: bool
```

pandas.Series.str.strip

`Series.str.strip(to_strip=None)`

Remove leading and trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left and right sides. Equivalent to `str.strip()`.

Parameters

to_strip [str or None, default None] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

Returns

Series/Index of objects

See also:

Series.str.strip Remove leading and trailing characters in Series/Index.

Series.str.lstrip Remove leading characters in Series/Index.

Series.str.rstrip Remove trailing characters in Series/Index.

Examples

```
>>> s = pd.Series(['1. Ant.  ', '2. Bee!\n', '3. Cat?\t', np.nan])
>>> s
0    1. Ant.
1    2. Bee!\n
2    3. Cat?\t
3          NaN
dtype: object
```

```
>>> s.str.strip()
0    1. Ant.
1    2. Bee!
2    3. Cat?
3        NaN
dtype: object
```

```
>>> s.str.lstrip('123.')
0    Ant.
1    Bee!\n
2    Cat?\t
3       NaN
dtype: object
```

```
>>> s.str.rstrip('.!? \n\t')
0    1. Ant
1    2. Bee
2    3. Cat
3       NaN
dtype: object
```

```
>>> s.str.strip('123.!? \n\t')
0    Ant
1    Bee
2    Cat
3    NaN
dtype: object
```

pandas.Series.str.swapcase

`Series.str.swapcase()`

Convert strings in the Series/Index to be swapcased.

Equivalent to `str.swapcase()`.

Returns

Series/Index of objects

See also:

Series.str.lower Converts all characters to lowercase.

Series.str.upper Converts all characters to uppercase.

Series.str.title Converts first character of each word to uppercase and remaining to lowercase.

Series.str.capitalize Converts first character to uppercase and remaining to lowercase.

Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.title

`Series.str.title()`

Convert strings in the Series/Index to titlecase.

Equivalent to `str.title()`.

Returns

Series/Index of objects

See also:

Series.str.lower Converts all characters to lowercase.

Series.str.upper Converts all characters to uppercase.

Series.str.title Converts first character of each word to uppercase and remaining to lowercase.

Series.str.capitalize Converts first character to uppercase and remaining to lowercase.

Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
```

(continues on next page)

(continued from previous page)

```
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.translate

`Series.str.translate(table, deletechars=None)`

Map all characters in the string through the given mapping table. Equivalent to standard `str.translate()`. Note that the optional argument deletechars is only valid if you are using python 2. For python 3, character deletion should be specified via the table argument.

Parameters

table [dict (python 3), str or None (python 2)] In python 3, table is a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. `str.maketrans()` is a helper function for making translation tables. In python 2, table is either a string of length 256 or None. If the table argument is None, no translation is applied and the operation simply removes the characters in deletechars. `string.maketrans()` is a helper function for making translation tables.

deletechars [str, optional (python 2)] A string of characters to delete. This argument is only valid in python 2.

Returns

translated [Series/Index of objects]

pandas.Series.str.upper

`Series.str.upper()`

Convert strings in the Series/Index to uppercase.

Equivalent to `str.upper()`.

Returns

Series/Index of objects

See also:

Series.str.lower Converts all characters to lowercase.

Series.str.upper Converts all characters to uppercase.

Series.str.title Converts first character of each word to uppercase and remaining to lowercase.

Series.str.capitalize Converts first character to uppercase and remaining to lowercase.

Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.wrap

`Series.str.wrap(width, **kwargs)`

Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given width.

This method has the same keyword parameters and defaults as `textwrap.TextWrapper`.

Parameters

width [int] Maximum line-width

expand_tabs [bool, optional] If true, tab characters will be expanded to spaces (default: True)

replace_whitespace [bool, optional] If true, each whitespace character (as defined by string.whitespace) remaining after tab expansion will be replaced by a single space (default: True)

drop_whitespace [bool, optional] If true, whitespace that, after wrapping, happens to end up at the beginning or end of a line is dropped (default: True)

break_long_words [bool, optional] If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width. (default: True)

break_on_hyphens [bool, optional] If true, wrapping will occur preferably on whitespace and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set break_long_words to false if you want truly insecable words. (default: True)

Returns

wrapped [Series/Index of objects]

Notes

Internally, this method uses a `textwrap.TextWrapper` instance with default settings. To achieve behavior matching R's stringr library str_wrap function, use the arguments:

- expand_tabs = False
- replace_whitespace = True
- drop_whitespace = True
- break_long_words = False
- break_on_hyphens = False

Examples

```
>>> s = pd.Series(['line to be wrapped', 'another line to be wrapped'])
>>> s.str.wrap(12)
0             line to be\nwrapped
1    another line\nto be\nwrapped
```

pandas.Series.str.zfill

`Series.str.`**`zfill`**(*width*)

Pad strings in the Series/Index by prepending '0' characters.

Strings in the Series/Index are padded with '0' characters on the left of the string to reach a total string length *width*. Strings in the Series/Index with length greater or equal to *width* are unchanged.

Parameters

width [int] Minimum length of resulting string; strings with length less than *width* be prepended with '0' characters.

Returns

Series/Index of objects

See also:

Series.str.rjust Fills the left side of strings with an arbitrary character.

Series.str.ljust Fills the right side of strings with an arbitrary character.

Series.str.pad Fills the specified sides of strings with an arbitrary character.

Series.str.center Fills boths sides of strings with an arbitrary character.

Notes

Differs from `str.zfill()` which has special handling for '+'/'-' in the string.

Examples

```
>>> s = pd.Series(['-1', '1', '1000', 10, np.nan])
>>> s
0      -1
1       1
2    1000
3      10
4     NaN
dtype: object
```

Note that `10` and `NaN` are not strings, therefore they are converted to `NaN`. The minus sign in `'-1'` is treated as a regular character and the zero is added to the left of it (`str.zfill()` would have moved it to the left). `1000` remains unchanged as it is longer than *width*.

```
>>> s.str.zfill(3)
0     0-1
1     001
2    1000
3     NaN
4     NaN
dtype: object
```

pandas.Series.str.isalnum

`Series.str.isalnum()`

Check whether all characters in each string are alphanumeric.

This is equivalent to running the Python string method `str.isalnum()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isalpha

`Series.str.isalpha()`

Check whether all characters in each string are alphabetic.

This is equivalent to running the Python string method `str.isalpha()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
```

(continues on next page)

(continued from previous page)

```
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isdigit

`Series.str.isdigit()`

Check whether all characters in each string are digits.

This is equivalent to running the Python string method `str.isdigit()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

`Series.str.isalpha` Check whether all characters are alphabetic.

`Series.str.isnumeric` Check whether all characters are numeric.

`Series.str.isalnum` Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isspace

`Series.str.isspace()`

Check whether all characters in each string are whitespace.

This is equivalent to running the Python string method `str.isspace()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
```

(continues on next page)

(continued from previous page)

```
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.islower

`Series.str.islower()`

Check whether all characters in each string are lowercase.

This is equivalent to running the Python string method `str.islower()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isupper

`Series.str.isupper()`

Check whether all characters in each string are uppercase.

This is equivalent to running the Python string method `str.isupper()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
```

(continues on next page)

(continued from previous page)

```
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.istitle

`Series.str.istitle()`

Check whether all characters in each string are titlecase.

This is equivalent to running the Python string method `str.istitle()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

`Series.str.isalpha` Check whether all characters are alphabetic.

`Series.str.isnumeric` Check whether all characters are numeric.

`Series.str.isalnum` Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isnumeric

`Series.str.isnumeric()`

Check whether all characters in each string are numeric.

This is equivalent to running the Python string method `str.isnumeric()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

Series.str.isalpha Check whether all characters are alphabetic.

Series.str.isnumeric Check whether all characters are numeric.

Series.str.isalnum Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
```

(continues on next page)

(continued from previous page)

```
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.isdecimal

`Series.str.isdecimal()`

Check whether all characters in each string are decimal.

This is equivalent to running the Python string method `str.isdecimal()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

Returns

Series or Index of bool Series or Index of boolean values with the same length as the original Series/Index.

See also:

`Series.str.isalpha` Check whether all characters are alphabetic.

`Series.str.isnumeric` Check whether all characters are numeric.

`Series.str.isalnum` Check whether all characters are alphanumeric.

Series.str.isdigit Check whether all characters are digits.

Series.str.isdecimal Check whether all characters are decimal.

Series.str.isspace Check whether all characters are whitespace.

Series.str.islower Check whether all characters are lowercase.

Series.str.isupper Check whether all characters are uppercase.

Series.str.istitle Check whether all characters are titlecase.

Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters seperated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.get_dummies

`Series.str.get_dummies(sep='|')`

Split each string in the Series by sep and return a frame of dummy/indicator variables.

Parameters

sep [string, default "|"] String to split on.

Returns

dummies [DataFrame]

See also:

get_dummies

Examples

```
>>> pd.Series(['a|b', 'a', 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  1  0  0
2  1  0  1
```

```
>>> pd.Series(['a|b', np.nan, 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  0  0  0
2  1  0  1
```

Categorical Accessor

Categorical-dtype specific methods and attributes are available under the `Series.cat` accessor.

Series.cat.categories	The categories of this categorical.
Series.cat.ordered	Whether the categories have an ordered relationship.
Series.cat.codes	Return Series of codes as well as the index.

pandas.Series.cat.categories

`Series.cat.categories`

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to *categories* is a inplace operation!

Raises

ValueError If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

pandas.Series.cat.ordered

Series.cat.**ordered**
Whether the categories have an ordered relationship.

pandas.Series.cat.codes

Series.cat.**codes**
Return Series of codes as well as the index.

Series.cat.rename_categories(*args, **kwargs)	Renames categories.
Series.cat.reorder_categories(*args, **kwargs)	Reorders categories as specified in new_categories.
Series.cat.add_categories(*args, **kwargs)	Add new categories.
Series.cat.remove_categories(*args, **kwargs)	Removes the specified categories.
Series.cat.remove_unused_categories(*args, ...)	Removes categories which are not used.
Series.cat.set_categories(*args, **kwargs)	Sets the categories to the specified new_categories.
Series.cat.as_ordered(*args, **kwargs)	Set the Categorical to be ordered.
Series.cat.as_unordered(*args, **kwargs)	Set the Categorical to be unordered.

pandas.Series.cat.rename_categories

Series.cat.**rename_categories**(**args*, ***kwargs*)
Renames categories.

Parameters

new_categories [list-like, dict-like or callable]

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.

  New in version 0.21.0.

- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

  New in version 0.23.0.

> **Warning:** Currently, Series are considered list like. In a future version of pandas they'll be considered dict-like.

inplace [boolean (default: False)] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns

cat [Categorical or None] With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

Raises

ValueError If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

See also:

reorder_categories, *add_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

Examples

```
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

pandas.Series.cat.reorder_categories

`Series.cat.`**`reorder_categories`**`(*args, **kwargs)`

Reorders categories as specified in new_categories.

new_categories need to include all old categories and no new category items.

Parameters

new_categories [Index-like] The categories in new order.

ordered [boolean, optional] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

inplace [boolean (default: False)] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If the new categories do not contain all old category items or any new ones

See also:

rename_categories, *add_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

pandas.Series.cat.add_categories

`Series.cat.add_categories(*args, **kwargs)`

Add new categories.

new_categories will be included at the last/highest place in the categories and will be unused directly after this call.

Parameters

new_categories [category or list-like of category] The new categories to be included.

inplace [boolean (default: False)] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical with new categories added or None if inplace.]

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories, *reorder_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

pandas.Series.cat.remove_categories

`Series.cat.remove_categories(*args, **kwargs)`

Removes the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [boolean (default: False)] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical with removed categories or None if inplace.]

Raises

ValueError If the removals are not contained in the categories

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_unused_categories*, *set_categories*

pandas.Series.cat.remove_unused_categories

`Series.cat.`**`remove_unused_categories`**`(*args, **kwargs)`

Removes categories which are not used.

Parameters

inplace [boolean (default: False)] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Returns

cat [Categorical with unused categories dropped or None if inplace.]

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_categories*, *set_categories*

pandas.Series.cat.set_categories

`Series.cat.`**`set_categories`**`(*args, **kwargs)`

Sets the categories to the specified new_categories.

new_categories can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes on python3, which does not considers a S1 string equal to a single char python string.

Parameters

new_categories [Index-like] The categories in new order.

ordered [boolean, (default: False)] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

rename [boolean (default: False)] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

inplace [boolean (default: False)] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If new_categories does not validate as categories

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_categories*, *remove_unused_categories*

pandas.Series.cat.as_ordered

`Series.cat.as_ordered(*args, **kwargs)`

Set the Categorical to be ordered.

Parameters

inplace [boolean (default: False)] Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True

pandas.Series.cat.as_unordered

`Series.cat.as_unordered(*args, **kwargs)`

Set the Categorical to be unordered.

Parameters

inplace [boolean (default: False)] Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to False

Sparse Accessor

Sparse-dtype specific methods and attributes are provided under the `Series.sparse` accessor.

Series.sparse.npoints	The number of non- `fill_value` points.
Series.sparse.density	The percent of non- `fill_value` points, as decimal.
Series.sparse.fill_value	Elements in *data* that are *fill_value* are not stored.
Series.sparse.sp_values	An ndarray containing the non- `fill_value` values.

pandas.Series.sparse.npoints

`Series.sparse.npoints`

The number of non- `fill_value` points.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.npoints
3
```

pandas.Series.sparse.density

`Series.sparse.density`
The percent of non- `fill_value` points, as decimal.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.density
0.6
```

pandas.Series.sparse.fill_value

`Series.sparse.fill_value`
Elements in *data* that are *fill_value* are not stored.

For memory savings, this should be the most common value in the array.

pandas.Series.sparse.sp_values

`Series.sparse.sp_values`
An ndarray containing the non- `fill_value` values.

Examples

```
>>> s = SparseArray([0, 0, 1, 0, 2], fill_value=0)
>>> s.sp_values
array([1, 2])
```

Series.sparse.from_coo(A[, dense_index])	Create a SparseSeries from a scipy.sparse.coo_matrix.
Series.sparse.to_coo([row_levels, ...])	Create a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.

pandas.Series.sparse.from_coo

classmethod `sparse.from_coo`(*A*, *dense_index=False*)
Create a SparseSeries from a scipy.sparse.coo_matrix.

Parameters

A [scipy.sparse.coo_matrix]

dense_index [bool, default False] If False (default), the SparseSeries index consists of only the coords of the non-null entries of the original coo_matrix. If True, the SparseSeries index consists of the full sorted (row, col) coordinates of the coo_matrix.

Returns

s [SparseSeries]

Examples

```
>>> from scipy import sparse
>>> A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
                       shape=(3, 4))
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> ss = pd.SparseSeries.from_coo(A)
>>> ss
0  2    1
   3    2
1  0    3
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

pandas.Series.sparse.to_coo

`sparse.to_coo`(*row_levels=(0,), column_levels=(1,), sort_labels=False*)

Create a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.

Use row_levels and column_levels to determine the row and column coordinates respectively. row_levels and column_levels are the names (labels) or numbers of the levels. {row_levels, column_levels} must be a partition of the MultiIndex level names (or numbers).

Parameters

row_levels [tuple/list]

column_levels [tuple/list]

sort_labels [bool, default False] Sort the row and column labels before forming the sparse matrix.

Returns

y [scipy.sparse.coo_matrix]

rows [list (row labels)]

columns [list (column labels)]

Examples

```
>>> s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])
>>> s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
                                         (1, 2, 'a', 1),
                                         (1, 1, 'b', 0),
                                         (1, 1, 'b', 1),
                                         (2, 1, 'b', 0),
                                         (2, 1, 'b', 1)],
```

(continues on next page)

(continued from previous page)

```
                                        names=['A', 'B', 'C', 'D'])
>>> ss = s.to_sparse()
>>> A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
                                 column_levels=['C', 'D'],
                                 sort_labels=True)
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  3.],
[ 3.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.]])
>>> rows
[(1, 1), (1, 2), (2, 1)]
>>> columns
[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

6.3.14 Plotting

`Series.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `Series.plot.<kind>`.

Series.plot([kind, ax, figsize, ….])	Series plotting accessor and method

Series.plot.area(**kwds)	Area plot.
Series.plot.bar(**kwds)	Vertical bar plot.
Series.plot.barh(**kwds)	Horizontal bar plot.
Series.plot.box(**kwds)	Boxplot.
Series.plot.density([bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
Series.plot.hist([bins])	Histogram.
Series.plot.kde([bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
Series.plot.line(**kwds)	Line plot.
Series.plot.pie(**kwds)	Pie chart.

pandas.Series.plot.area

`Series.plot.`**`area`**`(`***kwds*`)`

Area plot.

Parameters

‘kwds‘** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them]

pandas.Series.plot.bar

`Series.plot.bar(**kwds)`
Vertical bar plot.

Parameters

'kwds'** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

pandas.Series.plot.barh

`Series.plot.barh(**kwds)`
Horizontal bar plot.

Parameters

'kwds'** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

pandas.Series.plot.box

`Series.plot.box(**kwds)`
Boxplot.

Parameters

'kwds'** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

pandas.Series.plot.density

`Series.plot.density(bw_method=None, ind=None, **kwds)`
Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

Parameters

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See scipy.stats.gaussian_kde for more information.

ind [NumPy array or integer, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

****kwds** [optional] Additional keyword arguments are documented in `pandas.Series.plot()`.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

`scipy.stats.gaussian_kde` Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

`DataFrame.plot.kde` Generate a KDE plot for a DataFrame.

Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> ax = s.plot.kde(bw_method=0.3)
```

```
>>> ax = s.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

pandas.Series.plot.hist

`Series.plot.hist(bins=10, **kwds)`

Histogram.

Parameters

bins [integer, default 10] Number of histogram bins to be used

'kwds'** [optional] Additional keyword arguments are documented in `pandas.Series.plot()`.

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them]

pandas.Series.plot.kde

`Series.plot.kde(bw_method=None, ind=None, **kwds)`

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

Parameters

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

ind [NumPy array or integer, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

****kwds** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

DataFrame.plot.kde Generate a KDE plot for a DataFrame.

Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> ax = s.plot.kde(bw_method=0.3)
```

```
>>> ax = s.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

pandas.Series.plot.line

`Series.plot.line(**kwds)`

Line plot.

Parameters

‘kwds‘** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

Examples

```
>>> s = pd.Series([1, 3, 2])
>>> s.plot.line()
```

pandas.Series.plot.pie

Series.plot.**pie**(***kwds*)

Pie chart.

Parameters

‘kwds‘** [optional] Additional keyword arguments are documented in *pandas.Series.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

Series.hist([by, ax, grid, xlabelsize, . . .])	Draw histogram of the input series using matplotlib.

6.3.15 Serialization / IO / Conversion

Series.to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
Series.to_csv(*args, **kwargs)	Write object to a comma-separated values (csv) file.
Series.to_dict([into])	Convert Series to {label -> value} dict or dict-like object.
Series.to_excel(excel_writer[, sheet_name, . . .])	Write object to an Excel sheet.
Series.to_frame([name])	Convert Series to DataFrame.
Series.to_xarray()	Return an xarray object from the pandas object.
Series.to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
Series.to_sql(name, con[, schema, . . .])	Write records stored in a DataFrame to a SQL database.
Series.to_msgpack([path_or_buf, encoding])	Serialize object to input file path using msgpack format.
Series.to_json([path_or_buf, orient, . . .])	Convert the object to a JSON string.
Series.to_sparse([kind, fill_value])	Convert Series to SparseSeries.
Series.to_dense()	Return dense representation of NDFrame (as opposed to sparse).
Series.to_string([buf, na_rep, . . .])	Render a string representation of the Series.
Series.to_clipboard([excel, sep])	Copy object to the system clipboard.
Series.to_latex([buf, columns, col_space, . . .])	Render an object to a LaTeX tabular environment table.

6.3.16 Sparse

SparseSeries.to_coo([row_levels, . . .])	Create a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.
SparseSeries.from_coo(A[, dense_index])	Create a SparseSeries from a scipy.sparse.coo_matrix.

pandas.SparseSeries.to_coo

SparseSeries.**to_coo**(*row_levels=(0,), column_levels=(1,), sort_labels=False*)

Create a scipy.sparse.coo_matrix from a SparseSeries with MultiIndex.

Use row_levels and column_levels to determine the row and column coordinates respectively. row_levels and column_levels are the names (labels) or numbers of the levels. {row_levels, column_levels} must be a partition of the MultiIndex level names (or numbers).

Parameters

row_levels [tuple/list]

column_levels [tuple/list]

sort_labels [bool, default False] Sort the row and column labels before forming the sparse matrix.

Returns

y [scipy.sparse.coo_matrix]

rows [list (row labels)]

columns [list (column labels)]

Examples

```
>>> s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])
>>> s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
                                         (1, 2, 'a', 1),
                                         (1, 1, 'b', 0),
                                         (1, 1, 'b', 1),
                                         (2, 1, 'b', 0),
                                         (2, 1, 'b', 1)],
                                        names=['A', 'B', 'C', 'D'])
>>> ss = s.to_sparse()
>>> A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
                                 column_levels=['C', 'D'],
                                 sort_labels=True)
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  3.],
[ 3.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.]])
>>> rows
[(1, 1), (1, 2), (2, 1)]
>>> columns
[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

pandas.SparseSeries.from_coo

classmethod SparseSeries.**from_coo**(*A*, *dense_index=False*)

Create a SparseSeries from a scipy.sparse.coo_matrix.

Parameters

A [scipy.sparse.coo_matrix]

dense_index [bool, default False] If False (default), the SparseSeries index consists of only the coords of the non-null entries of the original coo_matrix. If True, the SparseSeries index consists of the full sorted (row, col) coordinates of the coo_matrix.

Returns

s [SparseSeries]

Examples

```
>>> from scipy import sparse
>>> A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
                       shape=(3, 4))
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,   0.,   1.,   2.],
        [ 3.,   0.,   0.,   0.],
        [ 0.,   0.,   0.,   0.]])
>>> ss = pd.SparseSeries.from_coo(A)
>>> ss
0  2    1
   3    2
1  0    3
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

6.4 DataFrame

6.4.1 Constructor

DataFrame([data, index, columns, dtype, copy])	Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

pandas.DataFrame

class pandas.**DataFrame**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters

data [ndarray (structured or homogeneous), Iterable, dict, or DataFrame] Dict can contain Series, arrays, constants, or list-like objects

Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index [Index or array-like] Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided

columns [Index or array-like] Column labels to use for resulting frame. Will default to RangeIndex (0, 1, 2, . . . , n) if no column labels are provided

dtype [dtype, default None] Data type to force. Only a single dtype is allowed. If None, infer

copy [boolean, default False] Copy data from inputs. Only affects DataFrame / 2d ndarray input

See also:

DataFrame.from_records Constructor from tuples, also record arrays.

DataFrame.from_dict From dicts of Series, arrays, or dicts.

DataFrame.from_items From sequence of (key, value) pairs pandas.read_csv, pandas.read_table, pandas.read_clipboard.

Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
```

(continues on next page)

(continued from previous page)

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Attributes

T	Transpose index and columns.
at	Access a single value for a row/column label pair.
axes	Return a list representing the axes of the DataFrame.
blocks	(DEPRECATED) Internal property, property synonym for as_blocks().
columns	The column labels of the DataFrame.
dtypes	Return the dtypes in the DataFrame.
empty	Indicator whether DataFrame is empty.
ftypes	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
iat	Access a single value for a row/column pair by integer position.
iloc	Purely integer-location based indexing for selection by position.
index	The index (row labels) of the DataFrame.
is_copy	Return the copy.
ix	A primarily label-location based indexer, with integer position fallback.
loc	Access a group of rows and columns by label(s) or a boolean array.
ndim	Return an int representing the number of axes / array dimensions.
shape	Return a tuple representing the dimensionality of the DataFrame.
size	Return an int representing the number of elements in this object.
style	Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.
values	Return a Numpy representation of the DataFrame.

pandas.DataFrame.T

DataFrame.**T**

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

Parameters

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

DataFrame The transposed DataFrame.

See also:

`numpy.transpose` Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

pandas.DataFrame.at

`DataFrame.at`

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

DataFrame.iat Access a single value for a row/column pair by integer position.

DataFrame.loc Access a group of rows and columns by label(s).

Series.at Access a single value using a label.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

pandas.DataFrame.axes

`DataFrame.axes`
Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['coll', 'col2'],
dtype='object')]
```

pandas.DataFrame.blocks

`DataFrame.blocks`
Internal property, property synonym for as_blocks().

Deprecated since version 0.21.0.

pandas.DataFrame.columns

`DataFrame.columns`
The column labels of the DataFrame.

pandas.DataFrame.dtypes

`DataFrame.dtypes`
Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See *the User Guide* for more.

Returns

pandas.Series The data type of each column.

See also:

pandas.DataFrame.ftypes Dtype and sparsity information.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

pandas.DataFrame.empty

`DataFrame.empty`

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If DataFrame is empty, return True, if not return False.

See also:

pandas.Series.dropna, *pandas.DataFrame.dropna*

Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

pandas.DataFrame.ftypes

`DataFrame.ftypes`

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See *the User Guide* for more.

Returns

pandas.Series The data type and indication of sparse/dense of each column.

See also:

pandas.DataFrame.dtypes Series with just dtype information.

pandas.SparseDataFrame Container for sparse tabular data.

Notes

Sparse data should have the same dtypes as its dense representation.

Examples

```
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

pandas.DataFrame.iat

`DataFrame.iat`

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.loc Access a group of rows and columns by label(s).

DataFrame.iloc Access a group of rows and columns by integer position(s).

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

pandas.DataFrame.iloc

`DataFrame.`**`iloc`**

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at ref:*Selection by Position <indexing.integer>*.

See also:

DataFrame.iat Fast integer location scalar accessor.

DataFrame.loc Purely label-location based indexer for selection by label.

Series.iloc Purely integer-location based indexing for selection by position.

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a *slice* object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

pandas.DataFrame.index

`DataFrame.`**`index`**
The index (row labels) of the DataFrame.

pandas.DataFrame.is_copy

`DataFrame.`**`is_copy`**
Return the copy.

pandas.DataFrame.ix

`DataFrame.ix`
A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at *Advanced Indexing*.

pandas.DataFrame.loc

`DataFrame.loc`
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at *Selection by Label*

Raises

KeyError: when any items are not found

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.iloc Access group of rows and columns by integer position(s).

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels.

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=['cobra', 'viper', 'sidewinder'],
...      columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...    ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...         [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

pandas.DataFrame.ndim

DataFrame.**ndim**

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

See also:

ndarray.ndim Number of array dimensions.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

pandas.DataFrame.shape

DataFrame.**shape**

Return a tuple representing the dimensionality of the DataFrame.

See also:

ndarray.shape

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

pandas.DataFrame.size

`DataFrame.size`

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also:

ndarray.size Number of elements in the array.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

pandas.DataFrame.style

`DataFrame.style`

Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

See also:

pandas.io.formats.style.Styler

pandas.DataFrame.values

`DataFrame.values`

Return a Numpy representation of the DataFrame.

Warning: We recommend using *DataFrame.to_numpy()* instead.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

See also:

DataFrame.to_numpy Recommended alternative to this method.

pandas.DataFrame.index Retrieve the index labels.

pandas.DataFrame.columns Retrieving the column names.

Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                   columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
```

(continues on next page)

(continued from previous page)

```
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

timetuple	

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	Suffix labels with string *suffix*.
agg(func[, axis])	Aggregate using one or more operations over the specified axis.
aggregate(func[, axis])	Aggregate using one or more operations over the specified axis.
align(other[, join, axis, level, copy, …])	Align two objects on their axes with the specified join method for each axis Index.
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
append(other[, ignore_index, …])	Append rows of *other* to the end of caller, returning a new object.
apply(func[, axis, broadcast, raw, reduce, …])	Apply a function along an axis of the DataFrame.
applymap(func)	Apply a function to a Dataframe elementwise.
as_blocks([copy])	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	(DEPRECATED) Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, …])	Convert TimeSeries to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
assign(**kwargs)	Assign new columns to a DataFrame.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g.
between_time(start_time, end_time[, …])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for *DataFrame.fillna()* with `method='bfill'`.
bool()	Return the bool of a single element PandasObject.
boxplot([column, by, ax, fontsize, rot, …])	Make a box plot from DataFrame columns.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.

Continued on next page

Table 57 – continued from previous page

clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.
combine(other, func[, fill_value, overwrite])	Perform column-wise combine with another DataFrame based on a passed function.
combine_first(other)	Update null elements with value in the same location in *other*.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis.
convert_objects([convert_dates, . . .])	(DEPRECATED) Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.
corr([method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values.
corrwith(other[, axis, drop, method])	Compute pairwise correlation between rows or columns of DataFrame with rows or columns of Series or DataFrame.
count([axis, level, numeric_only])	Count non-NA cells for each column or row.
cov([min_periods])	Compute pairwise covariance of columns, excluding NA/null values.
cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe([percentiles, include, exclude])	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
diff([periods, axis])	First discrete difference of element.
div(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Compute the matrix mutiplication between the DataFrame and other.
drop([labels, axis, index, columns, level, . . .])	Drop specified labels from rows or columns.
drop_duplicates([subset, keep, inplace])	Return DataFrame with duplicate rows removed, optionally only considering certain columns.
droplevel(level[, axis])	Return DataFrame with requested index / column level(s) removed.
dropna([axis, how, thresh, subset, inplace])	Remove missing values.
duplicated([subset, keep])	Return boolean Series denoting duplicate rows, optionally only considering certain columns.
eq(other[, axis, level])	Equal to of dataframe and other, element-wise (binary operator *eq*).
equals(other)	Test whether two objects contain the same elements.
eval(expr[, inplace])	Evaluate a string describing operations on DataFrame columns.
ewm([com, span, halflife, alpha, . . .])	Provides exponential weighted functions.

Continued on next page

Table 57 – continued from previous page

expanding([min_periods, center, axis])	Provides expanding transformations.
ffill([axis, inplace, limit, downcast])	Synonym for *DataFrame.fillna()* with method='ffill'.
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_csv(path[, header, sep, index_col, ...])	(DEPRECATED) Read CSV file.
from_dict(data[, orient, dtype, columns])	Construct DataFrame from dict of array-like or dicts.
from_items(items[, columns, orient])	(DEPRECATED) Construct a DataFrame from a list of tuples.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame.
ge(other[, axis, level])	Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	(DEPRECATED) Return counts of unique ftypes in this object.
get_value(index, col[, takeable])	(DEPRECATED) Quickly retrieve single value at passed column and index.
get_values()	Return an ndarray after converting sparse values to dense.
groupby([by, axis, level, as_index, sort, ...])	Group DataFrame or Series using a mapper or by a Series of columns.
gt(other[, axis, level])	Greater than of dataframe and other, element-wise (binary operator *gt*).
head([n])	Return the first *n* rows.
hist([column, by, grid, xlabelsize, xrot, ...])	Make a histogram of the DataFrame's.
idxmax([axis, skipna])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna])	Return index of first occurrence of minimum over requested axis.
infer_objects()	Attempt to infer better dtypes for object columns.
info([verbose, buf, max_cols, memory_usage, ...])	Print a concise summary of a DataFrame.
insert(loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isin(values)	Whether each element in the DataFrame is contained in values.
isna()	Detect missing values.
isnull()	Detect missing values.
items()	Iterator over (column name, Series) pairs.
iteritems()	Iterator over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples.
join(other[, on, how, lsuffix, rsuffix, sort])	Join columns of another DataFrame.

Continued on next page

Table 57 – continued from previous page

keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, axis, level])	Less than or equal to of dataframe and other, element-wise (binary operator *le*).
lookup(row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
lt(other[, axis, level])	Less than of dataframe and other, element-wise (binary operator *lt*).
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
mask(cond[, other, inplace, axis, level, ...])	Replace values where the condition is True.
max([axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis.
melt([id_vars, value_vars, var_name, ...])	Unpivots a DataFrame from wide format to long format, optionally leaving identifier variables set.
memory_usage([index, deep])	Return the memory usage of each column in bytes.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame or named Series objects with a database-style join.
min([axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
mod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
mul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
ne(other[, axis, level])	Not equal to of dataframe and other, element-wise (binary operator *ne*).
nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
nsmallest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
nunique([axis, dropna])	Count distinct observations over requested axis.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs).

Continued on next page

Table 57 – continued from previous page

pivot([index, columns, values])	Return reshaped DataFrame organized by given index / column values.
pivot_table([values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
plot	alias of `pandas.plotting._core.FramePlotMethods`
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis.
product([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis.
quantile([q, axis, numeric_only, interpolation])	Return values at the given quantile over requested axis.
query(expr[, inplace])	Query the columns of a DataFrame with a boolean expression.
radd(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex([labels, index, columns, axis, ...])	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	(DEPRECATED) Conform input object to new index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
rename([mapper, index, columns, axis, copy, ...])	Alter axes labels.
rename_axis([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
reorder_levels(order[, axis])	Rearrange index levels using input order.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, how, axis, fill_method, ...])	Resample time-series data.
reset_index([level, drop, inplace, ...])	Reset the index, or a level of it.
rfloordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, center, ...])	Provides rolling window calculations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Continued on next page

Table 57 – continued from previous page

rtruediv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, . . .])	Return a random sample of items from an axis of object.
select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.
set_index(keys[, drop, append, inplace, . . .])	Set the DataFrame index using existing columns.
set_value(index, col, value[, takeable])	(DEPRECATED) Put single value at passed column and index.
shift([periods, freq, axis, fill_value])	Shift index by desired number of periods with an optional time *freq*.
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1.
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, . . .])	Sort object by labels (along an axis)
sort_values(by[, axis, ascending, inplace, . . .])	Sort by the values along either axis
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
stack([level, dropna])	Stack the prescribed level(s) from columns to index.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, . . .])	Return the sum of the values for the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis.
tail([n])	Return the last *n* rows.
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path_or_buf, sep, na_rep, . . .])	Write object to a comma-separated values (csv) file.
to_dense()	Return dense representation of NDFrame (as opposed to sparse).
to_dict([orient, into])	Convert the DataFrame to a dictionary.
to_excel(excel_writer[, sheet_name, na_rep, . . .])	Write object to an Excel sheet.
to_feather(fname)	Write out the binary feather-format for DataFrames.
to_gbq(destination_table[, project_id, . . .])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_html([buf, columns, col_space, header, . . .])	Render a DataFrame as an HTML table.
to_json([path_or_buf, orient, date_format, . . .])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, . . .])	Render an object to a LaTeX tabular environment table.

Continued on next page

Table 57 – continued from previous page

to_msgpack([path_or_buf, encoding])	Serialize object to input file path using msgpack format.
to_numpy([dtype, copy])	Convert the DataFrame to a NumPy array.
to_panel()	(DEPRECATED) Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
to_parquet(fname[, engine, compression, . . .])	Write a DataFrame to the binary parquet format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_records([index, convert_datetime64, . . .])	Convert DataFrame to a NumPy record array.
to_sparse([fill_value, kind])	Convert to SparseDataFrame.
to_sql(name, con[, schema, if_exists, . . .])	Write records stored in a DataFrame to a SQL database.
to_stata(fname[, convert_dates, . . .])	Export DataFrame object to Stata dta format.
to_string([buf, columns, col_space, header, . . .])	Render a DataFrame to a console-friendly tabular output.
to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
to_xarray()	Return an xarray object from the pandas object.
transform(func[, axis])	Call `func` on self producing a DataFrame with transformed values and that has the same axis length as self.
transpose(*args, **kwargs)	Transpose index and columns.
truediv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, . . .])	Localize tz-naive index of a Series or DataFrame to target time zone.
unstack([level, fill_value])	Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
update(other[, join, overwrite, . . .])	Modify in place using non-NA values from another DataFrame.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
where(cond[, other, inplace, axis, level, . . .])	Replace values where the condition is False.
xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.

pandas.DataFrame.abs

`DataFrame.abs()`

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

pandas.DataFrame.add

`DataFrame.add`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.add_prefix

DataFrame.**add_prefix**(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix [str] The string to add before each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_suffix Suffix row labels with string *suffix*.

DataFrame.add_suffix Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0        1      3
1        2      4
2        3      5
3        4      6
```

pandas.DataFrame.add_suffix

`DataFrame.add_suffix(suffix)`

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix [str] The string to add after each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_prefix Prefix row labels with string *prefix*.

DataFrame.add_prefix Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

pandas.DataFrame.agg

DataFrame.**agg**(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`

- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

The aggregation operations are always performed over an axis, either the

index (default) or the column axis. This behavior is different from

'numpy' aggregation functions ('mean', 'median', 'prod', 'sum', 'std',

'var'), where the default is to compute the aggregation of the flattened

array, e.g., ''numpy.mean(arr_2d)'' as opposed to ''numpy.mean(arr_2d,

axis=0)''.

'agg' is an alias for 'aggregate'. Use the alias.

See also:

DataFrame.apply Perform any type of operations.

DataFrame.transform Perform transformation type operations.

pandas.core.groupby.GroupBy Perform operations over groups.

pandas.core.resample.Resampler Perform operations over resampled bins.

pandas.core.window.Rolling Perform operations over rolling window.

pandas.core.window.Expanding Perform operations over expanding window.

pandas.core.window.EWM Perform operation over exponential weighted window.

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

pandas.DataFrame.aggregate

`DataFrame.aggregate`(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

The aggregation operations are always performed over an axis, either the

index (default) or the column axis. This behavior is different from

‘numpy‘ aggregation functions (‘mean‘, ‘median‘, ‘prod‘, ‘sum‘, ‘std‘, ‘var‘), where the default is to compute the aggregation of the flattened array, e.g., ‘‘numpy.mean(arr_2d)‘‘ as opposed to ‘‘numpy.mean(arr_2d, axis=0)‘‘.

‘agg‘ is an alias for ‘aggregate‘. Use the alias.

See also:

DataFrame.apply Perform any type of operations.

DataFrame.transform Perform transformation type operations.

pandas.core.groupby.GroupBy Perform operations over groups.

pandas.core.resample.Resampler Perform operations over resampled bins.

pandas.core.window.Rolling Perform operations over rolling window.

pandas.core.window.Expanding Perform operations over expanding window.

pandas.core.window.EWM Perform operation over exponential weighted window.

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

pandas.DataFrame.align

DataFrame.**align**(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index.

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

axis [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

fill_axis [{0 or 'index', 1 or 'columns'}, default 0] Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

Returns

(left, right) [(DataFrame, type of other)] Aligned objects

pandas.DataFrame.all

DataFrame.**all**(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

Series.all Return True if all elements are True.

DataFrame.any Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([]).all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

pandas.DataFrame.any

`DataFrame.any`(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether any element is True, potentially over an axis.

Returns False unless there at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

`numpy.any` Numpy version of this method.

Series.any Return whether any element is True.

Series.all Return whether all elements are True.

DataFrame.any Return whether any element is True over requested axis.

DataFrame.all Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

pandas.DataFrame.append

`DataFrame.`**append**(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of caller, returning a new object.

Columns in *other* that are not in the caller are added as new columns.

Parameters

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

Returns

appended [DataFrame]

See also:

pandas.concat General function to concatenate DataFrame, Series or Panel objects.

Notes

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

pandas.DataFrame.apply

DataFrame.**apply**(*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

Parameters

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

Returns

applied [Series or DataFrame]

See also:

DataFrame.applymap For elementwise operations.

DataFrame.aggregate Only perform aggregating type operations.

DataFrame.transform Only perform transforming type operations.

Notes

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

Examples

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

pandas.DataFrame.applymap

`DataFrame.`**`applymap`**(*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func [callable] Python function, returns a single value from a single value.

Returns

DataFrame Transformed DataFrame.

See also:

DataFrame.apply Apply a function along input axis of DataFrame.

Notes

In the current implementation applymap calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

Examples

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0       1
```

(continues on next page)

(continued from previous page)

```
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

pandas.DataFrame.as_blocks

DataFrame.**as_blocks**(*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

pandas.DataFrame.as_matrix

DataFrame.**as_matrix**(*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

Parameters

columns [list, optional, default:None] If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

DataFrame.values

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.asfreq

DataFrame.**asfreq**(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, or string]

method [{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

Returns

converted [same type as caller]

See also:

reindex

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

pandas.DataFrame.asof

`DataFrame.`**`asof`**(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where [date or array-like of dates] Date(s) before which the last row(s) are returned.

subset [str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5     NaN
20    2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                      a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
                         a   b
2018-02-27 09:03:30   30.0 NaN
2018-02-27 09:04:30   40.0 NaN
```

pandas.DataFrame.assign

`DataFrame.assign(**kwargs)`

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters

****kwargs** [dict of {str: callable or Series}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns

DataFrame A new DataFrame with the new columns in addition to all the existing columns.

Notes

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

Examples

```
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
...                   index=['Portland', 'Berkeley'])
>>> df
          temp_c
Portland    17.0
Berkeley    25.0
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

In Python 3.6+, you can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...           temp_k=lambda x: (x['temp_f'] +  459.67) * 5 / 9)
          temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```

pandas.DataFrame.astype

`DataFrame.astype`(*dtype*, *copy=True*, *errors='raise'*, ***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [same type as caller]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> cat_dtype = pd.api.types.CategoricalDtype(
...                     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.DataFrame.at_time

`DataFrame.`**`at_time`**(*time*, *asof=False*, *axis=None*)
Select values at particular time of day (e.g. 9:30AM).

Parameters

time [datetime.time or string]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_at_time [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

between_time Select values between particular times of the day.

first Select initial periods of time series based on a date offset.

last Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

pandas.DataFrame.between_time

DataFrame.**between_time**(*start_time*, *end_time*, *include_start=True*, *include_end=True*, *axis=None*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time [datetime.time or string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_between_time [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

`at_time` Select values at a particular time of the day.

`first` Select initial periods of time series based on a date offset.

`last` Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_between_time` Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

pandas.DataFrame.bfill

`DataFrame.`**`bfill`**(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for `DataFrame.fillna()` with `method='bfill'`.

pandas.DataFrame.bool

`DataFrame.`**`bool`**()

Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

pandas.DataFrame.boxplot

`DataFrame.boxplot`(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

Parameters

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate system.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

Returns

result : The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes
- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a namedtuple with structure (ax, lines)

For data grouped with `by`:

- *Series*
- `array` (for `return_type = None`)

See also:

Series.plot.hist Make a histogram.

matplotlib.pyplot.boxplot Matplotlib equivalent plot.

Notes

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10,4),
...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                   columns=['Col1', 'Col2'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

```
>>> df = pd.DataFrame(np.random.randn(10,3),
...                   columns=['Col1', 'Col2', 'Col3'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                      'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
...                      'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

The layout of boxplot can be adjusted giving a tuple to `layout`:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      layout=(2, 1))
```

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

```
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

pandas.DataFrame.clip

`DataFrame.`**`clip`**(*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
```

(continues on next page)

(continued from previous page)

```
2       0       6
3      -1       8
4       5      -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

pandas.DataFrame.clip_lower

DataFrame.**clip_lower**(*threshold*, *axis=None*, *inplace=False*)

Trim values below a given threshold.

Deprecated since version 0.24.0: Use clip(lower=threshold) instead.

Elements below the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

Series.clip General purpose method to trim Series values to given threshold(s).

DataFrame.clip General purpose method to trim DataFrame values to given threshold(s).

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip(lower=8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip(lower=elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip(lower=3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip(lower=[3, 3, 5], axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip(lower=[4, 5], axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

pandas.DataFrame.clip_upper

`DataFrame.clip_upper`(*threshold*, *axis=None*, *inplace=False*)

Trim values above a given threshold.

Deprecated since version 0.24.0: Use clip(upper=threshold) instead.

Elements above the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Maximum value allowed. All values above threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align object with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

Series.clip General purpose method to trim Series values to given threshold(s).

DataFrame.clip General purpose method to trim DataFrame values to given threshold(s).

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.clip(upper=3)
0    1
1    2
2    3
3    3
4    3
dtype: int64
```

```
>>> elemwise_thresholds = [5, 4, 3, 2, 1]
>>> elemwise_thresholds
[5, 4, 3, 2, 1]
```

```
>>> s.clip(upper=elemwise_thresholds)
0    1
1    2
2    3
3    2
4    1
dtype: int64
```

pandas.DataFrame.combine

DataFrame.**combine**(*other*, *func*, *fill_value=None*, *overwrite=True*)

Perform column-wise combine with another DataFrame based on a passed function.

Combines a DataFrame with *other* DataFrame using *func* to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other [DataFrame] The DataFrame to merge column-wise.

func [function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value [scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite [boolean, default True] If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.

Returns

result [DataFrame]

See also:

DataFrame.combine_first Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using *fill_value* fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0  NaN
1  0  3.0
```

Example that demonstrates the use of *overwrite* and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1],}, index=[1, 2])
>>> df1.combine(df2, take_smaller)
     A    B     C
0  NaN  NaN   NaN
1  NaN  3.0 -10.0
2  NaN  3.0   1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)
     A    B     C
0  0.0  NaN   NaN
1  0.0  3.0 -10.0
2  NaN  3.0   1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1],}, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A    B   C
0  0.0  NaN NaN
1  0.0  3.0 NaN
2  NaN  3.0 NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)
     A    B   C
0  0.0  NaN NaN
1  0.0  3.0 1.0
2  NaN  3.0 1.0
```

pandas.DataFrame.combine_first

`DataFrame.combine_first(other)`

Update null elements with value in the same location in *other*.

Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other [DataFrame] Provided DataFrame to use to fill null values.

Returns

combined [DataFrame]

See also:

DataFrame.combine Perform series-wise operation on two DataFrames using a given function.

Examples

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
     A    B
0  1.0  3.0
1  0.0  4.0
```

Null values still persist if the location of that null value does not exist in *other*

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
     A    B    C
0  NaN  4.0  NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

pandas.DataFrame.compound

`DataFrame.compound`(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

compounded [Series or DataFrame (if level specified)]

pandas.DataFrame.convert_objects

`DataFrame.convert_objects`(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

pandas.DataFrame.copy

DataFrame.**copy**(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

copy [Series, DataFrame or Panel] Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

pandas.DataFrame.corr

`DataFrame.corr(method='pearson', min_periods=1)`

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method [{'pearson', 'kendall', 'spearman'} or callable]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

- **callable: callable with input two 1d ndarrays** and returning a float .. version-added:: 0.24.0

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

Returns

y [DataFrame]

See also:

DataFrame.corrwith, *Series.corr*

Examples

```
>>> histogram_intersection = lambda a, b: np.minimum(a, b
... ).sum().round(decimals=1)
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

pandas.DataFrame.corrwith

DataFrame.**corrwith**(*other*, *axis=0*, *drop=False*, *method='pearson'*)

Compute pairwise correlation between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other [DataFrame, Series]

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result

method [{'pearson', 'kendall', 'spearman'} or callable]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays** and returning a float

New in version 0.24.0.

Returns

correls [Series]

See Also

——-

DataFrame.corr

pandas.DataFrame.count

DataFrame.**count**(*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Returns

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.shape Number of DataFrame rows and columns (including NA elements).

DataFrame.isna Boolean same-sized DataFrame showing places of NA elements.

Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2   Lewis  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    5
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    3
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Lewis     1
Myla      1
```

pandas.DataFrame.cov

DataFrame.**cov**(*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

Returns

DataFrame The covariance matrix of the series of the DataFrame.

See also:

pandas.Series.cov Compute covariance with another Series.

pandas.core.window.EWM.cov Exponential weighted sample covariance.

pandas.core.window.Expanding.cov Expanding sample covariance.

pandas.core.window.Rolling.cov Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

pandas.DataFrame.cummax

`DataFrame.`**`cummax`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummax [Series or DataFrame]

See also:

core.window.Expanding.max Similar functionality but ignores `NaN` values.

DataFrame.max Return the maximum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
```

(continues on next page)

(continued from previous page)

```
...                          columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.DataFrame.cummin

`DataFrame.`**`cummin`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummin [Series or DataFrame]

See also:

`core.window.Expanding.min` Similar functionality but ignores `NaN` values.

`DataFrame.min` Return the minimum over DataFrame axis.

`DataFrame.cummax` Return cumulative maximum over DataFrame axis.

`DataFrame.cummin` Return cumulative minimum over DataFrame axis.

`DataFrame.cumsum` Return cumulative sum over DataFrame axis.

`DataFrame.cumprod` Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
```

(continues on next page)

(continued from previous page)

```
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.DataFrame.cumprod

`DataFrame.cumprod(axis=None, skipna=True, *args, **kwargs)`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumprod [Series or DataFrame]

See also:

core.window.Expanding.prod Similar functionality but ignores NaN values.

DataFrame.prod Return the product over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.DataFrame.cumsum

`DataFrame.`**`cumsum`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumsum [Series or DataFrame]

See also:

core.window.Expanding.sum Similar functionality but ignores `NaN` values.

DataFrame.sum Return the sum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.DataFrame.describe

`DataFrame.describe`(*percentiles=None*, *include=None*, *exclude=None*)

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

DataFrame.count Count number of non-NA/null observations.

DataFrame.max Maximum of the values in the object.

DataFrame.min Minimum of the values in the object.

DataFrame.mean Mean of the values.

DataFrame.std Standard deviation of the obersvations.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
```

(continues on next page)

(continued from previous page)

```
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      c
freq             1      NaN      1
```

(continues on next page)

(continued from previous page)

```
mean            NaN       2.0     NaN
std             NaN       1.0     NaN
min             NaN       1.0     NaN
25%             NaN       1.5     NaN
50%             NaN       2.0     NaN
75%             NaN       2.5     NaN
max             NaN       3.0     NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
```

(continues on next page)

(continued from previous page)

```
count          3      3
unique         3      3
top            f      c
freq           1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

pandas.DataFrame.diff

`DataFrame.diff`(*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

Parameters

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

Returns

diffed [DataFrame]

See also:

Series.diff First discrete difference for a Series.

DataFrame.pct_change Percent change over given number of periods.

DataFrame.shift Shift index by desired number of periods with an optional time freq.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
```

(continues on next page)

(continued from previous page)

```
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

pandas.DataFrame.div

`DataFrame.`**`div`**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.divide

`DataFrame.divide`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.dot

`DataFrame.dot(other)`

Compute the matrix mutiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series, DataFrame or a numpy array.

It can also be called using `self @ other` in Python >= 3.5.

Parameters

other [Series, DataFrame or array-like] The other object to compute the matrix product with.

Returns

Series or DataFrame If other is a Series, return the matrix product between self and other as a Serie. If other is a DataFrame or a numpy.array, return the matrix product of self and other in a DataFrame of a np.array.

See also:

Series.dot Similar method for Series.

Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

Examples

Here we multiply a DataFrame with a Series.

```
>>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> s = pd.Series([1, 1, 2, 1])
>>> df.dot(s)
0    -4
1     5
dtype: int64
```

Here we multiply a DataFrame with another DataFrame.

```
>>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(other)
    0   1
0   1   4
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other
    0   1
0   1   4
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
>>> df.dot(arr)
    0   1
0   1   4
1   2   2
```

pandas.DataFrame.drop

`DataFrame.drop`(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

Returns

dropped [pandas.DataFrame]

Raises

KeyError If none of the labels are found in the selected axis

See also:

DataFrame.loc Label-location based indexer for selection by label.

DataFrame.dropna Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

pandas.DataFrame.drop_duplicates

`DataFrame.drop_duplicates`(*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Parameters

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

Returns

deduplicated [DataFrame]

pandas.DataFrame.droplevel

DataFrame.**droplevel**(*level*, *axis=0*)

Return DataFrame with requested index / column level(s) removed.

New in version 0.24.0.

Parameters

level [int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis [{0 or 'index', 1 or 'columns'}, default 0]

Returns

DataFrame.droplevel()

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...    ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2      3   4
5 6      7   8
9 10    11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2        3   4
6        7   8
10      11  12
```

```
>>> df.droplevel('level2', axis=1)
level_1   c   d
a b
1 2      3   4
5 6      7   8
9 10    11  12
```

pandas.DataFrame.dropna

`DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)`

Remove missing values.

See the *User Guide* for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0: Pass tuple or list to drop on multiple axes. Only a single axis is allowed.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

Returns

DataFrame DataFrame with NA entries dropped from it.

See also:

DataFrame.isna Indicate missing values.

DataFrame.notna Indicate existing (non-missing) values.

DataFrame.fillna Replace missing values.

Series.dropna Drop missing values.

Index.dropna Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

pandas.DataFrame.duplicated

`DataFrame.duplicated`(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns.

Parameters

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

Returns

duplicated [Series]

pandas.DataFrame.eq

`DataFrame.eq(other, axis='columns', level=None)`

Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

`DataFrame.eq` Compare DataFrames for equality elementwise.

`DataFrame.ne` Compare DataFrames for inequality elementwise.

`DataFrame.le` Compare DataFrames for less than inequality or equality elementwise.

`DataFrame.lt` Compare DataFrames for strictly less than inequality elementwise.

`DataFrame.ge` Compare DataFrames for greater than inequality or equality elementwise.

`DataFrame.gt` Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
     cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
     cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
     cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
     cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
     cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.equals

`DataFrame.equals(other)`

Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal. The column headers do not need to have the same type, but the elements within the columns must be the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

assert_series_equal Return True if left and right Series are equal, False otherwise.

assert_frame_equal Return True if left and right DataFrames are equal, False otherwise.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Notes

This function requires that the elements have the same dtype as their respective elements in the other Series or DataFrame. However, the column labels do not need to have the same type, as long as they are still considered equal.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

pandas.DataFrame.eval

`DataFrame.eval`(*expr*, *inplace=False*, ***kwargs*)

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

Parameters

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

Returns

ndarray, scalar, or pandas object The result of the evaluation.

See also:

`DataFrame.query` Evaluates a boolean expression to query the columns of a frame.

`DataFrame.assign` Can evaluate an expression or function to create new values for a column.

`pandas.eval` Evaluate a Python expression as a string using various backends.

Notes

For more details see the API documentation for `eval()`. For detailed examples see *enhancing performance with eval*.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
```

(continues on next page)

(continued from previous page)

```
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

pandas.DataFrame.ewm

`DataFrame.ewm`(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)

Provides exponential weighted functions.

New in version 0.18.0.

Parameters

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com)$, for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1)$, for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife)$, for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [bool, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [bool, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations.

expanding Provides expanding transformations.

Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

pandas.DataFrame.expanding

`DataFrame.`**`expanding`**(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [bool, default False] Set the labels at the center of the window.

axis [int or str, default 0]

Returns

a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations.

ewm Provides exponential weighted functions.

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

pandas.DataFrame.ffill

`DataFrame.`**`ffill`**(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for *`DataFrame.fillna()`* with `method='ffill'`.

pandas.DataFrame.fillna

`DataFrame.`**`fillna`**(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)

Fill NA/NaN values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis [{0 or 'index', 1 or 'columns'}]

inplace [boolean, default False] If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns

filled [DataFrame]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

pandas.DataFrame.filter

`DataFrame.filter`(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items [list-like] List of axis to restrict to (must not all be present).

like [string] Keep axis where "arg in col == True".

regex [string (regular expression)] Keep axis with re.search(regex, col) == True.

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

same type as input object

See also:

DataFrame.loc

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1,2,3], [4,5,6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

pandas.DataFrame.first

DataFrame.**first**(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

last Select final periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

pandas.DataFrame.first_valid_index

DataFrame.**first_valid_index**()

Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.DataFrame.floordiv

DataFrame.**floordiv**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rfloordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multiindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.from_csv

classmethod DataFrame.**from_csv**(*path*, *header=0*, *sep=', '*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use *pandas.read_csv()* instead.

It is preferable to use the more powerful *pandas.read_csv()* for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred *pandas.read_csv()* in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

Parameters

path [string file path or file handle / StringIO]

header [int, default 0] Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format [boolean, default False] If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

Returns

y [DataFrame]

See also:

pandas.read_csv

pandas.DataFrame.from_dict

classmethod DataFrame.**from_dict**(*data*, *orient='columns'*, *dtype=None*, *columns=None*)

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

Parameters

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

Returns

pandas.DataFrame

See also:

DataFrame.from_records DataFrame from ndarray (structured dtype), list of tuples, dict, or DataFrame.

DataFrame DataFrame object creation using constructor.

Examples

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

pandas.DataFrame.from_items

classmethod DataFrame.**from_items**(*items*, *columns=None*, *orient='columns'*)

Construct a DataFrame from a list of tuples.

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

Parameters

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

Returns

frame [DataFrame]

pandas.DataFrame.from_records

classmethod DataFrame.**from_records**(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame.

Parameters

data [ndarray (structured dtype), list of tuples, dict, or DataFrame]

index [string, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

nrows [int, default None] Number of rows to read if data is an iterator

Returns

df [DataFrame]

pandas.DataFrame.ge

DataFrame.**ge**(*other*, *axis='columns'*, *level=None*)

Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.get

`DataFrame.get(key, default=None)`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

pandas.DataFrame.get_dtype_counts

`DataFrame.get_dtype_counts()`

Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
```

(continues on next page)

(continued from previous page)

```
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

pandas.DataFrame.get_ftype_counts

DataFrame.**get_ftype_counts**()

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()  # doctest: +SKIP
float64:dense     1
int64:dense       1
object:dense      1
dtype: int64
```

pandas.DataFrame.get_value

DataFrame.**get_value**(*index*, *col*, *takeable=False*)

Quickly retrieve single value at passed column and index.

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

Parameters

index [row label]

col [column label]

takeable [interpret the index/col as indexers, default False]

Returns

value [scalar value]

pandas.DataFrame.get_values

`DataFrame.get_values()`

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

Returns

numpy.ndarray Numpy representation of DataFrame

See also:

values Numpy representation of DataFrame.

pandas.SparseArray Container for sparse data.

Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

pandas.DataFrame.groupby

`DataFrame.groupby`(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group DataFrame or Series using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis [{0 or 'index', 1 or 'columns'}, default 0] Split along rows (0) or columns (1).

level [int, level name, or sequence of such, default None] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

as_index [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output.

sort [bool, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

group_keys [bool, default True] When calling apply, add group keys to index to identify pieces.

squeeze [bool, default False] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

observed [bool, default False] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

****kwargs** Optional, only accepts keyword argument 'mutated' and is passed to groupby.

Returns

DataFrameGroupBy or SeriesGroupBy Depends on the calling object and returns groupby object that contains information about the groups.

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

```
>>> df = pd.DataFrame({'Animal' : ['Falcon', 'Falcon',
...                                'Parrot', 'Parrot'],
...                    'Max Speed' : [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the *level* parameter:

```
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...           ['Capitve', 'Wild', 'Capitve', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed' : [390., 350., 30., 20.]},
...                    index=index)
>>> df
                Max Speed
Animal Type
Falcon Capitve      390.0
       Wild         350.0
Parrot Capitve       30.0
       Wild          20.0
>>> df.groupby(level=0).mean()
        Max Speed
Animal
Falcon      370.0
Parrot       25.0
>>> df.groupby(level=1).mean()
         Max Speed
Type
Capitve      210.0
Wild         185.0
```

pandas.DataFrame.gt

`DataFrame.gt`(*other*, *axis='columns'*, *level=None*)

Greater than of dataframe and other, element-wise (binary operator *gt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B   True     False
C  False      True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
```

(continues on next page)

(continued from previous page)

```
   B   150       250
   C   100       300
Q2 A   150       200
   B   300       175
   C   220       225
```

```
>>> df.le(df_multiindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.head

`DataFrame.head(n=5)`

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [same type as caller] The first *n* rows of the caller object.

See also:

DataFrame.tail Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

pandas.DataFrame.hist

`DataFrame.hist`(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)

Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

Parameters

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

axes [matplotlib.AxesSubplot or numpy.ndarray of them]

See also:

`matplotlib.pyplot.hist` Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index= ['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```

pandas.DataFrame.idxmax

`DataFrame.`**`idxmax`**(*axis=0, skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmax [Series]

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmax`

Notes

This method is the DataFrame version of `ndarray.argmax`.

pandas.DataFrame.idxmin

`DataFrame.idxmin(axis=0, skipna=True)`

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmin [Series]

Raises

ValueError

- If the row/column is empty

See also:

Series.idxmin

Notes

This method is the DataFrame version of `ndarray.argmin`.

pandas.DataFrame.infer_objects

`DataFrame.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
```

(continues on next page)

(continued from previous page)

```
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

pandas.DataFrame.info

`DataFrame.info`(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

Parameters

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

Returns

None This method prints a summary of a DataFrame and returns None.

See also:

DataFrame.describe Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage Memory usage of DataFrame columns.

Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...           encoding="utf-8") as f:  # doctest: +SKIP
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
```

(continues on next page)

(continued from previous page)

```
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

pandas.DataFrame.insert

`DataFrame.`**`insert`**(*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to True.

Parameters

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value [int, Series, or array-like]

allow_duplicates [bool, optional]

pandas.DataFrame.interpolate

`DataFrame.`**`interpolate`**(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.

- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

New in version 0.18.1: Added support for the 'akima' method. Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in SciPy 0.18; backwards-compatible with SciPy < 0.18

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.21.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all `NaN` values

See also:

fillna Fill missing values using different methods.

`scipy.interpolate.Akima1DInterpolator` Piecewise cubic polynomials (Akima interpolator).

`scipy.interpolate.BPoly.from_derivatives` Piecewise polynomial in the Bernstein basis.

`scipy.interpolate.interp1d` Interpolate a 1-D function.

`scipy.interpolate.KroghInterpolator` Interpolate polynomial (Krogh interpolator).

`scipy.interpolate.PchipInterpolator` PCHIP 1-d monotonic cubic interpolation.

`scipy.interpolate.CubicSpline` Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in `NaN` in a *Series* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series by padding, but filling at most two consecutive `NaN` at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in NaN in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an order (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains NaN, because there is no entry befofe it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0,  np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

pandas.DataFrame.isin

`DataFrame.isin(values)`

Whether each element in the DataFrame is contained in values.

Parameters

values [iterable, Series, DataFrame or dict] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dict, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

Returns

DataFrame DataFrame of booleans showing whether each element in the DataFrame is contained in values.

See also:

DataFrame.eq Equality test for DataFrame.

Series.isin Equivalent method on Series.

Series.str.contains Test if pattern or regex is contained within a string of a Series or Index.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                   index=['falcon', 'dog'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
```

When `values` is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```
>>> df.isin([0, 2])
        num_legs  num_wings
falcon      True       True
dog        False       True
```

When `values` is a dict, we can pass values to check for each column separately:

```
>>> df.isin({'num_wings': [0, 3]})
        num_legs  num_wings
falcon     False      False
dog        False       True
```

When `values` is a Series or DataFrame the index and column must match. Note that 'falcon' does not match based on the number of legs in df2.

```
>>> other = pd.DataFrame({'num_legs': [8, 2],'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon      True       True
dog        False      False
```

pandas.DataFrame.isna

`DataFrame.`**`isna`**`()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.isnull Alias of isna.

DataFrame.notna Boolean inverse of isna.

DataFrame.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.DataFrame.isnull

`DataFrame.`**`isnull`**`()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.isnull Alias of isna.

DataFrame.notna Boolean inverse of isna.

DataFrame.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.DataFrame.items

`DataFrame.items()`

Iterator over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

label [object] The column names for the DataFrame being iterated over.

content [Series] The column entries belonging to each label, as a Series.

See also:

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.itertuples Iterate over DataFrame rows as namedtuples of the values.

Examples

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                    'population': [1864, 22000, 80000]},
...                    index=['panda', 'polar', 'koala'])
>>> df
        species   population
panda   bear      1864
polar   bear      22000
koala   marsupial 80000
>>> for label, content in df.iteritems():
...     print('label:', label)
...     print('content:', content, sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala    marsupial
Name: species, dtype: object
label: population
content:
panda     1864
polar    22000
koala    80000
Name: population, dtype: int64
```

pandas.DataFrame.iteritems

`DataFrame.iteritems()`

Iterator over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

Yields

label [object] The column names for the DataFrame being iterated over.

content [Series] The column entries belonging to each label, as a Series.

See also:

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.itertuples Iterate over DataFrame rows as namedtuples of the values.

Examples

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                   'population': [1864, 22000, 80000]},
...                   index=['panda', 'polar', 'koala'])
>>> df
        species   population
panda   bear      1864
polar   bear      22000
koala   marsupial 80000
>>> for label, content in df.iteritems():
...     print('label:', label)
...     print('content:', content, sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala    marsupial
Name: species, dtype: object
label: population
content:
panda     1864
polar    22000
koala    80000
Name: population, dtype: int64
```

pandas.DataFrame.iterrows

`DataFrame.iterrows()`

Iterate over DataFrame rows as (index, Series) pairs.

Yields

index [label or tuple of label] The index of the row. A tuple for a *MultiIndex*.

data [Series] The data of the row as a Series.

it [generator] A generator that iterates over the rows of the frame.

See also:

itertuples Iterate over DataFrame rows as namedtuples of the values.

iteritems Iterate over (column name, Series) pairs.

Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use `itertuples()` which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

pandas.DataFrame.itertuples

`DataFrame.itertuples`(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples.

Parameters

index [bool, default True] If True, return the index as the first element of the tuple.

name [str, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

Yields

collections.namedtuple Yields a namedtuple for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

See also:

DataFrame.iterrows Iterate over DataFrame rows as (index, Series) pairs.

DataFrame.iteritems Iterate over (column name, Series) pairs.

Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> for row in df.itertuples():
```

(continues on next page)

(continued from previous page)

```
...     print(row)
...
Pandas(Index='dog', num_legs=4, num_wings=0)
Pandas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the *index* parameter to False we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):
...     print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)
```

With the *name* parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

pandas.DataFrame.join

DataFrame.**join**(*other*, *on=None*, *how='left'*, *lsuffix="*, *rsuffix="*, *sort=False*)

Join columns of another DataFrame.

Join columns with *other* DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

other [DataFrame, Series, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

on [str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how [{'left', 'right', 'outer', 'inner'}, default 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use *other*'s index.
- outer: form union of calling frame's index (or column if on is specified) with *other*'s index, and sort it. lexicographically.
- inner: form intersection of calling frame's index (or column if on is specified) with *other*'s index, preserving the order of the calling's one.

lsuffix [str, default ''] Suffix to use from left frame's overlapping columns.

rsuffix [str, default ''] Suffix to use from right frame's overlapping columns.

sort [bool, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

Returns

DataFrame A dataframe containing columns from both the caller and *other*.

See also:

DataFrame.merge For column(s)-on-columns(s) operations.

Notes

Parameters *on*, *lsuffix*, and *rsuffix* are not supported when passing a list of *DataFrame* objects.

Support for specifying index levels as the *on* parameter was added in version 0.23.0.

Examples

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
  key   A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
4  K4  A4
5  K5  A5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
  key   B
0  K0  B0
1  K1  B1
2  K2  B2
```

Join DataFrames using their indexes.

```
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
  key_caller   A key_other    B
0         K0  A0        K0   B0
1         K1  A1        K1   B1
2         K2  A2        K2   B2
3         K3  A3       NaN  NaN
4         K4  A4       NaN  NaN
5         K5  A5       NaN  NaN
```

If we want to join using the key columns, we need to set key to be the index in both *df* and *other*. The joined DataFrame will have key as its index.

```
>>> df.set_index('key').join(other.set_index('key'))
      A    B
key
K0   A0   B0
```

(continues on next page)

(continued from previous page)

```
K1    A1    B1
K2    A2    B2
K3    A3   NaN
K4    A4   NaN
K5    A5   NaN
```

Another option to join using the key columns is to use the *on* parameter. DataFrame.join always uses *other*'s index but we can use any column in *df*. This method preserves the original DataFrame's index in the result.

```
>>> df.join(other.set_index('key'), on='key')
  key   A    B
0  K0  A0   B0
1  K1  A1   B1
2  K2  A2   B2
3  K3  A3  NaN
4  K4  A4  NaN
5  K5  A5  NaN
```

pandas.DataFrame.keys

`DataFrame.keys()`

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

pandas.DataFrame.kurt

`DataFrame.kurt(`*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*`)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [Series or DataFrame (if level specified)]

pandas.DataFrame.kurtosis

`DataFrame.kurtosis(`*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*`)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal ==

0.0). Normalized by N-1.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [Series or DataFrame (if level specified)]

pandas.DataFrame.last

`DataFrame.last(offset)`

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

pandas.DataFrame.last_valid_index

DataFrame.**last_valid_index**()

Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.DataFrame.le

DataFrame.**le**(*other*, *axis='columns'*, *level=None*)

Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.lookup

`DataFrame.lookup`(*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame.

Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

Parameters

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Notes

Akin to:

```
result = [df.get_value(row, col)
          for row, col in zip(row_labels, col_labels)]
```

Examples

values [ndarray] The found values

pandas.DataFrame.lt

DataFrame.**lt**(*other*, *axis='columns'*, *level=None*)

Less than of dataframe and other, element-wise (binary operator *lt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
```

(continues on next page)

(continued from previous page)

```
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.mad

DataFrame.**mad**(*axis=None*, *skipna=None*, *level=None*)

Return the mean absolute deviation of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mad [Series or DataFrame (if level specified)]

pandas.DataFrame.mask

`DataFrame.mask(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)`

Replace values where the condition is True.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

`DataFrame.where()` Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `mask` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
```

(continues on next page)

(continued from previous page)

```
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.DataFrame.max

DataFrame.**max**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the maximum of the values for the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

max [Series or DataFrame (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

Max using level names, as well as indices.

```
>>> s.max(level='blooded')
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

```
>>> s.max(level=0)
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

pandas.DataFrame.mean

DataFrame.**mean**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mean [Series or DataFrame (if level specified)]

pandas.DataFrame.median

`DataFrame.median`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

median [Series or DataFrame (if level specified)]

pandas.DataFrame.melt

`DataFrame.melt`(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

Unpivots a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

Parameters

frame [DataFrame]

id_vars [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

See also:

melt, *pivot_table*, *DataFrame.pivot*

Examples

```
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

pandas.DataFrame.memory_usage

DataFrame.**memory_usage**(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

Returns

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of an ndarray.

Series.memory_usage Bytes consumed by a Series.

pandas.Categorical Memory-efficient array for string values with many repeated values.

DataFrame.info Concise summary of a DataFrame.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64          40000
float64        40000
complex128     80000
object         40000
bool            5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

pandas.DataFrame.merge

`DataFrame.merge`(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

right [DataFrame or named Series] Object to merge with.

how [{'left', 'right', 'outer', 'inner'}, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes [tuple of (str, str), default ('_x', '_y')] Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [str, optional] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Returns

DataFrame A DataFrame of the two merged objects.

See also:

merge_ordered Merge with optional filling/interpolation.

merge_asof Merge on nearest keys.

DataFrame.join Similar method using indices.

Notes

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0 Support for merging named Series objects was added in version 0.24.0

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
    lkey value
0   foo      1
1   bar      2
2   baz      3
3   foo      5
>>> df2
    rkey value
0   foo      5
1   bar      6
2   baz      7
3   foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
  lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
  lkey  value_left rkey  value_right
0  foo           1  foo            5
1  foo           1  foo            8
2  foo           5  foo            5
3  foo           5  foo            8
4  bar           2  bar            6
5  baz           3  baz            7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

pandas.DataFrame.min

`DataFrame.min`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the minimum of the values for the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

min [Series or DataFrame (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

Min using level names, as well as indices.

```
>>> s.min(level='blooded')
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

```
>>> s.min(level=0)
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

pandas.DataFrame.mod

`DataFrame.mod`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.mode

DataFrame.**mode**(*axis=0*, *numeric_only=False*, *dropna=True*)

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to iterate over while searching for the mode:

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [bool, default False] If True, only apply to numeric columns.

dropna [bool, default True] Don't consider counts of NaN/NaT.

New in version 0.24.0.

Returns

DataFrame The modes of each column or row.

See also:

Series.mode Return the highest frequency value in a Series.

Series.value_counts Return the counts of values in a Series.

Examples

```
>>> df = pd.DataFrame([('bird', 2, 2),
...                    ('mammal', 4, np.nan),
...                    ('arthropod', 8, 0),
...                    ('bird', 2, np.nan)],
...                   index=('falcon', 'horse', 'spider', 'ostrich'),
...                   columns=('species', 'legs', 'wings'))
>>> df
           species  legs  wings
falcon        bird     2    2.0
horse       mammal     4    NaN
spider   arthropod     8    0.0
ostrich       bird     2    NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. The second row of species and legs contains NaN, because they have only one mode, but the DataFrame has two rows.

```
>>> df.mode()
  species  legs  wings
0    bird   2.0    0.0
1     NaN   NaN    2.0
```

Setting dropna=False NaN values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
  species  legs  wings
0    bird     2    NaN
```

Setting `numeric_only=True`, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
   legs  wings
0   2.0    0.0
1   NaN    2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
           0    1
falcon   2.0  NaN
horse    4.0  NaN
spider   0.0  8.0
ostrich  2.0  NaN
```

pandas.DataFrame.mul

`DataFrame.mul`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
```

(continues on next page)

(continued from previous page)

```
B square          0.0       0.0
  pentagon        0.0       0.0
  hexagon         0.0       0.0
```

pandas.DataFrame.multiply

DataFrame.**multiply**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle          -1      359
triangle         2      179
rectangle        3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.ne

`DataFrame.ne`(*other*, *axis='columns'*, *level=None*)

Not equal to of dataframe and other, element-wise (binary operator *ne*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, =!, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
```

(continues on next page)

(continued from previous page)

```
B   False     False
C    True     False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
```

(continues on next page)

(continued from previous page)

```
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multiindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multiindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multiindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

pandas.DataFrame.nlargest

`DataFrame.nlargest(n, columns, keep='first')`

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

Parameters

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)
- **all** [do not drop any duplicates, even it means] selecting more than *n* items.

New in version 0.24.0.

Returns

DataFrame The first *n* rows ordered by the given columns in descending order.

See also:

DataFrame.nsmallest Return the first *n* rows ordered by *columns* in ascending order.

DataFrame.sort_values Sort DataFrame by the values.

DataFrame.head Return the first *n* rows without re-ordering.

Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column “population”.

```
>>> df.nlargest(3, 'population')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Malta       434000    12011      MT
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'population', keep='last')
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nlargest(3, 'population', keep='all')
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
        population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Brunei      434000    12128      BN
```

pandas.DataFrame.notna

DataFrame.**notna**()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull Alias of notna.

DataFrame.isna Boolean inverse of notna.

DataFrame.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.DataFrame.notnull

`DataFrame.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.notnull Alias of notna.

DataFrame.isna Boolean inverse of notna.

DataFrame.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.DataFrame.nsmallest

DataFrame.**nsmallest**(*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in ascending order.

Return the first *n* rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant.

Parameters

n [int] Number of items to retrieve.

columns [list or str] Column name or names to order by.

keep [{'first', 'last', 'all'}, default 'first'] Where there are duplicate values:

- `first` : take the first occurrence.
- `last` : take the last occurrence.
- `all` : do not drop any duplicates, even it means selecting more than *n* items.

New in version 0.24.0.

Returns

DataFrame

See also:

DataFrame.nlargest Return the first *n* rows ordered by *columns* in descending order.

DataFrame.sort_values Sort DataFrame by the values.

DataFrame.head Return the first *n* rows without re-ordering.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column "a".

```
>>> df.nsmallest(3, 'population')
          population  GDP alpha-2
Nauru          11300  182      NR
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nsmallest(3, 'population', keep='last')
          population  GDP alpha-2
Anguilla       11300  311      AI
Tuvalu         11300   38      TV
Nauru          11300  182      NR
```

When using `keep='all'`, all duplicate items are maintained:

```
>>> df.nsmallest(3, 'population', keep='all')
          population  GDP alpha-2
Nauru          11300  182      NR
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Nauru          11300  182      NR
Anguilla       11300  311      AI
```

pandas.DataFrame.nunique

DataFrame.**nunique**(*axis=0*, *dropna=True*)

Count distinct observations over requested axis.

Return Series with number of distinct observations. Can ignore NaN values.

New in version 0.20.0.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

dropna [bool, default True] Don't include NaN in the counts.

Returns

nunique [Series]

See also:

Series.nunique Method nunique for Series.

DataFrame.count Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

pandas.DataFrame.pct_change

DataFrame.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

chg [Series or DataFrame] The same type as the calling object.

See also:

Series.diff Compute the difference of two elements in a Series.

DataFrame.diff Compute the difference of two elements in a DataFrame.

Series.shift Shift the index by some number of periods.

DataFrame.shift Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pandas.DataFrame.pipe

`DataFrame.pipe(func, *args, **kwargs)`

Apply func(self, *args, **kwargs).

Parameters

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where data_keyword is a string indicating the keyword of callable that expects the NDFrame.

args [iterable, optional] positional arguments passed into func.

kwargs [mapping, optional] a dictionary of keyword arguments passed into func.

Returns

object [the return type of func.]

See also:

DataFrame.apply, *DataFrame.applymap*, *Series.map*

Notes

Use .pipe when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose f takes its data as arg2:

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.pivot

DataFrame.**pivot**(*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the *User Guide* for more on reshaping.

Parameters

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

Returns

DataFrame Returns reshaped DataFrame.

Raises

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

See also:

DataFrame.pivot_table Generalization of pivot that can handle duplicate values for one index/column pair.

DataFrame.unstack Pivot based on the index values instead of a column.

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pandas.DataFrame.pivot_table

DataFrame.**pivot_table**(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values [column to aggregate, optional]

index [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

Returns

table [DataFrame]

See also:

DataFrame.pivot Pivot without aggregation that can handle non-numeric data.

Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

This first example aggregates values by taking the sum.

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two    NaN      6
```

We can also fill missing values using the *fill_value* parameter.

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum, fill_value=0)
>>> table
C        large  small
A   B
bar one      4      5
    two      7      6
foo one      4      1
    two      0      6
```

The next example aggregates by taking the mean across multiple columns.

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': np.mean})
>>> table
                  D         E
               mean      mean
A   C
bar large  5.500000  7.500000
```

(continues on next page)

(continued from previous page)

```
    small  5.500000  8.500000
foo large  2.000000  4.500000
    small  2.333333  4.333333
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max      mean min
A   C
bar large  5.500000  9   7.500000   6
    small  5.500000  9   8.500000   8
foo large  2.000000  5   4.500000   4
    small  2.333333  6   4.333333   2
```

pandas.DataFrame.plot

`DataFrame.plot`(*x=None*, *y=None*, *kind='line'*, *ax=None*, *subplots=False*, *sharex=None*, *sharey=False*, *layout=None*, *figsize=None*, *use_index=True*, *title=None*, *grid=None*, *legend=True*, *style=None*, *logx=False*, *logy=False*, *loglog=False*, *xticks=None*, *yticks=None*, *xlim=None*, *ylim=None*, *rot=None*, *fontsize=None*, *colormap=None*, *table=False*, *yerr=None*, *xerr=None*, *secondary_y=False*, *sort_columns=False*, ***kwds*)

Make plots of DataFrame using matplotlib / pylab.

New in version 0.17.0: Each plot kind has a corresponding method on the `DataFrame.plot` accessor: `df.plot(kind='line')` is equivalent to `df.plot.line()`.

Parameters

data [DataFrame]

x [label or position, default None]

y [label, position or list of label, positions, default None] Allows plotting of one column versus another

kind [str]

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot
- 'scatter' : scatter plot

- 'hexbin' : hexbin plot

ax [matplotlib axes object, default None]

subplots [boolean, default False] Make separate subplots for each column

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in; Be aware, that passing in both an ax and sharex=True will alter all x axis labels for all axis in a figure!

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible

layout [tuple (optional)] (rows, columns) for the layout of subplots

figsize [a tuple (width, height) in inches]

use_index [boolean, default True] Use index as ticks for x axis

title [string or list] Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and *subplots* is True, print each item in the list above the corresponding subplot.

grid [boolean, default None (matlab style default)] Axis grid lines

legend [False/True/'reverse'] Place legend on axis subplots

style [list or dict] matplotlib line style per column

logx [boolean, default False] Use log scaling on x axis

logy [boolean, default False] Use log scaling on y axis

loglog [boolean, default False] Use log scaling on both x and y axes

xticks [sequence] Values to use for the xticks

yticks [sequence] Values to use for the yticks

xlim [2-tuple/list]

ylim [2-tuple/list]

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots)

fontsize [int, default None] Font size for xticks and yticks

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

colorbar [boolean, optional] If True, plot colorbar (only relevant for 'scatter' and 'hexbin' plots)

position [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

table [boolean, Series or DataFrame, default False] If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib's default layout. If a Series or DataFrame is passed, use passed data to draw a table.

yerr [DataFrame, Series, array-like, dict and str] See *Plotting with Error Bars* for detail.

xerr [same types as yerr.]

stacked [boolean, default False in line and] bar plots, and True in area plot. If True, create stacked plot.

sort_columns [boolean, default False] Sort column names to determine plot ordering

secondary_y [boolean or sequence, default False] Whether to plot on the secondary y-axis If a list/tuple, which columns to plot on secondary y-axis

mark_right [boolean, default True] When using a secondary_y axis, automatically mark the column labels with "(right)" in the legend

'kwds'** [keywords] Options to pass to matplotlib plotting method

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them]

Notes

- See matplotlib documentation online for more on this subject
- If *kind* = 'bar' or 'barh', you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)
- If *kind* = 'scatter' and the argument *c* is the name of a dataframe column, the values of that column are used to color each point.
- If *kind* = 'hexbin', you can control the size of the bins with the *gridsize* argument. By default, a histogram of the counts around each *(x, y)* point is computed. You can specify alternative aggregations by passing values to the *C* and *reduce_C_function* arguments. *C* specifies the value at each *(x, y)* point and *reduce_C_function* is a function of one argument that reduces all the values in a bin to a single number (e.g. *mean*, *max*, *sum*, *std*).

pandas.DataFrame.pop

`DataFrame.pop(item)`

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [Series]

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pandas.DataFrame.pow

`DataFrame.pow`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rpow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.prod

`DataFrame.prod(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the product of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [Series or DataFrame (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.DataFrame.product

`DataFrame.product(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the product of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [Series or DataFrame (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.DataFrame.quantile

`DataFrame.quantile`(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)
Return values at the given quantile over requested axis.

Parameters

q [float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis [{0, 1, 'index', 'columns'} (default 0)] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only [bool, default True] If False, the quantile of datetime and timedelta data will be computed as well.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

New in version 0.18.0.

Returns

quantiles [Series or DataFrame]

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

See also:

`core.window.Rolling.quantile` Rolling quantile.

`numpy.percentile` Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.DataFrame.query

`DataFrame.query(expr, inplace=False, **kwargs)`

Query the columns of a DataFrame with a boolean expression.

Parameters

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

Returns

q [DataFrame]

See also:

`pandas.eval`, `DataFrame.eval`

Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in *indexing*.

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

pandas.DataFrame.radd

`DataFrame.radd(other, axis='columns', level=None, fill_value=None)`

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *add*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

`DataFrame.add` Add DataFrames.

`DataFrame.sub` Subtract DataFrames.

`DataFrame.mul` Multiply DataFrames.

`DataFrame.div` Divide DataFrames (float division).

`DataFrame.truediv` Divide DataFrames (float division).

`DataFrame.floordiv` Divide DataFrames (integer division).

`DataFrame.mod` Calculate modulo (remainder after division).

`DataFrame.pow` Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rank

`DataFrame.rank`(*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

pandas.DataFrame.rdiv

`DataFrame.`**`rdiv`**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

`DataFrame.add` Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
```

(continues on next page)

(continued from previous page)

```
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multiindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.reindex

`DataFrame.reindex`(*labels=None*, *index=None*, *columns=None*, *axis=None*, *method=None*, *copy=True*, *level=None*, *fill_value=nan*, *limit=None*, *tolerance=None*)

Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional] New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit [int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

DataFrame with changed index.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...      'http_status': [200,200,404,404,301],
...      'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...       index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
```

(continues on next page)

(continued from previous page)

```
2009-12-30       NaN
2009-12-31       NaN
2010-01-01     100.0
2010-01-02     101.0
2010-01-03       NaN
2010-01-04     100.0
2010-01-05      89.0
2010-01-06      88.0
2010-01-07       NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the *user guide* for more.

pandas.DataFrame.reindex_axis

`DataFrame.`**`reindex_axis`**(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index.

Deprecated since version 0.21.0: Use *reindex* instead.

By default, places NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data.

axis [{0 or 'index', 1 or 'columns'}] Indicate whether to use rows or columns.

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps.

- pad / ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use next valid observation to fill gap.
- nearest: use nearest valid observations to fill gap.

level [int or str] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, optional] Maximum number of consecutive elements to forward or backward fill.

fill_value [float, default NaN] Value used to fill in locations having no value in the previous index.

New in version 0.21.0: (list-like tolerance)

Returns

DataFrame Returns a new DataFrame object with new indices, unless the new index is equivalent to the current one and copy=False.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex Change to new indices or expand indices.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> df.reindex(['num_wings', 'num_legs', 'num_heads'],
...            axis='columns')
      num_wings  num_legs  num_heads
dog           0         4        NaN
hawk          2         2        NaN
```

pandas.DataFrame.reindex_like

`DataFrame.reindex_like(other, method=None, copy=True, limit=None, tolerance=None)`

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other [Object of the same data type] Its row and column indices are used to define the new indices of this object.

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

Series or DataFrame Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...     columns=['temp_celsius', 'temp_fahrenheit', 'windspeed'],
...     index=pd.date_range(start='2014-02-12',
...                         end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...     columns=['temp_celsius', 'windspeed'],
...     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                             '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

pandas.DataFrame.rename

`DataFrame.rename`(*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*, *level=None*)

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the *user guide* for more.

Parameters

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

Returns

renamed [DataFrame]

See also:

pandas.DataFrame.rename_axis

Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

pandas.DataFrame.rename_axis

`DataFrame.`**`rename_axis`**(*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*)

Set the name of the axis for the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set the axis name attribute.

index, columns [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

Changed in version 0.24.0.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to rename.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None The same type as the caller or None if *inplace* is True.

See also:

Series.rename Alter Series index labels or name.

DataFrame.rename Alter DataFrame index labels or name.

Index.rename Set new names on index.

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
>>> df
limbs   num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

pandas.DataFrame.reorder_levels

`DataFrame.`**`reorder_levels`**(*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

Returns

type of caller (new object)

pandas.DataFrame.replace

`DataFrame.`**`replace`**`(`*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*`)`

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

  See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [bool, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

DataFrame Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

DataFrame.fillna Fill NA values.

DataFrame.where Replace values based on boolean condition.

Series.str.replace Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar 'to_replace' and 'value'

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like 'to_replace'

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
```

(continues on next page)

(continued from previous page)

```
2    3
3    3
4    4
dtype: int64
```

dict-like 'to_replace'

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
      A    B
```

(continues on next page)

(continued from previous page)

```
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

pandas.DataFrame.resample

`DataFrame.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)`

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or pass datetime-like values to the *on* or *level* keyword.

Parameters

rule [str] The offset string or object representing target conversion.

how [str] Method for down/re-sampling, default to 'mean' for downsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).mean()`, or `.resample(...).apply(<func>)`

axis [{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this will default to 0, i.e. along the rows. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

fill_method [str, default None] Filling method for upsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).<func>()`, e.g. `.resample(...).pad()`

closed [{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind [{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

loffset [timedelta, default None] Adjust the resampled time labels.

limit [int, default None] Maximum size gap when reindexing with *fill_method*.

Deprecated since version 0.18.0.

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0.

on [str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

Series.resample Resample a Series.

DataFrame.resample Resample a DataFrame.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
```

(continues on next page)

(continued from previous page)

```
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                    freq='Q',
...                                                    periods=4))
>>> q
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...           'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
```

(continues on next page)

(continued from previous page)

```
7     19       50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df2 = pd.DataFrame(d2,
...                    index=pd.MultiIndex.from_product([days,
...                                                      ['morning',
...                                                       'afternoon']]
...                                                      ))
>>> df2
                      price  volume
2000-01-01 morning       10      50
           afternoon     11      60
2000-01-02 morning        9      40
           afternoon     13     100
2000-01-03 morning       14      50
           afternoon     18     100
2000-01-04 morning       17      40
           afternoon     19      50
>>> df2.resample('D', level=0).sum()
            price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

pandas.DataFrame.reset_index

DataFrame.**reset_index**(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

Parameters

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.

drop [bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [bool, default False] Modify the DataFrame in place (do not create a new object).

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

Returns

DataFrame DataFrame with the new index.

See also:

DataFrame.set_index Opposite of reset_index.

DataFrame.reindex Change to new indices or expand indices.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
```

(continues on next page)

(continued from previous page)

```
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
```

(continues on next page)

(continued from previous page)

```
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

pandas.DataFrame.rfloordiv

DataFrame.**rfloordiv**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle          -1      359
triangle         2      179
rectangle        3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rmod

`DataFrame.rmod(other, axis='columns', level=None, fill_value=None)`

Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rmul

`DataFrame.`**`rmul`**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rolling

`DataFrame.rolling`(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

Parameters

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, *min_periods* will default to 1. Otherwise, *min_periods* will default to the size of the window.

center [bool, default False] Set the labels at the center of the window.

win_type [str, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [str, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

axis [int or str, default 0]

closed [str, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

Returns

a Window or Rolling sub-classed for the particular operation

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions.

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`
- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                       B
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
```

pandas.DataFrame.round

DataFrame.**round**(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

Parameters

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

Returns

DataFrame

See also:

numpy.around, *Series.round*

Examples

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...      columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
               A         B         C
first   0.028208  0.992815  0.173891
second  0.038683  0.645646  0.577595
third   0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second   0.0   1   0.58
third    0.9   0   0.49
```

pandas.DataFrame.rpow

`DataFrame.rpow(other, axis='columns', level=None, fill_value=None)`

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

`DataFrame.add` Add DataFrames.

`DataFrame.sub` Subtract DataFrames.

`DataFrame.mul` Multiply DataFrames.

`DataFrame.div` Divide DataFrames (float division).

`DataFrame.truediv` Divide DataFrames (float division).

`DataFrame.floordiv` Divide DataFrames (integer division).

`DataFrame.mod` Calculate modulo (remainder after division).

`DataFrame.pow` Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle          -1      359
triangle         2      179
rectangle        3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rsub

`DataFrame.rsub(other, axis='columns', level=None, fill_value=None)`

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

`DataFrame.add` Add DataFrames.

`DataFrame.sub` Subtract DataFrames.

`DataFrame.mul` Multiply DataFrames.

`DataFrame.div` Divide DataFrames (float division).

`DataFrame.truediv` Divide DataFrames (float division).

`DataFrame.floordiv` Divide DataFrames (integer division).

`DataFrame.mod` Calculate modulo (remainder after division).

`DataFrame.pow` Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle         3       180
rectangle        4       360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.rtruediv

`DataFrame.rtruediv`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.sample

`DataFrame.sample`(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [bool, default False] Sample with or without replacement.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

Series or DataFrame A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`numpy.random.choice` Generates a random sample from a given 1-D numpy array.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

pandas.DataFrame.select

`DataFrame.select`(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria.

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [same type as caller]

pandas.DataFrame.select_dtypes

`DataFrame.select_dtypes(include=None, exclude=None)`

Return a subset of the DataFrame's columns based on the column dtypes.

Parameters

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

Returns

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

Raises

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

Notes

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

Examples

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

pandas.DataFrame.sem

`DataFrame.sem`(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [Series or DataFrame (if level specified)]

pandas.DataFrame.set_axis

`DataFrame.set_axis(labels, axis=0, inplace=None)`

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

Parameters

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

DataFrame.rename_axis Alter the name of the index or columns.

Examples

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

pandas.DataFrame.set_index

`DataFrame.set_index`(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

Parameters

keys [label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, "array" encompasses `Series`, `Index` and `np.ndarray`.

drop [bool, default True] Delete columns to be used as the new index.

append [bool, default False] Whether to append columns to existing index.

inplace [bool, default False] Modify the DataFrame in place (do not create a new object).

verify_integrity [bool, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method.

Returns

DataFrame Changed row labels.

See also:

DataFrame.reset_index Opposite of set_index.

DataFrame.reindex Change to new indices or expand indices.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale': [55, 40, 84, 31]})
>>> df
   month  year  sale
0      1  2012    55
1      4  2014    40
2      7  2013    84
3     10  2014    31
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       year  sale
month
1      2012    55
4      2014    40
7      2013    84
10     2014    31
```

Create a MultiIndex using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1       55
2014  4       40
2013  7       84
2014  10      31
```

Create a MultiIndex using an Index and a column:

```
>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
         month  sale
   year
```

(continues on next page)

(continued from previous page)

```
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

Create a MultiIndex using two Series:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
      month  year  sale
1 1       1  2012    55
2 4       4  2014    40
3 9       7  2013    84
4 16     10  2014    31
```

pandas.DataFrame.set_value

`DataFrame.set_value`(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index.

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

Parameters

index [row label]

col [column label]

value [scalar value]

takeable [interpret the index/col as indexers, default False]

Returns

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

pandas.DataFrame.shift

`DataFrame.shift`(*periods=1*, *freq=None*, *axis=0*, *fill_value=None*)

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*.

Parameters

periods [int] Number of periods to shift. Can be positive or negative.

freq [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data.

axis [{0 or 'index', 1 or 'columns', None}, default None] Shift direction.

fill_value [object, optional] The scalar value to use for newly introduced missing values. the default depends on the dtype of *self*. For numeric data, `np.nan` is used. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

Changed in version 0.24.0.

Returns

DataFrame Copy of input object, shifted.

See also:

`Index.shift` Shift values of Index.

`DatetimeIndex.shift` Shift values of DatetimeIndex.

`PeriodIndex.shift` Shift values of PeriodIndex.

`tshift` Shift the time index, using the index's frequency if available.

Examples

```
>>> df = pd.DataFrame({'Col1': [10, 20, 15, 30, 45],
...                    'Col2': [13, 23, 18, 33, 48],
...                    'Col3': [17, 27, 22, 37, 52]})
```

```
>>> df.shift(periods=3)
   Col1  Col2  Col3
0   NaN   NaN   NaN
1   NaN   NaN   NaN
2   NaN   NaN   NaN
3  10.0  13.0  17.0
4  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis='columns')
   Col1  Col2  Col3
0   NaN  10.0  13.0
1   NaN  20.0  23.0
2   NaN  15.0  18.0
3   NaN  30.0  33.0
4   NaN  45.0  48.0
```

```
>>> df.shift(periods=3, fill_value=0)
   Col1  Col2  Col3
0     0     0     0
1     0     0     0
2     0     0     0
3    10    13    17
4    20    23    27
```

pandas.DataFrame.skew

`DataFrame.skew`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased skew over requested axis Normalized by N-1.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

skew [Series or DataFrame (if level specified)]

pandas.DataFrame.slice_shift

DataFrame.**slice_shift**(*periods=1*, *axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

pandas.DataFrame.sort_index

DataFrame.**sort_index**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)

Sort object by labels (along an axis)

Parameters

axis [index, columns to direct sorting]

level [int or level name or list of ints or list of level names] if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

Returns

sorted_obj [DataFrame]

pandas.DataFrame.sort_values

DataFrame.**sort_values**(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values along either axis

Parameters

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

Returns

sorted_obj [DataFrame]

Examples

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

pandas.DataFrame.squeeze

DataFrame.**squeeze**(*axis=None*)

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

New in version 0.20.0.

Returns

DataFrame, Series, or scalar The projection after squeezing *axis* or all the axes.

See also:

Series.iloc Integer-location based indexing for selecting scalars.

DataFrame.iloc Integer-location based indexing for selecting Series.

Series.to_frame Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes wil project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

pandas.DataFrame.stack

`DataFrame.stack`(*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

Parameters

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

Returns

DataFrame or Series Stacked dataframe or series.

See also:

DataFrame.unstack Unstack prescribed level(s) from index axis onto column axis.

DataFrame.pivot Reshape dataframe from long format to wide format.

DataFrame.pivot_table Create a spreadsheet-style pivot table as a DataFrame.

Notes

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Examples

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1        2
dog       2        4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
```

(continues on next page)

(continued from previous page)

```
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
```

(continues on next page)

(continued from previous page)

```
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

pandas.DataFrame.std

DataFrame.**std**(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [Series or DataFrame (if level specified)]

pandas.DataFrame.sub

DataFrame.**sub**(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
```

(continues on next page)

(continued from previous page)

```
>>> df_multiindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multiindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.subtract

`DataFrame.subtract`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
```

(continues on next page)

(continued from previous page)

```
B square        0.0       0.0
  pentagon      0.0       0.0
  hexagon       0.0       0.0
```

pandas.DataFrame.sum

`DataFrame.sum`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the sum of the values for the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

sum [Series or DataFrame (if level specified)]

See also:

`Series.sum` Return the sum.

`Series.min` Return the minimum.

`Series.max` Return the maximum.

`Series.idxmin` Return the index of the minimum.

`Series.idxmax` Return the index of the maximum.

`DataFrame.min` Return the sum over the requested axis.

`DataFrame.min` Return the minimum over the requested axis.

`DataFrame.max` Return the maximum over the requested axis.

`DataFrame.idxmin` Return the index of the minimum over the requested axis.

`DataFrame.idxmax` Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

Sum using level names, as well as indices.

```
>>> s.sum(level='blooded')
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

```
>>> s.sum(level=0)
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

pandas.DataFrame.swapaxes

`DataFrame.`**`swapaxes`**(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately.

Returns

y [same as input]

pandas.DataFrame.swaplevel

DataFrame.**swaplevel**(*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis.

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [same type as caller (new object)]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

pandas.DataFrame.tail

DataFrame.**tail**(*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

DataFrame.head The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

pandas.DataFrame.take

`DataFrame.take`(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)
Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [same type as caller] An array-like containing the elements taken from the object.

See also:

DataFrame.loc Select a subset of a DataFrame by labels.

DataFrame.iloc Select a subset of a DataFrame by positions.

numpy.take Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

pandas.DataFrame.to_clipboard

DataFrame.**to_clipboard**(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

See also:

DataFrame.to_csv Write a DataFrame to a comma-separated values (csv) file.

read_clipboard Read text from clipboard and pass to read_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

pandas.DataFrame.to_csv

`DataFrame.to_csv`(*path_or_buf=None*, *sep='*, *'*, *na_rep="*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression='infer'*, *quoting=None*, *quotechar='"'*, *line_terminator=None*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write object to a comma-separated values (csv) file.

Changed in version 0.24.0: The order of arguments for Series was changed.

Parameters

path_or_buf [str or file handle, default None] File path or object, if None is provided the result is returned as a string.

Changed in version 0.24.0: Was previously named "path" for Series.

sep [str, default ','] String of length 1. Field delimiter for the output file.

na_rep [str, default ''] Missing data representation.

float_format [str, default None] Format string for floating point numbers.

columns [sequence, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

Changed in version 0.24.0: Previously defaulted to False for Series.

index [bool, default True] Write row names (index).

index_label [str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode [str] Python write mode, default 'w'.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [str, default 'infer'] Compression mode among the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If 'infer' and *path_or_buf* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression).

Changed in version 0.24.0: 'infer' option added and set to default.

quoting [optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar [str, default '"'] String of length 1. Character used to quote fields.

line_terminator [string, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('n' for linux, 'rn' for Windows, i.e.).

Changed in version 0.24.0.

chunksize [int or None] Rows to write at a time.

tupleize_cols [bool, default False] Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

date_format [str, default None] Format string for datetime objects.

doublequote [bool, default True] Control quoting of *quotechar* inside a field.

escapechar [str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal [str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

Returns

None or str If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

read_csv Load a CSV file into a DataFrame.

to_excel Load an Excel file into a DataFrame.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

pandas.DataFrame.to_dense

`DataFrame.to_dense()`

Return dense representation of NDFrame (as opposed to sparse).

pandas.DataFrame.to_dict

`DataFrame.to_dict(orient='dict', into=<class 'dict'>)`

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, … , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

Returns

dict, list or collections.Mapping Return a collections.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

See also:

DataFrame.from_dict Create a DataFrame from a dictionary.

DataFrame.to_json Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

pandas.DataFrame.to_excel

`DataFrame.to_excel`(*excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None*)

Write object to an Excel sheet.

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer [str or ExcelWriter object] File path or existing ExcelWriter.

sheet_name [str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep [str, default ''] Missing data representation.

float_format [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns [sequence or list of str, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

index_label [str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow [int, default 0] Upper left cell row to dump data frame.

startcol [int, default 0] Upper left cell column to dump data frame.

engine [str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

verbose [bool, default True] Display more information in the error logs.

freeze_panes [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

New in version 0.20.0..

See also:

to_csv Write DataFrame to a comma-separated values (csv) file.

ExcelWriter Class for writing DataFrame objects into excel sheets.

read_excel Read an Excel file into a pandas DataFrame.

read_csv Read a comma-separated values (csv) file into DataFrame.

Notes

For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")  # doctest: +SKIP
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')  # doctest: +SKIP
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:  # doctest: +SKIP
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')  # doctest: +SKIP
```

pandas.DataFrame.to_feather

`DataFrame.`**`to_feather`**(*fname*)

Write out the binary feather-format for DataFrames.

New in version 0.20.0.

Parameters

fname [str] string file path

pandas.DataFrame.to_gbq

`DataFrame.`**`to_gbq`**(*destination_table, project_id=None, chunksize=None, reauth=False, if_exists='fail', auth_local_webserver=False, table_schema=None, location=None, progress_bar=True, credentials=None, verbose=None, private_key=None*)

Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

destination_table [str] Name of table to be written, in the form `dataset.tablename`.

project_id [str, optional] Google BigQuery Account project ID. Optional when available from the environment.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

location [str, optional] Location where the load job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of the target dataset.

New in version 0.5.0 of pandas-gbq.

progress_bar [bool, default True] Use the library *tqdm* to show the progress bar for the upload, chunk by chunk.

New in version 0.5.0 of pandas-gbq.

credentials [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

New in version 0.24.0.

verbose [bool, deprecated] Deprecated in pandas-gbq version 0.4.0. Use the logging module to adjust verbosity instead.

private_key [str, deprecated] Deprecated in pandas-gbq version 0.8.0. Use the `credentials` parameter and `google.oauth2.service_account.Credentials.from_service_account_info()` or `google.oauth2.service_account.Credentials.from_service_account_file()` instead.

Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

See also:

`pandas_gbq.to_gbq` This function in the pandas-gbq library.

`pandas.read_gbq` Read a DataFrame from Google BigQuery.

pandas.DataFrame.to_hdf

`DataFrame.`**`to_hdf`**(*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the *user guide*.

Parameters

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See *Query via Data Columns*. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

`DataFrame.read_hdf` Read from HDF file.

`DataFrame.to_parquet` Write a DataFrame to the binary parquet format.

`DataFrame.to_sql` Write to a sql table.

`DataFrame.to_feather` Write out feather-format for DataFrames.

`DataFrame.to_csv` Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

pandas.DataFrame.to_html

`DataFrame.to_html`(*buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, max_rows=None, max_cols=None, show_dimensions=False, decimal='.', bold_rows=True, classes=None, escape=True, notebook=False, border=None, table_id=None, render_links=False*)

Render a DataFrame as an HTML table.

Parameters

buf [StringIO-like, optional] Buffer to write to.

columns [sequence, optional, default None] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool, optional] Whether to print column labels, default True.

index [bool, optional, default True] Whether to print index (row) labels.

na_rep [str, optional, default 'NaN'] String representation of NAN to use.

formatters [list or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. The result of this function must be a unicode string.

sparsify [bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names [bool, optional, default True] Prints the names of the indexes.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows [int, optional] Maximum number of rows to display in the console.

max_cols [int, optional] Maximum number of columns to display in the console.

show_dimensions [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

bold_rows [bool, default True] Make the row labels bold in the output.

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

escape [bool, default True] Convert the characters <, >, and & to HTML-safe sequences.

notebook [{True, False}, default False] Whether the generated HTML is for IPython Notebook.

border [int] A `border=border` attribute is included in the opening *<table>* tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening *<table>* tag if specified.

New in version 0.23.0.

render_links [bool, default False] Convert URLs to HTML links.

New in version 0.24.0.

Returns

str (or unicode, depending on data and options) String representation of the dataframe.

See also:

to_string Convert DataFrame to a string.

pandas.DataFrame.to_json

`DataFrame.`**`to_json`**(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression='infer'*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index','table'}
- DataFrame

  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values','table'}

- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [bool, default True] Force encoded string to be ASCII.

date_unit [string, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [bool, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}] A string representing the compression to use in the output file, only used when the first argument is a filename. By default, the compression is inferred from the filename.

New in version 0.21.0.

Changed in version 0.24.0: 'infer' option added and set to default

index [bool, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

read_json

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

pandas.DataFrame.to_latex

`DataFrame.to_latex`(*buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multicolumn_format=None, multirow=None*)

Render an object to a LaTeX tabular environment table.

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

Parameters

buf [file descriptor or None] Buffer to write to. If None, the output is returned as a string.

columns [list of label, optional] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

na_rep [str, default 'NaN'] Missing data representation.

formatters [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [str, optional] Format string for floating point numbers.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names [bool, default True] Prints the names of the indexes.

bold_rows [bool, default False] Make the row labels bold in the output.

column_format [str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a usepackage{longtable} to your LaTeX preamble.

escape [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe. .. versionadded:: 0.18.0

multicolumn [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module. .. versionadded:: 0.20.0

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module. .. versionadded:: 0.20.0

multirow [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module. .. versionadded:: 0.20.0

Returns

str or None If buf is None, returns the resulting LateX format as a string. Otherwise returns None.

See also:

DataFrame.to_string Render a DataFrame to a console-friendly tabular output.

DataFrame.to_html Render a DataFrame as an HTML table.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_latex(index=False) # doctest: +NORMALIZE_WHITESPACE
'\\begin{tabular}{lll}\n\\toprule\n      name &    mask &    weapon
\\\\\n\\midrule\n   Raphael &     red &       sai \\\\\n Donatello &
 purple &  bo staff \\\\\n\\bottomrule\n\\end{tabular}\n'
```

pandas.DataFrame.to_msgpack

`DataFrame.`**`to_msgpack`**(*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)

Serialize object to input file path using msgpack format.

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [bool whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

pandas.DataFrame.to_numpy

`DataFrame.`**`to_numpy`**(*dtype=None*, *copy=False*)

Convert the DataFrame to a NumPy array.

New in version 0.24.0.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`

copy [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

Returns

array [numpy.ndarray]

See also:

Series.to_numpy Similar method for Series.

Examples

```
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogenous data, the lowest common type will have to be used.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
>>> df.to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df['C'] = pd.date_range('2000', periods=2)
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

pandas.DataFrame.to_panel

`DataFrame.`**`to_panel`**`()`

Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns

panel [Panel]

pandas.DataFrame.to_parquet

`DataFrame.`**`to_parquet`**`(`*fname*, *engine='auto'*, *compression='snappy'*, *index=None*, *partition_cols=None*, ***kwargs*`)`

Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See *the user guide* for more details.

Parameters

fname [str] File path or Root Directory path. Will be used as Root Directory path while writing a partitioned dataset.

Changed in version 0.24.0.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

index [bool, default None] If `True`, include the dataframe's index(es) in the file output. If `False`, they will not be written to the file. If `None`, the behavior depends on the chosen engine.

New in version 0.24.0.

partition_cols [list, optional, default None] Column names by which to partition the dataset Columns are partitioned in the order they are given

New in version 0.24.0.

****kwargs** Additional arguments passed to the parquet library. See *pandas io* for more details.

See also:

read_parquet Read a parquet file.

DataFrame.to_csv Write a csv file.

DataFrame.to_sql Write to a sql table.

DataFrame.to_hdf Write to hdf.

Notes

This function requires either the fastparquet or pyarrow library.

Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip',
...               compression='gzip')  # doctest: +SKIP
>>> pd.read_parquet('df.parquet.gzip')  # doctest: +SKIP
   col1  col2
0     1     3
1     2     4
```

pandas.DataFrame.to_period

DataFrame.**to_period**(*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

freq [string, default]

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

Returns

ts [TimeSeries with PeriodIndex]

pandas.DataFrame.to_pickle

DataFrame.**to_pickle**(*path*, *compression='infer'*, *protocol=4*)

Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [**?**] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

See also:

read_pickle Load pickled pandas object (or any object) from file.

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_sql Write DataFrame to a SQL database.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

pandas.DataFrame.to_records

DataFrame.**to_records**(*index=True*, *convert_datetime64=None*, *column_dtypes=None*, *index_dtypes=None*)

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

Parameters

index [bool, default True] Include index in resulting record array, stored in 'index' field or using the index label, if set.

convert_datetime64 [bool, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

column_dtypes [str, type, dict, default None] New in version 0.24.0.

If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.

index_dtypes [str, type, dict, default None] New in version 0.24.0.

If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.

This mapping is applied only if *index=True*.

Returns

numpy.recarray NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

See also:

DataFrame.from_records Convert structured or record ndarray to DataFrame.

numpy.recarray An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

If the DataFrame index has no label then the recarray field name is set to 'index'. If the index has a label then this is used as the field name:

```
>>> df.index = df.index.rename("I")
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

Data types can be specified for the columns:

```
>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])
```

As well as for the index:

```
>>> df.to_records(index_dtypes="<S2")
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])
```

```
>>> index_dtypes = "<S{}".format(df.index.str.len().max())
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
```

pandas.DataFrame.to_sparse

DataFrame.**to_sparse**(*fill_value=None, kind='block'*)

Convert to SparseDataFrame.

Implement the sparse version of the DataFrame meaning that any data matching a specific value it's omitted in the representation. The sparse DataFrame allows for a more efficient storage.

Parameters

fill_value [float, default None] The specific value that should be omitted in the representation.

kind [{'block', 'integer'}, default 'block'] The kind of the SparseIndex tracking where data is not equal to the fill value:

- 'block' tracks only the locations and sizes of blocks of data.
- 'integer' keeps an array with all the locations of the data.

In most cases 'block' is recommended, since it's more memory efficient.

Returns

SparseDataFrame The sparse representation of the DataFrame.

See also:

DataFrame.to_dense Converts the DataFrame back to the its dense form.

Examples

```
>>> df = pd.DataFrame([(np.nan, np.nan),
...                    (1., np.nan),
...                    (np.nan, 1.)])
>>> df
     0    1
0  NaN  NaN
1  1.0  NaN
2  NaN  1.0
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

```
>>> sdf = df.to_sparse()
>>> sdf
     0    1
0  NaN  NaN
1  1.0  NaN
2  NaN  1.0
>>> type(sdf)
<class 'pandas.core.sparse.frame.SparseDataFrame'>
```

pandas.DataFrame.to_stata

DataFrame.**to_stata**(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export DataFrame object to Stata dta format.

Writes the DataFrame to a Stata dataset file. "dta" files contain a Stata dataset.

Parameters

fname [str, buffer or path object] String, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() function. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str, optional] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}, default 114] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version

114 limits string variables to 244 characters or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

Raises

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

See also:

read_stata Import Stata data files.

io.stata.StataWriter Low-level writer for Stata data files.

io.stata.StataWriter117 Low-level writer for version 117 files.

Examples

```
>>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]})
>>> df.to_stata('animals.dta')  # doctest: +SKIP
```

pandas.DataFrame.to_string

DataFrame.**to_string**(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *decimal='.'*, *line_width=None*)

Render a DataFrame to a console-friendly tabular output.

Parameters

buf [StringIO-like, optional] Buffer to write to.

columns [sequence, optional, default None] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, optional, default True] Whether to print index (row) labels.

na_rep [str, optional, default 'NaN'] String representation of NAN to use.

formatters [list or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. The result of this function must be a unicode string.

sparsify [bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names [bool, optional, default True] Prints the names of the indexes.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows [int, optional] Maximum number of rows to display in the console.

max_cols [int, optional] Maximum number of columns to display in the console.

show_dimensions [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

decimal [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

line_width [int, optional] Width to wrap a line in characters.

Returns

str (or unicode, depending on data and options) String representation of the dataframe.

See also:

`to_html` Convert DataFrame to HTML.

Examples

```
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0     1     4
1     2     5
2     3     6
```

pandas.DataFrame.to_timestamp

DataFrame.**to_timestamp**(*freq=None*, *how='start'*, *axis=0*, *copy=True*)

Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

Returns

df [DataFrame with DatetimeIndex]

pandas.DataFrame.to_xarray

DataFrame.**to_xarray**()

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',  389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion',   'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                    'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]}).set_index(['date',
...                                                    'animal'])
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

pandas.DataFrame.transform

DataFrame.**transform**(*func*, *axis=0*, **args*, ***kwargs*)

Call `func` on self producing a DataFrame with transformed values and that has the same axis length as self.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.exp. 'sqrt']`
- dict of axis labels -> functions, function names or list of such.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame A DataFrame that must have the same length as self.

Raises

ValueError [If the returned DataFrame has a different length than self.]

See also:

DataFrame.agg Only perform aggregating type operations.

DataFrame.apply Invoke function on a DataFrame.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

pandas.DataFrame.transpose

`DataFrame.transpose(*args, **kwargs)`

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property `T` is an accessor to the method `transpose()`.

Parameters

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

DataFrame The transposed DataFrame.

See also:

numpy.transpose Permute the dimensions of a given array.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

pandas.DataFrame.truediv

`DataFrame.truediv`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

pandas.DataFrame.truncate

`DataFrame.truncate`(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

Returns

type of caller The truncated Series or DataFrame.

See also:

DataFrame.loc Select a subset of a DataFrame by label.

DataFrame.iloc Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
```

(continues on next page)

(continued from previous page)

```
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

pandas.DataFrame.tshift

`DataFrame.tshift`(*periods=1*, *freq=None*, *axis=0*)

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

pandas.DataFrame.tz_convert

`DataFrame.tz_convert(tz, axis=0, level=None, copy=True)`

Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

pandas.DataFrame.tz_localize

`DataFrame.tz_localize(tz, axis=0, level=None, copy=True, ambiguous='raise', nonexistent='raise')`

Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent [str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid valuse are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

Series or DataFrame Same type as the input.

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series([1],
... index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7), index=pd.DatetimeIndex([
... '2018-10-28 01:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 03:00:00',
... '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3), index=pd.DatetimeIndex([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
```

(continues on next page)

(continued from previous page)

```
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.Series(range(2), index=pd.DatetimeIndex([... '2015-03-29 02:30:00', ... '2015-03-29 03:30:00'])) >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 2015-03-29 03:00:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 2015-03-29 01:59:59.999999999+01:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 2015-03-29 03:30:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64

pandas.DataFrame.unstack

DataFrame.**unstack**(*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex).

The level involved will automatically get sorted.

Parameters

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

Returns

unstacked [DataFrame or Series]

See also:

DataFrame.pivot Pivot a table based on column values.

DataFrame.stack Pivot a level of the column labels (inverse operation from *unstack*).

Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a    b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

pandas.DataFrame.update

`DataFrame.update`(*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *errors='ignore'*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

Parameters

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> bool 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

errors [{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

Changed in version 0.24.0: Changed from *raise_conflict=False|True* to *errors='ignore'|'raise'*.

Returns

None [method directly changes calling object]

Raises

ValueError

- When *errors='raise'* and there's overlapping non-NA data.
- When *errors* is not either *'ignore'* or *'raise'*

NotImplementedError

- If *join != 'left'*

See also:

dict.update Similar method for dictionaries.

DataFrame.merge For column(s)-on-columns(s) operations.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

pandas.DataFrame.var

`DataFrame.var`(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{index (0), columns (1)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [Series or DataFrame (if level specified)]

pandas.DataFrame.where

`DataFrame.where`(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Replace values where the condition is False.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

DataFrame.mask() Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `where` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2     2
3     3
4     4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.DataFrame.xs

`DataFrame.xs`(*key*, *axis=0*, *level=None*, *drop_level=True*)

Return cross-section from the Series/DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

Parameters

key [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to retrieve cross-section on.

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [bool, default True] If False, returns object with same levels as self.

Returns

Series or DataFrame Cross-section from the original Series or DataFrame corresponding to the selected index levels.

See also:

DataFrame.loc Access a group of rows and columns by label(s) or a boolean array.

`DataFrame.iloc` Purely integer-location based indexing for selection by position.

Notes

xs can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see *MultiIndex Slicers*.

Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
                           num_legs  num_wings
class  animal  locomotion
mammal cat     walks              4          0
       dog     walks              4          0
       bat     flies              2          2
bird   penguin walks              2          2
```

Get values at specified index

```
>>> df.xs('mammal')
                   num_legs  num_wings
animal locomotion
cat    walks              4          0
dog    walks              4          0
bat    flies              2          2
```

Get values at several indexes

```
>>> df.xs(('mammal', 'dog'))
            num_legs  num_wings
locomotion
walks              4          0
```

Get values at specified index and level

```
>>> df.xs('cat', level=1)
                   num_legs  num_wings
class  locomotion
mammal walks              4          0
```

Get values at several indexes and levels

```
>>> df.xs(('bird', 'walks'),
...       level=[0, 'locomotion'])
         num_legs  num_wings
animal
penguin         2          2
```

Get values at specified column and axis

```
>>> df.xs('num_wings', axis=1)
class   animal   locomotion
mammal  cat      walks         0
        dog      walks         0
        bat      flies         2
bird    penguin  walks         2
Name: num_wings, dtype: int64
```

6.4.2 Attributes and underlying data

Axes

DataFrame.index	The index (row labels) of the DataFrame.
DataFrame.columns	The column labels of the DataFrame.

DataFrame.dtypes	Return the dtypes in the DataFrame.
DataFrame.ftypes	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
DataFrame.get_dtype_counts()	Return counts of unique dtypes in this object.
DataFrame.get_ftype_counts()	(DEPRECATED) Return counts of unique ftypes in this object.
DataFrame.select_dtypes([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
DataFrame.values	Return a Numpy representation of the DataFrame.
DataFrame.get_values()	Return an ndarray after converting sparse values to dense.
DataFrame.axes	Return a list representing the axes of the DataFrame.
DataFrame.ndim	Return an int representing the number of axes / array dimensions.
DataFrame.size	Return an int representing the number of elements in this object.
DataFrame.shape	Return a tuple representing the dimensionality of the DataFrame.
DataFrame.memory_usage([index, deep])	Return the memory usage of each column in bytes.
DataFrame.empty	Indicator whether DataFrame is empty.
DataFrame.is_copy	Return the copy.

6.4.3 Conversion

DataFrame.astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
DataFrame.convert_objects([convert_dates, . . .])	(DEPRECATED) Attempt to infer better dtype for object columns.
DataFrame.infer_objects()	Attempt to infer better dtypes for object columns.
DataFrame.copy([deep])	Make a copy of this object's indices and data.
DataFrame.isna()	Detect missing values.
DataFrame.notna()	Detect existing (non-missing) values.

Continued on next page

Table 60 – continued from previous page

DataFrame.bool()	Return the bool of a single element PandasObject.

6.4.4 Indexing, iteration

DataFrame.head([n])	Return the first *n* rows.
DataFrame.at	Access a single value for a row/column label pair.
DataFrame.iat	Access a single value for a row/column pair by integer position.
DataFrame.loc	Access a group of rows and columns by label(s) or a boolean array.
DataFrame.iloc	Purely integer-location based indexing for selection by position.
DataFrame.insert(loc, column, value[, …])	Insert column into DataFrame at specified location.
DataFrame.__iter__()	Iterate over infor axis
DataFrame.items()	Iterator over (column name, Series) pairs.
DataFrame.keys()	Get the 'info axis' (see Indexing for more)
DataFrame.iteritems()	Iterator over (column name, Series) pairs.
DataFrame.iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
DataFrame.itertuples([index, name])	Iterate over DataFrame rows as namedtuples.
DataFrame.lookup(row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
DataFrame.pop(item)	Return item and drop from frame.
DataFrame.tail([n])	Return the last *n* rows.
DataFrame.xs(key[, axis, level, drop_level])	Return cross-section from the Series/DataFrame.
DataFrame.get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
DataFrame.isin(values)	Whether each element in the DataFrame is contained in values.
DataFrame.where(cond[, other, inplace, …])	Replace values where the condition is False.
DataFrame.mask(cond[, other, inplace, axis, …])	Replace values where the condition is True.
DataFrame.query(expr[, inplace])	Query the columns of a DataFrame with a boolean expression.

pandas.DataFrame.__iter__

DataFrame.**__iter__**()

Iterate over infor axis

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the *indexing documentation*.

6.4.5 Binary operator functions

DataFrame.add(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *add*).
DataFrame.sub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
DataFrame.mul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
DataFrame.div(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).

Continued on next page

Table 62 – continued from previous page

`DataFrame.truediv`(other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
`DataFrame.floordiv`(other[, axis, level, ...])	Integer division of dataframe and other, element-wise (binary operator *floordiv*).
`DataFrame.mod`(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *mod*).
`DataFrame.pow`(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *pow*).
`DataFrame.dot`(other)	Compute the matrix mutiplication between the DataFrame and other.
`DataFrame.radd`(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *radd*).
`DataFrame.rsub`(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *rsub*).
`DataFrame.rmul`(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *rmul*).
`DataFrame.rdiv`(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
`DataFrame.rtruediv`(other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
`DataFrame.rfloordiv`(other[, axis, level, ...])	Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
`DataFrame.rmod`(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *rmod*).
`DataFrame.rpow`(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *rpow*).
`DataFrame.lt`(other[, axis, level])	Less than of dataframe and other, element-wise (binary operator *lt*).
`DataFrame.gt`(other[, axis, level])	Greater than of dataframe and other, element-wise (binary operator *gt*).
`DataFrame.le`(other[, axis, level])	Less than or equal to of dataframe and other, element-wise (binary operator *le*).
`DataFrame.ge`(other[, axis, level])	Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).
`DataFrame.ne`(other[, axis, level])	Not equal to of dataframe and other, element-wise (binary operator *ne*).
`DataFrame.eq`(other[, axis, level])	Equal to of dataframe and other, element-wise (binary operator *eq*).
`DataFrame.combine`(other, func[, fill_value, ...])	Perform column-wise combine with another DataFrame based on a passed function.
`DataFrame.combine_first`(other)	Update null elements with value in the same location in *other*.

6.4.6 Function application, GroupBy & Window

`DataFrame.apply`(func[, axis, broadcast, ...])	Apply a function along an axis of the DataFrame.
`DataFrame.applymap`(func)	Apply a function to a Dataframe elementwise.
`DataFrame.pipe`(func, *args, **kwargs)	Apply func(self, *args, **kwargs).
`DataFrame.agg`(func[, axis])	Aggregate using one or more operations over the specified axis.

Continued on next page

Table 63 – continued from previous page

DataFrame.aggregate(func[, axis])	Aggregate using one or more operations over the specified axis.
DataFrame.transform(func[, axis])	Call `func` on self producing a DataFrame with transformed values and that has the same axis length as self.
DataFrame.groupby([by, axis, level, . . .])	Group DataFrame or Series using a mapper or by a Series of columns.
DataFrame.rolling(window[, min_periods, . . .])	Provides rolling window calculations.
DataFrame.expanding([min_periods, center, axis])	Provides expanding transformations.
DataFrame.ewm([com, span, halflife, alpha, . . .])	Provides exponential weighted functions.

6.4.7 Computations / Descriptive Stats

DataFrame.abs()	Return a Series/DataFrame with absolute numeric value of each element.
DataFrame.all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
DataFrame.any([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
DataFrame.clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
DataFrame.clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.
DataFrame.clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.
DataFrame.compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis.
DataFrame.corr([method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values.
DataFrame.corrwith(other[, axis, drop, method])	Compute pairwise correlation between rows or columns of DataFrame with rows or columns of Series or DataFrame.
DataFrame.count([axis, level, numeric_only])	Count non-NA cells for each column or row.
DataFrame.cov([min_periods])	Compute pairwise covariance of columns, excluding NA/null values.
DataFrame.cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
DataFrame.cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
DataFrame.cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
DataFrame.cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
DataFrame.describe([percentiles, include, . . .])	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
DataFrame.diff([periods, axis])	First discrete difference of element.
DataFrame.eval(expr[, inplace])	Evaluate a string describing operations on DataFrame columns.
DataFrame.kurt([axis, skipna, level, . . .])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).

Continued on next page

Table 64 – continued from previous page

DataFrame.kurtosis([axis, skipna, level, ...])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
DataFrame.mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
DataFrame.max([axis, skipna, level, ...])	Return the maximum of the values for the requested axis.
DataFrame.mean([axis, skipna, level, ...])	Return the mean of the values for the requested axis.
DataFrame.median([axis, skipna, level, ...])	Return the median of the values for the requested axis.
DataFrame.min([axis, skipna, level, ...])	Return the minimum of the values for the requested axis.
DataFrame.mode([axis, numeric_only, dropna])	Get the mode(s) of each element along the selected axis.
DataFrame.pct_change([periods, fill_method, ...])	Percentage change between the current and a prior element.
DataFrame.prod([axis, skipna, level, ...])	Return the product of the values for the requested axis.
DataFrame.product([axis, skipna, level, ...])	Return the product of the values for the requested axis.
DataFrame.quantile([q, axis, numeric_only, ...])	Return values at the given quantile over requested axis.
DataFrame.rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
DataFrame.round([decimals])	Round a DataFrame to a variable number of decimal places.
DataFrame.sem([axis, skipna, level, ddof, ...])	Return unbiased standard error of the mean over requested axis.
DataFrame.skew([axis, skipna, level, ...])	Return unbiased skew over requested axis Normalized by N-1.
DataFrame.sum([axis, skipna, level, ...])	Return the sum of the values for the requested axis.
DataFrame.std([axis, skipna, level, ddof, ...])	Return sample standard deviation over requested axis.
DataFrame.var([axis, skipna, level, ddof, ...])	Return unbiased variance over requested axis.
DataFrame.nunique([axis, dropna])	Count distinct observations over requested axis.

6.4.8 Reindexing / Selection / Label manipulation

DataFrame.add_prefix(prefix)	Prefix labels with string *prefix*.
DataFrame.add_suffix(suffix)	Suffix labels with string *suffix*.
DataFrame.align(other[, join, axis, level, ...])	Align two objects on their axes with the specified join method for each axis Index.
DataFrame.at_time(time[, asof, axis])	Select values at particular time of day (e.g.
DataFrame.between_time(start_time, end_time)	Select values between particular times of the day (e.g., 9:00-9:30 AM).
DataFrame.drop([labels, axis, index, ...])	Drop specified labels from rows or columns.
DataFrame.drop_duplicates([subset, keep, ...])	Return DataFrame with duplicate rows removed, optionally only considering certain columns.
DataFrame.duplicated([subset, keep])	Return boolean Series denoting duplicate rows, optionally only considering certain columns.
DataFrame.equals(other)	Test whether two objects contain the same elements.
DataFrame.filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
DataFrame.first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
DataFrame.head([n])	Return the first *n* rows.

Continued on next page

Table 65 – continued from previous page

DataFrame.idxmax([axis, skipna])	Return index of first occurrence of maximum over requested axis.
DataFrame.idxmin([axis, skipna])	Return index of first occurrence of minimum over requested axis.
DataFrame.last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
DataFrame.reindex([labels, index, columns, …])	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
DataFrame.reindex_axis(labels[, axis, …])	(DEPRECATED) Conform input object to new index.
DataFrame.reindex_like(other[, method, …])	Return an object with matching indices as other object.
DataFrame.rename([mapper, index, columns, …])	Alter axes labels.
DataFrame.rename_axis([mapper, index, …])	Set the name of the axis for the index or columns.
DataFrame.reset_index([level, drop, …])	Reset the index, or a level of it.
DataFrame.sample([n, frac, replace, …])	Return a random sample of items from an axis of object.
DataFrame.select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
DataFrame.set_axis(labels[, axis, inplace])	Assign desired index to given axis.
DataFrame.set_index(keys[, drop, append, …])	Set the DataFrame index using existing columns.
DataFrame.tail([n])	Return the last *n* rows.
DataFrame.take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
DataFrame.truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.

6.4.9 Missing data handling

DataFrame.dropna([axis, how, thresh, …])	Remove missing values.
DataFrame.fillna([value, method, axis, …])	Fill NA/NaN values using the specified method.
DataFrame.replace([to_replace, value, …])	Replace values given in *to_replace* with *value*.
DataFrame.interpolate([method, axis, limit, …])	Interpolate values according to different methods.

6.4.10 Reshaping, sorting, transposing

DataFrame.droplevel(level[, axis])	Return DataFrame with requested index / column level(s) removed.
DataFrame.pivot([index, columns, values])	Return reshaped DataFrame organized by given index / column values.
DataFrame.pivot_table([values, index, …])	Create a spreadsheet-style pivot table as a DataFrame.
DataFrame.reorder_levels(order[, axis])	Rearrange index levels using input order.
DataFrame.sort_values(by[, axis, ascending, …])	Sort by the values along either axis
DataFrame.sort_index([axis, level, …])	Sort object by labels (along an axis)
DataFrame.nlargest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in descending order.
DataFrame.nsmallest(n, columns[, keep])	Return the first *n* rows ordered by *columns* in ascending order.
DataFrame.swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis.

Continued on next page

Table 67 – continued from previous page

DataFrame.stack([level, dropna])	Stack the prescribed level(s) from columns to index.
DataFrame.unstack([level, fill_value])	Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
DataFrame.swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
DataFrame.melt([id_vars, value_vars, ...])	Unpivots a DataFrame from wide format to long format, optionally leaving identifier variables set.
DataFrame.squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
DataFrame.to_panel()	(DEPRECATED) Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
DataFrame.to_xarray()	Return an xarray object from the pandas object.
DataFrame.T	Transpose index and columns.
DataFrame.transpose(*args, **kwargs)	Transpose index and columns.

6.4.11 Combining / joining / merging

DataFrame.append(other[, ignore_index, ...])	Append rows of *other* to the end of caller, returning a new object.
DataFrame.assign(**kwargs)	Assign new columns to a DataFrame.
DataFrame.join(other[, on, how, lsuffix, ...])	Join columns of another DataFrame.
DataFrame.merge(right[, how, on, left_on, ...])	Merge DataFrame or named Series objects with a database-style join.
DataFrame.update(other[, join, overwrite, ...])	Modify in place using non-NA values from another DataFrame.

6.4.12 Time series-related

DataFrame.asfreq(freq[, method, how, ...])	Convert TimeSeries to specified frequency.
DataFrame.asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
DataFrame.shift([periods, freq, axis, ...])	Shift index by desired number of periods with an optional time *freq*.
DataFrame.slice_shift([periods, axis])	Equivalent to *shift* without copying data.
DataFrame.tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
DataFrame.first_valid_index()	Return index for first non-NA/null value.
DataFrame.last_valid_index()	Return index for last non-NA/null value.
DataFrame.resample(rule[, how, axis, ...])	Resample time-series data.
DataFrame.to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).
DataFrame.to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period.
DataFrame.tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
DataFrame.tz_localize(tz[, axis, level, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.

6.4.13 Plotting

`DataFrame.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `DataFrame.plot.<kind>`.

DataFrame.plot([x, y, kind, ax, ...])	DataFrame plotting accessor and method

DataFrame.plot.area([x, y])	Draw a stacked area plot.
DataFrame.plot.bar([x, y])	Vertical bar plot.
DataFrame.plot.barh([x, y])	Make a horizontal bar plot.
DataFrame.plot.box([by])	Make a box plot of the DataFrame columns.
DataFrame.plot.density([bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
DataFrame.plot.hexbin(x, y[, C, ...])	Generate a hexagonal binning plot.
DataFrame.plot.hist([by, bins])	Draw one histogram of the DataFrame's columns.
DataFrame.plot.kde([bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
DataFrame.plot.line([x, y])	Plot DataFrame columns as lines.
DataFrame.plot.pie([y])	Generate a pie plot.
DataFrame.plot.scatter(x, y[, s, c])	Create a scatter plot with varying marker point size and color.

pandas.DataFrame.plot.area

`DataFrame.plot.`**`area`**(*x=None*, *y=None*, ***kwds*)

Draw a stacked area plot.

An area plot displays quantitative data visually. This function wraps the matplotlib area function.

Parameters

x [label or position, optional] Coordinates for the X axis. By default uses the index.

y [label or position, optional] Column to plot. By default uses all columns.

stacked [bool, default True] Area plots are stacked by default. Set to False to create a unstacked plot.

****kwds** [optional] Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

Returns

matplotlib.axes.Axes or numpy.ndarray Area plot, or array of area plots if subplots is True

See also:

DataFrame.plot Make plots of DataFrame using matplotlib / pylab.

Examples

Draw an area plot based on basic business metrics:

```
>>> df = pd.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                        freq='M'))
>>> ax = df.plot.area()
```

Area plots are stacked by default. To produce an unstacked plot, pass `stacked=False`:

```
>>> ax = df.plot.area(stacked=False)
```

Draw an area plot for a single column:

```
>>> ax = df.plot.area(y='sales')
```

Draw with a different *x*:

```
>>> df = pd.DataFrame({
...     'sales': [3, 2, 3],
...     'visits': [20, 42, 28],
...     'day': [1, 2, 3],
... })
>>> ax = df.plot.area(x='day')
```

pandas.DataFrame.plot.bar

`DataFrame.plot.bar(x=None, y=None, **kwds)`

Vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

Parameters

x [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.

y [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

****kwds** Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.axes.Axes or np.ndarray of them] An ndarray is returned with one `matplotlib.axes.Axes` per column when `subplots=True`.

See also:

pandas.DataFrame.plot.barh Horizontal bar plot.

pandas.DataFrame.plot Make plots of a DataFrame.

matplotlib.pyplot.bar Make a bar plot with matplotlib.

Examples

Basic plot.

```
>>> df = pd.DataFrame({'lab':['A', 'B', 'C'], 'val':[10, 30, 20]})
>>> ax = df.plot.bar(x='lab', y='val', rot=0)
```

Plot a whole dataframe to a bar plot. Each column is assigned a distinct color, and each row is nested in a group along the horizontal axis.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.bar(rot=0)
```

Instead of nesting, the figure can be split by column with `subplots=True`. In this case, a `numpy.ndarray` of `matplotlib.axes.Axes` are returned.

```
>>> axes = df.plot.bar(rot=0, subplots=True)
>>> axes[1].legend(loc=2)  # doctest: +SKIP
```

Plot a single column.

```
>>> ax = df.plot.bar(y='speed', rot=0)
```

Plot only selected categories for the DataFrame.

```
>>> ax = df.plot.bar(x='lifespan', rot=0)
```

pandas.DataFrame.plot.barh

`DataFrame.plot.`**`barh`**(*x=None*, *y=None*, ***kwds*)

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

Parameters

x [label or position, default DataFrame.index] Column to be used for categories.

y [label or position, default All numeric columns in dataframe] Columns to be plotted from the DataFrame.

****kwds** Keyword arguments to pass on to *`pandas.DataFrame.plot()`*.

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them.]

See also:

`pandas.DataFrame.plot.bar` Vertical bar plot.

`pandas.DataFrame.plot` Make plots of DataFrame using matplotlib.

matplotlib.axes.Axes.bar Plot a vertical bar plot using matplotlib.

Examples

Basic example

```
>>> df = pd.DataFrame({'lab':['A', 'B', 'C'], 'val':[10, 30, 20]})
>>> ax = df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh()
```

Plot a column of the DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(y='speed')
```

Plot DataFrame versus the desired column

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(x='lifespan')
```

pandas.DataFrame.plot.box

`DataFrame.plot.`**`box`**(*by=None*, ***kwds*)

Make a box plot of the DataFrame columns.

A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to 1.5*IQR (IQR = Q3 - Q1) from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

A consideration when using this chart is that the box and the whiskers can overlap, which is very common when plotting small sets of data.

Parameters

by [string or sequence] Column in the DataFrame to group by.

****kwds** [optional] Additional keywords are documented in *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

pandas.DataFrame.boxplot Another method to draw a box plot.

pandas.Series.plot.box Draw a box plot from a Series object.

matplotlib.pyplot.boxplot Draw a box plot in matplotlib.

Examples

Draw a box plot from a DataFrame with four columns of randomly generated data.

```
>>> data = np.random.randn(25, 4)
>>> df = pd.DataFrame(data, columns=list('ABCD'))
>>> ax = df.plot.box()
```

pandas.DataFrame.plot.density

DataFrame.plot.**density**(*bw_method=None*, *ind=None*, ***kwds*)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

Parameters

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See scipy.stats.gaussian_kde for more information.

ind [NumPy array or integer, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

****kwds** [optional] Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

Series.plot.kde Generate a KDE plot for a Series.

Examples

Given several Series of points randomly sampled from unknown distributions, estimate their PDFs using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> ax = df.plot.kde(bw_method=0.3)
```

```
>>> ax = df.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

pandas.DataFrame.plot.hexbin

`DataFrame.plot.`**`hexbin`**(*x*, *y*, *C=None*, *reduce_C_function=None*, *gridsize=None*, ***kwds*)

Generate a hexagonal binning plot.

Generate a hexagonal binning plot of *x* versus *y*. If *C* is *None* (the default), this is a histogram of the number of occurrences of the observations at `(x[i], y[i])`.

If *C* is specified, specifies values at given coordinates `(x[i], y[i])`. These values are accumulated for each hexagonal bin and then reduced according to *reduce_C_function*, having as default the NumPy's mean function (`numpy.mean()`). (If *C* is specified, it must also be a 1-D sequence of the same length as *x* and *y*, or a column label.)

Parameters

x [int or str] The column label or position for x points.

y [int or str] The column label or position for y points.

C [int or str, optional] The column label or position for the value of *(x, y)* point.

reduce_C_function [callable, default *np.mean*] Function of one argument that reduces all the values in a bin to a single number (e.g. *np.mean*, *np.max*, *np.sum*, *np.std*).

gridsize [int or tuple of (int, int), default 100] The number of hexagons in the x-direction. The corresponding number of hexagons in the y-direction is chosen in a way that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the x-direction and the y-direction.

****kwds** Additional keyword arguments are documented in *`pandas.DataFrame.plot()`*.

Returns

matplotlib.AxesSubplot The matplotlib `Axes` on which the hexbin is plotted.

See also:

DataFrame.plot Make plots of a DataFrame.

matplotlib.pyplot.hexbin Hexagonal binning plot using matplotlib, the matplotlib function that is used under the hood.

Examples

The following examples are generated with random data from a normal distribution.

```
>>> n = 10000
>>> df = pd.DataFrame({'x': np.random.randn(n),
...                    'y': np.random.randn(n)})
>>> ax = df.plot.hexbin(x='x', y='y', gridsize=20)
```

The next example uses *C* and *np.sum* as *reduce_C_function*. Note that *'observations'* values ranges from 1 to 5 but the result plot shows values up to more than 25. This is because of the *reduce_C_function*.

```
>>> n = 500
>>> df = pd.DataFrame({
...     'coord_x': np.random.uniform(-3, 3, size=n),
...     'coord_y': np.random.uniform(30, 50, size=n),
...     'observations': np.random.randint(1,5, size=n)
...     })
>>> ax = df.plot.hexbin(x='coord_x',
...                     y='coord_y',
...                     C='observations',
...                     reduce_C_function=np.sum,
...                     gridsize=10,
...                     cmap="viridis")
```

pandas.DataFrame.plot.hist

`DataFrame.plot.`**`hist`**(*by=None*, *bins=10*, ***kwds*)

Draw one histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`. This is useful when the DataFrame's Series are in a similar scale.

Parameters

by [str or sequence, optional] Column in the DataFrame to group by.

bins [int, default 10] Number of histogram bins to be used.

****kwds** Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.AxesSubplot histogram.]

See also:

DataFrame.hist Draw histograms per DataFrame's Series.

Series.hist Draw a histogram with Series' data.

Examples

When we draw a dice 6000 times, we expect to get each value around 1000 times. But when we draw two dices and sum the result, the distribution is going to be quite different. A histogram illustrates those distributions.

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns = ['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> ax = df.plot.hist(bins=12, alpha=0.5)
```

pandas.DataFrame.plot.kde

`DataFrame.plot.kde`(*bw_method=None*, *ind=None*, ***kwds*)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, kernel density estimation (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

Parameters

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

ind [NumPy array or integer, optional] Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

****kwds** [optional] Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

scipy.stats.gaussian_kde Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

Series.plot.kde Generate a KDE plot for a Series.

Examples

Given several Series of points randomly sampled from unknown distributions, estimate their PDFs using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> ax = df.plot.kde(bw_method=0.3)
```

```
>>> ax = df.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

pandas.DataFrame.plot.line

`DataFrame.plot.line`(*x=None*, *y=None*, ***kwds*)

Plot DataFrame columns as lines.

This function is useful to plot lines using DataFrame's values as coordinates.

Parameters

x [int or str, optional] Columns to use for the horizontal axis. Either the location or the label of the columns to be used. By default, it will use the DataFrame indices.

y [int, str, or list of them, optional] The values to be plotted. Either the location or the label of the columns to be used. By default, it will use the remaining DataFrame numeric columns.

****kwds** Keyword arguments to pass on to `pandas.DataFrame.plot()`.

Returns

axes [`matplotlib.axes.Axes` or `numpy.ndarray`] Returns an ndarray when `subplots=True`.

See also:

`matplotlib.pyplot.plot` Plot y versus x as lines and/or markers.

Examples

The following example shows the populations for some animals over the years.

```
>>> df = pd.DataFrame({
...    'pig': [20, 18, 489, 675, 1776],
...    'horse': [4, 25, 281, 600, 1900]
...    }, index=[1990, 1997, 2003, 2009, 2014])
>>> lines = df.plot.line()
```

An example with subplots, so an array of axes is returned.

```
>>> axes = df.plot.line(subplots=True)
>>> type(axes)
<class 'numpy.ndarray'>
```

The following example shows the relationship between both populations.

```
>>> lines = df.plot.line(x='pig', y='horse')
```

pandas.DataFrame.plot.pie

`DataFrame.plot.pie(y=None, **kwds)`

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps `matplotlib.pyplot.pie()` for the specified column. If no column reference is passed and `subplots=True` a pie plot is drawn for each numerical column independently.

Parameters

y [int or label, optional] Label or position of the column to plot. If not provided, `subplots=True` argument must be passed.

****kwds** Keyword arguments to pass on to *pandas.DataFrame.plot()*.

Returns

axes [matplotlib.axes.Axes or np.ndarray of them.] A NumPy array is returned when *subplots* is True.

See also:

Series.plot.pie Generate a pie plot for a Series.

DataFrame.plot Make plots of a DataFrame.

Examples

In the example below we have a DataFrame with the information about planet's mass and radius. We pass the the 'mass' column to the pie function to get a pie plot.

```
>>> df = pd.DataFrame({'mass': [0.330, 4.87 , 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                   index=['Mercury', 'Venus', 'Earth'])
>>> plot = df.plot.pie(y='mass', figsize=(5, 5))
```

```
>>> plot = df.plot.pie(subplots=True, figsize=(6, 3))
```

pandas.DataFrame.plot.scatter

`DataFrame.plot.scatter(x, y, s=None, c=None, **kwds)`

Create a scatter plot with varying marker point size and color.

The coordinates of each point are defined by two dataframe columns and filled circles are used to represent each point. This kind of plot is useful to see complex correlations between two variables. Points could be for instance natural 2D coordinates like longitude and latitude in a map or, in general, any pair of metrics that can be plotted against each other.

Parameters

x [int or str] The column name or column position to be used as horizontal coordinates for each point.

y [int or str] The column name or column position to be used as vertical coordinates for each point.

s [scalar or array_like, optional] The size of each point. Possible values are:

- A single scalar so all points have the same size.
- A sequence of scalars, which will be used for each point's size recursively. For instance, when passing [2,14] all points size will be either 2 or 14, alternatively.

c [str, int or array_like, optional] The color of each point. Possible values are:

- A single color string referred to by name, RGB or RGBA code, for instance 'red' or '#a98d19'.
- A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each point's color recursively. For instance ['green','yellow'] all points will be filled in green or yellow, alternatively.
- A column name or position whose values will be used to color the marker points according to a colormap.

****kwds** Keyword arguments to pass on to `pandas.DataFrame.plot()`.

Returns

axes [`matplotlib.axes.Axes` or numpy.ndarray of them]

See also:

`matplotlib.pyplot.scatter` Scatter plot using multiple input data formats.

Examples

Let's see how to draw a scatter plot using coordinates from the values in a DataFrame's columns.

```
>>> df = pd.DataFrame([[5.1, 3.5, 0], [4.9, 3.0, 0], [7.0, 3.2, 1],
...                    [6.4, 3.2, 1], [5.9, 3.0, 2]],
...                   columns=['length', 'width', 'species'])
>>> ax1 = df.plot.scatter(x='length',
...                       y='width',
...                       c='DarkBlue')
```

And now with the color determined by a column as well.

```
>>> ax2 = df.plot.scatter(x='length',
...                       y='width',
...                       c='species',
...                       colormap='viridis')
```

`DataFrame.boxplot([column, by, ax, ...])`	Make a box plot from DataFrame columns.
`DataFrame.hist([column, by, grid, ...])`	Make a histogram of the DataFrame's.

6.4.14 Serialization / IO / Conversion

`DataFrame.from_csv(path[, header, sep, ...])`	(DEPRECATED) Read CSV file.
`DataFrame.from_dict(data[, orient, dtype, ...])`	Construct DataFrame from dict of array-like or dicts.
`DataFrame.from_items(items[, columns, orient])`	(DEPRECATED) Construct a DataFrame from a list of tuples.
`DataFrame.from_records(data[, index, ...])`	Convert structured or record ndarray to DataFrame.

Continued on next page

Table 73 – continued from previous page

DataFrame.info([verbose, buf, max_cols, ...])	Print a concise summary of a DataFrame.
DataFrame.to_parquet(fname[, engine, ...])	Write a DataFrame to the binary parquet format.
DataFrame.to_pickle(path[, compression, ...])	Pickle (serialize) object to file.
DataFrame.to_csv([path_or_buf, sep, na_rep, ...])	Write object to a comma-separated values (csv) file.
DataFrame.to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
DataFrame.to_sql(name, con[, schema, ...])	Write records stored in a DataFrame to a SQL database.
DataFrame.to_dict([orient, into])	Convert the DataFrame to a dictionary.
DataFrame.to_excel(excel_writer[, ...])	Write object to an Excel sheet.
DataFrame.to_json([path_or_buf, orient, ...])	Convert the object to a JSON string.
DataFrame.to_html([buf, columns, col_space, ...])	Render a DataFrame as an HTML table.
DataFrame.to_feather(fname)	Write out the binary feather-format for DataFrames.
DataFrame.to_latex([buf, columns, ...])	Render an object to a LaTeX tabular environment table.
DataFrame.to_stata(fname[, convert_dates, ...])	Export DataFrame object to Stata dta format.
DataFrame.to_msgpack([path_or_buf, encoding])	Serialize object to input file path using msgpack format.
DataFrame.to_gbq(destination_table[, ...])	Write a DataFrame to a Google BigQuery table.
DataFrame.to_records([index, ...])	Convert DataFrame to a NumPy record array.
DataFrame.to_sparse([fill_value, kind])	Convert to SparseDataFrame.
DataFrame.to_dense()	Return dense representation of NDFrame (as opposed to sparse).
DataFrame.to_string([buf, columns, ...])	Render a DataFrame to a console-friendly tabular output.
DataFrame.to_clipboard([excel, sep])	Copy object to the system clipboard.
DataFrame.style	Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

6.4.15 Sparse

SparseDataFrame.to_coo()	Return the contents of the frame as a sparse SciPy COO matrix.

pandas.SparseDataFrame.to_coo

`SparseDataFrame.to_coo()`

Return the contents of the frame as a sparse SciPy COO matrix.

New in version 0.20.0.

Returns

coo_matrix [scipy.sparse.spmatrix] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Notes

The dtype will be the lowest-common-denominator type (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. By numpy.find_common_type convention, mixing int64 and and uint64 will result in a float64 dtype.

6.5 Pandas Arrays

For most data types, pandas uses NumPy arrays as the concrete objects contained with a `Index`, `Series`, or `DataFrame`.

For some data types, pandas extends NumPy's type system.

Kind of Data	Pandas Data Type	Scalar	Array
TZ-aware datetime	`DatetimeTZDtype`	`Timestamp`	*Datetime Data*
Timedeltas	(none)	`Timedelta`	*Timedelta Data*
Period (time spans)	`PeriodDtype`	`Period`	*Timespan Data*
Intervals	`IntervalDtype`	`Interval`	*Interval Data*
Nullable Integer	`Int64Dtype`, ...	(none)	*Nullable Integer*
Categorical	`CategoricalDtype`	(none)	*Categorical Data*
Sparse	`SparseDtype`	(none)	*Sparse Data*

Pandas and third-party libraries can extend NumPy's type system (see *Extension Types*). The top-level `array()` method can be used to create a new array, which may be stored in a `Series`, `Index`, or as a column in a `DataFrame`.

`array`(data[, dtype, copy])	Create an array.

6.5.1 pandas.array

`pandas.`**`array`**(*data*, *dtype=None*, *copy=True*)

Create an array.

New in version 0.24.0.

Parameters

data [Sequence of objects] The scalars inside *data* should be instances of the scalar type for *dtype*. It's expected that *data* represents a 1-dimensional array of data.

When *data* is an Index or Series, the underlying array will be extracted from *data*.

dtype [str, np.dtype, or ExtensionDtype, optional] The dtype to use for the array. This may be a NumPy dtype or an extension type registered with pandas using `pandas.api.extensions.register_extension_dtype()`.

If not specified, there are two possibilities:

1. When *data* is a `Series`, `Index`, or `ExtensionArray`, the *dtype* will be taken from the data.
2. Otherwise, pandas will attempt to infer the *dtype* from the data.

Note that when *data* is a NumPy array, `data.dtype` is *not* used for inferring the array type. This is because NumPy cannot represent all the types of data that can be held in extension arrays.

Currently, pandas will infer an extension dtype for sequences of

Scalar Type	Array Type
pandas.Interval	*pandas.arrays.IntervalArray*
pandas.Period	pandas.arrays.PeriodArray
datetime.datetime	*pandas.arrays.DatetimeArray*
datetime.timedelta	*pandas.arrays.TimedeltaArray*

For all other cases, NumPy's usual inference rules will be used.

copy [bool, default True] Whether to copy the data, even if not necessary. Depending on the type of *data*, creating the new array may require copying data, even if `copy=False`.

Returns

ExtensionArray The newly created array.

Raises

ValueError When *data* is not 1-dimensional.

See also:

numpy.array Construct a NumPy array.

Series Construct a pandas Series.

Index Construct a pandas Index.

arrays.PandasArray ExtensionArray wrapping a NumPy array.

Series.array Extract the array stored within a Series.

Notes

Omitting the *dtype* argument means pandas will attempt to infer the best array type from the values in the data. As new array types are added by pandas and 3rd party libraries, the "best" array type may change. We recommend specifying *dtype* to ensure that

1. the correct array type for the data is returned
2. the returned array type doesn't change as new extension types are added by pandas and third-party libraries

Additionally, if the underlying memory representation of the returned array matters, we recommend specifying the *dtype* as a concrete object rather than a string alias or allowing it to be inferred. For example, a future version of pandas or a 3rd-party library may include a dedicated ExtensionArray for string data. In this event, the following would no longer return a *arrays.PandasArray* backed by a NumPy array.

```
>>> pd.array(['a', 'b'], dtype=str)
<PandasArray>
['a', 'b']
Length: 2, dtype: str32
```

This would instead return the new ExtensionArray dedicated for string data. If you really need the new array to be backed by a NumPy array, specify that in the dtype.

```
>>> pd.array(['a', 'b'], dtype=np.dtype("<U1"))
<PandasArray>
['a', 'b']
Length: 2, dtype: str32
```

Or use the dedicated constructor for the array you're expecting, and wrap that in a PandasArray

```
>>> pd.array(np.array(['a', 'b'], dtype='<U1'))
<PandasArray>
['a', 'b']
Length: 2, dtype: str32
```

Finally, Pandas has arrays that mostly overlap with NumPy

- *arrays.DatetimeArray*
- *arrays.TimedeltaArray*

When data with a `datetime64[ns]` or `timedelta64[ns]` dtype is passed, pandas will always return a `DatetimeArray` or `TimedeltaArray` rather than a `PandasArray`. This is for symmetry with the case of timezone-aware data, which NumPy does not natively support.

```
>>> pd.array(['2015', '2016'], dtype='datetime64[ns]')
<DatetimeArray>
['2015-01-01 00:00:00', '2016-01-01 00:00:00']
Length: 2, dtype: datetime64[ns]
```

```
>>> pd.array(["1H", "2H"], dtype='timedelta64[ns]')
<TimedeltaArray>
['01:00:00', '02:00:00']
Length: 2, dtype: timedelta64[ns]
```

Examples

If a dtype is not specified, *data* is passed through to `numpy.array()`, and a *arrays.PandasArray* is returned.

```
>>> pd.array([1, 2])
<PandasArray>
[1, 2]
Length: 2, dtype: int64
```

Or the NumPy dtype can be specified

```
>>> pd.array([1, 2], dtype=np.dtype("int32"))
<PandasArray>
[1, 2]
Length: 2, dtype: int32
```

You can use the string alias for *dtype*

```
>>> pd.array(['a', 'b', 'a'], dtype='category')
[a, b, a]
Categories (2, object): [a, b]
```

Or specify the actual dtype

```
>>> pd.array(['a', 'b', 'a'],
...          dtype=pd.CategoricalDtype(['a', 'b', 'c'], ordered=True))
[a, b, a]
Categories (3, object): [a < b < c]
```

Because omitting the *dtype* passes the data through to NumPy, a mixture of valid integers and NA will return a floating-point NumPy array.

```
>>> pd.array([1, 2, np.nan])
<PandasArray>
[1.0,  2.0, nan]
Length: 3, dtype: float64
```

To use pandas' nullable `pandas.arrays.IntegerArray`, specify the dtype:

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

Pandas will infer an ExtensionArray for some types of data:

```
>>> pd.array([pd.Period('2000', freq="D"), pd.Period("2000", freq="D")])
<PeriodArray>
['2000-01-01', '2000-01-01']
Length: 2, dtype: period[D]
```

data must be 1-dimensional. A ValueError is raised when the input has the wrong dimensionality.

```
>>> pd.array(1)
Traceback (most recent call last):
  ...
ValueError: Cannot pass scalar '1' to 'pandas.array'.
```

6.5.2 Datetime Data

NumPy cannot natively represent timezone-aware datetimes. Pandas supports this with the `arrays.DatetimeArray` extension array, which can hold timezone-naive or timezone-aware values.

`Timestamp`, a subclass of `datetime.datetime`, is pandas' scalar type for timezone-naive or timezone-aware datetime data.

`Timestamp`	Pandas replacement for datetime.datetime

pandas.Timestamp

class pandas.**Timestamp**

Pandas replacement for datetime.datetime

Timestamp is the pandas equivalent of python's Datetime and is interchangeable with it in most cases. It's the type used for the entries that make up a DatetimeIndex, and other timeseries oriented data structures in pandas.

Parameters

ts_input [datetime-like, str, int, float] Value to be converted to Timestamp

freq [str, DateOffset] Offset which Timestamp will have

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will have.

unit [str] Unit used for conversion if ts_input is of type int or float. The valid values are 'D', 'h', 'm', 's', 'ms', 'us', and 'ns'. For example, 's' means seconds and 'ms' means milliseconds.

year, month, day [int] New in version 0.19.0.

hour, minute, second, microsecond [int, optional, default 0] New in version 0.19.0.

nanosecond [int, optional, default 0] New in version 0.23.0.

tzinfo [datetime.tzinfo, optional, default None] New in version 0.19.0.

Notes

There are essentially three calling conventions for the constructor. The primary form accepts four parameters. They can be passed by position or keyword.

The other two forms mimic the parameters from `datetime.datetime`. They can be passed by either position or keyword, but not both mixed together.

Examples

Using the primary calling convention:

This converts a datetime-like string >>> pd.Timestamp('2017-01-01T12') Timestamp('2017-01-01 12:00:00')

This converts a float representing a Unix epoch in units of seconds >>> pd.Timestamp(1513393355.5, unit='s') Timestamp('2017-12-16 03:02:35.500000')

This converts an int representing a Unix-epoch in units of seconds and for a particular timezone >>> pd.Timestamp(1513393355, unit='s', tz='US/Pacific') Timestamp('2017-12-15 19:02:35-0800', tz='US/Pacific')

Using the other two forms that mimic the API for `datetime.datetime`:

```
>>> pd.Timestamp(2017, 1, 1, 12)
Timestamp('2017-01-01 12:00:00')
```

```
>>> pd.Timestamp(year=2017, month=1, day=1, hour=12)
Timestamp('2017-01-01 12:00:00')
```

Attributes

resolution	Return resolution describing the smallest difference between two times that can be represented by Timestamp object_state
tz	Alias for tzinfo
weekday_name	(DEPRECATED) .. deprecated:: 0.23.0

pandas.Timestamp.resolution

`Timestamp.`**`resolution`**

Return resolution describing the smallest difference between two times that can be represented by Timestamp object_state

pandas.Timestamp.tz

Timestamp.**tz**
Alias for tzinfo

pandas.Timestamp.weekday_name

Timestamp.**weekday_name**
Deprecated since version 0.23.0: Use `Timestamp.day_name()` instead

asm8	
day	
dayofweek	
dayofyear	
days_in_month	
daysinmonth	
fold	
freq	
freqstr	
hour	
is_leap_year	
is_month_end	
is_month_start	
is_quarter_end	
is_quarter_start	
is_year_end	
is_year_start	
microsecond	
minute	
month	
nanosecond	
quarter	
second	
tzinfo	
value	
week	
weekofyear	
year	

Methods

astimezone	Convert tz-aware Timestamp to another time zone.
ceil	return a new Timestamp ceiled to this resolution
combine(date, time)	date, time -> datetime with same date and time fields
ctime	Return ctime() style string.
date	Return date object with same year, month and day.
day_name	Return the day name of the Timestamp with specified locale.

Continued on next page

Table 78 – continued from previous page

dst	Return self.tzinfo.dst(self).
floor	return a new Timestamp floored to this resolution
fromordinal(ordinal[, freq, tz])	passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself
fromtimestamp(ts)	timestamp[, tz] -> tz's local time from POSIX timestamp.
isocalendar	Return a 3-tuple containing ISO year, week number, and weekday.
isoweekday	Return the day of the week represented by the date.
month_name	Return the month name of the Timestamp with specified locale.
normalize	Normalize Timestamp to midnight, preserving tz information.
now([tz])	Returns new Timestamp object representing current time local to tz.
replace	implements datetime.replace, handles nanoseconds
round	Round the Timestamp to the specified resolution
strftime	format -> strftime() style string.
strptime	string, format -> new datetime parsed from a string (like time.strptime()).
time	Return time object with same time but with tzinfo=None.
timestamp	Return POSIX timestamp as float.
timetuple	Return time tuple, compatible with time.localtime().
timetz	Return time object with same time and tzinfo.
to_datetime64	Returns a numpy.datetime64 object with 'ns' precision
to_julian_date	Convert TimeStamp to a Julian Date.
to_period	Return an period of which this timestamp is an observation.
to_pydatetime	Convert a Timestamp object to a native Python datetime object.
today(cls[, tz])	Return the current time in the local timezone.
toordinal	Return proleptic Gregorian ordinal.
tz_convert	Convert tz-aware Timestamp to another time zone.
tz_localize	Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.
tzname	Return self.tzinfo.tzname(self).
utcfromtimestamp(ts)	Construct a naive UTC datetime from a POSIX timestamp.
utcnow()	Return a new Timestamp representing UTC day and time.
utcoffset	Return self.tzinfo.utcoffset(self).
utctimetuple	Return UTC time tuple, compatible with time.localtime().
weekday	Return the day of the week represented by the date.

pandas.Timestamp.astimezone

`Timestamp.astimezone`
Convert tz-aware Timestamp to another time zone.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

Returns

converted [Timestamp]

Raises

TypeError If Timestamp is tz-naive.

pandas.Timestamp.ceil

`Timestamp.ceil`
return a new Timestamp ceiled to this resolution

Parameters

freq [a freq string indicating the ceiling resolution]

ambiguous [bool, 'NaT', default 'raise']

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates)
- 'NaT' will return NaT for an ambiguous time
- 'raise' will raise an AmbiguousTimeError for an ambiguous time

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Raises

ValueError if the freq cannot be converted

pandas.Timestamp.combine

classmethod Timestamp.**combine**(*date*, *time*)
date, time -> datetime with same date and time fields

pandas.Timestamp.ctime

Timestamp.**ctime**()
Return ctime() style string.

pandas.Timestamp.date

Timestamp.**date**()
Return date object with same year, month and day.

pandas.Timestamp.day_name

Timestamp.**day_name**
Return the day name of the Timestamp with specified locale.

Parameters

locale [string, default None (English locale)] locale determining the language in which to return the day name

Returns

day_name [string]

.. versionadded:: 0.23.0

pandas.Timestamp.dst

Timestamp.**dst**()
Return self.tzinfo.dst(self).

pandas.Timestamp.floor

Timestamp.**floor**
return a new Timestamp floored to this resolution

Parameters

freq [a freq string indicating the flooring resolution]

ambiguous [bool, 'NaT', default 'raise']

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates)
- 'NaT' will return NaT for an ambiguous time
- 'raise' will raise an AmbiguousTimeError for an ambiguous time

New in version 0.24.0.

nonexistent [‘shift_forward’, ‘shift_backward, ‘NaT’, timedelta,]

default ‘raise’

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- ‘shift_forward’ will shift the nonexistent time forward to the closest existing time
- ‘shift_backward’ will shift the nonexistent time backward to the closest existing time
- ‘NaT’ will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- ‘raise’ will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Raises

ValueError if the freq cannot be converted

pandas.Timestamp.fromordinal

classmethod `Timestamp.`**`fromordinal`**(*ordinal*, *freq=None*, *tz=None*)

passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself

Parameters

ordinal [int] date corresponding to a proleptic Gregorian ordinal

freq [str, DateOffset] Offset which Timestamp will have

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will have.

pandas.Timestamp.fromtimestamp

classmethod `Timestamp.`**`fromtimestamp`**(*ts*)

timestamp[, tz] -> tz’s local time from POSIX timestamp.

pandas.Timestamp.isocalendar

`Timestamp.`**`isocalendar`**()

Return a 3-tuple containing ISO year, week number, and weekday.

pandas.Timestamp.isoweekday

`Timestamp.`**`isoweekday`**()

Return the day of the week represented by the date. Monday == 1 … Sunday == 7

pandas.Timestamp.month_name

`Timestamp.month_name`
Return the month name of the Timestamp with specified locale.

Parameters

locale [string, default None (English locale)] locale determining the language in which to return the month name

Returns

month_name [string]

.. versionadded:: 0.23.0

pandas.Timestamp.normalize

`Timestamp.normalize`
Normalize Timestamp to midnight, preserving tz information.

pandas.Timestamp.now

`classmethod Timestamp.now(tz=None)`
Returns new Timestamp object representing current time local to tz.

Parameters

tz [str or timezone object, default None] Timezone to localize to

pandas.Timestamp.replace

`Timestamp.replace`
implements datetime.replace, handles nanoseconds

Parameters

year [int, optional]

month [int, optional]

day [int, optional]

hour [int, optional]

minute [int, optional]

second [int, optional]

microsecond [int, optional]

nanosecond [int, optional]

tzinfo [tz-convertible, optional]

fold [int, optional, default is 0] added in 3.6, NotImplemented

Returns

Timestamp with fields replaced

pandas.Timestamp.round

`Timestamp.round`

Round the Timestamp to the specified resolution

Parameters

freq [a freq string indicating the rounding resolution]

ambiguous [bool, 'NaT', default 'raise']

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates)
- 'NaT' will return NaT for an ambiguous time
- 'raise' will raise an AmbiguousTimeError for an ambiguous time

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

a new Timestamp rounded to the given resolution of 'freq'

Raises

ValueError if the freq cannot be converted

pandas.Timestamp.strftime

`Timestamp.strftime()`

format -> strftime() style string.

pandas.Timestamp.strptime

`Timestamp.strptime()`

string, format -> new datetime parsed from a string (like time.strptime()).

pandas.Timestamp.time

`Timestamp.time()`
Return time object with same time but with tzinfo=None.

pandas.Timestamp.timestamp

`Timestamp.timestamp()`
Return POSIX timestamp as float.

pandas.Timestamp.timetuple

`Timestamp.timetuple()`
Return time tuple, compatible with time.localtime().

pandas.Timestamp.timetz

`Timestamp.timetz()`
Return time object with same time and tzinfo.

pandas.Timestamp.to_datetime64

`Timestamp.to_datetime64()`
Returns a numpy.datetime64 object with 'ns' precision

pandas.Timestamp.to_julian_date

`Timestamp.to_julian_date`
Convert TimeStamp to a Julian Date. 0 Julian date is noon January 1, 4713 BC.

pandas.Timestamp.to_period

`Timestamp.to_period`
Return an period of which this timestamp is an observation.

pandas.Timestamp.to_pydatetime

`Timestamp.to_pydatetime()`
Convert a Timestamp object to a native Python datetime object.

If warn=True, issue a warning if nanoseconds is nonzero.

pandas.Timestamp.today

classmethod Timestamp.**today**(*cls*, *tz=None*)

Return the current time in the local timezone. This differs from datetime.today() in that it can be localized to a passed timezone.

Parameters

tz [str or timezone object, default None] Timezone to localize to

pandas.Timestamp.toordinal

Timestamp.**toordinal**()

Return proleptic Gregorian ordinal. January 1 of year 1 is day 1.

pandas.Timestamp.tz_convert

Timestamp.**tz_convert**

Convert tz-aware Timestamp to another time zone.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

Returns

converted [Timestamp]

Raises

TypeError If Timestamp is tz-naive.

pandas.Timestamp.tz_localize

Timestamp.**tz_localize**

Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.

Parameters

tz [str, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time which Timestamp will be converted to. None will remove timezone holding local time.

ambiguous [bool, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates)
- 'NaT' will return NaT for an ambiguous time
- 'raise' will raise an AmbiguousTimeError for an ambiguous time

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

errors ['raise', 'coerce', default None]

- **'raise' will raise a NonExistentTimeError if a timestamp is not** valid in the specified timezone (e.g. due to a transition from or to DST time). Use `nonexistent='raise'` instead.
- 'coerce' will return NaT if the timestamp can not be converted into the specified timezone. Use `nonexistent='NaT'` instead.

  Deprecated since version 0.24.0.

Returns

localized [Timestamp]

Raises

TypeError If the Timestamp is tz-aware and tz is not None.

pandas.Timestamp.tzname

`Timestamp.`**`tzname`**`()`

Return self.tzinfo.tzname(self).

pandas.Timestamp.utcfromtimestamp

`classmethod` `Timestamp.`**`utcfromtimestamp`**`(`*ts*`)`

Construct a naive UTC datetime from a POSIX timestamp.

pandas.Timestamp.utcnow

`classmethod` `Timestamp.`**`utcnow`**`()`

Return a new Timestamp representing UTC day and time.

pandas.Timestamp.utcoffset

`Timestamp.`**`utcoffset`**`()`

Return self.tzinfo.utcoffset(self).

pandas.Timestamp.utctimetuple

`Timestamp.utctimetuple()`
Return UTC time tuple, compatible with time.localtime().

pandas.Timestamp.weekday

`Timestamp.weekday()`
Return the day of the week represented by the date. Monday == 0 … Sunday == 6

isoformat	

Properties

Timestamp.asm8	
Timestamp.day	
Timestamp.dayofweek	
Timestamp.dayofyear	
Timestamp.days_in_month	
Timestamp.daysinmonth	
Timestamp.fold	
Timestamp.hour	
Timestamp.is_leap_year	
Timestamp.is_month_end	
Timestamp.is_month_start	
Timestamp.is_quarter_end	
Timestamp.is_quarter_start	
Timestamp.is_year_end	
Timestamp.is_year_start	
Timestamp.max	
Timestamp.microsecond	
Timestamp.min	
Timestamp.minute	
Timestamp.month	
Timestamp.nanosecond	
Timestamp.quarter	
Timestamp.resolution	Return resolution describing the smallest difference between two times that can be represented by Timestamp object_state
Timestamp.second	
Timestamp.tz	Alias for tzinfo
Timestamp.tzinfo	
Timestamp.value	
Timestamp.week	
Timestamp.weekofyear	
Timestamp.year	

pandas.Timestamp.asm8

Timestamp.**asm8**

pandas.Timestamp.day

Timestamp.**day**

pandas.Timestamp.dayofweek

Timestamp.**dayofweek**

pandas.Timestamp.dayofyear

Timestamp.**dayofyear**

pandas.Timestamp.days_in_month

Timestamp.**days_in_month**

pandas.Timestamp.daysinmonth

Timestamp.**daysinmonth**

pandas.Timestamp.fold

Timestamp.**fold**

pandas.Timestamp.hour

Timestamp.**hour**

pandas.Timestamp.is_leap_year

Timestamp.**is_leap_year**

pandas.Timestamp.is_month_end

Timestamp.**is_month_end**

pandas.Timestamp.is_month_start

Timestamp.**is_month_start**

pandas.Timestamp.is_quarter_end

`Timestamp.is_quarter_end`

pandas.Timestamp.is_quarter_start

`Timestamp.is_quarter_start`

pandas.Timestamp.is_year_end

`Timestamp.is_year_end`

pandas.Timestamp.is_year_start

`Timestamp.is_year_start`

pandas.Timestamp.max

`Timestamp.max = Timestamp('2262-04-11 23:47:16.854775807')`

pandas.Timestamp.microsecond

`Timestamp.microsecond`

pandas.Timestamp.min

`Timestamp.min = Timestamp('1677-09-21 00:12:43.145225')`

pandas.Timestamp.minute

`Timestamp.minute`

pandas.Timestamp.month

`Timestamp.month`

pandas.Timestamp.nanosecond

`Timestamp.nanosecond`

pandas.Timestamp.quarter

`Timestamp.quarter`

pandas.Timestamp.second

Timestamp.**second**

pandas.Timestamp.tzinfo

Timestamp.**tzinfo**

pandas.Timestamp.value

Timestamp.**value**

pandas.Timestamp.week

Timestamp.**week**

pandas.Timestamp.weekofyear

Timestamp.**weekofyear**

pandas.Timestamp.year

Timestamp.**year**

Methods

Timestamp.astimezone	Convert tz-aware Timestamp to another time zone.
Timestamp.ceil	return a new Timestamp ceiled to this resolution
Timestamp.combine(date, time)	date, time -> datetime with same date and time fields
Timestamp.ctime	Return ctime() style string.
Timestamp.date	Return date object with same year, month and day.
Timestamp.day_name	Return the day name of the Timestamp with specified locale.
Timestamp.dst	Return self.tzinfo.dst(self).
Timestamp.floor	return a new Timestamp floored to this resolution
Timestamp.freq	
Timestamp.freqstr	
Timestamp.fromordinal(ordinal[, freq, tz])	passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself
Timestamp.fromtimestamp(ts)	timestamp[, tz] -> tz's local time from POSIX timestamp.
Timestamp.isocalendar	Return a 3-tuple containing ISO year, week number, and weekday.
Timestamp.isoformat	
Timestamp.isoweekday	Return the day of the week represented by the date.

Continued on next page

Table 80 – continued from previous page

Timestamp.month_name	Return the month name of the Timestamp with specified locale.
Timestamp.normalize	Normalize Timestamp to midnight, preserving tz information.
Timestamp.now([tz])	Returns new Timestamp object representing current time local to tz.
Timestamp.replace	implements datetime.replace, handles nanoseconds
Timestamp.round	Round the Timestamp to the specified resolution
Timestamp.strftime	format -> strftime() style string.
Timestamp.strptime	string, format -> new datetime parsed from a string (like time.strptime()).
Timestamp.time	Return time object with same time but with tzinfo=None.
Timestamp.timestamp	Return POSIX timestamp as float.
Timestamp.timetuple	Return time tuple, compatible with time.localtime().
Timestamp.timetz	Return time object with same time and tzinfo.
Timestamp.to_datetime64	Returns a numpy.datetime64 object with 'ns' precision
Timestamp.to_julian_date	Convert TimeStamp to a Julian Date.
Timestamp.to_period	Return an period of which this timestamp is an observation.
Timestamp.to_pydatetime	Convert a Timestamp object to a native Python datetime object.
Timestamp.today(cls[, tz])	Return the current time in the local timezone.
Timestamp.toordinal	Return proleptic Gregorian ordinal.
Timestamp.tz_convert	Convert tz-aware Timestamp to another time zone.
Timestamp.tz_localize	Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.
Timestamp.tzname	Return self.tzinfo.tzname(self).
Timestamp.utcfromtimestamp(ts)	Construct a naive UTC datetime from a POSIX timestamp.
Timestamp.utcnow()	Return a new Timestamp representing UTC day and time.
Timestamp.utcoffset	Return self.tzinfo.utcoffset(self).
Timestamp.utctimetuple	Return UTC time tuple, compatible with time.localtime().
Timestamp.weekday	Return the day of the week represented by the date.

pandas.Timestamp.freq

`Timestamp.`**`freq`**

pandas.Timestamp.freqstr

`Timestamp.`**`freqstr`**

pandas.Timestamp.isoformat

`Timestamp.`**`isoformat`**

A collection of timestamps may be stored in a *arrays.DatetimeArray*. For timezone-aware data, the `.dtype` of a `DatetimeArray` is a *DatetimeTZDtype*. For timezone-naive data, `np.dtype("datetime64[ns]")` is used.

If the data are tz-aware, then every value in the array must have the same timezone.

arrays.DatetimeArray(values[, dtype, freq, copy])	Pandas ExtensionArray for tz-naive or tz-aware datetime data.
DatetimeTZDtype([unit, tz])	A np.dtype duck-typed class, suitable for holding a custom datetime with tz dtype.

pandas.arrays.DatetimeArray

class pandas.arrays.**DatetimeArray**(*values*, *dtype=dtype('<M8[ns]')*, *freq=None*, *copy=False*)

Pandas ExtensionArray for tz-naive or tz-aware datetime data.

New in version 0.24.0.

> **Warning:** DatetimeArray is currently experimental, and its API may change without warning. In particular, *DatetimeArray.dtype* is expected to change to always be an instance of an `ExtensionDtype` subclass.

Parameters

values [Series, Index, DatetimeArray, ndarray] The datetime data.

For DatetimeArray *values* (or a Series or Index boxing one), *dtype* and *freq* will be extracted from *values*, with precedence given to

dtype [numpy.dtype or DatetimeTZDtype] Note that the only NumPy dtype allowed is 'datetime64[ns]'.

freq [str or Offset, optional]

copy [bool, default False] Whether to copy the underlying array of values.

Attributes

asi8	Integer representation of the values.
date	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
day	The days of the datetime.
dayofweek	The day of the week with Monday=0, Sunday=6.
dayofyear	The ordinal day of the year.
days_in_month	The number of days in the month.
daysinmonth	The number of days in the month.
dtype	The dtype for the DatetimeArray.
freq	Return the frequency object if it is set, otherwise None.
freqstr	Return the frequency object as a string if its set, otherwise None

Continued on next page

Table 82 – continued from previous page

hour	The hours of the datetime.
inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.
is_leap_year	Boolean indicator if the date belongs to a leap year.
is_month_end	Indicates whether the date is the last day of the month.
is_month_start	Indicates whether the date is the first day of the month.
is_normalized	Returns True if all of the dates are at midnight ("no time")
is_quarter_end	Indicator for whether the date is the last day of a quarter.
is_quarter_start	Indicator for whether the date is the first day of a quarter.
is_year_end	Indicate whether the date is the last day of the year.
is_year_start	Indicate whether the date is the first day of a year.
microsecond	The microseconds of the datetime.
minute	The minutes of the datetime.
month	The month as January=1, December=12.
nanosecond	The nanoseconds of the datetime.
nbytes	The number of bytes needed to store this object in memory.
quarter	The quarter of the date.
resolution	Returns day, hour, minute, second, millisecond or microsecond
second	The seconds of the datetime.
shape	Return a tuple of the array dimensions.
size	The number of elements in this array.
time	Returns numpy array of datetime.time.
timetz	Returns numpy array of datetime.time also containing timezone information.
tz	Return timezone, if any.
tzinfo	Alias for tz attribute
week	The week ordinal of the year.
weekday	The day of the week with Monday=0, Sunday=6.
weekday_name	(DEPRECATED) The name of day in a week (ex: Friday)
weekofyear	The week ordinal of the year.
year	The year of the datetime.

pandas.arrays.DatetimeArray.asi8

DatetimeArray.**asi8**

Integer representation of the values.

Returns

ndarray An ndarray with int64 dtype.

pandas.arrays.DatetimeArray.date

`DatetimeArray.date`
Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).

pandas.arrays.DatetimeArray.day

`DatetimeArray.day`
The days of the datetime.

pandas.arrays.DatetimeArray.dayofweek

`DatetimeArray.dayofweek`
The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

`Series.dt.dayofweek` Alias.

`Series.dt.weekday` Alias.

`Series.dt.day_name` Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.arrays.DatetimeArray.dayofyear

`DatetimeArray.dayofyear`
The ordinal day of the year.

pandas.arrays.DatetimeArray.days_in_month

`DatetimeArray.days_in_month`
The number of days in the month.

pandas.arrays.DatetimeArray.daysinmonth

`DatetimeArray.daysinmonth`
The number of days in the month.

pandas.arrays.DatetimeArray.dtype

`DatetimeArray.dtype`
The dtype for the DatetimeArray.

> **Warning:** A future version of pandas will change dtype to never be a `numpy.dtype`. Instead, *`DatetimeArray.dtype`* will always be an instance of an `ExtensionDtype` subclass.

Returns

numpy.dtype or DatetimeTZDtype If the values are tz-naive, then `np.dtype('datetime64[ns]')` is returned.

If the values are tz-aware, then the `DatetimeTZDtype` is returned.

pandas.arrays.DatetimeArray.freq

`DatetimeArray.freq`
Return the frequency object if it is set, otherwise None.

pandas.arrays.DatetimeArray.freqstr

`DatetimeArray.freqstr`
Return the frequency object as a string if its set, otherwise None

pandas.arrays.DatetimeArray.hour

`DatetimeArray.hour`
The hours of the datetime.

pandas.arrays.DatetimeArray.inferred_freq

`DatetimeArray.inferred_freq`
Tryies to return a string representing a frequency guess, generated by infer_freq. Returns None if it can't autodetect the frequency.

pandas.arrays.DatetimeArray.is_leap_year

`DatetimeArray.`**`is_leap_year`**

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

Returns

Series or ndarray Booleans indicating if dates belong to a leap year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```
>>> dates = pd.Series(idx)
>>> dates_series
0   2012-12-31
1   2013-12-31
2   2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool
```

pandas.arrays.DatetimeArray.is_month_end

`DatetimeArray.`**`is_month_end`**

Indicates whether the date is the last day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

`is_month_start` Return a boolean indicating whether the date is the first day of the month.

`is_month_end` Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
dtype: bool
>>> s.dt.is_month_end
0    False
1    True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.arrays.DatetimeArray.is_month_start

`DatetimeArray.`**`is_month_start`**

Indicates whether the date is the first day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

`is_month_start` Return a boolean indicating whether the date is the first day of the month.

`is_month_end` Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
```

(continues on next page)

(continued from previous page)

```
>>> s.dt.is_month_start
0    False
1    False
2     True
dtype: bool
>>> s.dt.is_month_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.arrays.DatetimeArray.is_normalized

`DatetimeArray.`**`is_normalized`**

Returns True if all of the dates are at midnight ("no time")

pandas.arrays.DatetimeArray.is_quarter_end

`DatetimeArray.`**`is_quarter_end`**

Indicator for whether the date is the last day of a quarter.

Returns

is_quarter_end [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

`quarter` Return the quarter of the date.

`is_quarter_start` Similar property indicating the quarter start.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                    periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
       dates  quarter    is_quarter_end
0 2017-03-30        1             False
1 2017-03-31        1              True
2 2017-04-01        2             False
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

pandas.arrays.DatetimeArray.is_quarter_start

DatetimeArray.**is_quarter_start**
Indicator for whether the date is the first day of a quarter.

Returns

is_quarter_start [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.

is_quarter_end Similar property for indicating the quarter start.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                   periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
       dates  quarter  is_quarter_start
0 2017-03-30        1             False
1 2017-03-31        1             False
2 2017-04-01        2              True
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

pandas.arrays.DatetimeArray.is_year_end

DatetimeArray.**is_year_end**
Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

pandas.arrays.DatetimeArray.is_year_start

DatetimeArray.**is_year_start**

Indicate whether the date is the first day of a year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_end Similar property indicating the last day of the year.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2    True
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

pandas.arrays.DatetimeArray.microsecond

`DatetimeArray.`**`microsecond`**

The microseconds of the datetime.

pandas.arrays.DatetimeArray.minute

`DatetimeArray.`**`minute`**

The minutes of the datetime.

pandas.arrays.DatetimeArray.month

`DatetimeArray.`**`month`**

The month as January=1, December=12.

pandas.arrays.DatetimeArray.nanosecond

`DatetimeArray.`**`nanosecond`**

The nanoseconds of the datetime.

pandas.arrays.DatetimeArray.nbytes

`DatetimeArray.`**`nbytes`**

The number of bytes needed to store this object in memory.

pandas.arrays.DatetimeArray.quarter

`DatetimeArray.quarter`
The quarter of the date.

pandas.arrays.DatetimeArray.resolution

`DatetimeArray.resolution`
Returns day, hour, minute, second, millisecond or microsecond

pandas.arrays.DatetimeArray.second

`DatetimeArray.second`
The seconds of the datetime.

pandas.arrays.DatetimeArray.shape

`DatetimeArray.shape`
Return a tuple of the array dimensions.

pandas.arrays.DatetimeArray.size

`DatetimeArray.size`
The number of elements in this array.

pandas.arrays.DatetimeArray.time

`DatetimeArray.time`
Returns numpy array of datetime.time. The time part of the Timestamps.

pandas.arrays.DatetimeArray.timetz

`DatetimeArray.timetz`
Returns numpy array of datetime.time also containing timezone information. The time part of the Timestamps.

pandas.arrays.DatetimeArray.tz

`DatetimeArray.tz`
Return timezone, if any.

Returns

datetime.tzinfo, pytz.tzinfo.BaseTZInfo, dateutil.tz.tz.tzfile, or None Returns None when the array is tz-naive.

pandas.arrays.DatetimeArray.tzinfo

`DatetimeArray.tzinfo`

Alias for tz attribute

pandas.arrays.DatetimeArray.week

`DatetimeArray.week`

The week ordinal of the year.

pandas.arrays.DatetimeArray.weekday

`DatetimeArray.weekday`

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

`Series.dt.dayofweek` Alias.

`Series.dt.weekday` Alias.

`Series.dt.day_name` Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.arrays.DatetimeArray.weekday_name

`DatetimeArray.weekday_name`

The name of day in a week (ex: Friday)

Deprecated since version 0.23.0.

pandas.arrays.DatetimeArray.weekofyear

DatetimeArray.**weekofyear**
The week ordinal of the year.

pandas.arrays.DatetimeArray.year

DatetimeArray.**year**
The year of the datetime.

timetuple	

Methods

argsort([ascending, kind])	Return the indices that would sort this array.
astype(dtype[, copy])	Cast to a NumPy array with 'dtype'.
ceil(freq[, ambiguous, nonexistent])	Perform ceil operation on the data to the specified *freq*.
copy([deep])	Return a copy of the array.
day_name([locale])	Return the day names of the DateTimeIndex with specified locale.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill NA/NaN values using the specified method.
floor(freq[, ambiguous, nonexistent])	Perform floor operation on the data to the specified *freq*.
isna()	A 1-D array indicating if each value is missing.
max([axis, skipna])	Return the maximum value of the Array or maximum along an axis.
min([axis, skipna])	Return the minimum value of the Array or minimum along an axis.
month_name([locale])	Return the month names of the DateTimeIndex with specified locale.
normalize()	Convert times to midnight.
repeat(repeats, *args, **kwargs)	Repeat elements of an array.
round(freq[, ambiguous, nonexistent])	Perform round operation on the data to the specified *freq*.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
strftime(date_format)	Convert to Index using specified date_format.
take(indices[, allow_fill, fill_value])	Take elements from an array.
to_julian_date()	Convert Datetime Array to float64 ndarray of Julian Dates.
to_period([freq])	Cast to PeriodArray/Index at a particular frequency.
to_perioddelta(freq)	Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq.

Continued on next page

Table 83 – continued from previous page

to_pydatetime()	Return Datetime Array/Index as object ndarray of datetime.datetime objects
tz_convert(tz)	Convert tz-aware Datetime Array/Index from one time zone to another.
tz_localize(tz[, ambiguous, nonexistent, errors])	Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.
unique()	Compute the ExtensionArray of unique values.
value_counts([dropna])	Return a Series containing counts of unique values.
view([dtype])	New view on this array with the same data.

pandas.arrays.DatetimeArray.argsort

DatetimeArray.**argsort**(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)

Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

numpy.argsort Sorting implementation used internally.

pandas.arrays.DatetimeArray.astype

DatetimeArray.**astype**(*dtype*, *copy=True*)

Cast to a NumPy array with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray] NumPy ndarray with 'dtype' for its dtype.

pandas.arrays.DatetimeArray.ceil

DatetimeArray.**ceil**(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.arrays.DatetimeArray.copy

DatetimeArray.**copy**(*deep=False*)

Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.arrays.DatetimeArray.day_name

DatetimeArray.**day_name**(*locale=None*)

Return the day names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the day name. Default is English locale.

Returns

Index Index of day names.

Examples

```
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03'],
              dtype='datetime64[ns]', freq='D')
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

pandas.arrays.DatetimeArray.dropna

DatetimeArray.**dropna**()

Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.arrays.DatetimeArray.factorize

`DatetimeArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

Note: uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a *sort* keyword as well.

pandas.arrays.DatetimeArray.fillna

`DatetimeArray.fillna(value=None, method=None, limit=None)`

Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

filled [ExtensionArray with NA/NaN filled]

pandas.arrays.DatetimeArray.floor

DatetimeArray.**floor**(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
```

(continues on next page)

(continued from previous page)

```
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.arrays.DatetimeArray.isna

`DatetimeArray.`**`isna`**`()`

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.arrays.DatetimeArray.max

`DatetimeArray.`**`max`**`(`*axis=None*, *skipna=True*, **args*, ***kwargs*`)`

Return the maximum value of the Array or maximum along an axis.

See also:

`numpy.ndarray.max`

`Index.max` Return the maximum value in an Index.

`Series.max` Return the maximum value in a Series.

pandas.arrays.DatetimeArray.min

`DatetimeArray.`**`min`**`(`*axis=None*, *skipna=True*, **args*, ***kwargs*`)`

Return the minimum value of the Array or minimum along an axis.

See also:

`numpy.ndarray.min`

Index.min Return the minimum value in an Index.

Series.min Return the minimum value in a Series.

pandas.arrays.DatetimeArray.month_name

DatetimeArray.**month_name**(*locale=None*)

Return the month names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the month name. Default is English locale.

Returns

Index Index of month names.

Examples

```
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'],
              dtype='datetime64[ns]', freq='M')
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

pandas.arrays.DatetimeArray.normalize

DatetimeArray.**normalize**()

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on Datetime Array/Index.

Returns

DatetimeArray, DatetimeIndex or Series The same type as the original data. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

floor Floor the datetimes to the specified freq.

ceil Ceil the datetimes to the specified freq.

round Round the datetimes to the specified freq.

Examples

```
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
...                     periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
                dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

pandas.arrays.DatetimeArray.repeat

`DatetimeArray.`**`repeat`**(*repeats*, **args*, ***kwargs*)

Repeat elements of an array.

See also:

`numpy.ndarray.repeat`

pandas.arrays.DatetimeArray.round

`DatetimeArray.`**`round`**(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time

- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.arrays.DatetimeArray.searchsorted

`DatetimeArray.`**`searchsorted`**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

pandas.arrays.DatetimeArray.shift

`DatetimeArray.shift(periods=1, fill_value=None)`

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.arrays.DatetimeArray.strftime

`DatetimeArray.strftime(date_format)`

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc

Parameters

date_format [str] Date format string (e.g. "%Y-%m-%d").

Returns

Index Index of formatted strings

See also:

`to_datetime` Convert the given argument to datetime.

`DatetimeIndex.normalize` Return DatetimeIndex with times to midnight.

`DatetimeIndex.round` Round the DatetimeIndex to the specified freq.

`DatetimeIndex.floor` Floor the DatetimeIndex to the specified freq.

Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

pandas.arrays.DatetimeArray.take

`DatetimeArray.`**`take`**(*indices*, *allow_fill=False*, *fill_value=None*)

Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.arrays.DatetimeArray.to_julian_date

`DatetimeArray.`**`to_julian_date`**`()`

Convert Datetime Array to float64 ndarray of Julian Dates. 0 Julian date is noon January 1, 4713 BC. http://en.wikipedia.org/wiki/Julian_day

pandas.arrays.DatetimeArray.to_period

`DatetimeArray.`**`to_period`**`(`*freq=None*`)`

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

Parameters

freq [string or Offset, optional] One of pandas' *offset strings* or an Offset object. Will be inferred by default.

Returns

PeriodArray/Index

Raises

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

See also:

`PeriodIndex` Immutable ndarray holding ordinal values.

`DatetimeIndex.to_pydatetime` Return DatetimeIndex as object.

Examples

```
>>> df = pd.DataFrame({"y": [1,2,3]},
...                   index=pd.to_datetime(["2000-03-31 00:00:00",
...                                         "2000-05-31 00:00:00",
...                                         "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

pandas.arrays.DatetimeArray.to_perioddelta

`DatetimeArray.`**`to_perioddelta`**(*freq*)

Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq. Used for vectorized offsets

Parameters

freq [Period frequency]

Returns

TimedeltaArray/Index

pandas.arrays.DatetimeArray.to_pydatetime

`DatetimeArray.`**`to_pydatetime`**()

Return Datetime Array/Index as object ndarray of datetime.datetime objects

Returns

datetimes [ndarray]

pandas.arrays.DatetimeArray.tz_convert

`DatetimeArray.`**`tz_convert`**(*tz*)

Convert tz-aware Datetime Array/Index from one time zone to another.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A *tz* of None will convert to UTC and remove the timezone information.

Returns

normalized [same type as self]

Raises

TypeError If Datetime Array/Index is tz-naive.

See also:

DatetimeIndex.tz A timezone that has a variable offset from UTC.

DatetimeIndex.tz_localize Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.

Examples

With the *tz* parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.date_range(start='2014-08-01 09:00',
...                     freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the tz=None, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.date_range(start='2014-08-01 09:00',freq='H',
...                     periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
                dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
                dtype='datetime64[ns]', freq='H')
```

pandas.arrays.DatetimeArray.tz_localize

DatetimeArray.**tz_localize**(*tz*, *ambiguous='raise'*, *nonexistent='raise'*, *errors=None*)

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

ambiguous ['infer', 'NaT', bool array, default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

errors [{'raise', 'coerce'}, default None]

- 'raise' will raise a NonExistentTimeError if a timestamp is not valid in the specified time zone (e.g. due to a transition from or to DST time). Use `nonexistent='raise'` instead.
- 'coerce' will return NaT if the timestamp can not be converted to the specified time zone. Use `nonexistent='NaT'` instead.

Deprecated since version 0.24.0.

Returns

result [same type as self] Array/Index converted to the specified time zone.

Raises

TypeError If the Datetime Array/Index is tz-aware and tz is not None.

See also:

`DatetimeIndex.tz_convert` Convert tz-aware DatetimeIndex from one time zone to another.

Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
               '2018-03-02 09:00:00-05:00',
               '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the `tz=None`, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time: >>> s = pd.to_datetime(pd.Series([… '2018-10-28 01:30:00', … '2018-10-28 02:00:00', … '2018-10-28 02:30:00', … '2018-10-28 02:00:00', … '2018-10-28 02:30:00', … '2018-10-28 03:00:00', … '2018-10-28 03:30:00'])) >>> s.dt.tz_localize('CET', ambiguous='infer') 2018-10-28 01:30:00+02:00 0 2018-10-28 02:00:00+02:00 1 2018-10-28 02:30:00+02:00 2 2018-10-28 02:00:00+01:00 3 2018-10-28 02:30:00+01:00 4 2018-10-28 03:00:00+01:00 5 2018-10-28 03:30:00+01:00 6 dtype: int64

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.to_datetime(pd.Series([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.dt.tz_localize('CET', ambiguous=np.array([True, True, False]))
0   2018-10-28 01:20:00+02:00
1   2018-10-28 02:36:00+02:00
2   2018-10-28 03:46:00+01:00
dtype: datetime64[ns, CET]
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.to_datetime(pd.Series([… '2015-03-29 02:30:00', … '2015-03-29 03:30:00'])) >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 0 2015-03-29 03:00:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 0 2015-03-29 01:59:59.999999999+01:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 0 2015-03-29 03:30:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw']

pandas.arrays.DatetimeArray.unique

`DatetimeArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

pandas.arrays.DatetimeArray.value_counts

`DatetimeArray.value_counts(dropna=False)`

Return a Series containing counts of unique values.

Parameters

dropna [boolean, default True] Don't include counts of NaT values.

Returns

Series

pandas.arrays.DatetimeArray.view

`DatetimeArray.view(dtype=None)`

New view on this array with the same data.

Parameters

dtype [numpy dtype, optional]

Returns

ndarray With the specified *dtype*.

map	

pandas.DatetimeTZDtype

`class pandas.DatetimeTZDtype(unit='ns', tz=None)`

A np.dtype duck-typed class, suitable for holding a custom datetime with tz dtype.

THIS IS NOT A REAL NUMPY DTYPE, but essentially a sub-class of np.datetime64[ns]

Attributes

name	A string representation of the dtype.
names	Ordered list of field names, or None if there are no fields.
tz	The timezone.
unit	The precision of the datetime data.

pandas.DatetimeTZDtype.name

DatetimeTZDtype.**name**
A string representation of the dtype.

pandas.DatetimeTZDtype.names

DatetimeTZDtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.DatetimeTZDtype.tz

DatetimeTZDtype.**tz**
The timezone.

pandas.DatetimeTZDtype.unit

DatetimeTZDtype.**unit**
The precision of the datetime data.

subdtype	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construct a DatetimeTZDtype from a string.
is_dtype(dtype)	Check if we match 'dtype'.
reset_cache()	clear the cache
type	alias of pandas._libs.tslibs.timestamps.Timestamp

pandas.DatetimeTZDtype.construct_array_type

classmethod DatetimeTZDtype.**construct_array_type**()
Return the array type associated with this dtype

Returns

type

pandas.DatetimeTZDtype.construct_from_string

classmethod DatetimeTZDtype.**construct_from_string**(*string*)
Construct a DatetimeTZDtype from a string.

Parameters

string [str] The string alias for this DatetimeTZDtype. Should be formatted like `datetime64[ns, <tz>]`, where `<tz>` is the timezone name.

Examples

```
>>> DatetimeTZDtype.construct_from_string('datetime64[ns, UTC]')
datetime64[ns, UTC]
```

pandas.DatetimeTZDtype.is_dtype

classmethod DatetimeTZDtype.**is_dtype**(*dtype*)
Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.DatetimeTZDtype.reset_cache

classmethod DatetimeTZDtype.**reset_cache**()
clear the cache

6.5.3 Timedelta Data

NumPy can natively represent timedeltas. Pandas provides *Timedelta* for symmetry with *Timestamp*.

Timedelta	Represents a duration, the difference between two dates or times.

pandas.Timedelta

class pandas.**Timedelta**
Represents a duration, the difference between two dates or times.

Timedelta is the pandas equivalent of python's `datetime.timedelta` and is interchangeable with it in most cases.

Parameters

value [Timedelta, timedelta, np.timedelta64, string, or integer]

unit [str, optional] Denote the unit of the input, if input is an integer. Default 'ns'. Possible values: {'Y', 'M', 'W', 'D', 'days', 'day', 'hours', hour', 'hr', 'h', 'm', 'minute', 'min', 'minutes', 'T', 'S', 'seconds', 'sec', 'second', 'ms', 'milliseconds', 'millisecond', 'milli', 'millis', 'L', 'us', 'microseconds', 'microsecond', 'micro', 'micros', 'U', 'ns', 'nanoseconds', 'nano', 'nanos', 'nanosecond', 'N'}

days, seconds, microseconds,

milliseconds, minutes, hours, weeks [numeric, optional] Values for construction in compat with datetime.timedelta. np ints and floats will be coerced to python ints and floats.

Notes

The `.value` attribute is always in ns.

Attributes

asm8	Return a numpy timedelta64 array scalar view.
components	Return a Components NamedTuple-like
days	Number of days.
delta	Return the timedelta in nanoseconds (ns), for internal compatibility.
microseconds	Number of microseconds (>= 0 and less than 1 second).
nanoseconds	Return the number of nanoseconds (n), where 0 <= n < 1 microsecond.
resolution	Return a string representing the lowest timedelta resolution.
seconds	Number of seconds (>= 0 and less than 1 day).

pandas.Timedelta.asm8

`Timedelta.`**`asm8`**

Return a numpy timedelta64 array scalar view.

Provides access to the array scalar view (i.e. a combination of the value and the units) associated with the numpy.timedelta64().view(), including a 64-bit integer representation of the timedelta in nanoseconds (Python int compatible).

Returns

numpy timedelta64 array scalar view Array scalar view of the timedelta in nanoseconds.

Examples

```
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
>>> td.asm8
numpy.timedelta64(86520000003042,'ns')
```

```
>>> td = pd.Timedelta('2 min 3 s')
>>> td.asm8
numpy.timedelta64(123000000000,'ns')
```

```
>>> td = pd.Timedelta('3 ms 5 us')
>>> td.asm8
numpy.timedelta64(3005000,'ns')
```

```
>>> td = pd.Timedelta(42, unit='ns')
>>> td.asm8
numpy.timedelta64(42,'ns')
```

pandas.Timedelta.components

Timedelta.**components**

Return a Components NamedTuple-like

pandas.Timedelta.days

Timedelta.**days**

Number of days.

pandas.Timedelta.delta

Timedelta.**delta**

Return the timedelta in nanoseconds (ns), for internal compatibility.

Returns

int Timedelta in nanoseconds.

Examples

```
>>> td = pd.Timedelta('1 days 42 ns')
>>> td.delta
86400000000042
```

```
>>> td = pd.Timedelta('3 s')
>>> td.delta
3000000000
```

```
>>> td = pd.Timedelta('3 ms 5 us')
>>> td.delta
3005000
```

```
>>> td = pd.Timedelta(42, unit='ns')
>>> td.delta
42
```

pandas.Timedelta.microseconds

`Timedelta.microseconds`
Number of microseconds (>= 0 and less than 1 second).

pandas.Timedelta.nanoseconds

`Timedelta.nanoseconds`
Return the number of nanoseconds (n), where 0 <= n < 1 microsecond.

Returns

int Number of nanoseconds.

See also:

Timedelta.components Return all attributes with assigned values (i.e. days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds).

Examples

Using string input

```
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
>>> td.nanoseconds
42
```

Using integer input

```
>>> td = pd.Timedelta(42, unit='ns')
>>> td.nanoseconds
42
```

pandas.Timedelta.resolution

`Timedelta.resolution`
Return a string representing the lowest timedelta resolution.

Each timedelta has a defined resolution that represents the lowest OR most granular level of precision. Each level of resolution is represented by a short string as defined below:

Resolution: Return value

- Days: 'D'
- Hours: 'H'
- Minutes: 'T'
- Seconds: 'S'
- Milliseconds: 'L'
- Microseconds: 'U'
- Nanoseconds: 'N'

Returns

str Timedelta resolution.

Examples

```
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
>>> td.resolution
'N'
```

```
>>> td = pd.Timedelta('1 days 2 min 3 us')
>>> td.resolution
'U'
```

```
>>> td = pd.Timedelta('2 min 3 s')
>>> td.resolution
'S'
```

```
>>> td = pd.Timedelta(36, unit='us')
>>> td.resolution
'U'
```

pandas.Timedelta.seconds

`Timedelta.`**`seconds`**
Number of seconds (>= 0 and less than 1 day).

freq	
is_populated	
value	

Methods

ceil	return a new Timedelta ceiled to this resolution
floor	return a new Timedelta floored to this resolution
isoformat	Format Timedelta as ISO 8601 Duration like `P[n]Y[n]M[n]DT[n]H[n]M[n]S`, where the `[n]` s are replaced by the values.
round	Round the Timedelta to the specified resolution
to_pytimedelta	return an actual datetime.timedelta object note: we lose nanosecond resolution if any
to_timedelta64	Returns a numpy.timedelta64 object with 'ns' precision
total_seconds	Total duration of timedelta in seconds (to ns precision)
view	array view compat

pandas.Timedelta.ceil

`Timedelta.ceil`
return a new Timedelta ceiled to this resolution

Parameters

freq [a freq string indicating the ceiling resolution]

pandas.Timedelta.floor

`Timedelta.floor`
return a new Timedelta floored to this resolution

Parameters

freq [a freq string indicating the flooring resolution]

pandas.Timedelta.isoformat

`Timedelta.isoformat()`
Format Timedelta as ISO 8601 Duration like `P[n]Y[n]M[n]DT[n]H[n]M[n]S`, where the `[n]` s are replaced by the values. See https://en.wikipedia.org/wiki/ISO_8601#Durations

New in version 0.20.0.

Returns

formatted [str]

See also:

Timestamp.isoformat

Notes

The longest component is days, whose value may be larger than 365. Every component is always included, even if its value is 0. Pandas uses nanosecond precision, so up to 9 decimal places may be included in the seconds component. Trailing 0's are removed from the seconds component after the decimal. We do not 0 pad components, so it's …*T5H*…, not …*T05H*…

Examples

```
>>> td = pd.Timedelta(days=6, minutes=50, seconds=3,
...                   milliseconds=10, microseconds=10, nanoseconds=12)
>>> td.isoformat()
'P6DT0H50M3.010010012S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(days=500.5).isoformat()
'P500DT12H0MS'
```

pandas.Timedelta.round

`Timedelta.round`
Round the Timedelta to the specified resolution

Parameters

freq [a freq string indicating the rounding resolution]

Returns

a new Timedelta rounded to the given resolution of 'freq'

Raises

ValueError if the freq cannot be converted

pandas.Timedelta.to_pytimedelta

`Timedelta.to_pytimedelta()`
return an actual datetime.timedelta object note: we lose nanosecond resolution if any

pandas.Timedelta.to_timedelta64

`Timedelta.to_timedelta64()`
Returns a numpy.timedelta64 object with 'ns' precision

pandas.Timedelta.total_seconds

`Timedelta.total_seconds()`
Total duration of timedelta in seconds (to ns precision)

pandas.Timedelta.view

`Timedelta.view()`
array view compat

Properties

Timedelta.asm8	Return a numpy timedelta64 array scalar view.
Timedelta.components	Return a Components NamedTuple-like
Timedelta.days	Number of days.
Timedelta.delta	Return the timedelta in nanoseconds (ns), for internal compatibility.
Timedelta.freq	
Timedelta.is_populated	
Timedelta.max	
Timedelta.microseconds	Number of microseconds (>= 0 and less than 1 second).
Timedelta.min	

Continued on next page

Table 89 – continued from previous page

Timedelta.nanoseconds	Return the number of nanoseconds (n), where 0 <= n < 1 microsecond.
Timedelta.resolution	Return a string representing the lowest timedelta resolution.
Timedelta.seconds	Number of seconds (>= 0 and less than 1 day).
Timedelta.value	
Timedelta.view	array view compat

pandas.Timedelta.freq

Timedelta.**freq**

pandas.Timedelta.is_populated

Timedelta.**is_populated**

pandas.Timedelta.max

Timedelta.**max = Timedelta('106751 days 23:47:16.854775')**

pandas.Timedelta.min

Timedelta.**min = Timedelta('-106752 days +00:12:43.145224')**

pandas.Timedelta.value

Timedelta.**value**

Methods

Timedelta.ceil	return a new Timedelta ceiled to this resolution
Timedelta.floor	return a new Timedelta floored to this resolution
Timedelta.isoformat	Format Timedelta as ISO 8601 Duration like `P[n]Y[n]M[n]DT[n]H[n]M[n]S`, where the `[n]` s are replaced by the values.
Timedelta.round	Round the Timedelta to the specified resolution
Timedelta.to_pytimedelta	return an actual datetime.timedelta object note: we lose nanosecond resolution if any
Timedelta.to_timedelta64	Returns a numpy.timedelta64 object with 'ns' precision
Timedelta.total_seconds	Total duration of timedelta in seconds (to ns precision)

A collection of timedeltas may be stored in a `TimedeltaArray`.

arrays.TimedeltaArray(values[, dtype, freq, …])	Pandas ExtensionArray for timedelta data.

pandas.arrays.TimedeltaArray

class pandas.arrays.**TimedeltaArray**(*values*, *dtype=dtype('<m8[ns]')*, *freq=None*, *copy=False*)

Pandas ExtensionArray for timedelta data.

New in version 0.24.0.

> **Warning:** TimedeltaArray is currently experimental, and its API may change without warning. In particular, `TimedeltaArray.dtype` is expected to change to be an instance of an `ExtensionDtype` subclass.

Parameters

values [array-like] The timedelta data.

dtype [numpy.dtype] Currently, only `numpy.dtype("timedelta64[ns]")` is accepted.

freq [Offset, optional]

copy [bool, default False] Whether to copy the underlying array of data.

Attributes

asi8	Integer representation of the values.
components	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.
days	Number of days for each element.
dtype	The dtype for the TimedeltaArray.
freq	Return the frequency object if it is set, otherwise None.
freqstr	Return the frequency object as a string if its set, otherwise None
inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.
microseconds	Number of microseconds (>= 0 and less than 1 second) for each element.
nanoseconds	Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
nbytes	The number of bytes needed to store this object in memory.
resolution	Returns day, hour, minute, second, millisecond or microsecond
seconds	Number of seconds (>= 0 and less than 1 day) for each element.
shape	Return a tuple of the array dimensions.
size	The number of elements in this array.

pandas.arrays.TimedeltaArray.asi8

`TimedeltaArray.asi8`

Integer representation of the values.

Returns

ndarray An ndarray with int64 dtype.

pandas.arrays.TimedeltaArray.components

`TimedeltaArray.components`

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns

a DataFrame

pandas.arrays.TimedeltaArray.days

`TimedeltaArray.days`

Number of days for each element.

pandas.arrays.TimedeltaArray.dtype

`TimedeltaArray.dtype`

The dtype for the TimedeltaArray.

> **Warning:** A future version of pandas will change dtype to be an instance of a *pandas.api.extensions.ExtensionDtype* subclass, not a `numpy.dtype`.

Returns

numpy.dtype

pandas.arrays.TimedeltaArray.freq

`TimedeltaArray.freq`

Return the frequency object if it is set, otherwise None.

pandas.arrays.TimedeltaArray.freqstr

`TimedeltaArray.freqstr`

Return the frequency object as a string if its set, otherwise None

pandas.arrays.TimedeltaArray.inferred_freq

`TimedeltaArray.inferred_freq`
Tryies to return a string representing a frequency guess, generated by infer_freq. Returns None if it can't autodetect the frequency.

pandas.arrays.TimedeltaArray.microseconds

`TimedeltaArray.microseconds`
Number of microseconds (>= 0 and less than 1 second) for each element.

pandas.arrays.TimedeltaArray.nanoseconds

`TimedeltaArray.nanoseconds`
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

pandas.arrays.TimedeltaArray.nbytes

`TimedeltaArray.nbytes`
The number of bytes needed to store this object in memory.

pandas.arrays.TimedeltaArray.resolution

`TimedeltaArray.resolution`
Returns day, hour, minute, second, millisecond or microsecond

pandas.arrays.TimedeltaArray.seconds

`TimedeltaArray.seconds`
Number of seconds (>= 0 and less than 1 day) for each element.

pandas.arrays.TimedeltaArray.shape

`TimedeltaArray.shape`
Return a tuple of the array dimensions.

pandas.arrays.TimedeltaArray.size

`TimedeltaArray.size`
The number of elements in this array.

Methods

argsort([ascending, kind])	Return the indices that would sort this array.

Continued on next page

Table 93 – continued from previous page

astype(dtype[, copy])	Cast to a NumPy array with 'dtype'.
ceil(freq[, ambiguous, nonexistent])	Perform ceil operation on the data to the specified *freq*.
copy([deep])	Return a copy of the array.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill NA/NaN values using the specified method.
floor(freq[, ambiguous, nonexistent])	Perform floor operation on the data to the specified *freq*.
isna()	A 1-D array indicating if each value is missing.
max([axis, skipna])	Return the maximum value of the Array or maximum along an axis.
min([axis, skipna])	Return the minimum value of the Array or minimum along an axis.
repeat(repeats, *args, **kwargs)	Repeat elements of an array.
round(freq[, ambiguous, nonexistent])	Perform round operation on the data to the specified *freq*.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
take(indices[, allow_fill, fill_value])	Take elements from an array.
to_pytimedelta()	Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.
total_seconds()	Return total duration of each element expressed in seconds.
unique()	Compute the ExtensionArray of unique values.
value_counts([dropna])	Return a Series containing counts of unique values.
view([dtype])	New view on this array with the same data.

pandas.arrays.TimedeltaArray.argsort

TimedeltaArray.**argsort**(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)

Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

numpy.argsort Sorting implementation used internally.

pandas.arrays.TimedeltaArray.astype

`TimedeltaArray.astype(dtype, copy=True)`
Cast to a NumPy array with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray] NumPy ndarray with 'dtype' for its dtype.

pandas.arrays.TimedeltaArray.ceil

`TimedeltaArray.ceil(freq, ambiguous='raise', nonexistent='raise')`
Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.arrays.TimedeltaArray.copy

`TimedeltaArray.copy(deep=False)`
Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.arrays.TimedeltaArray.dropna

`TimedeltaArray.dropna()`
Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.arrays.TimedeltaArray.factorize

`TimedeltaArray.factorize(na_sentinel=-1)`
Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

> **Note:** uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

`pandas.factorize()` offers a *sort* keyword as well.

pandas.arrays.TimedeltaArray.fillna

`TimedeltaArray.`**`fillna`**`(`*value=None*, *method=None*, *limit=None*`)`
Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

filled [ExtensionArray with NA/NaN filled]

pandas.arrays.TimedeltaArray.floor

`TimedeltaArray.`**`floor`**`(`*freq*, *ambiguous='raise'*, *nonexistent='raise'*`)`
Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.arrays.TimedeltaArray.isna

`TimedeltaArray.isna()`

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.arrays.TimedeltaArray.max

`TimedeltaArray.max(`*axis=None*, *skipna=True*, **args*, ***kwargs*`)`

Return the maximum value of the Array or maximum along an axis.

See also:

`numpy.ndarray.max`

`Index.max` Return the maximum value in an Index.

`Series.max` Return the maximum value in a Series.

pandas.arrays.TimedeltaArray.min

`TimedeltaArray.min(`*axis=None*, *skipna=True*, **args*, ***kwargs*`)`

Return the minimum value of the Array or minimum along an axis.

See also:

`numpy.ndarray.min`

`Index.min` Return the minimum value in an Index.

`Series.min` Return the minimum value in a Series.

pandas.arrays.TimedeltaArray.repeat

`TimedeltaArray.repeat(`*repeats*, **args*, ***kwargs*`)`

Repeat elements of an array.

See also:

`numpy.ndarray.repeat`

pandas.arrays.TimedeltaArray.round

`TimedeltaArray.round`(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
```

(continues on next page)

(continued from previous page)

```
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.arrays.TimedeltaArray.searchsorted

`TimedeltaArray.`**`searchsorted`**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

pandas.arrays.TimedeltaArray.shift

`TimedeltaArray.`**`shift`**(*periods=1*, *fill_value=None*)

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.arrays.TimedeltaArray.take

`TimedeltaArray.take(indices, allow_fill=False, fill_value=None)`
Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.arrays.TimedeltaArray.to_pytimedelta

TimedeltaArray.**to_pytimedelta**()

Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.

Returns

datetimes [ndarray]

pandas.arrays.TimedeltaArray.total_seconds

TimedeltaArray.**total_seconds**()

Return total duration of each element expressed in seconds.

This method is available directly on TimedeltaArray, TimedeltaIndex and on Series containing timedelta values under the `.dt` namespace.

Returns

seconds [[ndarray, Float64Index, Series]] When the calling object is a TimedeltaArray, the return type is ndarray. When the calling object is a TimedeltaIndex, the return type is a Float64Index. When the calling object is a Series, the return type is Series of type *float64* whose index is the same as the original.

See also:

datetime.timedelta.total_seconds Standard library version of this method.

TimedeltaIndex.components Return a DataFrame with components of each Timedelta.

Examples

Series

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0   0 days
```

(continues on next page)

(continued from previous page)

```
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.total_seconds()
0         0.0
1     86400.0
2    172800.0
3    259200.0
4    345600.0
dtype: float64
```

TimedeltaIndex

```
>>> idx = pd.to_timedelta(np.arange(5), unit='d')
>>> idx
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

```
>>> idx.total_seconds()
Float64Index([0.0, 86400.0, 172800.0, 259200.00000000003, 345600.0],
             dtype='float64')
```

pandas.arrays.TimedeltaArray.unique

`TimedeltaArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

pandas.arrays.TimedeltaArray.value_counts

`TimedeltaArray.value_counts(dropna=False)`

Return a Series containing counts of unique values.

Parameters

dropna [boolean, default True] Don't include counts of NaT values.

Returns

Series

pandas.arrays.TimedeltaArray.view

`TimedeltaArray.view(dtype=None)`

New view on this array with the same data.

Parameters

dtype [numpy dtype, optional]

Returns

ndarray With the specified *dtype*.

map	

6.5.4 Timespan Data

Pandas represents spans of times as *Period* objects.

6.5.5 Period

Period	Represents a period of time

pandas.Period

class pandas.**Period**

Represents a period of time

Parameters

value [Period or compat.string_types, default None] The time period represented (e.g., '4Q2005')

freq [str, default None] One of pandas period strings or corresponding objects

year [int, default None]

month [int, default 1]

quarter [int, default None]

day [int, default 1]

hour [int, default 0]

minute [int, default 0]

second [int, default 0]

Attributes

day	Get day of the month that a Period falls on.
dayofweek	Day of the week the period lies in, with Monday=0 and Sunday=6.
dayofyear	Return the day of the year.
days_in_month	Get the total number of days in the month that this period falls on.
daysinmonth	Get the total number of days of the month that the Period falls in.
hour	Get the hour of the day component of the Period.
minute	Get minute of the hour component of the Period.

Continued on next page

Table 95 – continued from previous page

qyear	Fiscal year the Period lies in according to its starting-quarter.
second	Get the second component of the Period.
start_time	Get the Timestamp for the start of the period.
week	Get the week of the year on the given Period.
weekday	Day of the week the period lies in, with Monday=0 and Sunday=6.

pandas.Period.day

Period.**day**

Get day of the month that a Period falls on.

Returns

int

See also:

Period.dayofweek Get the day of the week.

Period.dayofyear Get the day of the year.

Examples

```
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.day
11
```

pandas.Period.dayofweek

Period.**dayofweek**

Day of the week the period lies in, with Monday=0 and Sunday=6.

If the period frequency is lower than daily (e.g. hourly), and the period spans over multiple days, the day at the start of the period is used.

If the frequency is higher than daily (e.g. monthly), the last day of the period is used.

Returns

int Day of the week.

See also:

Period.dayofweek Day of the week the period lies in.

Period.weekday Alias of Period.dayofweek.

Period.day Day of the month.

Period.dayofyear Day of the year.

Examples

```
>>> per = pd.Period('2017-12-31 22:00', 'H')
>>> per.dayofweek
6
```

For periods that span over multiple days, the day at the beginning of the period is returned.

```
>>> per = pd.Period('2017-12-31 22:00', '4H')
>>> per.dayofweek
6
>>> per.start_time.dayofweek
6
```

For periods with a frequency higher than days, the last day of the period is returned.

```
>>> per = pd.Period('2018-01', 'M')
>>> per.dayofweek
2
>>> per.end_time.dayofweek
2
```

pandas.Period.dayofyear

`Period.dayofyear`

Return the day of the year.

This attribute returns the day of the year on which the particular date occurs. The return value ranges between 1 to 365 for regular years and 1 to 366 for leap years.

Returns

int The day of year.

See also:

Period.day Return the day of the month.

Period.dayofweek Return the day of week.

PeriodIndex.dayofyear Return the day of year of all indexes.

Examples

```
>>> period = pd.Period("2015-10-23", freq='H')
>>> period.dayofyear
296
>>> period = pd.Period("2012-12-31", freq='D')
>>> period.dayofyear
366
>>> period = pd.Period("2013-01-01", freq='D')
>>> period.dayofyear
1
```

pandas.Period.days_in_month

`Period.days_in_month`
Get the total number of days in the month that this period falls on.

Returns

int

See also:

Period.daysinmonth Gets the number of days in the month.

DatetimeIndex.daysinmonth Gets the number of days in the month.

calendar.monthrange Returns a tuple containing weekday (0-6 ~ Mon-Sun) and number of days (28-31).

Examples

```
>>> p = pd.Period('2018-2-17')
>>> p.days_in_month
28
```

```
>>> pd.Period('2018-03-01').days_in_month
31
```

Handles the leap year case as well:

```
>>> p = pd.Period('2016-2-17')
>>> p.days_in_month
29
```

pandas.Period.daysinmonth

`Period.daysinmonth`
Get the total number of days of the month that the Period falls in.

Returns

int

See also:

Period.days_in_month Return the days of the month.

Period.dayofyear Return the day of the year.

Examples

```
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.daysinmonth
31
```

pandas.Period.hour

Period.**hour**

Get the hour of the day component of the Period.

Returns

int The hour as an integer, between 0 and 23.

See also:

Period.second Get the second component of the Period.

Period.minute Get the minute component of the Period.

Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.hour
13
```

Period longer than a day

```
>>> p = pd.Period("2018-03-11", freq="M")
>>> p.hour
0
```

pandas.Period.minute

Period.**minute**

Get minute of the hour component of the Period.

Returns

int The minute as an integer, between 0 and 59.

See also:

Period.hour Get the hour component of the Period.

Period.second Get the second component of the Period.

Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.minute
3
```

pandas.Period.qyear

Period.**qyear**

Fiscal year the Period lies in according to its starting-quarter.

The *year* and the *qyear* of the period will be the same if the fiscal and calendar years are the same. When they are not, the fiscal year can be different from the calendar year of the period.

Returns

int The fiscal year of the period.

See also:

Period.year Return the calendar year of the period.

Examples

If the natural and fiscal year are the same, *qyear* and *year* will be the same.

```
>>> per = pd.Period('2018Q1', freq='Q')
>>> per.qyear
2018
>>> per.year
2018
```

If the fiscal year starts in April (*Q-MAR*), the first quarter of 2018 will start in April 2017. *year* will then be 2018, but *qyear* will be the fiscal year, 2018.

```
>>> per = pd.Period('2018Q1', freq='Q-MAR')
>>> per.start_time
Timestamp('2017-04-01 00:00:00')
>>> per.qyear
2018
>>> per.year
2017
```

pandas.Period.second

Period.**second**

Get the second component of the Period.

Returns

int The second of the Period (ranges from 0 to 59).

See also:

Period.hour Get the hour component of the Period.

Period.minute Get the minute component of the Period.

Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.second
12
```

pandas.Period.start_time

Period.**start_time**

Get the Timestamp for the start of the period.

Returns

Timestamp

See also:

Period.end_time Return the end Timestamp.

Period.dayofyear Return the day of year.

Period.daysinmonth Return the days in that month.

Period.dayofweek Return the day of the week.

Examples

```
>>> period = pd.Period('2012-1-1', freq='D')
>>> period
Period('2012-01-01', 'D')
```

```
>>> period.start_time
Timestamp('2012-01-01 00:00:00')
```

```
>>> period.end_time
Timestamp('2012-01-01 23:59:59.999999999')
```

pandas.Period.week

Period.**week**

Get the week of the year on the given Period.

Returns

int

See also:

Period.dayofweek Get the day component of the Period.

Period.weekday Get the day component of the Period.

Examples

```
>>> p = pd.Period("2018-03-11", "H")
>>> p.week
10
```

```
>>> p = pd.Period("2018-02-01", "D")
>>> p.week
5
```

```
>>> p = pd.Period("2018-01-06", "D")
>>> p.week
1
```

pandas.Period.weekday

`Period.weekday`

Day of the week the period lies in, with Monday=0 and Sunday=6.

If the period frequency is lower than daily (e.g. hourly), and the period spans over multiple days, the day at the start of the period is used.

If the frequency is higher than daily (e.g. monthly), the last day of the period is used.

Returns

int Day of the week.

See also:

Period.dayofweek Day of the week the period lies in.

Period.weekday Alias of Period.dayofweek.

Period.day Day of the month.

Period.dayofyear Day of the year.

Examples

```
>>> per = pd.Period('2017-12-31 22:00', 'H')
>>> per.dayofweek
6
```

For periods that span over multiple days, the day at the beginning of the period is returned.

```
>>> per = pd.Period('2017-12-31 22:00', '4H')
>>> per.dayofweek
6
>>> per.start_time.dayofweek
6
```

For periods with a frequency higher than days, the last day of the period is returned.

```
>>> per = pd.Period('2018-01', 'M')
>>> per.dayofweek
2
>>> per.end_time.dayofweek
2
```

end_time	
freq	
freqstr	
is_leap_year	
month	
ordinal	
quarter	
weekofyear	
year	

Methods

asfreq	Convert Period to desired frequency, either at the start or end of the interval
strftime	Returns the string representation of the *Period*, depending on the selected `fmt`.
to_timestamp	Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

pandas.Period.asfreq

`Period.`**`asfreq`**`()`

Convert Period to desired frequency, either at the start or end of the interval

Parameters

freq [string]

how [{'E', 'S', 'end', 'start'}, default 'end'] Start or end of the timespan

Returns

resampled [Period]

pandas.Period.strftime

`Period.`**`strftime`**`()`

Returns the string representation of the *Period*, depending on the selected `fmt`. `fmt` must be a string containing one or several directives. The method recognizes the same directives as the `time.strftime()` function of the standard Python distribution, as well as the specific additional directives `%f`, `%F`, `%q`. (formatting & docs originally from scikits.timeries)

Directive	Meaning	Notes
`%a`	Locale's abbreviated weekday name.	
`%A`	Locale's full weekday name.	
`%b`	Locale's abbreviated month name.	
`%B`	Locale's full month name.	
`%c`	Locale's appropriate date and time representation.	
`%d`	Day of the month as a decimal number [01,31].	
`%f`	'Fiscal' year without a century as a decimal number [00,99]	(1)
`%F`	'Fiscal' year with a century as a decimal number	(2)
`%H`	Hour (24-hour clock) as a decimal number [00,23].	
`%I`	Hour (12-hour clock) as a decimal number [01,12].	
`%j`	Day of the year as a decimal number [001,366].	
`%m`	Month as a decimal number [01,12].	
`%M`	Minute as a decimal number [00,59].	
`%p`	Locale's equivalent of either AM or PM.	(3)
`%q`	Quarter as a decimal number [01,04]	
`%S`	Second as a decimal number [00,61].	(4)
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(5)
`%w`	Weekday as a decimal number [0(Sunday),6].	
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(5)
`%x`	Locale's appropriate date representation.	
`%X`	Locale's appropriate time representation.	
`%y`	Year without century as a decimal number [00,99].	
`%Y`	Year with century as a decimal number.	
`%Z`	Time zone name (no characters if no time zone exists).	
`%%`	A literal `'%'` character.	

Notes

(1) The `%f` directive is the same as `%y` if the frequency is not quarterly. Otherwise, it corresponds to the 'fiscal' year, as defined by the *qyear* attribute.

(2) The `%F` directive is the same as `%Y` if the frequency is not quarterly. Otherwise, it corresponds to the 'fiscal' year, as defined by the *qyear* attribute.

(3) The `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) The range really is `0` to `61`; this accounts for leap seconds and the (very rare) double leap seconds.

(5) The `%U` and `%W` directives are only used in calculations when the day of the week and the year are specified.

Examples

```
>>> a = Period(freq='Q-JUL', year=2006, quarter=1)
>>> a.strftime('%F-Q%q')
```

(continues on next page)

(continued from previous page)

```
'2006-Q1'
>>> # Output the last month in the quarter of this date
>>> a.strftime('%b-%Y')
'Oct-2005'
>>>
>>> a = Period(freq='D', year=2001, month=1, day=1)
>>> a.strftime('%d-%b-%Y')
'01-Jan-2006'
>>> a.strftime('%b. %d, %Y was a %A')
'Jan. 01, 2001 was a Monday'
```

pandas.Period.to_timestamp

Period.**to_timestamp**()

Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

Parameters

freq [string or DateOffset] Target frequency. Default is 'D' if self.freq is week or longer and 'S' otherwise

how [str, default 'S' (start)] 'S', 'E'. Can be aliased as case insensitive 'Start', 'Finish', 'Begin', 'End'

Returns

Timestamp

now	

Properties

Period.day	Get day of the month that a Period falls on.
Period.dayofweek	Day of the week the period lies in, with Monday=0 and Sunday=6.
Period.dayofyear	Return the day of the year.
Period.days_in_month	Get the total number of days in the month that this period falls on.
Period.daysinmonth	Get the total number of days of the month that the Period falls in.
Period.end_time	
Period.freq	
Period.freqstr	
Period.hour	Get the hour of the day component of the Period.
Period.is_leap_year	
Period.minute	Get minute of the hour component of the Period.
Period.month	
Period.ordinal	
Period.quarter	

Continued on next page

Table 97 – continued from previous page

Period.qyear	Fiscal year the Period lies in according to its starting-quarter.
Period.second	Get the second component of the Period.
Period.start_time	Get the Timestamp for the start of the period.
Period.week	Get the week of the year on the given Period.
Period.weekday	Day of the week the period lies in, with Monday=0 and Sunday=6.
Period.weekofyear	
Period.year	

pandas.Period.end_time

Period.**end_time**

pandas.Period.freq

Period.**freq**

pandas.Period.freqstr

Period.**freqstr**

pandas.Period.is_leap_year

Period.**is_leap_year**

pandas.Period.month

Period.**month**

pandas.Period.ordinal

Period.**ordinal**

pandas.Period.quarter

Period.**quarter**

pandas.Period.weekofyear

Period.**weekofyear**

pandas.Period.year

Period.**year**

Methods

Period.asfreq	Convert Period to desired frequency, either at the start or end of the interval
Period.now	
Period.strftime	Returns the string representation of the *Period*, depending on the selected `fmt`.
Period.to_timestamp	Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

pandas.Period.now

`Period.now()`

A collection of timedeltas may be stored in a `arrays.PeriodArray`. Every period in a `PeriodArray` must have the same `freq`.

arrays.DatetimeArray(values[, dtype, freq, copy])	Pandas ExtensionArray for tz-naive or tz-aware datetime data.
PeriodDtype	A Period duck-typed class, suitable for holding a period with freq dtype.

pandas.PeriodDtype

class pandas.**PeriodDtype**

A Period duck-typed class, suitable for holding a period with freq dtype.

THIS IS NOT A REAL NUMPY DTYPE, but essentially a sub-class of np.int64.

Attributes

na_value	float(x) -> floating point number
name	A string identifying the data type.
names	Ordered list of field names, or None if there are no fields.

pandas.PeriodDtype.na_value

`PeriodDtype.na_value`

float(x) -> floating point number

Convert a string or number to a floating point number, if possible.

pandas.PeriodDtype.name

`PeriodDtype.name`

A string identifying the data type.

Will be used for display in, e.g. `Series.dtype`

pandas.PeriodDtype.names

PeriodDtype.**names**

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

subdtype	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Strict construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Return a boolean if we if the passed type is an actual dtype that we can match (via string or type)
reset_cache()	clear the cache
type	alias of pandas._libs.tslibs.period.Period

pandas.PeriodDtype.construct_array_type

classmethod PeriodDtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.PeriodDtype.construct_from_string

classmethod PeriodDtype.**construct_from_string**(*string*)

Strict construction from a string, raise a TypeError if not possible

pandas.PeriodDtype.is_dtype

classmethod PeriodDtype.**is_dtype**(*dtype*)

Return a boolean if we if the passed type is an actual dtype that we can match (via string or type)

pandas.PeriodDtype.reset_cache

classmethod PeriodDtype.**reset_cache**()

clear the cache

6.5.6 Interval Data

Arbitrary intervals can be represented as *Interval* objects.

Interval	Immutable object implementing an Interval, a bounded slice-like interval.

pandas.Interval

class pandas.**Interval**

Immutable object implementing an Interval, a bounded slice-like interval.

New in version 0.20.0.

Parameters

left [orderable scalar] Left bound for the interval.

right [orderable scalar] Right bound for the interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the interval is closed on the left-side, right-side, both or neither.

closed [{'right', 'left', 'both', 'neither'}, default 'right'] Whether the interval is closed on the left-side, right-side, both or neither. See the Notes for more detailed explanation.

See also:

IntervalIndex An Index of Interval objects that are all closed on the same side.

cut Convert continuous data into discrete bins (Categorical of Interval objects).

qcut Convert continuous data into bins (Categorical of Interval objects) based on quantiles.

Period Represents a period of time.

Notes

The parameters *left* and *right* must be from the same type, you must be able to compare them and they must satisfy `left <= right`.

A closed interval (in mathematics denoted by square brackets) contains its endpoints, i.e. the closed interval `[0, 5]` is characterized by the conditions `0 <= x <= 5`. This is what `closed='both'` stands for. An open interval (in mathematics denoted by parentheses) does not contain its endpoints, i.e. the open interval `(0, 5)` is characterized by the conditions `0 < x < 5`. This is what `closed='neither'` stands for. Intervals can also be half-open or half-closed, i.e. `[0, 5)` is described by `0 <= x < 5` (`closed='left'`) and `(0, 5]` is described by `0 < x <= 5` (`closed='right'`).

Examples

It is possible to build Intervals of different types, like numeric ones:

```
>>> iv = pd.Interval(left=0, right=5)
>>> iv
Interval(0, 5, closed='right')
```

You can check if an element belongs to it

```
>>> 2.5 in iv
True
```

You can test the bounds (`closed='right'`, so `0 < x <= 5`):

```
>>> 0 in iv
False
>>> 5 in iv
True
>>> 0.0001 in iv
True
```

Calculate its length

```
>>> iv.length
5
```

You can operate with + and * over an Interval and the operation is applied to each of its bounds, so the result depends on the type of the bound elements

```
>>> shifted_iv = iv + 3
>>> shifted_iv
Interval(3, 8, closed='right')
>>> extended_iv = iv * 10.0
>>> extended_iv
Interval(0.0, 50.0, closed='right')
```

To create a time interval you can use Timestamps as the bounds

```
>>> year_2017 = pd.Interval(pd.Timestamp('2017-01-01 00:00:00'),
...                         pd.Timestamp('2018-01-01 00:00:00'),
...                         closed='left')
>>> pd.Timestamp('2017-01-01 00:00') in year_2017
True
>>> year_2017.length
Timedelta('365 days 00:00:00')
```

And also you can create string intervals

```
>>> volume_1 = pd.Interval('Ant', 'Dog', closed='both')
>>> 'Bee' in volume_1
True
```

Attributes

closed	Whether the interval is closed on the left-side, right-side, both or neither
closed_left	Check if the interval is closed on the left side.
closed_right	Check if the interval is closed on the right side.
left	Left bound for the interval
length	Return the length of the Interval
mid	Return the midpoint of the Interval
open_left	Check if the interval is open on the left side.
open_right	Check if the interval is open on the right side.
right	Right bound for the interval

pandas.Interval.closed

`Interval.closed`
Whether the interval is closed on the left-side, right-side, both or neither

pandas.Interval.closed_left

`Interval.closed_left`
Check if the interval is closed on the left side.

For the meaning of *closed* and *open* see `Interval`.

Returns

bool `True` if the Interval is closed on the left-side, else `False`.

pandas.Interval.closed_right

`Interval.closed_right`
Check if the interval is closed on the right side.

For the meaning of *closed* and *open* see `Interval`.

Returns

bool `True` if the Interval is closed on the left-side, else `False`.

pandas.Interval.left

`Interval.left`
Left bound for the interval

pandas.Interval.length

`Interval.length`
Return the length of the Interval

pandas.Interval.mid

`Interval.mid`
Return the midpoint of the Interval

pandas.Interval.open_left

`Interval.open_left`
Check if the interval is open on the left side.

For the meaning of *closed* and *open* see `Interval`.

Returns

bool `True` if the Interval is closed on the left-side, else `False`.

pandas.Interval.open_right

`Interval.open_right`
Check if the interval is open on the right side.

For the meaning of *closed* and *open* see `Interval`.

Returns

bool `True` if the Interval is closed on the left-side, else `False`.

pandas.Interval.right

`Interval.right`
Right bound for the interval

Methods

overlaps	Check whether two Interval objects overlap.

pandas.Interval.overlaps

`Interval.overlaps()`
Check whether two Interval objects overlap.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

New in version 0.24.0.

Parameters

other [Interval] The interval to check against for an overlap.

Returns

bool `True` if the two intervals overlap, else `False`.

See also:

IntervalArray.overlaps The corresponding method for IntervalArray.

IntervalIndex.overlaps The corresponding method for IntervalIndex.

Examples

```
>>> i1 = pd.Interval(0, 2)
>>> i2 = pd.Interval(1, 3)
>>> i1.overlaps(i2)
True
>>> i3 = pd.Interval(4, 5)
>>> i1.overlaps(i3)
False
```

Intervals that share closed endpoints overlap:

```
>>> i4 = pd.Interval(0, 1, closed='both')
>>> i5 = pd.Interval(1, 2, closed='both')
>>> i4.overlaps(i5)
True
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> i6 = pd.Interval(1, 2, closed='neither')
>>> i4.overlaps(i6)
False
```

Properties

Interval.closed	Whether the interval is closed on the left-side, right-side, both or neither
Interval.closed_left	Check if the interval is closed on the left side.
Interval.closed_right	Check if the interval is closed on the right side.
Interval.left	Left bound for the interval
Interval.length	Return the length of the Interval
Interval.mid	Return the midpoint of the Interval
Interval.open_left	Check if the interval is open on the left side.
Interval.open_right	Check if the interval is open on the right side.
Interval.overlaps	Check whether two Interval objects overlap.
Interval.right	Right bound for the interval

A collection of intervals may be stored in an *arrays.IntervalArray*.

arrays.IntervalArray	Pandas array for interval data that are closed on the same side.
IntervalDtype	A Interval duck-typed class, suitable for holding an interval

pandas.arrays.IntervalArray

class pandas.arrays.**IntervalArray**

Pandas array for interval data that are closed on the same side.

New in version 0.24.0.

> **Warning:** The indexing behaviors are provisional and may change in a future version of pandas.

Parameters

data [array-like (1-dimensional)] Array-like containing Interval objects from which to build the IntervalArray.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

dtype [dtype or None, default None] If None, dtype will be inferred.

New in version 0.23.0.

copy [bool, default False] Copy the input data.

verify_integrity [bool, default True] Verify that the IntervalArray is valid.

See also:

Index The base pandas Index type.

Interval A bounded slice-like interval; the elements of an IntervalArray.

interval_range Function to create a fixed frequency IntervalIndex.

cut Bin values into discrete Intervals.

qcut Bin values into equal-sized Intervals based on rank or sample quantiles.

Notes

See the user guide for more.

Examples

A new `IntervalArray` can be constructed directly from an array-like of `Interval` objects:

```
>>> pd.arrays.IntervalArray([pd.Interval(0, 1), pd.Interval(1, 5)])
IntervalArray([(0, 1], (1, 5]],
              closed='right',
              dtype='interval[int64]')
```

It may also be constructed using one of the constructor methods: *IntervalArray.from_arrays()*, *IntervalArray.from_breaks()*, and *IntervalArray.from_tuples()*.

Attributes

left	Return the left endpoints of each Interval in the IntervalArray as an Index
right	Return the right endpoints of each Interval in the IntervalArray as an Index
closed	Whether the intervals are closed on the left-side, right-side, both or neither
mid	Return the midpoint of each Interval in the IntervalArray as an Index
length	Return an Index with entries denoting the length of each Interval in the IntervalArray
is_non_overlapping_monotonic	Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False

pandas.arrays.IntervalArray.left

IntervalArray.**left**

Return the left endpoints of each Interval in the IntervalArray as an Index

pandas.arrays.IntervalArray.right

`IntervalArray.right`
Return the right endpoints of each Interval in the IntervalArray as an Index

pandas.arrays.IntervalArray.closed

`IntervalArray.closed`
Whether the intervals are closed on the left-side, right-side, both or neither

pandas.arrays.IntervalArray.mid

`IntervalArray.mid`
Return the midpoint of each Interval in the IntervalArray as an Index

pandas.arrays.IntervalArray.length

`IntervalArray.length`
Return an Index with entries denoting the length of each Interval in the IntervalArray

pandas.arrays.IntervalArray.is_non_overlapping_monotonic

`IntervalArray.is_non_overlapping_monotonic`
Return True if the IntervalArray is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False

Methods

from_arrays(left, right[, closed, copy, dtype])	Construct from two arrays defining the left and right bounds.
from_tuples(data[, closed, copy, dtype])	Construct an IntervalArray from an array-like of tuples
from_breaks(breaks[, closed, copy, dtype])	Construct an IntervalArray from an array of splits.
overlaps(other)	Check elementwise if an Interval overlaps the values in the IntervalArray.
set_closed(closed)	Return an IntervalArray identical to the current one, but closed on the specified side
to_tuples([na_tuple])	Return an ndarray of tuples of the form (left, right)

pandas.arrays.IntervalArray.from_arrays

classmethod `IntervalArray.from_arrays`(*left, right, closed='right', copy=False, dtype=None*)
Construct from two arrays defining the left and right bounds.

Parameters

left [array-like (1-dimensional)] Left bounds for each interval.

right [array-like (1-dimensional)] Right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] Copy the data.

dtype [dtype, optional] If None, dtype will be inferred.

New in version 0.23.0.

Returns

IntervalArray

Raises

ValueError When a value is missing in only one of *left* or *right*. When a value in *left* is greater than the corresponding value in *right*.

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_breaks Construct an IntervalArray from an array of splits.

IntervalArray.from_tuples Construct an IntervalArray from an array-like of tuples.

Notes

Each element of *left* must be less than or equal to the *right* element at the same position. If an element is missing, it must be missing in both *left* and *right*. A TypeError is raised when using an unsupported type for *left* or *right*. At the moment, 'category', 'object', and 'string' subtypes are not supported.

Examples

```
>>> IntervalArray.from_arrays([0, 1, 2], [1, 2, 3])
IntervalArray([(0, 1], (1, 2], (2, 3]],
              closed='right',
              dtype='interval[int64]')
```

pandas.arrays.IntervalArray.from_tuples

classmethod IntervalArray.**from_tuples**(*data*, *closed='right'*, *copy=False*, *dtype=None*)

Construct an IntervalArray from an array-like of tuples

Parameters

data [array-like (1-dimensional)] Array of tuples

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] by-default copy the data, this is compat only and ignored

dtype [dtype or None, default None] If None, dtype will be inferred

..versionadded:: 0.23.0

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_arrays Construct an IntervalArray from a left and right array.

IntervalArray.from_breaks Construct an IntervalArray from an array of splits.

Examples

```
>>> pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 2)])
IntervalArray([(0, 1], (1, 2]],
              closed='right', dtype='interval[int64]')
```

pandas.arrays.IntervalArray.from_breaks

classmethod IntervalArray.**from_breaks**(*breaks*, *closed='right'*, *copy=False*, *dtype=None*)

Construct an IntervalArray from an array of splits.

Parameters

breaks [array-like (1-dimensional)] Left and right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] copy the data

dtype [dtype or None, default None] If None, dtype will be inferred

New in version 0.23.0.

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalArray.from_arrays Construct from a left and right array.

IntervalArray.from_tuples Construct from a sequence of tuples.

Examples

```
>>> pd.arrays.IntervalArray.from_breaks([0, 1, 2, 3])
IntervalArray([(0, 1], (1, 2], (2, 3]],
                closed='right',
                dtype='interval[int64]')
```

pandas.arrays.IntervalArray.overlaps

IntervalArray.**overlaps**(*other*)

Check elementwise if an Interval overlaps the values in the IntervalArray.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

New in version 0.24.0.

Parameters

other [Interval] Interval to check against for an overlap.

Returns

ndarray Boolean array positionally indicating where an overlap occurs.

See also:

Interval.overlaps Check whether two Interval objects overlap.

Examples

```
>>> intervals = pd.arrays.IntervalArray.from_tuples([(0, 1), (1, 3), (2, 4)])
>>> intervals
IntervalArray([(0, 1], (1, 3], (2, 4]],
      closed='right',
      dtype='interval[int64]')
>>> intervals.overlaps(pd.Interval(0.5, 1.5))
array([ True,  True, False])
```

Intervals that share closed endpoints overlap:

```
>>> intervals.overlaps(pd.Interval(1, 3, closed='left'))
array([ True,  True, True])
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> intervals.overlaps(pd.Interval(1, 2, closed='right'))
array([False,  True, False])
```

pandas.arrays.IntervalArray.set_closed

IntervalArray.**set_closed**(*closed*)

Return an IntervalArray identical to the current one, but closed on the specified side

New in version 0.24.0.

Parameters

closed [{'left', 'right', 'both', 'neither'}] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

new_index [IntervalArray]

Examples

```
>>> index = pd.interval_range(0, 3)
>>> index
IntervalIndex([(0, 1], (1, 2], (2, 3]],
      closed='right',
```

(continues on next page)

(continued from previous page)

```
      dtype='interval[int64]')
>>> index.set_closed('both')
IntervalIndex([[0, 1], [1, 2], [2, 3]],
      closed='both',
      dtype='interval[int64]')
```

pandas.arrays.IntervalArray.to_tuples

`IntervalArray.to_tuples(na_tuple=True)`

Return an ndarray of tuples of the form (left, right)

Parameters

na_tuple [boolean, default True] Returns NA as a tuple if True, `(nan, nan)`, or just as the NA value itself if False, `nan`.

New in version 0.23.0.

Returns

tuples: ndarray

pandas.IntervalDtype

class `pandas.IntervalDtype`

A Interval duck-typed class, suitable for holding an interval

THIS IS NOT A REAL NUMPY DTYPE

Attributes

names	Ordered list of field names, or None if there are no fields.
type	The scalar type for the array, e.g.

pandas.IntervalDtype.names

`IntervalDtype.names`

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.IntervalDtype.type

`IntervalDtype.type`

The scalar type for the array, e.g. `int`

It's expected `ExtensionArray[item]` returns an instance of `ExtensionDtype.type` for scalar `item`, assuming that value is valid (not NA). NA values do not need to be instances of *type*.

kind	
subdtype	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	attempt to construct this type from a string, raise a TypeError if its not possible
is_dtype(dtype)	Return a boolean if we if the passed type is an actual dtype that we can match (via string or type)
reset_cache()	clear the cache

pandas.IntervalDtype.construct_array_type

classmethod IntervalDtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.IntervalDtype.construct_from_string

classmethod IntervalDtype.**construct_from_string**(*string*)

attempt to construct this type from a string, raise a TypeError if its not possible

pandas.IntervalDtype.is_dtype

classmethod IntervalDtype.**is_dtype**(*dtype*)

Return a boolean if we if the passed type is an actual dtype that we can match (via string or type)

pandas.IntervalDtype.reset_cache

classmethod IntervalDtype.**reset_cache**()

clear the cache

6.5.7 Nullable Integer

`numpy.ndarray` cannot natively represent integer-data with missing values. Pandas provides this through *arrays.IntegerArray*.

arrays.IntegerArray(values, mask[, copy])	Array of integer (optional missing) values.
Int8Dtype	

Attributes

Continued on next page

Table 111 – continued from previous page

Int16Dtype **Attributes**
Int32Dtype **Attributes**
Int64Dtype **Attributes**
UInt8Dtype **Attributes**
UInt16Dtype **Attributes**
UInt32Dtype **Attributes**
UInt64Dtype **Attributes**

pandas.arrays.IntegerArray

class pandas.arrays.**IntegerArray**(*values*, *mask*, *copy=False*)

Array of integer (optional missing) values.

New in version 0.24.0.

> **Warning:** IntegerArray is currently experimental, and its API or internal implementation may change without warning.

We represent an IntegerArray with 2 numpy arrays:

- data: contains a numpy integer array of the appropriate dtype
- mask: a boolean array holding a mask on the data, True is missing

To construct an IntegerArray from generic array-like input, use *pandas.array()* with one of the integer dtypes (see examples).

See *Nullable Integer Data Type* for more.

Parameters

values [numpy.ndarray] A 1-d integer-dtype array.

mask [numpy.ndarray] A 1-d boolean-dtype array indicating missing values.

copy [bool, default False] Whether to copy the *values* and *mask*.

Returns

IntegerArray

Examples

Create an IntegerArray with `pandas.array()`.

```
>>> int_array = pd.array([1, None, 3], dtype=pd.Int32Dtype())
>>> int_array
<IntegerArray>
[1, NaN, 3]
Length: 3, dtype: Int32
```

String aliases for the dtypes are also available. They are capitalized.

```
>>> pd.array([1, None, 3], dtype='Int32')
<IntegerArray>
[1, NaN, 3]
Length: 3, dtype: Int32
```

```
>>> pd.array([1, None, 3], dtype='UInt16')
<IntegerArray>
[1, NaN, 3]
Length: 3, dtype: UInt16
```

Attributes

nbytes	The number of bytes needed to store this object in memory.
ndim	Extension Arrays are only allowed to be 1-dimensional.
shape	Return a tuple of the array dimensions.

pandas.arrays.IntegerArray.nbytes

`IntegerArray.`**`nbytes`**
The number of bytes needed to store this object in memory.

pandas.arrays.IntegerArray.ndim

`IntegerArray.`**`ndim`**
Extension Arrays are only allowed to be 1-dimensional.

pandas.arrays.IntegerArray.shape

IntegerArray.**shape**

Return a tuple of the array dimensions.

dtype	

Methods

argsort([ascending, kind])	Return the indices that would sort this array.
astype(dtype[, copy])	Cast to a NumPy array or IntegerArray with 'dtype'.
copy([deep])	Return a copy of the array.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill NA/NaN values using the specified method.
isna()	A 1-D array indicating if each value is missing.
repeat(repeats[, axis])	Repeat elements of a ExtensionArray.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
take(indexer[, allow_fill, fill_value])	Take elements from an array.
unique()	Compute the ExtensionArray of unique values.
value_counts([dropna])	Returns a Series containing counts of each category.

pandas.arrays.IntegerArray.argsort

IntegerArray.**argsort**(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)

Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

numpy.argsort Sorting implementation used internally.

pandas.arrays.IntegerArray.astype

IntegerArray.**astype**(*dtype*, *copy=True*)

Cast to a NumPy array or IntegerArray with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray or IntegerArray] NumPy ndarray or IntergerArray with 'dtype' for its dtype.

Raises

TypeError if incompatible type with an IntegerDtype, equivalent of same_kind casting

pandas.arrays.IntegerArray.copy

`IntegerArray.copy(deep=False)`

Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.arrays.IntegerArray.dropna

`IntegerArray.dropna()`

Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.arrays.IntegerArray.factorize

`IntegerArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

Note: uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

`pandas.factorize()` offers a *sort* keyword as well.

pandas.arrays.IntegerArray.fillna

`IntegerArray.`**`fillna`**`(`*value=None*, *method=None*, *limit=None*`)`
Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

filled [ExtensionArray with NA/NaN filled]

pandas.arrays.IntegerArray.isna

`IntegerArray.`**`isna`**`()`
A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.arrays.IntegerArray.repeat

`IntegerArray.`**`repeat`**`(`*repeats*, *axis=None*`)`
Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_array [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

`Series.repeat` Equivalent function for Series.

`Index.repeat` Equivalent function for Index.

`numpy.repeat` Similar method for `numpy.ndarray`.

`ExtensionArray.take` Take arbitrary positions.

Examples

```
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

pandas.arrays.IntegerArray.searchsorted

`IntegerArray.`**`searchsorted`**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array *self* (a) such that, if the corresponding elements in *v* were inserted before the indices, the order of *self* would be preserved.

Assuming that *a* is sorted:

side	returned index *i* satisfies
left	`self[i-1] < v <= self[i]`
right	`self[i-1] <= v < self[i]`

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

See also:

`numpy.searchsorted` Similar method from NumPy.

pandas.arrays.IntegerArray.shift

`IntegerArray.shift(periods=1, fill_value=None)`

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.arrays.IntegerArray.take

`IntegerArray.take(indexer, allow_fill=False, fill_value=None)`

Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.

- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.arrays.IntegerArray.unique

`IntegerArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

pandas.arrays.IntegerArray.value_counts

`IntegerArray.value_counts(dropna=True)`

Returns a Series containing counts of each category.

Every category will have an entry, even those with a count of 0.

Parameters

dropna [boolean, default True] Don't include counts of NaN.

Returns

counts [Series]

See also:

`Series.value_counts`

pandas.Int8Dtype

`class pandas.Int8Dtype`

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.Int8Dtype.itemsize

`Int8Dtype.itemsize`

Return the number of bytes in this dtype

pandas.Int8Dtype.names

`Int8Dtype.names`

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.Int8Dtype.numpy_dtype

`Int8Dtype.numpy_dtype`
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of numpy.int8

pandas.Int8Dtype.construct_array_type

classmethod `Int8Dtype.construct_array_type()`
Return the array type associated with this dtype

Returns

type

pandas.Int8Dtype.construct_from_string

classmethod `Int8Dtype.construct_from_string(string)`
Construction from a string, raise a TypeError if not possible

pandas.Int8Dtype.is_dtype

classmethod `Int8Dtype.is_dtype(dtype)`
Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`

3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.Int16Dtype

class pandas.**Int16Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.Int16Dtype.itemsize

Int16Dtype.**itemsize**
Return the number of bytes in this dtype

pandas.Int16Dtype.names

Int16Dtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.Int16Dtype.numpy_dtype

Int16Dtype.**numpy_dtype**
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of `numpy.int16`

pandas.Int16Dtype.construct_array_type

classmethod Int16Dtype.**construct_array_type**()
Return the array type associated with this dtype

Returns

type

pandas.Int16Dtype.construct_from_string

classmethod Int16Dtype.**construct_from_string**(*string*)

Construction from a string, raise a TypeError if not possible

pandas.Int16Dtype.is_dtype

classmethod Int16Dtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.Int32Dtype

class pandas.**Int32Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.Int32Dtype.itemsize

Int32Dtype.**itemsize**

Return the number of bytes in this dtype

pandas.Int32Dtype.names

`Int32Dtype.names`

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.Int32Dtype.numpy_dtype

`Int32Dtype.numpy_dtype`

Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of `numpy.int32`

pandas.Int32Dtype.construct_array_type

classmethod `Int32Dtype.construct_array_type()`

Return the array type associated with this dtype

Returns

type

pandas.Int32Dtype.construct_from_string

classmethod `Int32Dtype.construct_from_string`(*string*)

Construction from a string, raise a TypeError if not possible

pandas.Int32Dtype.is_dtype

classmethod `Int32Dtype.is_dtype`(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.Int64Dtype

class pandas.**Int64Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.Int64Dtype.itemsize

Int64Dtype.**itemsize**
Return the number of bytes in this dtype

pandas.Int64Dtype.names

Int64Dtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.Int64Dtype.numpy_dtype

Int64Dtype.**numpy_dtype**
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible

Continued on next page

Table 121 – continued from previous page

is_dtype(dtype)	Check if we match 'dtype'.
type	alias of numpy.int64

pandas.Int64Dtype.construct_array_type

classmethod Int64Dtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.Int64Dtype.construct_from_string

classmethod Int64Dtype.**construct_from_string**(*string*)

Construction from a string, raise a TypeError if not possible

pandas.Int64Dtype.is_dtype

classmethod Int64Dtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.UInt8Dtype

class pandas.**UInt8Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.UInt8Dtype.itemsize

`UInt8Dtype.`**`itemsize`**
Return the number of bytes in this dtype

pandas.UInt8Dtype.names

`UInt8Dtype.`**`names`**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.UInt8Dtype.numpy_dtype

`UInt8Dtype.`**`numpy_dtype`**
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
`type`	alias of `numpy.uint8`

pandas.UInt8Dtype.construct_array_type

`classmethod` `UInt8Dtype.`**`construct_array_type`**`()`
Return the array type associated with this dtype

Returns

type

pandas.UInt8Dtype.construct_from_string

`classmethod` `UInt8Dtype.`**`construct_from_string`**`(`*string*`)`
Construction from a string, raise a TypeError if not possible

pandas.UInt8Dtype.is_dtype

`classmethod` `UInt8Dtype.`**`is_dtype`**`(`*dtype*`)`
Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.UInt16Dtype

class pandas.**UInt16Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.UInt16Dtype.itemsize

UInt16Dtype.**itemsize**
Return the number of bytes in this dtype

pandas.UInt16Dtype.names

UInt16Dtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.UInt16Dtype.numpy_dtype

UInt16Dtype.**numpy_dtype**
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of numpy.uint16

pandas.UInt16Dtype.construct_array_type

classmethod UInt16Dtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.UInt16Dtype.construct_from_string

classmethod UInt16Dtype.**construct_from_string**(*string*)

Construction from a string, raise a TypeError if not possible

pandas.UInt16Dtype.is_dtype

classmethod UInt16Dtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.UInt32Dtype

class pandas.**UInt32Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.UInt32Dtype.itemsize

UInt32Dtype.**itemsize**
Return the number of bytes in this dtype

pandas.UInt32Dtype.names

UInt32Dtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.UInt32Dtype.numpy_dtype

UInt32Dtype.**numpy_dtype**
Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of `numpy.uint32`

pandas.UInt32Dtype.construct_array_type

classmethod UInt32Dtype.**construct_array_type**()
Return the array type associated with this dtype

Returns

type

pandas.UInt32Dtype.construct_from_string

classmethod UInt32Dtype.**construct_from_string**(*string*)
Construction from a string, raise a TypeError if not possible

pandas.UInt32Dtype.is_dtype

classmethod UInt32Dtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.UInt64Dtype

class pandas.**UInt64Dtype**

Attributes

itemsize	Return the number of bytes in this dtype
names	Ordered list of field names, or None if there are no fields.
numpy_dtype	Return an instance of our numpy dtype

pandas.UInt64Dtype.itemsize

UInt64Dtype.**itemsize**

Return the number of bytes in this dtype

pandas.UInt64Dtype.names

UInt64Dtype.**names**

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.UInt64Dtype.numpy_dtype

UInt64Dtype.**numpy_dtype**

Return an instance of our numpy dtype

base	
is_signed_integer	
is_unsigned_integer	
kind	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construction from a string, raise a TypeError if not possible
is_dtype(dtype)	Check if we match 'dtype'.
type	alias of numpy.uint64

pandas.UInt64Dtype.construct_array_type

classmethod UInt64Dtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.UInt64Dtype.construct_from_string

classmethod UInt64Dtype.**construct_from_string**(*string*)

Construction from a string, raise a TypeError if not possible

pandas.UInt64Dtype.is_dtype

classmethod UInt64Dtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

6.5.8 Categorical Data

Pandas defines a custom data type for representing data that can take only a limited, fixed set of values. The dtype of a `Categorical` can be described by a `pandas.api.types.CategoricalDtype`.

CategoricalDtype([categories, ordered])	Type for categorical data with the categories and orderedness

pandas.CategoricalDtype

class pandas.**CategoricalDtype**(*categories=None*, *ordered=None*)

Type for categorical data with the categories and orderedness

Changed in version 0.21.0.

Parameters

categories [sequence, optional] Must be unique, and must not contain any nulls.

ordered [bool, default False]

See also:

pandas.Categorical

Notes

This class is useful for specifying the type of a `Categorical` independent of the values. See *CategoricalDtype* for more.

Examples

```
>>> t = pd.CategoricalDtype(categories=['b', 'a'], ordered=True)
>>> pd.Series(['a', 'b', 'a', 'c'], dtype=t)
0      a
1      b
2      a
3    NaN
dtype: category
Categories (2, object): [b < a]
```

Attributes

categories	An `Index` containing the unique categories allowed.
ordered	Whether the categories have an ordered relationship.

pandas.CategoricalDtype.categories

CategoricalDtype.**categories**

An `Index` containing the unique categories allowed.

pandas.CategoricalDtype.ordered

`CategoricalDtype.ordered`
Whether the categories have an ordered relationship.

Methods

None	

CategoricalDtype.categories	An `Index` containing the unique categories allowed.
CategoricalDtype.ordered	Whether the categories have an ordered relationship.

Categorical data can be stored in a *pandas.Categorical*

Categorical(values[, categories, ordered, ...])	Represents a categorical variable in classic R / S-plus fashion

pandas.Categorical

class pandas.**Categorical**(*values*, *categories=None*, *ordered=None*, *dtype=None*, *fastpath=False*)
Represents a categorical variable in classic R / S-plus fashion

Categoricals can only take on only a limited, and usually fixed, number of possible values (*categories*). In contrast to statistical categorical variables, a *Categorical* might have an order, but numerical operations (additions, divisions, ...) are not possible.

All values of the *Categorical* are either in *categories* or *np.nan*. Assigning values outside of *categories* will raise a *ValueError*. Order is defined by the order of the *categories*, not lexical order of the values.

Parameters

values [list-like] The values of the categorical. If categories are given, values not in categories will be replaced with NaN.

categories [Index-like (unique), optional] The unique categories for this categorical. If not given, the categories are assumed to be the unique values of *values* (sorted, if possible, otherwise in the order in which they appear).

ordered [boolean, (default False)] Whether or not this categorical is treated as a ordered categorical. If True, the resulting categorical will be ordered. An ordered categorical respects, when sorted, the order of its *categories* attribute (which in turn is the *categories* argument, if provided).

dtype [CategoricalDtype] An instance of `CategoricalDtype` to use for this categorical

New in version 0.21.0.

Raises

ValueError If the categories do not validate.

TypeError If an explicit `ordered=True` is given but no *categories* and the *values* are not sortable.

See also:

pandas.api.types.CategoricalDtype Type for categorical data.

CategoricalIndex An Index with an underlying `Categorical`.

Notes

See the user guide for more.

Examples

```
>>> pd.Categorical([1, 2, 3, 1, 2, 3])
[1, 2, 3, 1, 2, 3]
Categories (3, int64): [1, 2, 3]
```

```
>>> pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
[a, b, c, a, b, c]
Categories (3, object): [a, b, c]
```

Ordered *Categoricals* can be sorted according to the custom order of the categories and can have a min and max value.

```
>>> c = pd.Categorical(['a','b','c','a','b','c'], ordered=True,
...                    categories=['c', 'b', 'a'])
>>> c
[a, b, c, a, b, c]
Categories (3, object): [c < b < a]
>>> c.min()
'c'
```

Attributes

categories	The categories of this categorical.
codes	The category codes of this categorical.
ordered	Whether the categories have an ordered relationship.
dtype	The `CategoricalDtype` for this instance

pandas.Categorical.categories

`Categorical.`**categories**

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to *categories* is a inplace operation!

Raises

ValueError If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

`rename_categories`, `reorder_categories`, `add_categories`, `remove_categories`, `remove_unused_categories`, `set_categories`

pandas.Categorical.codes

`Categorical.codes`
The category codes of this categorical.

Level codes are an array if integer which are the positions of the real values in the categories array.

There is not setter, use the other categorical methods and the normal item setter to change values in the categorical.

pandas.Categorical.ordered

`Categorical.ordered`
Whether the categories have an ordered relationship.

pandas.Categorical.dtype

`Categorical.dtype`
The `CategoricalDtype` for this instance

Methods

from_codes(codes[, categories, ordered, dtype])	Make a Categorical type from codes and categories or dtype.
__array__([dtype])	The numpy array interface.

pandas.Categorical.from_codes

classmethod `Categorical.from_codes`(*codes*, *categories=None*, *ordered=None*, *dtype=None*)
Make a Categorical type from codes and categories or dtype.

This constructor is useful if you already have codes and categories/dtype and so do not need the (computation intensive) factorization step, which is usually done on the constructor.

If your data does not follow this convention, please use the normal constructor.

Parameters

codes [array-like, integers] An integer array, where each integer points to a category in categories or dtype.categories, or else is -1 for NaN

categories [index-like, optional] The categories for the categorical. Items need to be unique. If the categories are not given here, then they must be provided in *dtype*.

ordered [bool, optional] Whether or not this categorical is treated as an ordered categorical. If not given here or in *dtype*, the resulting categorical will be unordered.

dtype [CategoricalDtype or the string "category", optional] If `CategoricalDtype`, cannot be used together with *categories* or *ordered*.

New in version 0.24.0: When *dtype* is provided, neither *categories* nor *ordered* should be provided.

Examples

```
>>> dtype = pd.CategoricalDtype(['a', 'b'], ordered=True)
>>> pd.Categorical.from_codes(codes=[0, 1, 0, 1], dtype=dtype)
[a, b, a, b]
Categories (2, object): [a < b]
```

pandas.Categorical.__array__

`Categorical.__array__`(*dtype=None*)

The numpy array interface.

Returns

values [numpy array] A numpy array of either the specified dtype or, if dtype==None (default), the same dtype as categorical.categories.dtype

The alternative *Categorical.from_codes()* constructor can be used when you have the categories and integer codes already:

Categorical.from_codes(codes[, categories, ...])	Make a Categorical type from codes and categories or dtype.

The dtype information is available on the `Categorical`

Categorical.dtype	The `CategoricalDtype` for this instance
Categorical.categories	The categories of this categorical.
Categorical.ordered	Whether the categories have an ordered relationship.
Categorical.codes	The category codes of this categorical.

`np.asarray(categorical)` works by implementing the array interface. Be aware, that this converts the Categorical back to a NumPy array, so categories and order information is not preserved!

Categorical.__array__([dtype])	The numpy array interface.

A `Categorical` can be stored in a `Series` or `DataFrame`. To create a Series of dtype `category`, use `cat = s.astype(dtype)` or `Series(..., dtype=dtype)` where `dtype` is either

- the string `'category'`
- an instance of `CategoricalDtype`.

If the Series is of dtype `CategoricalDtype`, `Series.cat` can be used to change the categorical data. See *Categorical Accessor* for more.

6.5.9 Sparse Data

Data where a single value is repeated many times (e.g. `0` or `NaN`) may be stored efficiently as a *SparseArray*.

SparseArray(data[, sparse_index, index, . . .])	An ExtensionArray for storing sparse data.
SparseDtype([dtype, fill_value])	Dtype for data stored in *SparseArray*.

pandas.SparseArray

class pandas.**SparseArray**(*data*, *sparse_index=None*, *index=None*, *fill_value=None*, *kind='integer'*, *dtype=None*, *copy=False*)

An ExtensionArray for storing sparse data.

Changed in version 0.24.0: Implements the ExtensionArray interface.

Parameters

data [array-like] A dense array of values to store in the SparseArray. This may contain *fill_value*.

sparse_index [SparseIndex, optional]

index [Index]

fill_value [scalar, optional] Elements in *data* that are *fill_value* are not stored in the SparseArray. For memory savings, this should be the most common value in *data*. By default, *fill_value* depends on the dtype of *data*:

data.dtype	na_value
float	`np.nan`
int	`0`
bool	False
datetime64	`pd.NaT`
timedelta64	`pd.NaT`

The fill value is potentiall specified in three ways. In order of precedence, these are

1. The *fill_value* argument
2. `dtype.fill_value` if *fill_value* is None and *dtype* is a `SparseDtype`
3. `data.dtype.fill_value` if *fill_value* is None and *dtype* is not a `SparseDtype` and *data* is a `SparseArray`.

kind [{'integer', 'block'}, default 'integer'] The type of storage for sparse locations.

- 'block': Stores a *block* and *block_length* for each contiguous *span* of sparse values. This is best when sparse data tends to be clumped together, with large regsions of `fill-value` values between sparse values.
- 'integer': uses an integer to store the location of each sparse value.

dtype [np.dtype or SparseDtype, optional] The dtype to use for the SparseArray. For numpy dtypes, this determines the dtype of `self.sp_values`. For SparseDtype, this determines `self.sp_values` and `self.fill_value`.

copy [bool, default False] Whether to explicitly copy the incoming *data* array.

Attributes

T	Returns the SparseArray.
density	The percent of non- `fill_value` points, as decimal.
dtype	An instance of 'ExtensionDtype'.
fill_value	Elements in *data* that are *fill_value* are not stored.
kind	The kind of sparse index for this array.
nbytes	The number of bytes needed to store this object in memory.
ndim	Extension Arrays are only allowed to be 1-dimensional.
npoints	The number of non- `fill_value` points.
shape	Return a tuple of the array dimensions.
sp_index	The SparseIndex containing the location of non-`fill_value` points.
sp_values	An ndarray containing the non- `fill_value` values.
values	Dense values

pandas.SparseArray.T

SparseArray.**T**
Returns the SparseArray.

pandas.SparseArray.density

SparseArray.**density**
The percent of non- `fill_value` points, as decimal.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.density
0.6
```

pandas.SparseArray.dtype

SparseArray.**dtype**
An instance of 'ExtensionDtype'.

pandas.SparseArray.fill_value

SparseArray.**fill_value**
Elements in *data* that are *fill_value* are not stored.

For memory savings, this should be the most common value in the array.

pandas.SparseArray.kind

`SparseArray.kind`
The kind of sparse index for this array. One of {'integer', 'block'}.

pandas.SparseArray.nbytes

`SparseArray.nbytes`
The number of bytes needed to store this object in memory.

pandas.SparseArray.ndim

`SparseArray.ndim`
Extension Arrays are only allowed to be 1-dimensional.

pandas.SparseArray.npoints

`SparseArray.npoints`
The number of non- `fill_value` points.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.npoints
3
```

pandas.SparseArray.shape

`SparseArray.shape`
Return a tuple of the array dimensions.

pandas.SparseArray.sp_index

`SparseArray.sp_index`
The SparseIndex containing the location of non- `fill_value` points.

pandas.SparseArray.sp_values

`SparseArray.sp_values`
An ndarray containing the non- `fill_value` values.

Examples

```
>>> s = SparseArray([0, 0, 1, 0, 2], fill_value=0)
>>> s.sp_values
array([1, 2])
```

pandas.SparseArray.values

SparseArray.**values**
Dense values

Methods

all([axis])	Tests whether all elements evaluate True
any([axis])	Tests whether at least one of elements evaluate True
argsort([ascending, kind])	Return the indices that would sort this array.
astype([dtype, copy])	Change the dtype of a SparseArray.
copy([deep])	Return a copy of the array.
cumsum([axis])	Cumulative sum of non-NA/null values.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill missing values with *value*.
get_values()	Convert SparseArray to a NumPy array.
isna()	A 1-D array indicating if each value is missing.
map(mapper)	Map categories using input correspondence (dict, Series, or function).
mean([axis])	Mean of non-NA/null values
repeat(repeats[, axis])	Repeat elements of a ExtensionArray.
searchsorted(v[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
sum([axis])	Sum of non-NA/null values
take(indices[, allow_fill, fill_value])	Take elements from an array.
to_dense()	Convert SparseArray to a NumPy array.
transpose(*axes)	Returns the SparseArray.
unique()	Compute the ExtensionArray of unique values.
value_counts([dropna])	Returns a Series containing counts of unique values.

pandas.SparseArray.all

SparseArray.**all**(*axis=None*, **args*, ***kwargs*)
Tests whether all elements evaluate True

Returns

all [bool]

See also:

numpy.all

pandas.SparseArray.any

SparseArray.**any**(*axis=0*, **args*, ***kwargs*)
Tests whether at least one of elements evaluate True

Returns

any [bool]

See also:

numpy.any

pandas.SparseArray.argsort

SparseArray.**argsort**(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)

Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

numpy.argsort Sorting implementation used internally.

pandas.SparseArray.astype

SparseArray.**astype**(*dtype=None*, *copy=True*)

Change the dtype of a SparseArray.

The output will always be a SparseArray. To convert to a dense ndarray with a certain dtype, use `numpy.asarray()`.

Parameters

dtype [np.dtype or ExtensionDtype] For SparseDtype, this changes the dtype of `self.sp_values` and the `self.fill_value`.

For other dtypes, this only changes the dtype of `self.sp_values`.

copy [bool, default True] Whether to ensure a copy is made, even if not necessary.

Returns

SparseArray

Examples

```
>>> arr = SparseArray([0, 0, 1, 2])
>>> arr
[0, 0, 1, 2]
Fill: 0
IntIndex
Indices: array([2, 3], dtype=int32)
```

```
>>> arr.astype(np.dtype('int32'))
[0, 0, 1, 2]
Fill: 0
IntIndex
Indices: array([2, 3], dtype=int32)
```

Using a NumPy dtype with a different kind (e.g. float) will coerce just `self.sp_values`.

```
>>> arr.astype(np.dtype('float64'))
... # doctest: +NORMALIZE_WHITESPACE
[0, 0, 1.0, 2.0]
Fill: 0
IntIndex
Indices: array([2, 3], dtype=int32)
```

Use a SparseDtype if you wish to be change the fill value as well.

```
>>> arr.astype(SparseDtype("float64", fill_value=np.nan))
... # doctest: +NORMALIZE_WHITESPACE
[nan, nan, 1.0, 2.0]
Fill: nan
IntIndex
Indices: array([2, 3], dtype=int32)
```

pandas.SparseArray.copy

`SparseArray.`**`copy`**(*deep=False*)

Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.SparseArray.cumsum

`SparseArray.`**`cumsum`**(*axis=0*, **args*, ***kwargs*)

Cumulative sum of non-NA/null values.

When performing the cumulative summation, any non-NA/null values will be skipped. The resulting SparseArray will preserve the locations of NaN values, but the fill value will be *np.nan* regardless.

Parameters

axis [int or None] Axis over which to perform the cumulative summation. If None, perform cumulative summation over flattened array.

Returns

cumsum [SparseArray]

pandas.SparseArray.dropna

`SparseArray.dropna()`

Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.SparseArray.factorize

`SparseArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

> **Note:** uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a *sort* keyword as well.

pandas.SparseArray.fillna

`SparseArray.fillna(value=None, method=None, limit=None)`

Fill missing values with *value*.

Parameters

value [scalar, optional]

method [str, optional]

> **Warning:** Using 'method' will result in high memory use, as all *fill_value* methods will be converted to an in-memory ndarray

limit [int, optional]

Returns

SparseArray

Notes

When *value* is specified, the result's `fill_value` depends on `self.fill_value`. The goal is to maintain low-memory use.

If `self.fill_value` is NA, the result dtype will be `SparseDtype(self.dtype, fill_value=value)`. This will preserve amount of memory used before and after filling.

When `self.fill_value` is not NA, the result dtype will be `self.dtype`. Again, this preserves the amount of memory used.

pandas.SparseArray.get_values

`SparseArray.`**`get_values`**`()`

Convert SparseArray to a NumPy array.

Returns

arr [NumPy array]

pandas.SparseArray.isna

`SparseArray.`**`isna`**`()`

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.SparseArray.map

`SparseArray.`**`map`**`(`*mapper*`)`

Map categories using input correspondence (dict, Series, or function).

Parameters

mapper [dict, Series, callable] The correspondence from old values to new.

Returns

SparseArray The output array will have the same density as the input. The output fill value will be the result of applying the mapping to `self.fill_value`

Examples

```
>>> arr = pd.SparseArray([0, 1, 2])
>>> arr.apply(lambda x: x + 10)
[10, 11, 12]
Fill: 10
IntIndex
Indices: array([1, 2], dtype=int32)
```

```
>>> arr.apply({0: 10, 1: 11, 2: 12})
[10, 11, 12]
Fill: 10
IntIndex
Indices: array([1, 2], dtype=int32)
```

```
>>> arr.apply(pd.Series([10, 11, 12], index=[0, 1, 2]))
[10, 11, 12]
Fill: 10
IntIndex
Indices: array([1, 2], dtype=int32)
```

pandas.SparseArray.mean

`SparseArray.mean(axis=0, *args, **kwargs)`

Mean of non-NA/null values

Returns

mean [float]

pandas.SparseArray.repeat

`SparseArray.repeat(repeats, axis=None)`

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_array [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

Series.repeat Equivalent function for Series.

Index.repeat Equivalent function for Index.

numpy.repeat Similar method for `numpy.ndarray`.

ExtensionArray.take Take arbitrary positions.

Examples

```
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

pandas.SparseArray.searchsorted

SparseArray.**searchsorted**(*v*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array *self* (a) such that, if the corresponding elements in *v* were inserted before the indices, the order of *self* would be preserved.

Assuming that *a* is sorted:

side	returned index *i* satisfies
left	`self[i-1] < v <= self[i]`
right	`self[i-1] <= v < self[i]`

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

See also:

numpy.searchsorted Similar method from NumPy.

pandas.SparseArray.shift

SparseArray.**shift**(*periods=1*, *fill_value=None*)

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.SparseArray.sum

`SparseArray.`**`sum`**`(`*axis=0*, **args*, ***kwargs*`)`

Sum of non-NA/null values

Returns

sum [float]

pandas.SparseArray.take

`SparseArray.`**`take`**`(`*indices*, *allow_fill=False*, *fill_value=None*`)`

Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.SparseArray.to_dense

`SparseArray.to_dense()`

Convert SparseArray to a NumPy array.

Returns

arr [NumPy array]

pandas.SparseArray.transpose

`SparseArray.transpose(*axes)`

Returns the SparseArray.

pandas.SparseArray.unique

`SparseArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

pandas.SparseArray.value_counts

`SparseArray.value_counts(dropna=True)`

Returns a Series containing counts of unique values.

Parameters

dropna [boolean, default True] Don't include counts of NaN, even if NaN is in sp_values.

Returns

counts [Series]

nonzero	

pandas.SparseDtype

class `pandas.SparseDtype(dtype=<class 'numpy.float64'>, fill_value=None)`

Dtype for data stored in `SparseArray`.

This dtype implements the pandas ExtensionDtype interface.

New in version 0.24.0.

Parameters

dtype [str, ExtensionDtype, numpy.dtype, type, default numpy.float64] The dtype of the underlying array storing the non-fill value values.

fill_value [scalar, optional] The scalar value not stored in the SparseArray. By default, this depends on *dtype*.

dtype	na_value
float	`np.nan`
int	`0`
bool	`False`
datetime64	`pd.NaT`
timedelta64	`pd.NaT`

The default value may be overridden by specifying a *fill_value*.

Attributes

fill_value	The fill value of the array.
kind	The sparse kind.

Continued on next page

Table 142 – continued from previous page

name	A string identifying the data type.
names	Ordered list of field names, or None if there are no fields.
type	The scalar type for the array, e.g.

pandas.SparseDtype.fill_value

`SparseDtype.`**`fill_value`**

The fill value of the array.

Converting the SparseArray to a dense ndarray will fill the array with this value.

> **Warning:** It's possible to end up with a SparseArray that has `fill_value` values in `sp_values`. This can occur, for example, when setting `SparseArray.fill_value` directly.

pandas.SparseDtype.kind

`SparseDtype.`**`kind`**

The sparse kind. Either 'integer', or 'block'.

pandas.SparseDtype.name

`SparseDtype.`**`name`**

A string identifying the data type.

Will be used for display in, e.g. `Series.dtype`

pandas.SparseDtype.names

`SparseDtype.`**`names`**

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.SparseDtype.type

`SparseDtype.`**`type`**

The scalar type for the array, e.g. `int`

It's expected `ExtensionArray[item]` returns an instance of `ExtensionDtype.type` for scalar `item`, assuming that value is valid (not NA). NA values do not need to be instances of *type*.

subtype	

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Construct a SparseDtype from a string form.
is_dtype(dtype)	Check if we match 'dtype'.
update_dtype(dtype)	Convert the SparseDtype to a new dtype.

pandas.SparseDtype.construct_array_type

classmethod SparseDtype.**construct_array_type**()

Return the array type associated with this dtype

Returns

type

pandas.SparseDtype.construct_from_string

classmethod SparseDtype.**construct_from_string**(*string*)

Construct a SparseDtype from a string form.

Parameters

string [str] Can take the following forms.

string dtype ================ ============================ 'int' SparseDtype[np.int64, 0] 'Sparse' SparseDtype[np.float64, nan] 'Sparse[int]' SparseDtype[np.int64, 0] 'Sparse[int, 0]' SparseDtype[np.int64, 0] ================ ============================

It is not possible to specify non-default fill values with a string. An argument like `'Sparse[int, 1]'` will raise a `TypeError` because the default fill value for integers is 0.

Returns

SparseDtype

pandas.SparseDtype.is_dtype

classmethod SparseDtype.**is_dtype**(*dtype*)

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`

3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

pandas.SparseDtype.update_dtype

SparseDtype.**update_dtype**(*dtype*)

Convert the SparseDtype to a new dtype.

This takes care of converting the `fill_value`.

Parameters

dtype [Union[str, numpy.dtype, SparseDtype]] The new dtype to use.

- For a SparseDtype, it is simply returned
- For a NumPy dtype (or str), the current fill value is converted to the new dtype, and a SparseDtype with *dtype* and the new fill value is returned.

Returns

SparseDtype A new SparseDtype with the corret *dtype* and fill value for that *dtype*.

Raises

ValueError When the current fill value cannot be converted to the new *dtype* (e.g. trying to convert `np.nan` to an integer dtype).

Examples

```
>>> SparseDtype(int, 0).update_dtype(float)
Sparse[float64, 0.0]
```

```
>>> SparseDtype(int, 1).update_dtype(SparseDtype(float, np.nan))
Sparse[float64, nan]
```

The `Series.sparse` accessor may be used to access sparse-specific attributes and methods if the *Series* contains sparse values. See *Sparse Accessor* for more.

6.6 Panel

6.6.1 Constructor

Panel([data, items, major_axis, minor_axis, ...])	(DEPRECATED) Represents wide format panel data, stored as 3-dimensional array.

pandas.Panel

class pandas.**Panel**(*data=None*, *items=None*, *major_axis=None*, *minor_axis=None*, *copy=False*, *dtype=None*)

Represents wide format panel data, stored as 3-dimensional array.

Deprecated since version 0.20.0: The recommended way to represent 3-D data are with a MultiIndex on a DataFrame via the *to_frame()* method or with the xarray package. Pandas provides a *to_xarray()* method to automate this conversion.

Parameters

data [ndarray (items x major x minor), or dict of DataFrames]

items [Index or array-like] axis=0

major_axis [Index or array-like] axis=1

minor_axis [Index or array-like] axis=2

copy [boolean, default False] Copy data from inputs. Only affects DataFrame / 2d ndarray input

dtype [dtype, default None] Data type to force, otherwise infer

Attributes

at	Access a single value for a row/column label pair.
axes	Return index label(s) of the internal NDFrame
blocks	(DEPRECATED) Internal property, property synonym for as_blocks().
dtypes	Return the dtypes in the DataFrame.
empty	Indicator whether DataFrame is empty.
ftypes	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
iat	Access a single value for a row/column pair by integer position.
iloc	Purely integer-location based indexing for selection by position.
is_copy	Return the copy.
items	
ix	A primarily label-location based indexer, with integer position fallback.
loc	Access a group of rows and columns by label(s) or a boolean array.
major_axis	
minor_axis	
ndim	Return an int representing the number of axes / array dimensions.
shape	Return a tuple of axis dimensions
size	Return an int representing the number of elements in this object.
values	Return a Numpy representation of the DataFrame.

pandas.Panel.at

Panel.**at**

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError When label does not exist in DataFrame

See also:

DataFrame.iat Access a single value for a row/column pair by integer position.

DataFrame.loc Access a group of rows and columns by label(s).

Series.at Access a single value using a label.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

pandas.Panel.axes

Panel.**axes**
Return index label(s) of the internal NDFrame

pandas.Panel.blocks

Panel.**blocks**
Internal property, property synonym for as_blocks().

Deprecated since version 0.21.0.

pandas.Panel.dtypes

Panel.**dtypes**
Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See *the User Guide* for more.

Returns

pandas.Series The data type of each column.

See also:

pandas.DataFrame.ftypes Dtype and sparsity information.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

pandas.Panel.empty

`Panel.empty`

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If DataFrame is empty, return True, if not return False.

See also:

pandas.Series.dropna, *pandas.DataFrame.dropna*

Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

pandas.Panel.ftypes

`Panel.ftypes`

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See *the User Guide* for more.

Returns

pandas.Series The data type and indication of sparse/dense of each column.

See also:

pandas.DataFrame.dtypes Series with just dtype information.

pandas.SparseDataFrame Container for sparse tabular data.

Notes

Sparse data should have the same dtypes as its dense representation.

Examples

```
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

pandas.Panel.iat

`Panel.iat`

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises

IndexError When integer position is out of bounds

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.loc Access a group of rows and columns by label(s).

DataFrame.iloc Access a group of rows and columns by integer position(s).

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

pandas.Panel.iloc

`Panel.`**`iloc`**

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.

- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at ref:*Selection by Position <indexing.integer>*.

See also:

DataFrame.iat Fast integer location scalar accessor.

DataFrame.loc Purely label-location based indexer for selection by label.

Series.iloc Purely integer-location based indexing for selection by position.

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a *slice* object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
```

(continues on next page)

(continued from previous page)

```
1   100   300
2  1000  3000
```

pandas.Panel.is_copy

Panel.**is_copy**

Return the copy.

pandas.Panel.items

Panel.**items**

pandas.Panel.ix

Panel.**ix**

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at *Advanced Indexing*.

pandas.Panel.loc

Panel.**loc**

Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at *Selection by Label*

Raises

KeyError: when any items are not found

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.iloc Access group of rows and columns by integer position(s).

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels.

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=['cobra', 'viper', 'sidewinder'],
...      columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
```

(continues on next page)

(continued from previous page)

```
viper                 0        0
sidewinder            0        0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...    ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...         [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
```

(continues on next page)

(continued from previous page)

```
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

pandas.Panel.major_axis

Panel.**major_axis**

pandas.Panel.minor_axis

Panel.**minor_axis**

pandas.Panel.ndim

Panel.**ndim**

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

See also:

ndarray.ndim Number of array dimensions.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

pandas.Panel.shape

Panel.**shape**

Return a tuple of axis dimensions

pandas.Panel.size

Panel.**size**

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

See also:

ndarray.size Number of elements in the array.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

pandas.Panel.values

Panel.**values**

Return a Numpy representation of the DataFrame.

> **Warning:** We recommend using *DataFrame.to_numpy()* instead.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

See also:

DataFrame.to_numpy Recommended alternative to this method.

pandas.DataFrame.index Retrieve the index labels.

pandas.DataFrame.columns Retrieving the column names.

Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By numpy.find_common_type() convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                   columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

timetuple	

Methods

abs()	Return a Series/DataFrame with absolute numeric value of each element.
add(other[, axis])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Prefix labels with string *prefix*.
add_suffix(suffix)	Suffix labels with string *suffix*.
align(other, **kwargs)	Align two objects on their axes with the specified join method for each axis Index.
all([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
any([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
apply(func[, axis])	Applies function along axis (or axes) of the Panel.
as_blocks([copy])	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix()	Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	Return the last row(s) without any NaNs before *where*.
astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
at_time(time[, asof, axis])	Select values at particular time of day (e.g.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for *DataFrame.fillna()* with `method='bfill'`.
bool()	Return the bool of a single element PandasObject.
clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.
clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.

Continued on next page

Table 146 – continued from previous page

compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis.
conform(frame[, axis])	Conform input DataFrame to align with chosen axis pair.
convert_objects([convert_dates, . . .])	(DEPRECATED) Attempt to infer better dtype for object columns.
copy([deep])	Make a copy of this object's indices and data.
count([axis])	Return number of observations over requested axis.
cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
describe([percentiles, include, exclude])	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
div(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divide(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
droplevel(level[, axis])	Return DataFrame with requested index / column level(s) removed.
dropna([axis, how, inplace])	Drop 2D from panel, holding passed axis constant.
eq(other[, axis])	Wrapper for comparison method eq
equals(other)	Test whether two objects contain the same elements.
ffill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna()` with `method='ffill'`.
fillna([value, method, axis, inplace, . . .])	Fill NA/NaN values using the specified method.
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return index for first non-NA/null value.
floordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
fromDict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects.
from_dict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects.
ge(other[, axis])	Wrapper for comparison method ge
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return counts of unique dtypes in this object.
get_ftype_counts()	(DEPRECATED) Return counts of unique ftypes in this object.
get_value(*args, **kwargs)	(DEPRECATED) Quickly retrieve single value at (item, major, minor) location.
get_values()	Return an ndarray after converting sparse values to dense.
groupby(function[, axis])	Group data on given axis, returning GroupBy object.

Continued on next page

Table 146 – continued from previous page

gt(other[, axis])	Wrapper for comparison method gt
head([n])	Return the first *n* rows.
infer_objects()	Attempt to infer better dtypes for object columns.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isna()	Detect missing values.
isnull()	Detect missing values.
iteritems()	Iterate over (label, values) on info axis
join(other[, how, lsuffix, rsuffix])	Join items with other Panel either on major and minor axes column.
keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return index for last non-NA/null value.
le(other[, axis])	Wrapper for comparison method le
lt(other[, axis])	Wrapper for comparison method lt
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis.
major_xs(key)	Return slice of panel along major axis.
mask(cond[, other, inplace, axis, level, ...])	Replace values where the condition is True.
max([axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis.
min([axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
minor_xs(key)	Return slice of panel along minor axis.
mod(other[, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mul(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
ne(other[, axis])	Wrapper for comparison method ne
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
pct_change([periods, fill_method, limit, freq])	Percentage change between the current and a prior element.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs).
pop(item)	Return item and drop from frame.
pow(other[, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis.

Continued on next page

Table 146 – continued from previous page

product([axis, skipna, level, numeric_only, ...])	Return the product of the values for the requested axis.
radd(other[, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
reindex(*args, **kwargs)	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	(DEPRECATED) Conform input object to new index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices as other object.
rename([items, major_axis, minor_axis])	Alter axes input function or functions.
rename_axis([mapper, index, columns, axis, ...])	Set the name of the axis for the index or columns.
replace([to_replace, value, inplace, limit, ...])	Replace values given in *to_replace* with *value*.
resample(rule[, how, axis, fill_method, ...])	Resample time-series data.
rfloordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
round([decimals])	Round each value in Panel to a specified number of decimal places.
rpow(other[, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(labels[, axis, inplace])	Assign desired index to given axis.
set_value(*args, **kwargs)	(DEPRECATED) Quickly set single value at (item, major, minor) location.
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq.
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1.
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)

Continued on next page

Table 146 – continued from previous page

sort_values(*args, **kwargs)	NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.
squeeze([axis])	Squeeze 1 dimensional axis objects into scalars.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only, . . .])	Return the sum of the values for the requested axis.
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis
tail([n])	Return the last *n* rows.
take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
to_clipboard([excel, sep])	Copy object to the system clipboard.
to_csv([path_or_buf, sep, na_rep, . . .])	Write object to a comma-separated values (csv) file.
to_dense()	Return dense representation of NDFrame (as opposed to sparse).
to_excel(path[, na_rep, engine])	Write each DataFrame in Panel to a separate excel sheet.
to_frame([filter_observations])	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, . . .])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, . . .])	Render an object to a LaTeX tabular environment table.
to_msgpack([path_or_buf, encoding])	Serialize object to input file path using msgpack format.
to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
to_sparse(*args, **kwargs)	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
to_sql(name, con[, schema, if_exists, . . .])	Write records stored in a DataFrame to a SQL database.
to_xarray()	Return an xarray object from the pandas object.
transform(func, *args, **kwargs)	Call `func` on self producing a NDFrame with transformed values and that has the same axis length as self.
transpose(*args, **kwargs)	Permute the dimensions of the Panel
truediv(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.

Continued on next page

Table 146 – continued from previous page

tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(tz[, axis, level, copy, …])	Localize tz-naive index of a Series or DataFrame to target time zone.
update(other[, join, overwrite, …])	Modify Panel in place using non-NA values from other Panel.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
where(cond[, other, inplace, axis, level, …])	Replace values where the condition is False.
xs(key[, axis])	Return slice of panel along selected axis.

pandas.Panel.abs

`Panel.abs()`

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

See also:

numpy.absolute Calculate the absolute value element-wise.

Notes

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

pandas.Panel.add

Panel.**add**(*other*, *axis=0*)
Addition of series and other, element-wise (binary operator *add*). Equivalent to `panel + other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.radd

pandas.Panel.add_prefix

Panel.**add_prefix**(*prefix*)
Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

Parameters

prefix [str] The string to add before each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_suffix Suffix row labels with string *suffix*.

DataFrame.add_suffix Suffix column labels with string *suffix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

pandas.Panel.add_suffix

`Panel.add_suffix`(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

Parameters

suffix [str] The string to add after each label.

Returns

Series or DataFrame New Series or DataFrame with updated labels.

See also:

Series.add_prefix Prefix row labels with string *prefix*.

DataFrame.add_prefix Prefix column labels with string *prefix*.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

pandas.Panel.align

Panel.**align**(*other*, ***kwargs*)

Align two objects on their axes with the specified join method for each axis Index.

Parameters

other [DataFrame or Series]

join [{'outer', 'inner', 'left', 'right'}, default 'outer']

axis [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

fill_axis [int or labels for object, default 0] Filling axis, method and limit

broadcast_axis [int or labels for object, default None] Broadcast values along this axis, if aligning two objects of different dimensions

Returns

(left, right) [(NDFrame, type of other)] Aligned objects

pandas.Panel.all

`Panel.all`(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

DataFrame or Panel If level is specified, then, Panel is returned; otherwise, DataFrame is returned.

See also:

`Series.all` Return True if all elements are True.

`DataFrame.any` Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([]).all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

pandas.Panel.any

`Panel.`**any**(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether any element is True, potentially over an axis.

Returns False unless there at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

bool_only [bool, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

skipna [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

DataFrame or Panel If level is specified, then, Panel is returned; otherwise, DataFrame is returned.

See also:

`numpy.any` Numpy version of this method.

`Series.any` Return whether any element is True.

`Series.all` Return whether all elements are True.

`DataFrame.any` Return whether any element is True over requested axis.

`DataFrame.all` Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0     True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

pandas.Panel.apply

`Panel.apply`(*func*, *axis='major'*, ***kwargs*)

Applies function along axis (or axes) of the Panel.

Parameters

func [function] Function to apply to each combination of 'other' axes e.g. if axis = 'items', the combination of major_axis/minor_axis will each be passed as a Series;

if axis = ('items', 'major'), DataFrames of items & major axis will be passed

axis [{'items', 'minor', 'major'}, or {0, 1, 2}, or a tuple with two] axes

Additional keyword arguments will be passed as keywords to the function

Returns

result [Panel, DataFrame, or Series]

Examples

Returns a Panel with the square root of each element

```
>>> p = pd.Panel(np.random.rand(4, 3, 2))  # doctest: +SKIP
>>> p.apply(np.sqrt)
```

Equivalent to p.sum(1), returning a DataFrame

```
>>> p.apply(lambda x: x.sum(), axis=1)  # doctest: +SKIP
```

Equivalent to previous:

```
>>> p.apply(lambda x: x.sum(), axis='major')  # doctest: +SKIP
```

Return the shapes of each DataFrame over axis 2 (i.e the shapes of items x major), as a Series

```
>>> p.apply(lambda x: x.shape, axis=(0,1))  # doctest: +SKIP
```

pandas.Panel.as_blocks

Panel.**as_blocks**(*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

Parameters

copy [boolean, default True]

Returns

values [a dict of dtype -> Constructor Types]

pandas.Panel.as_matrix

Panel.**as_matrix**()

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use *DataFrame.values()* instead.

Parameters

columns [list, optional, default:None] If None, return all columns, otherwise, returns specified columns.

Returns

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

See also:

DataFrame.values

Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.Panel.asfreq

`Panel.asfreq`(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

Parameters

freq [DateOffset object, or string]

method [{'backfill'/'bfill', 'pad'/'ffill'}, default None] Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

Returns

converted [same type as caller]

See also:

reindex

Notes

To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

pandas.Panel.asof

`Panel.asof(where, subset=None)`

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

where [date or array-like of dates] Date(s) before which the last row(s) are returned.

subset [str or array-like of str, default *None*] For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

scalar, Series, or DataFrame

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
5      NaN
20     2.0
dtype: float64
```

Missing values are not considered. The following is `2.0`, not NaN, even though NaN is at the index location for `30`.

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                      a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...         subset=['a'])
                         a   b
2018-02-27 09:03:30   30.0 NaN
2018-02-27 09:04:30   40.0 NaN
```

pandas.Panel.astype

`Panel.`**`astype`**(*dtype*, *copy=True*, *errors='raise'*, ***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, …}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

kwargs [keyword arguments to pass on to the constructor]

Returns

casted [same type as caller]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

numpy.ndarray.astype Cast a numpy array to a specified type.

Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> cat_dtype = pd.api.types.CategoricalDtype(
...                     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.Panel.at_time

Panel.**at_time**(*time*, *asof=False*, *axis=None*)
Select values at particular time of day (e.g. 9:30AM).

Parameters

time [datetime.time or string]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_at_time [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

between_time Select values between particular times of the day.

first Select initial periods of time series based on a date offset.

last Select final periods of time series based on a date offset.

DatetimeIndex.indexer_at_time Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

pandas.Panel.between_time

Panel.**between_time**(*start_time*, *end_time*, *include_start=True*, *include_end=True*, *axis=None*)
Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

Parameters

start_time [datetime.time or string]

end_time [datetime.time or string]

include_start [boolean, default True]

include_end [boolean, default True]

axis [{0 or 'index', 1 or 'columns'}, default 0] New in version 0.24.0.

Returns

values_between_time [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

at_time Select values at a particular time of the day.

first Select initial periods of time series based on a date offset.

last Select final periods of time series based on a date offset.

DatetimeIndex.indexer_between_time Get just the index locations for values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

pandas.Panel.bfill

`Panel.`**`bfill`**(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for *DataFrame.fillna()* with `method='bfill'`.

pandas.Panel.bool

`Panel.`**`bool`**()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

pandas.Panel.clip

`Panel.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)`

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

Parameters

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

pandas.Panel.clip_lower

`Panel.clip_lower(threshold, axis=None, inplace=False)`

Trim values below a given threshold.

Deprecated since version 0.24.0: Use clip(lower=threshold) instead.

Elements below the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

Series.clip General purpose method to trim Series values to given threshold(s).

DataFrame.clip General purpose method to trim DataFrame values to given threshold(s).

Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip(lower=8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip(lower=elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip(lower=3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip(lower=[3, 3, 5], axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip(lower=[4, 5], axis='columns')
   A  B
0  4  5
```

(continues on next page)

(continued from previous page)

```
1   4   5
2   5   6
```

pandas.Panel.clip_upper

Panel.**clip_upper**(*threshold*, *axis=None*, *inplace=False*)

Trim values above a given threshold.

Deprecated since version 0.24.0: Use clip(upper=threshold) instead.

Elements above the *threshold* will be changed to match the *threshold* value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

Parameters

threshold [numeric or array-like] Maximum value allowed. All values above threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align object with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Returns

Series or DataFrame Original data with values trimmed.

See also:

Series.clip General purpose method to trim Series values to given threshold(s).

DataFrame.clip General purpose method to trim DataFrame values to given threshold(s).

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.clip(upper=3)
0    1
1    2
2    3
```

(continues on next page)

(continued from previous page)

```
3    3
4    3
dtype: int64
```

```
>>> elemwise_thresholds = [5, 4, 3, 2, 1]
>>> elemwise_thresholds
[5, 4, 3, 2, 1]
```

```
>>> s.clip(upper=elemwise_thresholds)
0    1
1    2
2    3
3    2
4    1
dtype: int64
```

pandas.Panel.compound

`Panel.compound`(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

compounded [DataFrame or Panel (if level specified)]

pandas.Panel.conform

`Panel.conform`(*frame*, *axis='items'*)

Conform input DataFrame to align with chosen axis pair.

Parameters

frame [DataFrame]

axis [{'items', 'major', 'minor'}] Axis the input corresponds to. E.g., if axis='major', then the frame's columns would be items, and the index would be values of the minor axis

Returns

DataFrame

pandas.Panel.convert_objects

`Panel.convert_objects`(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

Parameters

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

Returns

converted [same as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

pandas.Panel.copy

`Panel.copy`(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns

copy [Series, DataFrame or Panel] Object type matches caller.

Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

pandas.Panel.count

`Panel.count(axis='major')`

Return number of observations over requested axis.

Parameters

axis [{'items', 'major', 'minor'} or {0, 1, 2}]

Returns

count [DataFrame]

pandas.Panel.cummax

`Panel.cummax(axis=None, skipna=True, *args, **kwargs)`

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummax [DataFrame or Panel]

See also:

core.window.Expanding.max Similar functionality but ignores `NaN` values.

Panel.max Return the maximum over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.Panel.cummin

`Panel.cummin`(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummin [DataFrame or Panel]

See also:

`core.window.Expanding.min` Similar functionality but ignores `NaN` values.

`Panel.min` Return the minimum over Panel axis.

`Panel.cummax` Return cumulative maximum over Panel axis.

`Panel.cummin` Return cumulative minimum over Panel axis.

`Panel.cumsum` Return cumulative sum over Panel axis.

`Panel.cumprod` Return cumulative product over Panel axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
```

(continues on next page)

(continued from previous page)

```
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.Panel.cumprod

Panel.**cumprod**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumprod [DataFrame or Panel]

See also:

core.window.Expanding.prod Similar functionality but ignores NaN values.

Panel.prod Return the product over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use skipna=False

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.Panel.cumsum

`Panel.`**`cumsum`**(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumsum [DataFrame or Panel]

See also:

core.window.Expanding.sum Similar functionality but ignores NaN values.

Panel.sum Return the sum over Panel axis.

Panel.cummax Return cumulative maximum over Panel axis.

Panel.cummin Return cumulative minimum over Panel axis.

Panel.cumsum Return cumulative sum over Panel axis.

Panel.cumprod Return cumulative product over Panel axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use skipna=False

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.Panel.describe

`Panel.describe`(*percentiles=None*, *include=None*, *exclude=None*)

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

DataFrame.count Count number of non-NA/null observations.

DataFrame.max Maximum of the values in the object.

DataFrame.min Minimum of the values in the object.

DataFrame.mean Mean of the values.

DataFrame.std Standard deviation of the obersvations.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
```

(continues on next page)

(continued from previous page)

```
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      c
freq             1      NaN      1
mean           NaN      2.0    NaN
std            NaN      1.0    NaN
min            NaN      1.0    NaN
25%            NaN      1.5    NaN
50%            NaN      2.0    NaN
75%            NaN      2.5    NaN
max            NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
```

(continues on next page)

(continued from previous page)

```
mean      2.0
std       1.0
min       1.0
25%       1.5
50%       2.0
75%       2.5
max       3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
       categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
```

(continues on next page)

(continued from previous page)

```
freq              1       NaN
mean            NaN       2.0
std             NaN       1.0
min             NaN       1.0
25%             NaN       1.5
50%             NaN       2.0
75%             NaN       2.5
max             NaN       3.0
```

pandas.Panel.div

Panel.**div**(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rtruediv

pandas.Panel.divide

Panel.**divide**(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rtruediv

pandas.Panel.droplevel

Panel.**droplevel**(*level*, *axis=0*)

Return DataFrame with requested index / column level(s) removed.

New in version 0.24.0.

Parameters

level [int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

axis [{0 or 'index', 1 or 'columns'}, default 0]

Returns

DataFrame.droplevel()

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['a', 'b'])
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...    ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1   c   d
level_2   e   f
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

```
>>> df.droplevel('a')
level_1   c   d
level_2   e   f
b
2         3   4
6         7   8
10       11  12
```

```
>>> df.droplevel('level2', axis=1)
level_1   c   d
a b
1 2       3   4
5 6       7   8
9 10     11  12
```

pandas.Panel.dropna

`Panel.dropna(axis=0, how='any', inplace=False)`
Drop 2D from panel, holding passed axis constant.

Parameters

axis [int, default 0] Axis to hold constant. E.g. axis=1 will drop major_axis entries having a certain amount of NA data

how [{'all', 'any'}, default 'any'] 'any': one or more values are NA in the DataFrame along the axis. For 'all' they all must be.

inplace [bool, default False] If True, do operation inplace and return None.

Returns

dropped [Panel]

pandas.Panel.eq

`Panel.eq(other, axis=None)`
Wrapper for comparison method eq

pandas.Panel.equals

`Panel.equals(other)`
Test whether two objects contain the same elements.

This function allows two Series or DataFrames to be compared against each other to see if they have the same shape and elements. NaNs in the same location are considered equal. The column headers do not need to have the same type, but the elements within the columns must be the same dtype.

Parameters

other [Series or DataFrame] The other Series or DataFrame to be compared with the first.

Returns

bool True if all elements are the same in both objects, False otherwise.

See also:

Series.eq Compare two Series objects of the same length and return a Series where each element is True if the element in each Series is equal, False otherwise.

DataFrame.eq Compare two DataFrame objects of the same shape and return a DataFrame where each element is True if the respective element in each DataFrame is equal, False otherwise.

assert_series_equal Return True if left and right Series are equal, False otherwise.

assert_frame_equal Return True if left and right DataFrames are equal, False otherwise.

numpy.array_equal Return True if two arrays have the same shape and elements, False otherwise.

Notes

This function requires that the elements have the same dtype as their respective elements in the other Series or DataFrame. However, the column labels do not need to have the same type, as long as they are still considered equal.

Examples

```
>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
    1   2
0  10  20
```

DataFrames df and exactly_equal have the same types and values for their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
    1   2
0  10  20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different_column_type have the same element types and values, but have different types for the column labels, which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
   1.0  2.0
0   10   20
>>> df.equals(different_column_type)
True
```

DataFrames df and different_data_type have different types for the same values for their elements, and will return False even though their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
      1     2
0  10.0  20.0
>>> df.equals(different_data_type)
False
```

pandas.Panel.ffill

`Panel.ffill`(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna()` with `method='ffill'`.

pandas.Panel.fillna

`Panel.fillna`(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis [{0, 1, 2, 'items', 'major_axis', 'minor_axis'}]

inplace [boolean, default False] If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns

filled [Panel]

See also:

interpolate Fill NaN values using interpolation.

reindex, *asfreq*

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
```

(continues on next page)

(continued from previous page)

```
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

pandas.Panel.filter

Panel.**filter**(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items [list-like] List of axis to restrict to (must not all be present).

like [string] Keep axis where "arg in col == True".

regex [string (regular expression)] Keep axis with re.search(regex, col) == True.

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

same type as input object

See also:

DataFrame.loc

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> df = pd.DataFrame(np.array(([1,2,3], [4,5,6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
         one  three
mouse      1      3
rabbit     4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit     4    5      6
```

pandas.Panel.first

Panel.**first**(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a `DatetimeIndex`

See also:

last Select final periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

pandas.Panel.first_valid_index

Panel.**first_valid_index**()

Return index for first non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.Panel.floordiv

Panel.**floordiv**(*other*, *axis=0*)

Integer division of series and other, element-wise (binary operator *floordiv*). Equivalent to `panel // other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rfloordiv

pandas.Panel.fromDict

classmethod Panel.**fromDict**(*data*, *intersect=False*, *orient='items'*, *dtype=None*)

Construct Panel from dict of DataFrame objects.

Parameters

data [dict] {field : DataFrame}

intersect [boolean] Intersect indexes of input DataFrames

orient [{'items', 'minor'}, default 'items'] The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype [dtype, default None] Data type to force, otherwise infer

Returns

Panel

pandas.Panel.from_dict

`classmethod Panel.from_dict(data, intersect=False, orient='items', dtype=None)`

Construct Panel from dict of DataFrame objects.

Parameters

data [dict] {field : DataFrame}

intersect [boolean] Intersect indexes of input DataFrames

orient [{'items', 'minor'}, default 'items'] The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype [dtype, default None] Data type to force, otherwise infer

Returns

Panel

pandas.Panel.ge

`Panel.ge(other, axis=None)`

Wrapper for comparison method ge

pandas.Panel.get

`Panel.get(key, default=None)`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

pandas.Panel.get_dtype_counts

`Panel.get_dtype_counts()`

Return counts of unique dtypes in this object.

Returns

dtype [Series] Series with the count of columns with each dtype.

See also:

dtypes Return the dtypes in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

pandas.Panel.get_ftype_counts

Panel.**get_ftype_counts**()

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

Returns

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

See also:

ftypes Return ftypes (indication of sparse/dense and dtype) in this object.

Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()  # doctest: +SKIP
float64:dense     1
int64:dense       1
object:dense      1
dtype: int64
```

pandas.Panel.get_value

Panel.**get_value**(**args*, ***kwargs*)

Quickly retrieve single value at (item, major, minor) location.

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

Parameters

item [item label (panel item)]

major [major axis label (panel item row)]

minor [minor axis label (panel item column)]

takeable [interpret the passed labels as indexers, default False]

Returns

value [scalar value]

pandas.Panel.get_values

`Panel.get_values()`

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

Returns

numpy.ndarray Numpy representation of DataFrame

See also:

values Numpy representation of DataFrame.

pandas.SparseArray Container for sparse data.

Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

pandas.Panel.groupby

Panel.**groupby**(*function*, *axis='major'*)
Group data on given axis, returning GroupBy object.

Parameters

function [callable] Mapping function for chosen access

axis [{'major', 'minor', 'items'}, default 'major']

Returns

grouped [PanelGroupBy]

pandas.Panel.gt

Panel.**gt**(*other*, *axis=None*)
Wrapper for comparison method gt

pandas.Panel.head

Panel.**head**(*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters

n [int, default 5] Number of rows to select.

Returns

obj_head [same type as caller] The first *n* rows of the caller object.

See also:

DataFrame.tail Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
```

(continues on next page)

(continued from previous page)

```
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

pandas.Panel.infer_objects

`Panel.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

Returns

converted [same type as input object]

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

pandas.Panel.interpolate

Panel.**interpolate**(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.
- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

New in version 0.18.1: Added support for the 'akima' method. Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in SciPy 0.18; backwards-compatible with SciPy < 0.18

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.

- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.21.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all `NaN` values

See also:

fillna Fill missing values using different methods.

scipy.interpolate.Akima1DInterpolator Piecewise cubic polynomials (Akima interpolator).

scipy.interpolate.BPoly.from_derivatives Piecewise polynomial in the Bernstein basis.

scipy.interpolate.interp1d Interpolate a 1-D function.

scipy.interpolate.KroghInterpolator Interpolate polynomial (Krogh interpolator).

scipy.interpolate.PchipInterpolator PCHIP 1-d monotonic cubic interpolation.

scipy.interpolate.CubicSpline Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in `NaN` in a *Series* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in `NaN` in a Series by padding, but filling at most two consecutive `NaN` at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in NaN in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an order (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains NaN, because there is no entry befofe it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0,  np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
```

(continues on next page)

(continued from previous page)

```
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

pandas.Panel.isna

`Panel.isna()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

`NDFrame.isnull` Alias of isna.

`NDFrame.notna` Boolean inverse of isna.

`NDFrame.dropna` Omit axes labels with missing values.

`isna` Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Panel.isnull

`Panel.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

`NDFrame.isnull` Alias of isna.

`NDFrame.notna` Boolean inverse of isna.

`NDFrame.dropna` Omit axes labels with missing values.

`isna` Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
```

(continues on next page)

(continued from previous page)

```
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

pandas.Panel.iteritems

Panel.**iteritems**()

Iterate over (label, values) on info axis

This is index for Series, columns for DataFrame, major_axis for Panel, and so on.

pandas.Panel.join

Panel.**join**(*other*, *how='left'*, *lsuffix=''*, *rsuffix=''*)

Join items with other Panel either on major and minor axes column.

Parameters

other [Panel or list of Panels] Index should be similar to one of the columns in this one

how [{'left', 'right', 'outer', 'inner'}] How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise * left: use calling frame's index * right: use input frame's index * outer: form union of indexes * inner: use intersection of indexes

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

Returns

joined [Panel]

pandas.Panel.keys

`Panel.keys()`

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

pandas.Panel.kurt

`Panel.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [DataFrame or Panel (if level specified)]

pandas.Panel.kurtosis

`Panel.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

kurt [DataFrame or Panel (if level specified)]

pandas.Panel.last

`Panel.last(offset)`

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a *DatetimeIndex*

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

pandas.Panel.last_valid_index

`Panel.last_valid_index()`

Return index for last non-NA/null value.

Returns

scalar [type of index]

Notes

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

pandas.Panel.le

`Panel.le(other, axis=None)`
Wrapper for comparison method le

pandas.Panel.lt

`Panel.lt(other, axis=None)`
Wrapper for comparison method lt

pandas.Panel.mad

`Panel.mad(axis=None, skipna=None, level=None)`
Return the mean absolute deviation of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mad [DataFrame or Panel (if level specified)]

pandas.Panel.major_xs

`Panel.major_xs(key)`
Return slice of panel along major axis.

Parameters

key [object] Major axis label

Returns

y [DataFrame] index -> minor axis, columns -> items

Notes

major_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of major_xs functionality, see *MultiIndex Slicers*

pandas.Panel.mask

`Panel.mask`(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)

Replace values where the condition is True.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

DataFrame.where() Return an object of same shape as self.

Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `mask` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.Panel.max

`Panel.`**`max`**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the maximum of the values for the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

max [DataFrame or Panel (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

Max using level names, as well as indices.

```
>>> s.max(level='blooded')
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

```
>>> s.max(level=0)
blooded
warm    4
cold    8
Name: legs, dtype: int64
```

pandas.Panel.mean

`Panel.mean(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`
Return the mean of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mean [DataFrame or Panel (if level specified)]

pandas.Panel.median

`Panel.median(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`
Return the median of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

median [DataFrame or Panel (if level specified)]

pandas.Panel.min

`Panel.min(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return the minimum of the values for the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

min [DataFrame or Panel (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
```

(continues on next page)

(continued from previous page)

```
warm      dog        4
          falcon     2
cold      fish       0
          spider     8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

Min using level names, as well as indices.

```
>>> s.min(level='blooded')
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

```
>>> s.min(level=0)
blooded
warm    2
cold    0
Name: legs, dtype: int64
```

pandas.Panel.minor_xs

`Panel.minor_xs(key)`

Return slice of panel along minor axis.

Parameters

key [object] Minor axis label

Returns

y [DataFrame] index -> major axis, columns -> items

Notes

minor_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of minor_xs functionality, see *MultiIndex Slicers*

pandas.Panel.mod

`Panel.mod(other, axis=0)`

Modulo of series and other, element-wise (binary operator *mod*). Equivalent to `panel % other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rmod

pandas.Panel.mul

`Panel.mul(other, axis=0)`

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rmul

pandas.Panel.multiply

`Panel.multiply(other, axis=0)`

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rmul

pandas.Panel.ne

`Panel.ne(other, axis=None)`

Wrapper for comparison method ne

pandas.Panel.notna

`Panel.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

NDFrame.notnull Alias of notna.

NDFrame.isna Boolean inverse of notna.

NDFrame.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Panel.notnull

Panel.**notnull**()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

Returns

NDFrame Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

`NDFrame.notnull` Alias of notna.

`NDFrame.isna` Boolean inverse of notna.

`NDFrame.dropna` Omit axes labels with missing values.

`notna` Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

pandas.Panel.pct_change

`Panel.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)`

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns

chg [Series or DataFrame] The same type as the calling object.

See also:

Series.diff Compute the difference of two elements in a Series.

DataFrame.diff Compute the difference of two elements in a DataFrame.

Series.shift Shift the index by some number of periods.

DataFrame.shift Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pandas.Panel.pipe

`Panel.pipe(func, *args, **kwargs)`

Apply func(self, *args, **kwargs).

Parameters

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

Returns

object [the return type of `func`.]

See also:

DataFrame.apply, *DataFrame.applymap*, *Series.map*

Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.Panel.pop

`Panel.pop(item)`

Return item and drop from frame. Raise KeyError if not found.

Parameters

item [str] Column label to be popped

Returns

popped [Series]

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pandas.Panel.pow

Panel.**pow**(*other*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*). Equivalent to `panel ** other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rpow

pandas.Panel.prod

Panel.**prod**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [DataFrame or Panel (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.Panel.product

`Panel.product`(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

prod [DataFrame or Panel (if level specified)]

Examples

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

pandas.Panel.radd

`Panel.`**`radd`**(*other*, *axis=0*)

Addition of series and other, element-wise (binary operator *radd*). Equivalent to `other + panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.add`

pandas.Panel.rank

`Panel.`**`rank`**(*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

Returns

ranks [same type as caller]

pandas.Panel.rdiv

`Panel.`**`rdiv`**(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.truediv`

pandas.Panel.reindex

`Panel.`**`reindex`**(**args*, ***kwargs*)

Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

Parameters

items, major_axis, minor_axis [array-like, optional] New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit [int, default None] Maximum number of consecutive elements to forward or backward fill.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

Panel with changed index.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...      'http_status': [200,200,404,404,301],
...      'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...       index=index)
>>> df
           http_status  response_time
Firefox            200           0.04
Chrome             200           0.02
Safari             404           0.07
IE10               404           0.08
Konqueror          301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
2009-12-31   100.0
2010-01-01   100.0
2010-01-02   101.0
2010-01-03     NaN
2010-01-04   100.0
2010-01-05    89.0
2010-01-06    88.0
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the *user guide* for more.

pandas.Panel.reindex_axis

`Panel.`**`reindex_axis`**(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index.

Deprecated since version 0.21.0: Use *reindex* instead.

By default, places NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data.

axis [{0, 1, 2, 'items', 'major_axis', 'minor_axis'}] Indicate whether to use rows or columns.

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps.
- pad / ffill: propagate last valid observation forward to next valid.
- backfill / bfill: use next valid observation to fill gap.
- nearest: use nearest valid observations to fill gap.

level [int or str] Broadcast across a level, matching Index values on the passed MultiIndex level.

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, optional] Maximum number of consecutive elements to forward or backward fill.

fill_value [float, default NaN] Value used to fill in locations having no value in the previous index.

New in version 0.21.0: (list-like tolerance)

Returns

Panel Returns a new DataFrame object with new indices, unless the new index is equivalent to the current one and copy=False.

See also:

`DataFrame.set_index` Set row labels.

`DataFrame.reset_index` Remove row labels or move them to new columns.

`DataFrame.reindex` Change to new indices or expand indices.

`DataFrame.reindex_like` Change to same indices as other DataFrame.

Examples

```
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                   index=['dog', 'hawk'])
>>> df
      num_legs  num_wings
dog          4          0
hawk         2          2
>>> df.reindex(['num_wings', 'num_legs', 'num_heads'],
...            axis='columns')
      num_wings  num_legs  num_heads
dog           0         4        NaN
hawk          2         2        NaN
```

pandas.Panel.reindex_like

`Panel.reindex_like`(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False.

Parameters

other [Object of the same data type] Its row and column indices are used to define the new indices of this object.

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}] Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [bool, default True] Return a new object, even if the passed indexes are the same.

limit [int, default None] Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

Series or DataFrame Same type as caller, but with changed indices on each axis.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex Change to new indices or expand indices.

Notes

Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...     columns=['temp_celsius', 'temp_fahrenheit', 'windspeed'],
...     index=pd.date_range(start='2014-02-12',
...                         end='2014-02-15', freq='D'))
```

```
>>> df1
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df2 = pd.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...     columns=['temp_celsius', 'windspeed'],
...     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                             '2014-02-15']))
```

```
>>> df2
            temp_celsius windspeed
2014-02-12          28.0       low
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

pandas.Panel.rename

`Panel.`**`rename`**(*items=None*, *major_axis=None*, *minor_axis=None*, ***kwargs*)

Alter axes input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters

items, major_axis, minor_axis [scalar, list-like, dict-like or function, optional] Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new Panel. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

Returns

renamed [Panel (new object)]

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
```

Since `DataFrame` doesn't have a `.name` attribute, only mapping-type arguments are allowed.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

See the *user guide* for more.

pandas.Panel.rename_axis

Panel.**rename_axis**(*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*)

Set the name of the axis for the index or columns.

Parameters

mapper [scalar, list-like, optional] Value to set the axis name attribute.

index, columns [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

Changed in version 0.24.0.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to rename.

copy [bool, default True] Also copy underlying data.

inplace [bool, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

Returns

Series, DataFrame, or None The same type as the caller or None if *inplace* is True.

See also:

Series.rename Alter Series index labels or name.

DataFrame.rename Alter DataFrame index labels or name.

Index.rename Set new names on index.

Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
```

(continues on next page)

(continued from previous page)

```
cat             4          0
monkey          2          2
>>> df = df.rename_axis("limbs", axis="columns")
>>> df
limbs    num_legs  num_arms
animal
dog             4          0
cat             4          0
monkey          2          2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

pandas.Panel.replace

`Panel.`**`replace`**(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the NDFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*

- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [bool, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

Returns

NDFrame Object after replacement.

Raises

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

See also:

`NDFrame.fillna` Fill NA values.

`NDFrame.where` Replace values based on boolean condition.

`Series.str.replace` Simple string replacement.

Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like ‘to_replace‘

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like ‘to_replace‘

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
```

(continues on next page)

(continued from previous page)

```
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

pandas.Panel.resample

`Panel.`**`resample`**(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or pass datetime-like values to the *on* or *level* keyword.

Parameters

rule [str] The offset string or object representing target conversion.

how [str] Method for down/re-sampling, default to 'mean' for downsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).mean()`, or `.resample(...).apply(<func>)`

axis [{0 or 'index', 1 or 'columns'}, default 0] Which axis to use for up- or down-sampling. For *Series* this will default to 0, i.e. along the rows. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.

fill_method [str, default None] Filling method for upsampling.

Deprecated since version 0.18.0: The new syntax is `.resample(...).<func>()`, e.g. `.resample(...).pad()`

closed [{'right', 'left'}, default None] Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}, default None] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}, default 'start'] For *PeriodIndex* only, controls whether to use the start or end of *rule*.

kind [{'timestamp', 'period'}, optional, default None] Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.

loffset [timedelta, default None] Adjust the resampled time labels.

limit [int, default None] Maximum size gap when reindexing with *fill_method*.

Deprecated since version 0.18.0.

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0.

on [str, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [str or int, optional] For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

`groupby` Group by mapping, function, label, or list of labels.

`Series.resample` Resample a Series.

`DataFrame.resample` Resample a DataFrame.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
2018-06    2.0
```

(continues on next page)

(continued from previous page)

```
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...           'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df2 = pd.DataFrame(d2,
...                    index=pd.MultiIndex.from_product([days,
...                                                     ['morning',
...                                                      'afternoon']]
...                                                     ))
>>> df2
                      price  volume
2000-01-01 morning       10      50
           afternoon     11      60
2000-01-02 morning        9      40
           afternoon     13     100
2000-01-03 morning       14      50
           afternoon     18     100
2000-01-04 morning       17      40
           afternoon     19      50
>>> df2.resample('D', level=0).sum()
            price  volume
```

(continues on next page)

(continued from previous page)

```
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

pandas.Panel.rfloordiv

`Panel.`**`rfloordiv`**(*other*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*). Equivalent to `other // panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.floordiv`

pandas.Panel.rmod

`Panel.`**`rmod`**(*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *rmod*). Equivalent to `other % panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.mod`

pandas.Panel.rmul

`Panel.`**`rmul`**(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *rmul*). Equivalent to `other * panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.mul

pandas.Panel.round

`Panel.round(decimals=0, *args, **kwargs)`

Round each value in Panel to a specified number of decimal places.

New in version 0.18.0.

Parameters

decimals [int] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Panel object

See also:

`numpy.around`

pandas.Panel.rpow

`Panel.rpow(other, axis=0)`

Exponential power of series and other, element-wise (binary operator *rpow*). Equivalent to `other ** panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.pow

pandas.Panel.rsub

`Panel.rsub(other, axis=0)`

Subtraction of series and other, element-wise (binary operator *rsub*). Equivalent to `other - panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.sub

pandas.Panel.rtruediv

`Panel.rtruediv(other, axis=0)`

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.truediv

pandas.Panel.sample

`Panel.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)`

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [bool, default False] Sample with or without replacement.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

Returns

Series or DataFrame A new object of same type as caller containing *n* items randomly sampled from the caller object.

See also:

`numpy.random.choice` Generates a random sample from a given 1-D numpy array.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                   index=['falcon', 'dog', 'spider', 'fish'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

pandas.Panel.select

`Panel.`**`select`**(*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria.

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

Parameters

crit [function] To be called on each index (label). Should return True or False

axis [int]

Returns

selection [same type as caller]

pandas.Panel.sem

`Panel.sem`(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

sem [DataFrame or Panel (if level specified)]

pandas.Panel.set_axis

`Panel.set_axis`(*labels*, *axis=0*, *inplace=None*)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

Parameters

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

Returns

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

See also:

DataFrame.rename_axis Alter the name of the index or columns.

Examples

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

pandas.Panel.set_value

Panel.**set_value**(**args*, ***kwargs*)

Quickly set single value at (item, major, minor) location.

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

Parameters

item [item label (panel item)]

major [major axis label (panel item row)]

minor [minor axis label (panel item column)]

value [scalar]

takeable [interpret the passed labels as indexers, default False]

Returns

panel [Panel] If label combo is contained, will be reference to calling Panel, otherwise a new object

pandas.Panel.shift

Panel.**shift**(*periods=1*, *freq=None*, *axis='major'*)

Shift index by desired number of periods with an optional time freq.

The shifted data will not include the dropped periods and the shifted axis will be smaller than the original. This is different from the behavior of DataFrame.shift()

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional]

axis [{'items', 'major', 'minor'} or {0, 1, 2}]

Returns

shifted [Panel]

pandas.Panel.skew

Panel.**skew**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased skew over requested axis Normalized by N-1.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

skew [DataFrame or Panel (if level specified)]

pandas.Panel.slice_shift

`Panel.`**`slice_shift`**(*periods=1*, *axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

Parameters

periods [int] Number of periods to move, can be positive or negative

Returns

shifted [same type as caller]

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

pandas.Panel.sort_index

`Panel.`**`sort_index`**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*)

Sort object by labels (along an axis)

Parameters

axis [%(axes)s to direct sorting]

level [int or level name or list of ints or list of level names] if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

Returns

sorted_obj [%(klass)s]

pandas.Panel.sort_values

`Panel.sort_values(*args, **kwargs)`
NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.

pandas.Panel.squeeze

`Panel.squeeze(axis=None)`
Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

Parameters

axis [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

New in version 0.20.0.

Returns

DataFrame, Series, or scalar The projection after squeezing *axis* or all the axes.

See also:

Series.iloc Integer-location based indexing for selecting scalars.

DataFrame.iloc Integer-location based indexing for selecting Series.

Series.to_frame Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```
>>> primes = pd.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a    1
Name: 0, dtype: int64
```

Squeezing all axes wil project directly into a scalar:

```
>>> df_0a.squeeze()
1
```

pandas.Panel.std

`Panel.std`(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

std [DataFrame or Panel (if level specified)]

pandas.Panel.sub

`Panel.`**`sub`**(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rsub`

pandas.Panel.subtract

`Panel.`**`subtract`**(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

`Panel.rsub`

pandas.Panel.sum

`Panel.`**`sum`**(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the sum of the values for the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

sum [DataFrame or Panel (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

Sum using level names, as well as indices.

```
>>> s.sum(level='blooded')
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

```
>>> s.sum(level=0)
blooded
warm    6
cold    8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

pandas.Panel.swapaxes

`Panel.`**`swapaxes`**(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately.

Returns

y [same as input]

pandas.Panel.swaplevel

`Panel.`**`swaplevel`**(*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

Returns

swapped [same type as caller (new object)]

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

pandas.Panel.tail

`Panel.tail(n=5)`

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Parameters

n [int, default 5] Number of rows to select.

Returns

type of caller The last *n* rows of the caller object.

See also:

DataFrame.head The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
  animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

pandas.Panel.take

`Panel.take`(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [same type as caller] An array-like containing the elements taken from the object.

See also:

DataFrame.loc Select a subset of a DataFrame by labels.

DataFrame.iloc Select a subset of a DataFrame by positions.

numpy.take Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

pandas.Panel.to_clipboard

`Panel.to_clipboard(excel=True, sep=None, **kwargs)`

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

Parameters

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

See also:

DataFrame.to_csv Write a DataFrame to a comma-separated values (csv) file.

read_clipboard Read text from clipboard and pass to read_table.

Notes

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none

- OS X : none

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

pandas.Panel.to_csv

`Panel.to_csv`(*path_or_buf=None, sep=', ', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding=None, compression='infer', quoting=None, quotechar='"', line_terminator=None, chunksize=None, tupleize_cols=None, date_format=None, doublequote=True, escapechar=None, decimal='.'*)

Write object to a comma-separated values (csv) file.

Changed in version 0.24.0: The order of arguments for Series was changed.

Parameters

path_or_buf [str or file handle, default None] File path or object, if None is provided the result is returned as a string.

Changed in version 0.24.0: Was previously named "path" for Series.

sep [str, default ','] String of length 1. Field delimiter for the output file.

na_rep [str, default ''] Missing data representation.

float_format [str, default None] Format string for floating point numbers.

columns [sequence, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

Changed in version 0.24.0: Previously defaulted to False for Series.

index [bool, default True] Write row names (index).

index_label [str or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.

mode [str] Python write mode, default 'w'.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [str, default 'infer'] Compression mode among the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If 'infer' and *path_or_buf* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression).

Changed in version 0.24.0: 'infer' option added and set to default.

quoting [optional constant from csv module] Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.

quotechar [str, default '"'] String of length 1. Character used to quote fields.

line_terminator [string, optional] The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('n' for linux, 'rn' for Windows, i.e.).

Changed in version 0.24.0.

chunksize [int or None] Rows to write at a time.

tupleize_cols [bool, default False] Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

date_format [str, default None] Format string for datetime objects.

doublequote [bool, default True] Control quoting of *quotechar* inside a field.

escapechar [str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

decimal [str, default '.'] Character recognized as decimal separator. E.g. use ',' for European data.

Returns

None or str If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

See also:

read_csv Load a CSV file into a DataFrame.

to_excel Load an Excel file into a DataFrame.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

pandas.Panel.to_dense

`Panel.to_dense()`

Return dense representation of NDFrame (as opposed to sparse).

pandas.Panel.to_excel

`Panel.to_excel(path, na_rep='', engine=None, **kwargs)`

Write each DataFrame in Panel to a separate excel sheet.

Parameters

path [string or ExcelWriter object] File path or existing ExcelWriter

na_rep [string, default ''] Missing data representation

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

Other Parameters

float_format [string, default None] Format string for floating point numbers

cols [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out column names. If a list of string is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow [upper left cell row to dump data frame]

startcol [upper left cell column to dump data frame]

Notes

Keyword arguments (and na_rep) are passed to the `to_excel` method for each DataFrame written.

pandas.Panel.to_frame

`Panel.to_frame(filter_observations=True)`

Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.

Parameters

filter_observations [boolean, default True] Drop (major, minor) pairs without a complete set of observations across all the items

Returns

y [DataFrame]

pandas.Panel.to_hdf

`Panel.to_hdf(path_or_buf, key, **kwargs)`

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the *user guide*.

Parameters

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See *Query via Data Columns*. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

`DataFrame.read_hdf` Read from HDF file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

DataFrame.to_sql Write to a sql table.

DataFrame.to_feather Write out feather-format for DataFrames.

DataFrame.to_csv Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

pandas.Panel.to_json

Panel.**to_json**(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression='infer'*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

Parameters

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {'split','records','index','table'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values','table'}
- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, … , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, 'epoch', 'iso'}] Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [bool, default True] Force encoded string to be ASCII.

date_unit [string, default 'ms' (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [bool, default False] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}] A string representing the compression to use in the output file, only used when the first argument is a filename. By default, the compression is inferred from the filename.

New in version 0.21.0.

Changed in version 0.24.0: 'infer' option added and set to default

index [bool, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

See also:

read_json

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

pandas.Panel.to_latex

`Panel.to_latex`(*buf=None, columns=None, col_space=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multicolumn_format=None, multirow=None*)

Render an object to a LaTeX tabular environment table.

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

Parameters

buf [file descriptor or None] Buffer to write to. If None, the output is returned as a string.

columns [list of label, optional] The subset of columns to write. Writes all columns by default.

col_space [int, optional] The minimum width of each column.

header [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

na_rep [str, default 'NaN'] Missing data representation.

formatters [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [str, optional] Format string for floating point numbers.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

index_names [bool, default True] Prints the names of the indexes.

bold_rows [bool, default False] Make the row labels bold in the output.

column_format [str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

longtable [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a usepackage{longtable} to your LaTeX preamble.

escape [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

encoding [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe. .. versionadded:: 0.18.0

multicolumn [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module. .. versionadded:: 0.20.0

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module. .. versionadded:: 0.20.0

multirow [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module. .. versionadded:: 0.20.0

Returns

str or None If buf is None, returns the resulting LateX format as a string. Otherwise returns None.

See also:

DataFrame.to_string Render a DataFrame to a console-friendly tabular output.

DataFrame.to_html Render a DataFrame as an HTML table.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_latex(index=False) # doctest: +NORMALIZE_WHITESPACE
'\\begin{tabular}{lll}\n\\toprule\n      name &    mask &    weapon
\\\\\n\\midrule\n   Raphael &     red &       sai \\\\\n Donatello &
 purple &  bo staff \\\\\n\\bottomrule\n\\end{tabular}\n'
```

pandas.Panel.to_msgpack

Panel.**to_msgpack**(*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)

Serialize object to input file path using msgpack format.

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

Parameters

path [string File path, buffer-like, or None] if None, return generated string

append [bool whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

pandas.Panel.to_pickle

Panel.**to_pickle**(*path*, *compression='infer'*, *protocol=4*)

Pickle (serialize) object to file.

Parameters

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [**?**] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

See also:

read_pickle Load pickled pandas object (or any object) from file.

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_sql Write DataFrame to a SQL database.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

pandas.Panel.to_sparse

Panel.**to_sparse**(**args*, ***kwargs*)

NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Convert to SparsePanel.

pandas.Panel.to_sql

Panel.**to_sql**(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [**?**] are supported. Tables can be newly created, appended to, or overwritten.

Parameters

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [bool, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

method [{None, 'multi', callable}, default None] Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section *insert method*.

New in version 0.24.0.

Raises

ValueError When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

New in version 0.24.0.

References

[?], [?]

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

pandas.Panel.to_xarray

`Panel.to_xarray()`

Return an xarray object from the pandas object.

Returns

xarray.DataArray or xarray.Dataset Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

See also:

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',  389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion',   'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                    columns=['name', 'class', 'max_speed',
...                             'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                    'animal': ['falcon', 'parrot', 'falcon',
...                               'parrot'],
...                    'speed': [350, 18, 361, 15]}).set_index(['date',
...                                                    'animal'])
>>> df_multiindex
                   speed
date       animal
```

(continues on next page)

(continued from previous page)

```
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

pandas.Panel.transform

`Panel.transform`(*func*, **args*, ***kwargs*)

Call `func` on self producing a NDFrame with transformed values and that has the same axis length as self.

New in version 0.20.0.

Parameters

func [function, str, list or dict] Function to use for transforming the data. If a function, must either work when passed a NDFrame or when passed to NDFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.exp. 'sqrt']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

NDFrame A NDFrame that must have the same length as self.

Raises

ValueError [If the returned NDFrame has a different length than self.]

See also:

`NDFrame.agg` Only perform aggregating type operations.

`NDFrame.apply` Invoke function on a NDFrame.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting NDFrame must have the same length as the input NDFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

pandas.Panel.transpose

Panel.**transpose**(**args*, ***kwargs*)

Permute the dimensions of the Panel

Parameters

args [three positional arguments: each one of]

{0, 1, 2, 'items', 'major_axis', 'minor_axis'}

copy [boolean, default False] Make a copy of the underlying data. Mixed-dtype data will always result in a copy

Returns

y [same as input]

Examples

```
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

pandas.Panel.truediv

Panel.**truediv**(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters

other [DataFrame or Panel]

axis [{items, major_axis, minor_axis}] Axis to broadcast over

Returns

Panel

See also:

Panel.rtruediv

pandas.Panel.truncate

Panel.**truncate**(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

Parameters

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

Returns

type of caller The truncated Series or DataFrame.

See also:

DataFrame.loc Select a subset of a DataFrame by label.

DataFrame.iloc Select a subset of a DataFrame by position.

Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

pandas.Panel.tshift

Panel.**tshift**(*periods=1*, *freq=None*, *axis='major'*)
Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

pandas.Panel.tz_convert

Panel.**tz_convert**(*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters

tz [string or pytz.timezone object]

axis [the axis to convert]

level [int, str, default None] If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

Raises

TypeError If the axis is tz-naive.

pandas.Panel.tz_localize

Panel.**tz_localize**(*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*, *nonexistent='raise'*)
Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use *Series.dt.tz_localize()*.

Parameters

tz [string or pytz.timezone object]

axis [the axis to localize]

level [int, str, default None] If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent [str, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid valuse are:

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

Series or DataFrame Same type as the input.

Raises

TypeError If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series([1],
... index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7), index=pd.DatetimeIndex([
... '2018-10-28 01:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 03:00:00',
... '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3), index=pd.DatetimeIndex([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.Series(range(2), index=pd.DatetimeIndex([… '2015-03-29 02:30:00', … '2015-03-29 03:30:00'])) >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 2015-03-29 03:00:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 2015-03-29 01:59:59.999999999+01:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64 >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 2015-03-29 03:30:00+02:00 0 2015-03-29 03:30:00+02:00 1 dtype: int64

pandas.Panel.update

`Panel.update`(*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *errors='ignore'*)

Modify Panel in place using non-NA values from other Panel.

May also use object coercible to Panel. Will align on items.

Parameters

other [Panel, or object coercible to Panel] The object from which the caller will be updated.

join [{'left', 'right', 'outer', 'inner'}, default 'left'] How individual DataFrames are joined.

overwrite [bool, default True] If True then overwrite values for common keys in the calling Panel.

filter_func [callable(1d-array) -> 1d-array<bool>, default None] Can choose to replace values other than NA. Return True for values that should be updated.

errors [{'raise', 'ignore'}, default 'ignore'] If 'raise', will raise an error if a DataFrame and other both.

Changed in version 0.24.0: Changed from *raise_conflict=False|True* to *errors='ignore'|'raise'*.

See also:

DataFrame.update Similar method for DataFrames.

dict.update Similar method for dictionaries.

pandas.Panel.var

Panel.**var**(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

axis [{items (0), major_axis (1), minor_axis (2)}]

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

var [DataFrame or Panel (if level specified)]

pandas.Panel.where

Panel.**where**(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Replace values where the condition is False.

Parameters

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data.

axis [int, default None] Alignment axis if needed.

level [int, default None] Alignment level if needed.

errors [str, {'raise', 'ignore'}, default *raise*] Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

- *raise* : allow exceptions to be raised.
- *ignore* : suppress exceptions. On error return original object.

try_cast [boolean, default False] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use *errors*.

Returns

wh [same type as caller]

See also:

DataFrame.mask() Return an object of same shape as self.

Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

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

For further details and examples see the `where` documentation in *indexing*.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
```

(continues on next page)

(continued from previous page)

```
4    NaN
dtype: float64
```

```
>>> s.where(s > 1, 10)
0    10
1    10
2     2
3     3
4     4
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

pandas.Panel.xs

`Panel.xs`(*key*, *axis=1*)

Return slice of panel along selected axis.

Parameters

key [object] Label

axis [{'items', 'major', 'minor}, default 1/'major']

Returns

y [ndim(self)-1]

Notes

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of xs functionality, see *MultiIndex Slicers*

agg	
aggregate	
drop	

6.6.2 Properties and underlying data

Axes

- **items**: axis 0; each item corresponds to a DataFrame contained inside
- **major_axis**: axis 1; the index (rows) of each of the DataFrames
- **minor_axis**: axis 2; the columns of each of the DataFrames

Panel.values	Return a Numpy representation of the DataFrame.
Panel.axes	Return index label(s) of the internal NDFrame
Panel.ndim	Return an int representing the number of axes / array dimensions.
Panel.size	Return an int representing the number of elements in this object.
Panel.shape	Return a tuple of axis dimensions
Panel.dtypes	Return the dtypes in the DataFrame.
Panel.ftypes	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
Panel.get_dtype_counts()	Return counts of unique dtypes in this object.
Panel.get_ftype_counts()	(DEPRECATED) Return counts of unique ftypes in this object.

6.6.3 Conversion

Panel.astype(dtype[, copy, errors])	Cast a pandas object to a specified dtype `dtype`.
Panel.copy([deep])	Make a copy of this object's indices and data.
Panel.isna()	Detect missing values.
Panel.notna()	Detect existing (non-missing) values.

6.6.4 Getting and setting

Panel.get_value(*args, **kwargs)	(DEPRECATED) Quickly retrieve single value at (item, major, minor) location.
Panel.set_value(*args, **kwargs)	(DEPRECATED) Quickly set single value at (item, major, minor) location.

6.6.5 Indexing, iteration, slicing

Panel.at	Access a single value for a row/column label pair.
Panel.iat	Access a single value for a row/column pair by integer position.

Continued on next page

Table 150 – continued from previous page

Panel.loc	Access a group of rows and columns by label(s) or a boolean array.
Panel.iloc	Purely integer-location based indexing for selection by position.
Panel.__iter__()	Iterate over infor axis
Panel.iteritems()	Iterate over (label, values) on info axis
Panel.pop(item)	Return item and drop from frame.
Panel.xs(key[, axis])	Return slice of panel along selected axis.
Panel.major_xs(key)	Return slice of panel along major axis.
Panel.minor_xs(key)	Return slice of panel along minor axis.

pandas.Panel.__iter__

Panel.**__iter__**()

Iterate over infor axis

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the *indexing documentation*.

6.6.6 Binary operator functions

Panel.add(other[, axis])	Addition of series and other, element-wise (binary operator *add*).
Panel.sub(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
Panel.mul(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
Panel.div(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
Panel.truediv(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
Panel.floordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
Panel.mod(other[, axis])	Modulo of series and other, element-wise (binary operator *mod*).
Panel.pow(other[, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
Panel.radd(other[, axis])	Addition of series and other, element-wise (binary operator *radd*).
Panel.rsub(other[, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
Panel.rmul(other[, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
Panel.rdiv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
Panel.rtruediv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
Panel.rfloordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
Panel.rmod(other[, axis])	Modulo of series and other, element-wise (binary operator *rmod*).

Continued on next page

Table 151 – continued from previous page

Panel.rpow(other[, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
Panel.lt(other[, axis])	Wrapper for comparison method lt
Panel.gt(other[, axis])	Wrapper for comparison method gt
Panel.le(other[, axis])	Wrapper for comparison method le
Panel.ge(other[, axis])	Wrapper for comparison method ge
Panel.ne(other[, axis])	Wrapper for comparison method ne
Panel.eq(other[, axis])	Wrapper for comparison method eq

6.6.7 Function application, GroupBy

Panel.apply(func[, axis])	Applies function along axis (or axes) of the Panel.
Panel.groupby(function[, axis])	Group data on given axis, returning GroupBy object.

6.6.8 Computations / Descriptive Stats

Panel.abs()	Return a Series/DataFrame with absolute numeric value of each element.
Panel.clip([lower, upper, axis, inplace])	Trim values at input threshold(s).
Panel.clip_lower(threshold[, axis, inplace])	(DEPRECATED) Trim values below a given threshold.
Panel.clip_upper(threshold[, axis, inplace])	(DEPRECATED) Trim values above a given threshold.
Panel.count([axis])	Return number of observations over requested axis.
Panel.cummax([axis, skipna])	Return cumulative maximum over a DataFrame or Series axis.
Panel.cummin([axis, skipna])	Return cumulative minimum over a DataFrame or Series axis.
Panel.cumprod([axis, skipna])	Return cumulative product over a DataFrame or Series axis.
Panel.cumsum([axis, skipna])	Return cumulative sum over a DataFrame or Series axis.
Panel.max([axis, skipna, level, numeric_only])	Return the maximum of the values for the requested axis.
Panel.mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis.
Panel.median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis.
Panel.min([axis, skipna, level, numeric_only])	Return the minimum of the values for the requested axis.
Panel.pct_change([periods, fill_method, . . .])	Percentage change between the current and a prior element.
Panel.prod([axis, skipna, level, . . .])	Return the product of the values for the requested axis.
Panel.sem([axis, skipna, level, ddof, . . .])	Return unbiased standard error of the mean over requested axis.
Panel.skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1.
Panel.sum([axis, skipna, level, . . .])	Return the sum of the values for the requested axis.
Panel.std([axis, skipna, level, ddof, . . .])	Return sample standard deviation over requested axis.
Panel.var([axis, skipna, level, ddof, . . .])	Return unbiased variance over requested axis.

6.6.9 Reindexing / Selection / Label manipulation

Panel.add_prefix(prefix)	Prefix labels with string *prefix*.
Panel.add_suffix(suffix)	Suffix labels with string *suffix*.
Panel.drop([labels, axis, index, columns, . . .])	
Panel.equals(other)	Test whether two objects contain the same elements.
Panel.filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
Panel.first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
Panel.last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
Panel.reindex(*args, **kwargs)	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
Panel.reindex_axis(labels[, axis, method, . . .])	(DEPRECATED) Conform input object to new index.
Panel.reindex_like(other[, method, copy, . . .])	Return an object with matching indices as other object.
Panel.rename([items, major_axis, minor_axis])	Alter axes input function or functions.
Panel.sample([n, frac, replace, weights, . . .])	Return a random sample of items from an axis of object.
Panel.select(crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria.
Panel.take(indices[, axis, convert, is_copy])	Return the elements in the given *positional* indices along an axis.
Panel.truncate([before, after, axis, copy])	Truncate a Series or DataFrame before and after some index value.

pandas.Panel.drop

Panel.**drop**(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

6.6.10 Missing data handling

Panel.dropna([axis, how, inplace])	Drop 2D from panel, holding passed axis constant.

6.6.11 Reshaping, sorting, transposing

Panel.sort_index([axis, level, ascending, . . .])	Sort object by labels (along an axis)
Panel.swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis
Panel.transpose(*args, **kwargs)	Permute the dimensions of the Panel
Panel.swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately.
Panel.conform(frame[, axis])	Conform input DataFrame to align with chosen axis pair.

6.6.12 Combining / joining / merging

Panel.join(other[, how, lsuffix, rsuffix])	Join items with other Panel either on major and minor axes column.

Continued on next page

Table 157 – continued from previous page

Panel.update(other[, join, overwrite, ...])	Modify Panel in place using non-NA values from other Panel.

6.6.13 Time series-related

Panel.asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
Panel.shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq.
Panel.resample(rule[, how, axis, ...])	Resample time-series data.
Panel.tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
Panel.tz_localize(tz[, axis, level, copy, ...])	Localize tz-naive index of a Series or DataFrame to target time zone.

6.6.14 Serialization / IO / Conversion

Panel.from_dict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects.
Panel.to_pickle(path[, compression, protocol])	Pickle (serialize) object to file.
Panel.to_excel(path[, na_rep, engine])	Write each DataFrame in Panel to a separate excel sheet.
Panel.to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
Panel.to_sparse(*args, **kwargs)	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
Panel.to_frame([filter_observations])	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
Panel.to_clipboard([excel, sep])	Copy object to the system clipboard.

6.7 Indexing

6.7.1 Index

Many of these methods or variants thereof are available on the objects that contain an index (Series/DataFrame) and those should most likely be used before calling these methods directly.

Index	Immutable ndarray implementing an ordered, sliceable set.

pandas.Index

class pandas.**Index**

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects.

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: object)] If dtype is None, we find the dtype that best fits the data. If an actual dtype is provided, we coerce to that dtype if it's safe. Otherwise, an error will be raised.

copy [bool] Make a copy of input ndarray

name [object] Name to be stored in the index

tupleize_cols [bool (default: True)] When True, attempt to create a MultiIndex if possible

See also:

RangeIndex Index implementing a monotonic integer range.

CategoricalIndex Index of *Categorical* s.

MultiIndex A multi-level, or hierarchical, Index.

IntervalIndex An Index of *Interval* s.

DatetimeIndex, *TimedeltaIndex*, *PeriodIndex*, *Int64Index*, *UInt64Index*, *Float64Index*

Notes

An Index instance can **only** contain hashable objects

Examples

```
>>> pd.Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')
```

```
>>> pd.Index(list('abc'))
Index(['a', 'b', 'c'], dtype='object')
```

Attributes

T	Return the transpose, which is by definition self.
array	The ExtensionArray of the data backing this Series or Index.
base	Return the base object if the memory of the underlying data is shared.
data	Return the data pointer of the underlying data.
dtype	Return the dtype object of the underlying data.
dtype_str	Return the dtype str of the underlying data.
flags	
hasnans	Return if I have any nans; enables various perf speedups.
inferred_type	Return a string of the type inferred from the values.
is_monotonic	Alias for is_monotonic_increasing.
is_monotonic_decreasing	Return if the index is monotonic decreasing (only equal or decreasing) values.

Continued on next page

Table 161 – continued from previous page

is_monotonic_increasing	Return if the index is monotonic increasing (only equal or increasing) values.
is_unique	Return if the index has unique values.
itemsize	Return the size of the dtype of the item of the underlying data.
nbytes	Return the number of bytes in the underlying data.
ndim	Number of dimensions of the underlying data, by definition 1.
shape	Return a tuple of the shape of the underlying data.
size	Return the number of elements in the underlying data.
strides	Return the strides of the underlying data.
values	Return an array representing the data in the Index.

pandas.Index.T

Index.**T**

Return the transpose, which is by definition self.

pandas.Index.array

Index.**array**

The ExtensionArray of the data backing this Series or Index.

New in version 0.24.0.

Returns

array [ExtensionArray] An ExtensionArray of the values stored within. For extension types, this is the actual array. For NumPy native types, this is a thin (no copy) wrapper around `numpy.ndarray`.

`.array` differs `.values` which may require converting the data to a different form.

See also:

Index.to_numpy Similar method that always returns a NumPy array.

Series.to_numpy Similar method that always returns a NumPy array.

Notes

This table lays out the different array types for each extension dtype within pandas.

dtype	array type
category	Categorical
period	PeriodArray
interval	IntervalArray
IntegerNA	IntegerArray
datetime64[ns, tz]	DatetimeArray

For any 3rd-party extension types, the array type will be an ExtensionArray.

For all remaining dtypes `.array` will be a `arrays.NumpyExtensionArray` wrapping the actual ndarray stored within. If you absolutely need a NumPy array (possibly with copying / coercing data), then use *Series.to_numpy()* instead.

Examples

For regular NumPy types like int, and float, a PandasArray is returned.

```
>>> pd.Series([1, 2, 3]).array
<PandasArray>
[1, 2, 3]
Length: 3, dtype: int64
```

For extension types, like Categorical, the actual ExtensionArray is returned

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.array
[a, b, a]
Categories (2, object): [a, b]
```

pandas.Index.base

`Index.`**`base`**
Return the base object if the memory of the underlying data is shared.

pandas.Index.data

`Index.`**`data`**
Return the data pointer of the underlying data.

pandas.Index.dtype

`Index.`**`dtype`**
Return the dtype object of the underlying data.

pandas.Index.dtype_str

`Index.`**`dtype_str`**
Return the dtype str of the underlying data.

pandas.Index.flags

`Index.`**`flags`**

pandas.Index.hasnans

`Index.`**`hasnans`**
Return if I have any nans; enables various perf speedups.

pandas.Index.inferred_type

Index.**inferred_type**
Return a string of the type inferred from the values.

pandas.Index.is_monotonic

Index.**is_monotonic**
Alias for is_monotonic_increasing.

pandas.Index.is_monotonic_decreasing

Index.**is_monotonic_decreasing**
Return if the index is monotonic decreasing (only equal or decreasing) values.

Examples

```
>>> Index([3, 2, 1]).is_monotonic_decreasing
True
>>> Index([3, 2, 2]).is_monotonic_decreasing
True
>>> Index([3, 1, 2]).is_monotonic_decreasing
False
```

pandas.Index.is_monotonic_increasing

Index.**is_monotonic_increasing**
Return if the index is monotonic increasing (only equal or increasing) values.

Examples

```
>>> Index([1, 2, 3]).is_monotonic_increasing
True
>>> Index([1, 2, 2]).is_monotonic_increasing
True
>>> Index([1, 3, 2]).is_monotonic_increasing
False
```

pandas.Index.is_unique

Index.**is_unique**
Return if the index has unique values.

pandas.Index.itemsize

Index.**itemsize**
Return the size of the dtype of the item of the underlying data.

pandas.Index.nbytes

`Index.nbytes`
Return the number of bytes in the underlying data.

pandas.Index.ndim

`Index.ndim`
Number of dimensions of the underlying data, by definition 1.

pandas.Index.shape

`Index.shape`
Return a tuple of the shape of the underlying data.

pandas.Index.size

`Index.size`
Return the number of elements in the underlying data.

pandas.Index.strides

`Index.strides`
Return the strides of the underlying data.

pandas.Index.values

`Index.values`
Return an array representing the data in the Index.

> **Warning:** We recommend using `Index.array` or `Index.to_numpy()`, depending on whether you need a reference to the underlying data or a NumPy array.

Returns

array: numpy.ndarray or ExtensionArray

See also:

Index.array Reference to the underlying data.

Index.to_numpy A NumPy array representing the underlying data.

`Return`

asi8	
empty	
has_duplicates	
is_all_dates	
name	
names	
nlevels	

Methods

`all`(*args, **kwargs)	Return whether all elements are True.
`any`(*args, **kwargs)	Return whether any element is True.
`append`(other)	Append a collection of Index options together.
`argmax`([axis, skipna])	Return a ndarray of the maximum argument indexer.
`argmin`([axis, skipna])	Return a ndarray of the minimum argument indexer.
`argsort`(*args, **kwargs)	Return the integer indices that would sort the index.
`asof`(label)	Return the label from the index, or, if not present, the previous one.
`asof_locs`(where, mask)	Finds the locations (indices) of the labels from the index for every entry in the *where* argument.
`astype`(dtype[, copy])	Create an Index with values cast to dtypes.
`contains`(key)	Return a boolean indicating whether the provided key is in the index.
`copy`([name, deep, dtype])	Make a copy of this object.
`delete`(loc)	Make new Index with passed location(-s) deleted.
`difference`(other[, sort])	Return a new Index with elements from the index that are not in *other*.
`drop`(labels[, errors])	Make new Index with passed list of labels deleted.
`drop_duplicates`([keep])	Return Index with duplicate values removed.
`droplevel`([level])	Return index with requested level(s) removed.
`dropna`([how])	Return Index without NA/NaN values
`duplicated`([keep])	Indicate duplicate index values.
`equals`(other)	Determines if two Index objects contain the same elements.
`factorize`([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
`fillna`([value, downcast])	Fill NA/NaN values with the specified value
`format`([name, formatter])	Render a string representation of the Index.
`get_duplicates`()	(DEPRECATED) Extract duplicated index elements.
`get_indexer`(target[, method, limit, tolerance])	Compute indexer and mask for new index given the current index.
`get_indexer_for`(target, **kwargs)	Guaranteed return of an indexer even when non-unique.
`get_indexer_non_unique`(target)	Compute indexer and mask for new index given the current index.
`get_level_values`(level)	Return an Index of values for requested level.
`get_loc`(key[, method, tolerance])	Get integer location, slice or boolean mask for requested label.
`get_slice_bound`(label, side, kind)	Calculate slice bound that corresponds to given label.

Continued on next page

Table 162 – continued from previous page

get_value(series, key)	Fast lookup of value from 1-dimensional ndarray.
get_values()	Return *Index* data as an *numpy.ndarray*.
groupby(values)	Group the index labels by a given array of values.
identical(other)	Similar to equals, but check that other comparable attributes are also equal.
insert(loc, item)	Make new Index inserting new item at location.
intersection(other[, sort])	Form the intersection of two Index objects.
*is_*(other)	More flexible, faster check like `is` but that works through views.
is_categorical()	Check if the Index holds categorical data.
isin(values[, level])	Return a boolean array where the index values are in *values*.
isna()	Detect missing values.
isnull()	Detect missing values.
item()	Return the first element of the underlying data as a python scalar.
join(other[, how, level, return_indexers, sort])	Compute join_index and indexers to conform data structures to the new index.
map(mapper[, na_action])	Map values using input correspondence (a dict, Series, or function).
max([axis, skipna])	Return the maximum value of the Index.
memory_usage([deep])	Memory usage of the values
min([axis, skipna])	Return the minimum value of the Index.
notna()	Detect existing (non-missing) values.
notnull()	Detect existing (non-missing) values.
nunique([dropna])	Return number of unique elements in the object.
putmask(mask, value)	Return a new Index of the values set with the mask.
ravel([order])	Return an ndarray of the flattened values of the underlying data.
reindex(target[, method, level, limit, …])	Create index with target's values (move/add/delete values as necessary).
rename(name[, inplace])	Alter Index or MultiIndex name.
repeat(repeats[, axis])	Repeat elements of a Index.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
set_names(names[, level, inplace])	Set Index or MultiIndex name.
set_value(arr, key, value)	Fast lookup of value from 1-dimensional ndarray.
shift([periods, freq])	Shift index by desired number of time frequency increments.
slice_indexer([start, end, step, kind])	For an ordered or unique index, compute the slice indexer for input labels and step.
slice_locs([start, end, step, kind])	Compute slice locations for input labels.
sort_values([return_indexer, ascending])	Return a sorted copy of the index.
sortlevel([level, ascending, sort_remaining])	For internal compatibility with with the Index API.
str	alias of `pandas.core.strings.StringMethods`
summary([name])	(DEPRECATED) Return a summarized representation.
symmetric_difference(other[, result_name, sort])	Compute the symmetric difference of two Index objects.

Continued on next page

Table 162 – continued from previous page

take(indices[, axis, allow_fill, fill_value])	Return a new Index of the values selected by the indices.
to_flat_index()	Identity method.
to_frame([index, name])	Create a DataFrame with a column containing the Index.
to_list()	Return a list of the values.
to_native_types([slicer])	Format specified values of *self* and return them.
to_numpy([dtype, copy])	A NumPy ndarray representing the values in this Series or Index.
to_series([index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
tolist()	Return a list of the values.
transpose(*args, **kwargs)	Return the transpose, which is by definition self.
union(other[, sort])	Form the union of two Index objects.
unique([level])	Return unique values in the index.
value_counts([normalize, sort, ascending, ...])	Return a Series containing counts of unique values.
where(cond[, other])	Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

pandas.Index.all

`Index.all(*args, **kwargs)`

Return whether all elements are True.

Parameters

***args** These parameters will be passed to numpy.all.

****kwargs** These parameters will be passed to numpy.all.

Returns

all [bool or array_like (if axis is specified)] A single element array_like may be converted to bool.

See also:

pandas.Index.any Return whether any element in an Index is True.

pandas.Series.any Return whether any element in a Series is True.

pandas.Series.all Return whether all elements in a Series are True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

all

True, because nonzero integers are considered True.

```
>>> pd.Index([1, 2, 3]).all()
True
```

False, because 0 is considered False.

```
>>> pd.Index([0, 1, 2]).all()
False
```

any

True, because 1 is considered True.

```
>>> pd.Index([0, 0, 1]).any()
True
```

False, because 0 is considered False.

```
>>> pd.Index([0, 0, 0]).any()
False
```

pandas.Index.any

Index.**any**(**args*, ***kwargs*)

Return whether any element is True.

Parameters

***args** These parameters will be passed to numpy.any.

****kwargs** These parameters will be passed to numpy.any.

Returns

any [bool or array_like (if axis is specified)] A single element array_like may be converted to bool.

See also:

pandas.Index.all Return whether all elements are True.

pandas.Series.all Return whether all elements are True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```
>>> index = pd.Index([0, 1, 2])
>>> index.any()
True
```

```
>>> index = pd.Index([0, 0, 0])
>>> index.any()
False
```

pandas.Index.append

`Index.append(other)`

Append a collection of Index options together.

Parameters

other [Index or list/tuple of indices]

Returns

appended [Index]

pandas.Index.argmax

`Index.argmax(axis=None, skipna=True)`

Return a ndarray of the maximum argument indexer.

Parameters

axis [{None}] Dummy argument for consistency with Series

skipna [bool, default True]

See also:

`numpy.ndarray.argmax`

pandas.Index.argmin

`Index.argmin(axis=None, skipna=True)`

Return a ndarray of the minimum argument indexer.

Parameters

axis [{None}] Dummy argument for consistency with Series

skipna [bool, default True]

See also:

`numpy.ndarray.argmin`

pandas.Index.argsort

`Index.argsort(*args, **kwargs)`

Return the integer indices that would sort the index.

Parameters

***args** Passed to *numpy.ndarray.argsort*.

****kwargs** Passed to *numpy.ndarray.argsort*.

Returns

numpy.ndarray Integer indices that would sort the index if used as an indexer.

See also:

`numpy.argsort` Similar method for NumPy arrays.

Index.sort_values Return sorted copy of Index.

Examples

```
>>> idx = pd.Index(['b', 'a', 'd', 'c'])
>>> idx
Index(['b', 'a', 'd', 'c'], dtype='object')
```

```
>>> order = idx.argsort()
>>> order
array([1, 0, 3, 2])
```

```
>>> idx[order]
Index(['a', 'b', 'c', 'd'], dtype='object')
```

pandas.Index.asof

Index.**asof**(*label*)

Return the label from the index, or, if not present, the previous one.

Assuming that the index is sorted, return the passed index label if it is in the index, or return the previous index label if the passed one is not in the index.

Parameters

label [object] The label up to which the method returns the latest index label.

Returns

object The passed label if it is in the index. The previous label if the passed label is not in the sorted index or *NaN* if there is no such label.

See also:

Series.asof Return the latest value in a Series up to the passed index.

merge_asof Perform an asof merge (similar to left join but it matches on nearest key rather than equal key).

Index.get_loc An *asof* is a thin wrapper around *get_loc* with method='pad'.

Examples

Index.asof returns the latest index label up to the passed label.

```
>>> idx = pd.Index(['2013-12-31', '2014-01-02', '2014-01-03'])
>>> idx.asof('2014-01-01')
'2013-12-31'
```

If the label is in the index, the method returns the passed label.

```
>>> idx.asof('2014-01-02')
'2014-01-02'
```

If all of the labels in the index are later than the passed label, NaN is returned.

```
>>> idx.asof('1999-01-02')
nan
```

If the index is not sorted, an error is raised.

```
>>> idx_not_sorted = pd.Index(['2013-12-31', '2015-01-02',
...                            '2014-01-03'])
>>> idx_not_sorted.asof('2013-12-31')
Traceback (most recent call last):
ValueError: index must be monotonic increasing or decreasing
```

pandas.Index.asof_locs

`Index.asof_locs(where, mask)`

Finds the locations (indices) of the labels from the index for every entry in the *where* argument.

As in the *asof* function, if the label (a particular entry in *where*) is not in the index, the latest index label upto the passed label is chosen and its index returned.

If all of the labels in the index are later than a label in *where*, -1 is returned.

mask is used to ignore NA values in the index during calculation.

Parameters

where [Index] An Index consisting of an array of timestamps.

mask [array-like] Array of booleans denoting where values in the original data are not NA.

Returns

numpy.ndarray An array of locations (indices) of the labels from the Index which correspond to the return values of the *asof* function for every element in *where*.

pandas.Index.astype

`Index.astype(dtype, copy=True)`

Create an Index with values cast to dtypes. The class of a new Index is determined by dtype. When conversion is impossible, a ValueError exception is raised.

Parameters

dtype [numpy dtype or pandas type] Note that any signed integer *dtype* is treated as `'int64'`, and any unsigned integer *dtype* is treated as `'uint64'`, regardless of the size.

copy [bool, default True] By default, astype always returns a newly allocated object. If copy is set to False and internal requirements on dtype are satisfied, the original data is used to create a new Index or the original Index is returned.

New in version 0.19.0.

pandas.Index.contains

Index.**contains**(*key*)

Return a boolean indicating whether the provided key is in the index.

Parameters

key [label] The key to check if it is present in the index.

Returns

bool Whether the key search is in the index.

See also:

Index.isin Returns an ndarray of boolean dtype indicating whether the list-like key is in the index.

Examples

```
>>> idx = pd.Index([1, 2, 3, 4])
>>> idx
Int64Index([1, 2, 3, 4], dtype='int64')
```

```
>>> idx.contains(2)
True
>>> idx.contains(6)
False
```

This is equivalent to:

```
>>> 2 in idx
True
>>> 6 in idx
False
```

pandas.Index.copy

Index.**copy**(*name=None*, *deep=False*, *dtype=None*, ***kwargs*)

Make a copy of this object. Name and dtype sets those attributes on the new object.

Parameters

name [string, optional]

deep [boolean, default False]

dtype [numpy dtype or pandas type]

Returns

copy [Index]

Notes

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy.

pandas.Index.delete

`Index.delete(loc)`

Make new Index with passed location(-s) deleted.

Returns

new_index [Index]

pandas.Index.difference

`Index.difference(other, sort=True)`

Return a new Index with elements from the index that are not in *other*.

This is the set difference of two Index objects.

Parameters

other [Index or array-like]

sort [bool, default True] Sort the resulting index if possible

New in version 0.24.0.

Returns

difference [Index]

Examples

```
>>> idx1 = pd.Index([2, 1, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
>>> idx1.difference(idx2, sort=False)
Int64Index([2, 1], dtype='int64')
```

pandas.Index.drop

`Index.drop(labels, errors='raise')`

Make new Index with passed list of labels deleted.

Parameters

labels [array-like]

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and existing labels are dropped.

Returns

dropped [Index]

Raises

KeyError If not all of the labels are found in the selected axis

pandas.Index.drop_duplicates

`Index.drop_duplicates(keep='first')`

Return Index with duplicate values removed.

Parameters

keep [{'first', 'last', `False`}, default 'first']

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- `False` : Drop all duplicates.

Returns

deduplicated [Index]

See also:

Series.drop_duplicates Equivalent method on Series.

DataFrame.drop_duplicates Equivalent method on DataFrame.

Index.duplicated Related method on Index, indicating duplicate Index values.

Examples

Generate an pandas.Index with duplicate values.

```
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'])
```

The *keep* parameter controls which duplicate values are removed. The value 'first' keeps the first occurrence for each set of duplicated entries. The default value of keep is 'first'.

```
>>> idx.drop_duplicates(keep='first')
Index(['lama', 'cow', 'beetle', 'hippo'], dtype='object')
```

The value 'last' keeps the last occurrence for each set of duplicated entries.

```
>>> idx.drop_duplicates(keep='last')
Index(['cow', 'beetle', 'lama', 'hippo'], dtype='object')
```

The value `False` discards all sets of duplicated entries.

```
>>> idx.drop_duplicates(keep=False)
Index(['cow', 'beetle', 'hippo'], dtype='object')
```

pandas.Index.droplevel

`Index.droplevel(level=0)`

Return index with requested level(s) removed.

If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

New in version 0.23.1: (support for non-MultiIndex)

Parameters

level [int, str, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

Returns

index [Index or MultiIndex]

pandas.Index.dropna

`Index.dropna(how='any')`

Return Index without NA/NaN values

Parameters

how [{'any', 'all'}, default 'any'] If the Index is a MultiIndex, drop the value when any or all levels are NaN.

Returns

valid [Index]

pandas.Index.duplicated

`Index.duplicated(keep='first')`

Indicate duplicate index values.

Duplicated values are indicated as `True` values in the resulting array. Either all duplicates, all except the first, or all except the last occurrence of duplicates can be indicated.

Parameters

keep [{'first', 'last', False}, default 'first'] The value or values in a set of duplicates to mark as missing.

- 'first' : Mark duplicates as `True` except for the first occurrence.
- 'last' : Mark duplicates as `True` except for the last occurrence.
- `False` : Mark all duplicates as `True`.

Returns

numpy.ndarray

See also:

pandas.Series.duplicated Equivalent method on pandas.Series.

pandas.DataFrame.duplicated Equivalent method on pandas.DataFrame.

pandas.Index.drop_duplicates Remove duplicate values from Index.

Examples

By default, for each set of duplicated values, the first occurrence is set to False and all others to True:

```
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> idx.duplicated()
array([False, False,  True, False,  True])
```

which is equivalent to

```
>>> idx.duplicated(keep='first')
array([False, False,  True, False,  True])
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```
>>> idx.duplicated(keep='last')
array([ True, False,  True, False, False])
```

By setting keep on `False`, all duplicates are True:

```
>>> idx.duplicated(keep=False)
array([ True, False,  True, False,  True])
```

pandas.Index.equals

`Index.`**`equals`**(*other*)

Determines if two Index objects contain the same elements.

pandas.Index.factorize

`Index.`**`factorize`**(*sort=False*, *na_sentinel=-1*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

Parameters

sort [boolean, default False] Sort *uniques* and shuffle *labels* to maintain the relationship.

na_sentinel [int, default -1] Value to mark "not found".

Returns

labels [ndarray] An integer ndarray that's an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

uniques [ndarray, Index, or Categorical] The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

Note: Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

See also:

`cut` Discretize continuous-valued array.

`unique` Find the unique value in an array.

Examples

These examples all show factorize as a top-level method like `pd.factorize(values)`. The results are identical for methods like *Series.factorize()*.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na_sentinel* (`-1` by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that `'b'` is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

pandas.Index.fillna

`Index.`**`fillna`**(*value=None*, *downcast=None*)

Fill NA/NaN values with the specified value

Parameters

value [scalar] Scalar value to use to fill holes (e.g. 0). This value cannot be a list-likes.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns

filled [Index]

pandas.Index.format

`Index.format(name=False, formatter=None, **kwargs)`

Render a string representation of the Index.

pandas.Index.get_duplicates

`Index.get_duplicates()`

Extract duplicated index elements.

Deprecated since version 0.23.0: Use idx[idx.duplicated()].unique() instead

Returns a sorted list of index elements which appear more than once in the index.

Returns

array-like List of duplicated indexes.

See also:

Index.duplicated Return boolean array denoting duplicates.

Index.drop_duplicates Return Index with duplicates removed.

Examples

Works on different Index of types.

```
>>> pd.Index([1, 2, 2, 3, 3, 3, 4]).get_duplicates()  # doctest: +SKIP
[2, 3]
```

Note that for a DatetimeIndex, it does not return a list but a new DatetimeIndex:

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-02', '2018-01-03',
...                         '2018-01-03', '2018-01-04', '2018-01-04'],
...                        format='%Y-%m-%d')
>>> pd.Index(dates).get_duplicates()  # doctest: +SKIP
DatetimeIndex(['2018-01-03', '2018-01-04'],
              dtype='datetime64[ns]', freq=None)
```

Sorts duplicated elements even when indexes are unordered.

```
>>> pd.Index([1, 2, 3, 2, 3, 4, 3]).get_duplicates()  # doctest: +SKIP
[2, 3]
```

Return empty array-like structure when all elements are unique.

```
>>> pd.Index([1, 2, 3, 4]).get_duplicates()  # doctest: +SKIP
[]
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-02', '2018-01-03'],
...                        format='%Y-%m-%d')
>>> pd.Index(dates).get_duplicates()  # doctest: +SKIP
DatetimeIndex([], dtype='datetime64[ns]', freq=None)
```

pandas.Index.get_indexer

`Index.get_indexer`(*target*, *method=None*, *limit=None*, *tolerance=None*)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [Index]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit [int, optional] Maximum number of consecutive labels in `target` to match for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

pandas.Index.get_indexer_for

`Index.get_indexer_for(target, **kwargs)`

Guaranteed return of an indexer even when non-unique.

This dispatches to get_indexer or get_indexer_nonunique as appropriate.

pandas.Index.get_indexer_non_unique

`Index.get_indexer_non_unique(target)`

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [Index]

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

missing [ndarray of int] An indexer into the target of the values not found. These correspond to the -1 in the indexer array

pandas.Index.get_level_values

`Index.get_level_values(level)`

Return an Index of values for requested level.

This is primarily useful to get an individual level of values from a MultiIndex, but is provided on Index as well for compatability.

Parameters

level [int or str] It is either the integer position or the name of the level.

Returns

values [Index] Calling object, as there is only one level in the Index.

See also:

MultiIndex.get_level_values Get values for a level of a MultiIndex.

Notes

For Index, level should be 0, since there are no multiple levels.

Examples

```
>>> idx = pd.Index(list('abc'))
>>> idx
Index(['a', 'b', 'c'], dtype='object')
```

Get level values by supplying *level* as integer:

```
>>> idx.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object')
```

pandas.Index.get_loc

`Index.get_loc(key, method=None, tolerance=None)`
Get integer location, slice or boolean mask for requested label.

Parameters

key [label]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

tolerance [optional] Maximum distance from index value for inexact matches. The value of the index at the matching location most satisfy the equation `abs(index[loc] - key) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

loc [int if unique index, slice if monotonic index, else mask]

Examples

```
>>> unique_index = pd.Index(list('abc'))
>>> unique_index.get_loc('b')
1
```

```
>>> monotonic_index = pd.Index(list('abbc'))
>>> monotonic_index.get_loc('b')
slice(1, 3, None)
```

```
>>> non_monotonic_index = pd.Index(list('abcb'))
>>> non_monotonic_index.get_loc('b')
array([False,  True, False,  True], dtype=bool)
```

pandas.Index.get_slice_bound

`Index.get_slice_bound(label, side, kind)`
Calculate slice bound that corresponds to given label.

Returns leftmost (one-past-the-rightmost if `side=='right'`) position of given label.

Parameters

label [object]

side [{'left', 'right'}]

kind [{'ix', 'loc', 'getitem'}]

pandas.Index.get_value

`Index.get_value(series, key)`
Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing.

pandas.Index.get_values

`Index.get_values()`
Return *Index* data as an *numpy.ndarray*.

Returns

numpy.ndarray A one-dimensional numpy array of the *Index* values.

See also:

Index.values The attribute that get_values wraps.

Examples

Getting the *Index* values of a *DataFrame*:

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
...                     index=['a', 'b', 'c'], columns=['A', 'B', 'C'])
>>> df
   A  B  C
a  1  2  3
b  4  5  6
c  7  8  9
>>> df.index.get_values()
array(['a', 'b', 'c'], dtype=object)
```

Standalone *Index* values:

```
>>> idx = pd.Index(['1', '2', '3'])
>>> idx.get_values()
array(['1', '2', '3'], dtype=object)
```

MultiIndex arrays also have only one dimension:

```
>>> midx = pd.MultiIndex.from_arrays([[1, 2, 3], ['a', 'b', 'c']],
...                                  names=('number', 'letter'))
>>> midx.get_values()
array([(1, 'a'), (2, 'b'), (3, 'c')], dtype=object)
>>> midx.get_values().ndim
1
```

pandas.Index.groupby

`Index.groupby(values)`

Group the index labels by a given array of values.

Parameters

values [array] Values used to determine the groups.

Returns

groups [dict] {group name -> group labels}

pandas.Index.identical

`Index.identical(other)`

Similar to equals, but check that other comparable attributes are also equal.

pandas.Index.insert

`Index.insert(loc, item)`

Make new Index inserting new item at location.

Follows Python list.append semantics for negative values.

Parameters

loc [int]

item [object]

Returns

new_index [Index]

pandas.Index.intersection

`Index.intersection(other, sort=True)`

Form the intersection of two Index objects.

This returns a new Index with elements common to the index and *other*.

Parameters

other [Index or array-like]

sort [bool, default True] Sort the resulting index if possible

New in version 0.24.0.

Returns

intersection [Index]

Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.intersection(idx2)
Int64Index([3, 4], dtype='int64')
```

pandas.Index.is_

`Index.is_(other)`

More flexible, faster check like `is` but that works through views.

Note: this is *not* the same as `Index.identical()`, which checks that metadata is also the same.

Parameters

other [object] other object to compare against.

Returns

True if both have same underlying data, False otherwise [bool]

pandas.Index.is_categorical

`Index.is_categorical()`

Check if the Index holds categorical data.

Returns

boolean True if the Index is categorical.

See also:

CategoricalIndex Index for categorical data.

Examples

```
>>> idx = pd.Index(["Watermelon", "Orange", "Apple",
...                 "Watermelon"]).astype("category")
>>> idx.is_categorical()
True
```

```
>>> idx = pd.Index([1, 3, 5, 7])
>>> idx.is_categorical()
False
```

```
>>> s = pd.Series(["Peter", "Victor", "Elisabeth", "Mar"])
>>> s
0        Peter
1       Victor
```

(continues on next page)

(continued from previous page)

```
2    Elisabeth
3          Mar
dtype: object
>>> s.index.is_categorical()
False
```

pandas.Index.isin

Index.**isin**(*values*, *level=None*)

Return a boolean array where the index values are in *values*.

Compute boolean array of whether each index value is found in the passed set of values. The length of the returned boolean array matches the length of the index.

Parameters

values [set or list-like] Sought values.

New in version 0.18.1: Support for values as a set.

level [str or int, optional] Name or position of the index level to use (if the index is a *MultiIndex*).

Returns

is_contained [ndarray] NumPy array of boolean values.

See also:

Series.isin Same for Series.

DataFrame.isin Same method for DataFrames.

Notes

In the case of *MultiIndex* you must either specify *values* as a list-like object containing tuples that are the same length as the number of levels, or specify *level*. Otherwise it will raise a ValueError.

If *level* is specified:

- if it is the name of one *and only one* index level, use that level;
- otherwise it should be a number indicating level position.

Examples

```
>>> idx = pd.Index([1,2,3])
>>> idx
Int64Index([1, 2, 3], dtype='int64')
```

Check whether each index value in a list of values. >>> idx.isin([1, 4]) array([True, False, False])

```
>>> midx = pd.MultiIndex.from_arrays([[1,2,3],
...                                  ['red', 'blue', 'green']],
...                                  names=('number', 'color'))
>>> midx
```

(continues on next page)

(continued from previous page)

```
MultiIndex(levels=[[1, 2, 3], ['blue', 'green', 'red']],
           codes=[[0, 1, 2], [2, 0, 1]],
           names=['number', 'color'])
```

Check whether the strings in the 'color' level of the MultiIndex are in a list of colors.

```
>>> midx.isin(['red', 'orange', 'yellow'], level='color')
array([ True, False, False])
```

To check across the levels of a MultiIndex, pass a list of tuples:

```
>>> midx.isin([(1, 'red'), (3, 'red')])
array([ True, False, False])
```

For a DatetimeIndex, string values in *values* are converted to Timestamps.

```
>>> dates = ['2000-03-11', '2000-03-12', '2000-03-13']
>>> dti = pd.to_datetime(dates)
>>> dti
DatetimeIndex(['2000-03-11', '2000-03-12', '2000-03-13'],
dtype='datetime64[ns]', freq=None)
```

```
>>> dti.isin(['2000-03-11'])
array([ True, False, False])
```

pandas.Index.isna

`Index.`**`isna`**`()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

New in version 0.20.0.

Returns

numpy.ndarray A boolean array of whether my values are NA

See also:

`pandas.Index.notna` Boolean inverse of isna.

`pandas.Index.dropna` Omit entries with missing values.

`pandas.isna` Top-level isna.

`Series.isna` Detect missing values in Series object.

Examples

Show which entries in a pandas.Index are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, *NaT* (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                         pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

pandas.Index.isnull

`Index.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

New in version 0.20.0.

Returns

numpy.ndarray A boolean array of whether my values are NA

See also:

pandas.Index.notna Boolean inverse of isna.

pandas.Index.dropna Omit entries with missing values.

pandas.isna Top-level isna.

Series.isna Detect missing values in Series object.

Examples

Show which entries in a pandas.Index are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, *NaT* (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                         pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

pandas.Index.item

Index.**item**()

Return the first element of the underlying data as a python scalar.

pandas.Index.join

Index.**join**(*other*, *how='left'*, *level=None*, *return_indexers=False*, *sort=False*)

Compute join_index and indexers to conform data structures to the new index.

Parameters

other [Index]

how [{'left', 'right', 'inner', 'outer'}]

level [int or level name, default None]

return_indexers [boolean, default False]

sort [boolean, default False] Sort the join keys lexicographically in the result Index. If False, the order of the join keys depends on the join type (how keyword)

New in version 0.20.0.

Returns

join_index, (left_indexer, right_indexer)

pandas.Index.map

Index.**map**(*mapper*, *na_action=None*)

Map values using input correspondence (a dict, Series, or function).

Parameters

mapper [function, dict, or Series] Mapping correspondence.

na_action [{None, 'ignore'}] If 'ignore', propagate NA values, without passing them to the mapping correspondence.

Returns

applied [Union[Index, MultiIndex], inferred] The output of the mapping function applied to the index. If the function returns a tuple with more than one element a MultiIndex will be returned.

pandas.Index.max

Index.**max**(*axis=None*, *skipna=True*)

Return the maximum value of the Index.

Parameters

axis [int, optional] For compatibility with NumPy. Only 0 or None are allowed.

skipna [bool, default True]

Returns

scalar Maximum value.

See also:

Index.min Return the minimum value in an Index.

Series.max Return the maximum value in a Series.

DataFrame.max Return the maximum values in a DataFrame.

Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.max()
3
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.max()
('b', 2)
```

pandas.Index.memory_usage

`Index.memory_usage(deep=False)`
Memory usage of the values

Parameters

deep [bool] Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption

Returns

bytes used

See also:

`numpy.ndarray.nbytes`

Notes

Memory usage does not include memory consumed by elements that are not components of the array if deep=False or if used on PyPy

pandas.Index.min

`Index.min(axis=None, skipna=True)`
Return the minimum value of the Index.

Parameters

axis [{None}] Dummy argument for consistency with Series

skipna [bool, default True]

Returns

scalar Minimum value.

See also:

`Index.max` Return the maximum value of the object.

`Series.min` Return the minimum value in a Series.

`DataFrame.min` Return the minimum values in a DataFrame.

Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.min()
1
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```

For a MultiIndex, the minimum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.min()
('a', 1)
```

pandas.Index.notna

`Index.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to `False` values.

New in version 0.20.0.

Returns

numpy.ndarray Boolean array to indicate which entries are not NA.

See also:

Index.notnull Alias of notna.

Index.isna Inverse of notna.

pandas.notna Top-level notna.

Examples

Show which entries in an Index are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

pandas.Index.notnull

`Index.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to `False` values.

New in version 0.20.0.

Returns

numpy.ndarray Boolean array to indicate which entries are not NA.

See also:

Index.notnull Alias of notna.

Index.isna Inverse of notna.

pandas.notna Top-level notna.

Examples

Show which entries in an Index are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

pandas.Index.nunique

`Index.`**`nunique`**(*dropna=True*)

Return number of unique elements in the object.

Excludes NA values by default.

Parameters

dropna [boolean, default True] Don't include NaN in the count.

Returns

nunique [int]

pandas.Index.putmask

`Index.`**`putmask`**(*mask*, *value*)

Return a new Index of the values set with the mask.

See also:

`numpy.ndarray.putmask`

pandas.Index.ravel

`Index.ravel(order='C')`

Return an ndarray of the flattened values of the underlying data.

See also:

`numpy.ndarray.ravel`

pandas.Index.reindex

`Index.reindex(target, method=None, level=None, limit=None, tolerance=None)`

Create index with target's values (move/add/delete values as necessary).

Parameters

target [an iterable]

Returns

new_index [pd.Index] Resulting index

indexer [np.ndarray or None] Indices of output values in original index

pandas.Index.rename

`Index.rename(name, inplace=False)`

Alter Index or MultiIndex name.

Able to set new names without level. Defaults to returning new index. Length of names must match number of levels in MultiIndex.

Parameters

name [label or list of labels] Name(s) to set.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

Returns

Index The same type as the caller or None if inplace is True.

See also:

Index.set_names Able to set new names partially and by level.

Examples

```
>>> idx = pd.Index(['A', 'C', 'A', 'B'], name='score')
>>> idx.rename('grade')
Index(['A', 'C', 'A', 'B'], dtype='object', name='grade')
```

```
>>> idx = pd.MultiIndex.from_product([['python', 'cobra'],
...                                   [2018, 2019]],
...                                   names=['kind', 'year'])
>>> idx
```

(continues on next page)

(continued from previous page)

```
MultiIndex(levels=[['cobra', 'python'], [2018, 2019]],
           codes=[[1, 1, 0, 0], [0, 1, 0, 1]],
           names=['kind', 'year'])
>>> idx.rename(['species', 'year'])
MultiIndex(levels=[['cobra', 'python'], [2018, 2019]],
           codes=[[1, 1, 0, 0], [0, 1, 0, 1]],
           names=['species', 'year'])
>>> idx.rename('species')
Traceback (most recent call last):
TypeError: Must pass list-like as `names`.
```

pandas.Index.repeat

Index.**repeat**(*repeats*, *axis=None*)

Repeat elements of a Index.

Returns a new Index where each element of the current Index is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Index.

axis [None] Must be None. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_index [Index] Newly created Index with repeated elements.

See also:

Series.repeat Equivalent function for Series.

numpy.repeat Similar method for numpy.ndarray.

Examples

```
>>> idx = pd.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.repeat(2)
Index(['a', 'a', 'b', 'b', 'c', 'c'], dtype='object')
>>> idx.repeat([1, 2, 3])
Index(['a', 'b', 'b', 'c', 'c', 'c'], dtype='object')
```

pandas.Index.searchsorted

Index.**searchsorted**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted IndexOpsMixin *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns

int or array of int A scalar or array of insertion points with the same shape as *value*.

Changed in version 0.24.0: If *value* is a scalar, an int is now always returned. Previously, scalar inputs returned an 1-item array for *Series* and *Categorical*.

See also:

`numpy.searchsorted`

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
3
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',
                        'cheese', 'milk'], ordered=True)
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
1
```

```
>>> x.searchsorted(['bread'], side='right')
array([3])
```

pandas.Index.set_names

`Index.set_names(names, level=None, inplace=False)`

Set Index or MultiIndex name.

Able to set new names partially and by level.

Parameters

names [label or list of label] Name(s) to set.

level [int, label or list of int or label, optional] If the index is a MultiIndex, level(s) to set (None for all levels). Otherwise level must be None.

inplace [bool, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

Returns

Index The same type as the caller or None if inplace is True.

See also:

Index.rename Able to set new names without level.

Examples

```
>>> idx = pd.Index([1, 2, 3, 4])
>>> idx
Int64Index([1, 2, 3, 4], dtype='int64')
>>> idx.set_names('quarter')
Int64Index([1, 2, 3, 4], dtype='int64', name='quarter')
```

```
>>> idx = pd.MultiIndex.from_product([['python', 'cobra'],
...                                   [2018, 2019]])
>>> idx
MultiIndex(levels=[['cobra', 'python'], [2018, 2019]],
           codes=[[1, 1, 0, 0], [0, 1, 0, 1]])
>>> idx.set_names(['kind', 'year'], inplace=True)
>>> idx
MultiIndex(levels=[['cobra', 'python'], [2018, 2019]],
           codes=[[1, 1, 0, 0], [0, 1, 0, 1]],
           names=['kind', 'year'])
>>> idx.set_names('species', level=0)
MultiIndex(levels=[['cobra', 'python'], [2018, 2019]],
           codes=[[1, 1, 0, 0], [0, 1, 0, 1]],
           names=['species', 'year'])
```

pandas.Index.set_value

`Index.set_value(arr, key, value)`

Fast lookup of value from 1-dimensional ndarray.

Notes

Only use this if you know what you're doing.

pandas.Index.shift

`Index.shift(periods=1, freq=None)`

Shift index by desired number of time frequency increments.

This method is for shifting the values of datetime-like indexes by a specified time increment a given number of times.

Parameters

periods [int, default 1] Number of periods (or increments) to shift by, can be positive or negative.

freq [pandas.DateOffset, pandas.Timedelta or string, optional] Frequency increment to shift by. If None, the index is shifted by its own *freq* attribute. Offset aliases are valid strings, e.g., 'D', 'W', 'M' etc.

Returns

pandas.Index shifted index

See also:

Series.shift Shift values of Series.

Notes

This method is only implemented for datetime-like index classes, i.e., DatetimeIndex, PeriodIndex and TimedeltaIndex.

Examples

Put the first 5 month starts of 2011 into an index.

```
>>> month_starts = pd.date_range('1/1/2011', periods=5, freq='MS')
>>> month_starts
DatetimeIndex(['2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-05-01'],
              dtype='datetime64[ns]', freq='MS')
```

Shift the index by 10 days.

```
>>> month_starts.shift(10, freq='D')
DatetimeIndex(['2011-01-11', '2011-02-11', '2011-03-11', '2011-04-11',
               '2011-05-11'],
              dtype='datetime64[ns]', freq=None)
```

The default value of *freq* is the *freq* attribute of the index, which is 'MS' (month start) in this example.

```
>>> month_starts.shift(10)
DatetimeIndex(['2011-11-01', '2011-12-01', '2012-01-01', '2012-02-01',
               '2012-03-01'],
              dtype='datetime64[ns]', freq='MS')
```

pandas.Index.slice_indexer

Index.**slice_indexer**(*start=None*, *end=None*, *step=None*, *kind=None*)
For an ordered or unique index, compute the slice indexer for input labels and step.

Parameters

start [label, default None] If None, defaults to the beginning

end [label, default None] If None, defaults to the end

step [int, default None]

kind [string, default None]

Returns

indexer [slice]

Raises

KeyError [If key does not exist, or key is not unique and index is] not ordered.

Notes

This function assumes that the data is sorted, so use at your own peril

Examples

This is a method on all index types. For example you can do:

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_indexer(start='b', end='c')
slice(1, 3)
```

```
>>> idx = pd.MultiIndex.from_arrays([list('abcd'), list('efgh')])
>>> idx.slice_indexer(start='b', end=('c', 'g'))
slice(1, 3)
```

pandas.Index.slice_locs

Index.**slice_locs**(*start=None*, *end=None*, *step=None*, *kind=None*)
Compute slice locations for input labels.

Parameters

start [label, default None] If None, defaults to the beginning

end [label, default None] If None, defaults to the end

step [int, defaults None] If None, defaults to 1

kind [{'ix', 'loc', 'getitem'} or None]

Returns

start, end [int]

See also:

Index.get_loc Get location for a single label.

Notes

This method only works if the index is monotonic or unique.

Examples

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

pandas.Index.sort_values

Index.**sort_values**(*return_indexer=False*, *ascending=True*)
Return a sorted copy of the index.

Return a sorted copy of the index, and optionally return the indices that sorted the index itself.

Parameters

return_indexer [bool, default False] Should the indices that would sort the index be returned.

ascending [bool, default True] Should the index values be sorted in an ascending order.

Returns

sorted_index [pandas.Index] Sorted copy of the index.

indexer [numpy.ndarray, optional] The indices that the index itself was sorted by.

See also:

pandas.Series.sort_values Sort values of a Series.

pandas.DataFrame.sort_values Sort values in a DataFrame.

Examples

```
>>> idx = pd.Index([10, 100, 1, 1000])
>>> idx
Int64Index([10, 100, 1, 1000], dtype='int64')
```

Sort values in ascending order (default behavior).

```
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order, and also get the indices *idx* was sorted by.

```
>>> idx.sort_values(ascending=False, return_indexer=True)
(Int64Index([1000, 100, 10, 1], dtype='int64'), array([3, 1, 0, 2]))
```

pandas.Index.sortlevel

`Index.sortlevel(level=None, ascending=True, sort_remaining=None)`

For internal compatibility with with the Index API.

Sort the Index. This is for compat with MultiIndex

Parameters

ascending [boolean, default True] False to sort in descending order

level, sort_remaining are compat parameters

Returns

sorted_index [Index]

pandas.Index.str

`Index.str()`

Vectorized string functions for Series and Index. NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

Examples

```
>>> s.str.split('_')
>>> s.str.replace('_', '')
```

pandas.Index.summary

`Index.summary(name=None)`

Return a summarized representation.

Deprecated since version 0.23.0.

pandas.Index.symmetric_difference

`Index.symmetric_difference(other, result_name=None, sort=True)`

Compute the symmetric difference of two Index objects.

Parameters

other [Index or array-like]

result_name [str]

sort [bool, default True] Sort the resulting index if possible

New in version 0.24.0.

Returns

symmetric_difference [Index]

Notes

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the ^ operator:

```
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

pandas.Index.take

`Index.take(indices, axis=0, allow_fill=True, fill_value=None, **kwargs)`

Return a new Index of the values selected by the indices.

For internal compatibility with numpy arrays.

Parameters

indices [list] Indices to be taken

axis [int, optional] The axis over which to select values, always 0.

allow_fill [bool, default True]

fill_value [bool, default None] If allow_fill=True and fill_value is not None, indices specified by -1 is regarded as NA. If Index doesn't hold NA, raise ValueError

See also:

`numpy.ndarray.take`

pandas.Index.to_flat_index

`Index.to_flat_index()`

Identity method.

New in version 0.24.0.

This is implemented for compatability with subclass implementations when chaining.

Returns

pd.Index Caller.

See also:

`MultiIndex.to_flat_index` Subclass implementation.

pandas.Index.to_frame

`Index.to_frame(index=True, name=None)`

Create a DataFrame with a column containing the Index.

New in version 0.24.0.

Parameters

index [boolean, default True] Set the index of the returned DataFrame as the original Index.

name [object, default None] The passed name should substitute for the index name (if it has one).

Returns

DataFrame DataFrame containing the original Index data.

See also:

Index.to_series Convert an Index to a Series.

Series.to_frame Convert Series to DataFrame.

Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
       animal
animal
Ant       Ant
Bear     Bear
Cow       Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
    animal
0   Ant
1  Bear
2   Cow
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(index=False, name='zoo')
    zoo
0   Ant
1  Bear
2   Cow
```

pandas.Index.to_list

`Index.to_list()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

See also:

`numpy.ndarray.tolist`

pandas.Index.to_native_types

`Index.to_native_types(slicer=None, **kwargs)`

Format specified values of *self* and return them.

Parameters

slicer [int, array-like] An indexer into *self* that specifies which values are used in the formatting process.

kwargs [dict] Options for specifying how the values should be formatted. These options include the following:

1) **na_rep** [str] The value that serves as a placeholder for NULL values
2) **quoting** [bool or None] Whether or not there are quoted values in *self*
3) **date_format** [str] The format used to represent date-like values

pandas.Index.to_numpy

`Index.to_numpy(dtype=None, copy=False)`

A NumPy ndarray representing the values in this Series or Index.

New in version 0.24.0.

Parameters

dtype [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`

copy [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

Returns

numpy.ndarray

See also:

`Series.array` Get the actual data stored within.

`Index.array` Get the actual data stored within.

`DataFrame.to_numpy` Similar method for DataFrame.

Notes

The returned array will be the same up to equality (values equal in *self* will be equal in the returned array; likewise for values that are not equal). When *self* contains an ExtensionArray, the dtype may be different. For example, for a category-dtype Series, `to_numpy()` will return a NumPy array and the categorical dtype will be lost.

For NumPy dtypes, this will be a reference to the actual data stored in this Series or Index (assuming `copy=False`). Modifying the result in place will modify the data stored in the Series or Index (not that we recommend doing that).

For extension types, `to_numpy()` *may* require copying data and coercing the result to a NumPy type (possibly object), which may be expensive. When you need a no-copy reference to the underlying data, `Series.array` should be used instead.

This table lays out the different dtypes and default return types of `to_numpy()` for various dtypes within pandas.

dtype	array type
category[T]	ndarray[T] (same dtype as input)
period	ndarray[object] (Periods)
interval	ndarray[object] (Intervals)
IntegerNA	ndarray[object]
datetime64[ns]	datetime64[ns]
datetime64[ns, tz]	ndarray[object] (Timestamps)

Examples

```
>>> ser = pd.Series(pd.Categorical(['a', 'b', 'a']))
>>> ser.to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

Specify the *dtype* to control how datetime-aware data is represented. Use `dtype=object` to return an ndarray of pandas `Timestamp` objects, each with the correct `tz`.

```
>>> ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))
>>> ser.to_numpy(dtype=object)
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or `dtype='datetime64[ns]'` to return an ndarray of native datetime64 values. The values are converted to UTC and the timezone info is dropped.

```
>>> ser.to_numpy(dtype="datetime64[ns]")
... # doctest: +ELLIPSIS
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00...'],
      dtype='datetime64[ns]')
```

pandas.Index.to_series

`Index.`**`to_series`**`(`*index=None*, *name=None*`)`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.

Parameters

index [Index, optional] index of resulting Series. If None, defaults to original index

name [string, optional] name of resulting Series. If None, defaults to name of original index

Returns

Series [dtype will be based on the type of the Index values.]

pandas.Index.tolist

`Index.tolist()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

See also:

`numpy.ndarray.tolist`

pandas.Index.transpose

`Index.transpose(*args, **kwargs)`

Return the transpose, which is by definition self.

pandas.Index.union

`Index.union(other, sort=True)`

Form the union of two Index objects.

Parameters

other [Index or array-like]

sort [bool, default True] Sort the resulting index if possible

New in version 0.24.0.

Returns

union [Index]

Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.union(idx2)
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

pandas.Index.unique

`Index.unique(level=None)`

Return unique values in the index. Uniques are returned in order of appearance, this does NOT sort.

Parameters

level [int or str, optional, default None] Only return values from specified level (for MultiIndex)

New in version 0.23.0.

Returns

Index without duplicates

See also:

unique, *Series.unique*

pandas.Index.value_counts

`Index.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)`
Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters

normalize [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.

sort [boolean, default True] Sort by values.

ascending [boolean, default False] Sort in ascending order.

bins [integer, optional] Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data.

dropna [boolean, default True] Don't include counts of NaN.

Returns

counts [Series]

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.count Number of non-NA elements in a DataFrame.

Examples

```
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0    2
4.0    1
2.0    1
1.0    1
dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])
>>> s.value_counts(normalize=True)
3.0    0.4
4.0    0.2
2.0    0.2
1.0    0.2
dtype: float64
```

bins

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

```
>>> s.value_counts(bins=3)
(2.0, 3.0]      2
(0.996, 2.0]    2
(3.0, 4.0]      1
dtype: int64
```

dropna

With *dropna* set to *False* we can also see NaN index values.

```
>>> s.value_counts(dropna=False)
3.0    2
NaN    1
4.0    1
2.0    1
1.0    1
dtype: int64
```

pandas.Index.where

`Index.where(cond, other=None)`

Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

New in version 0.19.0.

Parameters

cond [boolean array-like with the same length as self]

other [scalar, or array-like]

holds_integer	
is_boolean	
is_floating	
is_integer	
is_interval	
is_lexsorted_for_tuple	
is_mixed	
is_numeric	
is_object	
is_type_compatible	
sort	
view	

Properties

Index.values	Return an array representing the data in the Index.
Index.is_monotonic	Alias for is_monotonic_increasing.
Index.is_monotonic_increasing	Return if the index is monotonic increasing (only equal or increasing) values.

Continued on next page

Table 163 – continued from previous page

Index.is_monotonic_decreasing	Return if the index is monotonic decreasing (only equal or decreasing) values.
Index.is_unique	Return if the index has unique values.
Index.has_duplicates	
Index.hasnans	Return if I have any nans; enables various perf speedups.
Index.dtype	Return the dtype object of the underlying data.
Index.dtype_str	Return the dtype str of the underlying data.
Index.inferred_type	Return a string of the type inferred from the values.
Index.is_all_dates	
Index.shape	Return a tuple of the shape of the underlying data.
Index.name	
Index.names	
Index.nbytes	Return the number of bytes in the underlying data.
Index.ndim	Number of dimensions of the underlying data, by definition 1.
Index.size	Return the number of elements in the underlying data.
Index.empty	
Index.strides	Return the strides of the underlying data.
Index.itemsize	Return the size of the dtype of the item of the underlying data.
Index.base	Return the base object if the memory of the underlying data is shared.
Index.T	Return the transpose, which is by definition self.
Index.memory_usage([deep])	Memory usage of the values

pandas.Index.has_duplicates

Index.**has_duplicates**

pandas.Index.is_all_dates

Index.**is_all_dates**

pandas.Index.name

Index.**name = None**

pandas.Index.names

Index.**names**

pandas.Index.empty

Index.**empty**

Modifying and Computations

Index.all(*args, **kwargs)	Return whether all elements are True.
Index.any(*args, **kwargs)	Return whether any element is True.
Index.argmin([axis, skipna])	Return a ndarray of the minimum argument indexer.
Index.argmax([axis, skipna])	Return a ndarray of the maximum argument indexer.
Index.copy([name, deep, dtype])	Make a copy of this object.
Index.delete(loc)	Make new Index with passed location(-s) deleted.
Index.drop(labels[, errors])	Make new Index with passed list of labels deleted.
Index.drop_duplicates([keep])	Return Index with duplicate values removed.
Index.duplicated([keep])	Indicate duplicate index values.
Index.equals(other)	Determines if two Index objects contain the same elements.
Index.factorize([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
Index.identical(other)	Similar to equals, but check that other comparable attributes are also equal.
Index.insert(loc, item)	Make new Index inserting new item at location.
*Index.is_*(other)	More flexible, faster check like `is` but that works through views.
Index.is_boolean()	
Index.is_categorical()	Check if the Index holds categorical data.
Index.is_floating()	
Index.is_integer()	
Index.is_interval()	
Index.is_mixed()	
Index.is_numeric()	
Index.is_object()	
Index.min([axis, skipna])	Return the minimum value of the Index.
Index.max([axis, skipna])	Return the maximum value of the Index.
Index.reindex(target[, method, level, …])	Create index with target's values (move/add/delete values as necessary).
Index.rename(name[, inplace])	Alter Index or MultiIndex name.
Index.repeat(repeats[, axis])	Repeat elements of a Index.
Index.where(cond[, other])	Return an Index of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.
Index.take(indices[, axis, allow_fill, …])	Return a new Index of the values selected by the indices.
Index.putmask(mask, value)	Return a new Index of the values set with the mask.
Index.unique([level])	Return unique values in the index.
Index.nunique([dropna])	Return number of unique elements in the object.
Index.value_counts([normalize, sort, …])	Return a Series containing counts of unique values.

pandas.Index.is_boolean

`Index.is_boolean()`

pandas.Index.is_floating

`Index.is_floating()`

pandas.Index.is_integer

`Index.is_integer()`

pandas.Index.is_interval

`Index.is_interval()`

pandas.Index.is_mixed

`Index.is_mixed()`

pandas.Index.is_numeric

`Index.is_numeric()`

pandas.Index.is_object

`Index.is_object()`

Compatibility with MultiIndex

Index.set_names(names[, level, inplace])	Set Index or MultiIndex name.
Index.is_lexsorted_for_tuple(tup)	
Index.droplevel([level])	Return index with requested level(s) removed.

pandas.Index.is_lexsorted_for_tuple

`Index.is_lexsorted_for_tuple(tup)`

Missing Values

Index.fillna([value, downcast])	Fill NA/NaN values with the specified value
Index.dropna([how])	Return Index without NA/NaN values
Index.isna()	Detect missing values.
Index.notna()	Detect existing (non-missing) values.

Conversion

Index.astype(dtype[, copy])	Create an Index with values cast to dtypes.
Index.item()	Return the first element of the underlying data as a python scalar.
Index.map(mapper[, na_action])	Map values using input correspondence (a dict, Series, or function).

Continued on next page

Table 167 – continued from previous page

Index.ravel([order])	Return an ndarray of the flattened values of the underlying data.
Index.to_list()	Return a list of the values.
Index.to_native_types([slicer])	Format specified values of *self* and return them.
Index.to_series([index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
Index.to_frame([index, name])	Create a DataFrame with a column containing the Index.
Index.view([cls])	

pandas.Index.view

Index.**view**(*cls=None*)

Sorting

Index.argsort(*args, **kwargs)	Return the integer indices that would sort the index.
Index.searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
Index.sort_values([return_indexer, ascending])	Return a sorted copy of the index.

Time-specific operations

Index.shift([periods, freq])	Shift index by desired number of time frequency increments.

Combining / joining / set operations

Index.append(other)	Append a collection of Index options together.
Index.join(other[, how, level, . . .])	Compute join_index and indexers to conform data structures to the new index.
Index.intersection(other[, sort])	Form the intersection of two Index objects.
Index.union(other[, sort])	Form the union of two Index objects.
Index.difference(other[, sort])	Return a new Index with elements from the index that are not in *other*.
Index.symmetric_difference(other[, . . .])	Compute the symmetric difference of two Index objects.

Selecting

Index.asof(label)	Return the label from the index, or, if not present, the previous one.
Index.asof_locs(where, mask)	Finds the locations (indices) of the labels from the index for every entry in the *where* argument.
Index.contains(key)	Return a boolean indicating whether the provided key is in the index.
Index.get_duplicates()	(DEPRECATED) Extract duplicated index elements.

Continued on next page

Table 171 – continued from previous page

Index.get_indexer(target[, method, limit, ...])	Compute indexer and mask for new index given the current index.
Index.get_indexer_for(target, **kwargs)	Guaranteed return of an indexer even when non-unique.
Index.get_indexer_non_unique(target)	Compute indexer and mask for new index given the current index.
Index.get_level_values(level)	Return an Index of values for requested level.
Index.get_loc(key[, method, tolerance])	Get integer location, slice or boolean mask for requested label.
Index.get_slice_bound(label, side, kind)	Calculate slice bound that corresponds to given label.
Index.get_value(series, key)	Fast lookup of value from 1-dimensional ndarray.
Index.get_values()	Return *Index* data as an *numpy.ndarray*.
Index.set_value(arr, key, value)	Fast lookup of value from 1-dimensional ndarray.
Index.isin(values[, level])	Return a boolean array where the index values are in *values*.
Index.slice_indexer([start, end, step, kind])	For an ordered or unique index, compute the slice indexer for input labels and step.
Index.slice_locs([start, end, step, kind])	Compute slice locations for input labels.

6.7.2 Numeric Index

RangeIndex	Immutable Index implementing a monotonic integer range.
Int64Index	Immutable ndarray implementing an ordered, sliceable set.
UInt64Index	Immutable ndarray implementing an ordered, sliceable set.
Float64Index	Immutable ndarray implementing an ordered, sliceable set.

pandas.RangeIndex

class pandas.**RangeIndex**

Immutable Index implementing a monotonic integer range.

RangeIndex is a memory-saving special case of Int64Index limited to representing monotonic ranges. Using RangeIndex may in some instances improve computing speed.

This is the default index type used by DataFrame and Series when no explicit index is provided by the user.

Parameters

start [int (default: 0), or other RangeIndex instance] If int and “stop” is not given, interpreted as “stop” instead.

stop [int (default: 0)]

step [int (default: 1)]

name [object, optional] Name to be stored in the index

copy [bool, default False] Unused, accepted for homogeneity with other index types.

See also:

Index The base pandas Index type.

Int64Index Index of int64 data.

Attributes

None	

Methods

from_range(data[, name, dtype])	Create RangeIndex from a range (py3), or xrange (py2) object.

pandas.RangeIndex.from_range

classmethod RangeIndex.**from_range**(*data*, *name=None*, *dtype=None*, ***kwargs*)
Create RangeIndex from a range (py3), or xrange (py2) object.

pandas.Int64Index

class pandas.**Int64Index**
Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Int64Index is a special case of *Index* with purely integer labels.

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: int64)]

copy [bool] Make a copy of input ndarray

name [object] Name to be stored in the index

See also:

Index The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

pandas.UInt64Index

class pandas.**UInt64Index**

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. UInt64Index is a special case of *Index* with purely unsigned integer labels.

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: uint64)]

copy [bool] Make a copy of input ndarray

name [object] Name to be stored in the index

See also:

Index The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

pandas.Float64Index

class pandas.**Float64Index**

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Float64Index is a special case of *Index* with purely float labels.

Parameters

data [array-like (1-dimensional)]

dtype [NumPy dtype (default: float64)]

copy [bool] Make a copy of input ndarray

name [object] Name to be stored in the index

See also:

Index The base pandas Index type.

Notes

An Index instance can **only** contain hashable objects.

Attributes

None	

Methods

None	

RangeIndex.from_range(data[, name, dtype])	Create RangeIndex from a range (py3), or xrange (py2) object.

6.7.3 CategoricalIndex

CategoricalIndex	Immutable Index implementing an ordered, sliceable set.

pandas.CategoricalIndex

class pandas.**CategoricalIndex**

Immutable Index implementing an ordered, sliceable set. CategoricalIndex represents a sparsely populated Index with an underlying Categorical.

Parameters

data [array-like or Categorical, (1-dimensional)]

categories [optional, array-like] categories for the CategoricalIndex

ordered [boolean,] designating if the categories are ordered

copy [bool] Make a copy of input ndarray

name [object] Name to be stored in the index

See also:

Categorical, *Index*

Attributes

codes	
categories	
ordered	

Methods

rename_categories(*args, **kwargs)	Renames categories.
reorder_categories(*args, **kwargs)	Reorders categories as specified in new_categories.

Continued on next page

Table 176 – continued from previous page

add_categories(*args, **kwargs)	Add new categories.
remove_categories(*args, **kwargs)	Removes the specified categories.
remove_unused_categories(*args, **kwargs)	Removes categories which are not used.
set_categories(*args, **kwargs)	Sets the categories to the specified new_categories.
as_ordered(*args, **kwargs)	Set the Categorical to be ordered.
as_unordered(*args, **kwargs)	Set the Categorical to be unordered.
map(mapper)	Map values using input correspondence (a dict, Series, or function).

pandas.CategoricalIndex.rename_categories

CategoricalIndex.**rename_categories**(**args*, ***kwargs*)

Renames categories.

Parameters

new_categories [list-like, dict-like or callable]

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.

  New in version 0.21.0.
- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

  New in version 0.23.0.

> **Warning:** Currently, Series are considered list like. In a future version of pandas they'll be considered dict-like.

inplace [boolean (default: False)] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns

cat [Categorical or None] With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

Raises

ValueError If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

See also:

reorder_categories, *add_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

Examples

```
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

pandas.CategoricalIndex.reorder_categories

`CategoricalIndex.`**`reorder_categories`**(**args*, ***kwargs*)

Reorders categories as specified in new_categories.

new_categories need to include all old categories and no new category items.

Parameters

new_categories [Index-like] The categories in new order.

ordered [boolean, optional] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

inplace [boolean (default: False)] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If the new categories do not contain all old category items or any new ones

See also:

`rename_categories`, *`add_categories`*, *`remove_categories`*, *`remove_unused_categories`*, *`set_categories`*

pandas.CategoricalIndex.add_categories

`CategoricalIndex.`**`add_categories`**(**args*, ***kwargs*)

Add new categories.

new_categories will be included at the last/highest place in the categories and will be unused directly after this call.

Parameters

new_categories [category or list-like of category] The new categories to be included.

inplace [boolean (default: False)] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical with new categories added or None if inplace.]

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories, *reorder_categories*, *remove_categories*, *remove_unused_categories*, *set_categories*

pandas.CategoricalIndex.remove_categories

CategoricalIndex.**remove_categories**(**args*, ***kwargs*)

Removes the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [boolean (default: False)] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical with removed categories or None if inplace.]

Raises

ValueError If the removals are not contained in the categories

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_unused_categories*, *set_categories*

pandas.CategoricalIndex.remove_unused_categories

CategoricalIndex.**remove_unused_categories**(**args*, ***kwargs*)

Removes categories which are not used.

Parameters

inplace [boolean (default: False)] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Returns

cat [Categorical with unused categories dropped or None if inplace.]

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_categories*, *set_categories*

pandas.CategoricalIndex.set_categories

CategoricalIndex.**set_categories**(**args*, ***kwargs*)

Sets the categories to the specified new_categories.

new_categories can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes on python3, which does not considers a S1 string equal to a single char python string.

Parameters

new_categories [Index-like] The categories in new order.

ordered [boolean, (default: False)] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

rename [boolean (default: False)] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

inplace [boolean (default: False)] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If new_categories does not validate as categories

See also:

rename_categories, *reorder_categories*, *add_categories*, *remove_categories*, *remove_unused_categories*

pandas.CategoricalIndex.as_ordered

CategoricalIndex.**as_ordered**(**args*, ***kwargs*)

Set the Categorical to be ordered.

Parameters

inplace [boolean (default: False)] Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True

pandas.CategoricalIndex.as_unordered

`CategoricalIndex.as_unordered(*args, **kwargs)`
Set the Categorical to be unordered.

Parameters

inplace [boolean (default: False)] Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to False

pandas.CategoricalIndex.map

`CategoricalIndex.map(mapper)`
Map values using input correspondence (a dict, Series, or function).

Maps the values (their categories, not the codes) of the index to new categories. If the mapping correspondence is one-to-one the result is a `CategoricalIndex` which has the same order property as the original, otherwise an `Index` is returned.

If a *dict* or `Series` is used any unmapped category is mapped to *NaN*. Note that if this happens an `Index` will be returned.

Parameters

mapper [function, dict, or Series] Mapping correspondence.

Returns

pandas.CategoricalIndex or pandas.Index Mapped index.

See also:

`Index.map` Apply a mapping correspondence on an `Index`.

`Series.map` Apply a mapping correspondence on a `Series`.

`Series.apply` Apply more complex functions on a `Series`.

Examples

```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'])
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                 ordered=False, dtype='category')
>>> idx.map(lambda x: x.upper())
CategoricalIndex(['A', 'B', 'C'], categories=['A', 'B', 'C'],
                 ordered=False, dtype='category')
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'third'})
CategoricalIndex(['first', 'second', 'third'], categories=['first',
                 'second', 'third'], ordered=False, dtype='category')
```

If the mapping is one-to-one the ordering of the categories is preserved:

```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'], ordered=True)
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                 ordered=True, dtype='category')
>>> idx.map({'a': 3, 'b': 2, 'c': 1})
```

(continues on next page)

(continued from previous page)

```
CategoricalIndex([3, 2, 1], categories=[3, 2, 1], ordered=True,
                 dtype='category')
```

If the mapping is not one-to-one an `Index` is returned:

```
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'first'})
Index(['first', 'second', 'first'], dtype='object')
```

If a *dict* is used, all unmapped categories are mapped to *NaN* and the result is an `Index`:

```
>>> idx.map({'a': 'first', 'b': 'second'})
Index(['first', 'second', nan], dtype='object')
```

Categorical Components

CategoricalIndex.codes	
CategoricalIndex.categories	
CategoricalIndex.ordered	
CategoricalIndex.rename_categories(*args, …)	Renames categories.
CategoricalIndex.reorder_categories(*args, …)	Reorders categories as specified in new_categories.
CategoricalIndex.add_categories(*args, **kwargs)	Add new categories.
CategoricalIndex.remove_categories(*args, …)	Removes the specified categories.
CategoricalIndex.remove_unused_categories(…)	Removes categories which are not used.
CategoricalIndex.set_categories(*args, **kwargs)	Sets the categories to the specified new_categories.
CategoricalIndex.as_ordered(*args, **kwargs)	Set the Categorical to be ordered.
CategoricalIndex.as_unordered(*args, **kwargs)	Set the Categorical to be unordered.

pandas.CategoricalIndex.codes

`CategoricalIndex.`**`codes`**

pandas.CategoricalIndex.categories

`CategoricalIndex.`**`categories`**

pandas.CategoricalIndex.ordered

`CategoricalIndex.`**`ordered`**

Modifying and Computations

CategoricalIndex.map(mapper)	Map values using input correspondence (a dict, Series, or function).
CategoricalIndex.equals(other)	Determines if two CategorialIndex objects contain the same elements.

pandas.CategoricalIndex.equals

`CategoricalIndex.`**`equals`**`(`*other*`)`

Determines if two CategorialIndex objects contain the same elements.

6.7.4 IntervalIndex

IntervalIndex	Immutable index of intervals that are closed on the same side.

pandas.IntervalIndex

`class` `pandas.`**`IntervalIndex`**

Immutable index of intervals that are closed on the same side.

New in version 0.20.0.

> **Warning:** The indexing behaviors are provisional and may change in a future version of pandas.

Parameters

data [array-like (1-dimensional)] Array-like containing Interval objects from which to build the IntervalIndex.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

dtype [dtype or None, default None] If None, dtype will be inferred.

New in version 0.23.0.

copy [bool, default False] Copy the input data.

name [object, optional] Name to be stored in the index.

verify_integrity [bool, default True] Verify that the IntervalIndex is valid.

See also:

Index The base pandas Index type.

Interval A bounded slice-like interval; the elements of an IntervalIndex.

interval_range Function to create a fixed frequency IntervalIndex.

cut Bin values into discrete Intervals.

qcut Bin values into equal-sized Intervals based on rank or sample quantiles.

Notes

See the user guide for more.

Examples

A new `IntervalIndex` is typically constructed using *interval_range()*:

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]],
              closed='right',
              dtype='interval[int64]')
```

It may also be constructed using one of the constructor methods: *IntervalIndex.from_arrays()*, *IntervalIndex.from_breaks()*, and *IntervalIndex.from_tuples()*.

See further examples in the doc strings of `interval_range` and the mentioned constructor methods.

Attributes

left	Return the left endpoints of each Interval in the IntervalIndex as an Index
right	Return the right endpoints of each Interval in the IntervalIndex as an Index
closed	Whether the intervals are closed on the left-side, right-side, both or neither
mid	Return the midpoint of each Interval in the IntervalIndex as an Index
length	Return an Index with entries denoting the length of each Interval in the IntervalIndex
is_non_overlapping_monotonic	Return True if the IntervalIndex is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False
is_overlapping	Return True if the IntervalIndex has overlapping intervals, else False.
values	Return the IntervalIndex's data as an IntervalArray.

pandas.IntervalIndex.left

`IntervalIndex.`**`left`**

Return the left endpoints of each Interval in the IntervalIndex as an Index

pandas.IntervalIndex.right

`IntervalIndex.`**`right`**

Return the right endpoints of each Interval in the IntervalIndex as an Index

pandas.IntervalIndex.closed

IntervalIndex.**closed**
Whether the intervals are closed on the left-side, right-side, both or neither

pandas.IntervalIndex.mid

IntervalIndex.**mid**
Return the midpoint of each Interval in the IntervalIndex as an Index

pandas.IntervalIndex.length

IntervalIndex.**length**
Return an Index with entries denoting the length of each Interval in the IntervalIndex

pandas.IntervalIndex.is_non_overlapping_monotonic

IntervalIndex.**is_non_overlapping_monotonic**
Return True if the IntervalIndex is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False

pandas.IntervalIndex.is_overlapping

IntervalIndex.**is_overlapping**
Return True if the IntervalIndex has overlapping intervals, else False.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

New in version 0.24.0.

Returns

bool Boolean indicating if the IntervalIndex has overlapping intervals.

See also:

Interval.overlaps Check whether two Interval objects overlap.

IntervalIndex.overlaps Check an IntervalIndex elementwise for overlaps.

Examples

```
>>> index = pd.IntervalIndex.from_tuples([(0, 2), (1, 3), (4, 5)])
>>> index
IntervalIndex([(0, 2], (1, 3], (4, 5]],
      closed='right',
      dtype='interval[int64]')
>>> index.is_overlapping
True
```

Intervals that share closed endpoints overlap:

```
>>> index = pd.interval_range(0, 3, closed='both')
>>> index
IntervalIndex([[0, 1], [1, 2], [2, 3]],
      closed='both',
      dtype='interval[int64]')
>>> index.is_overlapping
True
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> index = pd.interval_range(0, 3, closed='left')
>>> index
IntervalIndex([[0, 1), [1, 2), [2, 3)],
      closed='left',
      dtype='interval[int64]')
>>> index.is_overlapping
False
```

pandas.IntervalIndex.values

`IntervalIndex.values`
Return the IntervalIndex's data as an IntervalArray.

Methods

from_arrays(left, right[, closed, name, ...])	Construct from two arrays defining the left and right bounds.
from_tuples(data[, closed, name, copy, dtype])	Construct an IntervalIndex from an array-like of tuples
from_breaks(breaks[, closed, name, copy, dtype])	Construct an IntervalIndex from an array of splits.
overlaps(other)	Check elementwise if an Interval overlaps the values in the IntervalIndex.
set_closed(closed)	Return an IntervalIndex identical to the current one, but closed on the specified side
to_tuples([na_tuple])	Return an Index of tuples of the form (left, right)
contains(key)	Return a boolean indicating if the key is IN the index

pandas.IntervalIndex.from_arrays

classmethod `IntervalIndex.`**`from_arrays`**(*left*, *right*, *closed='right'*, *name=None*, *copy=False*, *dtype=None*)
Construct from two arrays defining the left and right bounds.

Parameters

left [array-like (1-dimensional)] Left bounds for each interval.

right [array-like (1-dimensional)] Right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] Copy the data.

dtype [dtype, optional] If None, dtype will be inferred.

New in version 0.23.0.

Returns

IntervalIndex

Raises

ValueError When a value is missing in only one of *left* or *right*. When a value in *left* is greater than the corresponding value in *right*.

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalIndex.from_breaks Construct an IntervalIndex from an array of splits.

IntervalIndex.from_tuples Construct an IntervalIndex from an array-like of tuples.

Notes

Each element of *left* must be less than or equal to the *right* element at the same position. If an element is missing, it must be missing in both *left* and *right*. A TypeError is raised when using an unsupported type for *left* or *right*. At the moment, 'category', 'object', and 'string' subtypes are not supported.

Examples

```
>>> IntervalIndex.from_arrays([0, 1, 2], [1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
              closed='right',
              dtype='interval[int64]')
```

pandas.IntervalIndex.from_tuples

classmethod `IntervalIndex.`**`from_tuples`**(*data*, *closed='right'*, *name=None*, *copy=False*, *dtype=None*)

Construct an IntervalIndex from an array-like of tuples

Parameters

data [array-like (1-dimensional)] Array of tuples

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] by-default copy the data, this is compat only and ignored

dtype [dtype or None, default None] If None, dtype will be inferred

..versionadded:: 0.23.0

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalIndex.from_arrays Construct an IntervalIndex from a left and right array.

IntervalIndex.from_breaks Construct an IntervalIndex from an array of splits.

Examples

```
>>> pd.IntervalIndex.from_tuples([(0, 1), (1, 2)])
IntervalIndex([(0, 1], (1, 2]],
              closed='right', dtype='interval[int64]')
```

pandas.IntervalIndex.from_breaks

classmethod IntervalIndex.**from_breaks**(*breaks*, *closed='right'*, *name=None*, *copy=False*, *dtype=None*)

Construct an IntervalIndex from an array of splits.

Parameters

breaks [array-like (1-dimensional)] Left and right bounds for each interval.

closed [{'left', 'right', 'both', 'neither'}, default 'right'] Whether the intervals are closed on the left-side, right-side, both or neither.

copy [boolean, default False] copy the data

dtype [dtype or None, default None] If None, dtype will be inferred

New in version 0.23.0.

See also:

interval_range Function to create a fixed frequency IntervalIndex.

IntervalIndex.from_arrays Construct from a left and right array.

IntervalIndex.from_tuples Construct from a sequence of tuples.

Examples

```
>>> pd.IntervalIndex.from_breaks([0, 1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3]],
              closed='right',
              dtype='interval[int64]')
```

pandas.IntervalIndex.overlaps

IntervalIndex.**overlaps**(*other*)

Check elementwise if an Interval overlaps the values in the IntervalIndex.

Two intervals overlap if they share a common point, including closed endpoints. Intervals that only have an open endpoint in common do not overlap.

New in version 0.24.0.

Parameters

other [Interval] Interval to check against for an overlap.

Returns

ndarray Boolean array positionally indicating where an overlap occurs.

See also:

Interval.overlaps Check whether two Interval objects overlap.

Examples

```
>>> intervals = pd.IntervalIndex.from_tuples([(0, 1), (1, 3), (2, 4)])
>>> intervals
IntervalIndex([(0, 1], (1, 3], (2, 4]],
      closed='right',
      dtype='interval[int64]')
>>> intervals.overlaps(pd.Interval(0.5, 1.5))
array([ True,  True, False])
```

Intervals that share closed endpoints overlap:

```
>>> intervals.overlaps(pd.Interval(1, 3, closed='left'))
array([ True,  True, True])
```

Intervals that only have an open endpoint in common do not overlap:

```
>>> intervals.overlaps(pd.Interval(1, 2, closed='right'))
array([False,  True, False])
```

pandas.IntervalIndex.set_closed

`IntervalIndex.`**`set_closed`**(*closed*)

Return an IntervalIndex identical to the current one, but closed on the specified side

New in version 0.24.0.

Parameters

closed [{'left', 'right', 'both', 'neither'}] Whether the intervals are closed on the left-side, right-side, both or neither.

Returns

new_index [IntervalIndex]

Examples

```
>>> index = pd.interval_range(0, 3)
>>> index
IntervalIndex([(0, 1], (1, 2], (2, 3]],
      closed='right',
      dtype='interval[int64]')
>>> index.set_closed('both')
IntervalIndex([[0, 1], [1, 2], [2, 3]],
      closed='both',
      dtype='interval[int64]')
```

pandas.IntervalIndex.to_tuples

`IntervalIndex.to_tuples`(*na_tuple=True*)

Return an Index of tuples of the form (left, right)

Parameters

na_tuple [boolean, default True] Returns NA as a tuple if True, `(nan, nan)`, or just as the NA value itself if False, `nan`.

New in version 0.23.0.

Returns

tuples: Index

Examples

```
>>> idx = pd.IntervalIndex.from_arrays([0, np.nan, 2], [1, np.nan, 3])
>>> idx.to_tuples()
Index([(0.0, 1.0), (nan, nan), (2.0, 3.0)], dtype='object')
>>> idx.to_tuples(na_tuple=False)
Index([(0.0, 1.0), nan, (2.0, 3.0)], dtype='object')
```

pandas.IntervalIndex.contains

`IntervalIndex.contains`(*key*)

Return a boolean indicating if the key is IN the index

We accept / allow keys to be not *just* actual objects.

Parameters

key [int, float, Interval]

Returns

boolean

IntervalIndex Components

IntervalIndex.from_arrays(left, right[, …])	Construct from two arrays defining the left and right bounds.
IntervalIndex.from_tuples(data[, closed, …])	Construct an IntervalIndex from an array-like of tuples
IntervalIndex.from_breaks(breaks[, closed, …])	Construct an IntervalIndex from an array of splits.
IntervalIndex.contains(key)	Return a boolean indicating if the key is IN the index
IntervalIndex.left	Return the left endpoints of each Interval in the IntervalIndex as an Index
IntervalIndex.right	Return the right endpoints of each Interval in the IntervalIndex as an Index
IntervalIndex.mid	Return the midpoint of each Interval in the IntervalIndex as an Index

Continued on next page

Table 182 – continued from previous page

IntervalIndex.closed	Whether the intervals are closed on the left-side, right-side, both or neither
IntervalIndex.length	Return an Index with entries denoting the length of each Interval in the IntervalIndex
IntervalIndex.values	Return the IntervalIndex's data as an IntervalArray.
IntervalIndex.is_non_overlapping_monotonic	Return True if the IntervalIndex is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False
IntervalIndex.is_overlapping	Return True if the IntervalIndex has overlapping intervals, else False.
IntervalIndex.get_loc(key[, method])	Get integer location, slice or boolean mask for requested label.
IntervalIndex.get_indexer(target[, method, ...])	Compute indexer and mask for new index given the current index.
IntervalIndex.set_closed(closed)	Return an IntervalIndex identical to the current one, but closed on the specified side
IntervalIndex.overlaps(other)	Check elementwise if an Interval overlaps the values in the IntervalIndex.
IntervalIndex.to_tuples([na_tuple])	Return an Index of tuples of the form (left, right)

pandas.IntervalIndex.get_loc

IntervalIndex.**get_loc**(*key*, *method=None*)

Get integer location, slice or boolean mask for requested label.

Parameters

key [label]

method [{None}, optional]

- default: matches where the label is within an interval only.

Returns

loc [int if unique index, slice if monotonic index, else mask]

Examples

```
>>> i1, i2 = pd.Interval(0, 1), pd.Interval(1, 2)
>>> index = pd.IntervalIndex([i1, i2])
>>> index.get_loc(1)
0
```

You can also supply an interval or an location for a point inside an interval.

```
>>> index.get_loc(pd.Interval(0, 2))
array([0, 1], dtype=int64)
>>> index.get_loc(1.5)
1
```

If a label is in several intervals, you get the locations of all the relevant intervals.

```
>>> i3 = pd.Interval(0, 2)
>>> overlapping_index = pd.IntervalIndex([i2, i3])
>>> overlapping_index.get_loc(1.5)
array([0, 1], dtype=int64)
```

pandas.IntervalIndex.get_indexer

`IntervalIndex.get_indexer`(*target*, *method=None*, *limit=None*, *tolerance=None*)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [IntervalIndex or list of Intervals]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit [int, optional] Maximum number of consecutive labels in `target` to match for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

6.7.5 MultiIndex

MultiIndex	A multi-level, or hierarchical, index object for pandas objects.

pandas.MultiIndex

class pandas.**MultiIndex**

A multi-level, or hierarchical, index object for pandas objects.

Parameters

levels [sequence of arrays] The unique labels for each level.

codes [sequence of arrays] Integers for each level designating which label at each location.

New in version 0.24.0.

labels [sequence of arrays] Integers for each level designating which label at each location.

Deprecated since version 0.24.0: Use `codes` instead

sortorder [optional int] Level of sortedness (must be lexicographically sorted by that level).

names [optional sequence of objects] Names for each of the index levels. (name is accepted for compat).

copy [bool, default False] Copy the meta-data.

verify_integrity [bool, default True] Check that the levels/codes are consistent and valid.

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_product Create a MultiIndex from the cartesian product of iterables.

MultiIndex.from_tuples Convert list of tuples to a MultiIndex.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Index The base pandas Index type.

Notes

See the user guide for more.

Examples

A new `MultiIndex` is typically constructed using one of the helper methods *MultiIndex.from_arrays()*, *MultiIndex.from_product()* and *MultiIndex.from_tuples()*. For example (using `.from_arrays`):

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex(levels=[[1, 2], ['blue', 'red']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['number', 'color'])
```

See further examples for how to construct a MultiIndex in the doc strings of the mentioned helper methods.

Attributes

names	Names of levels in MultiIndex
nlevels	Integer number of levels in this MultiIndex.
levshape	A tuple with the length of each level.

pandas.MultiIndex.names

MultiIndex.**names**
Names of levels in MultiIndex

pandas.MultiIndex.nlevels

MultiIndex.**nlevels**
Integer number of levels in this MultiIndex.

pandas.MultiIndex.levshape

MultiIndex.**levshape**
A tuple with the length of each level.

levels	
codes	

Methods

from_arrays(arrays[, sortorder, names])	Convert arrays to MultiIndex.
from_tuples(tuples[, sortorder, names])	Convert list of tuples to MultiIndex.
from_product(iterables[, sortorder, names])	Make a MultiIndex from the cartesian product of multiple iterables.
from_frame(df[, sortorder, names])	Make a MultiIndex from a DataFrame.
set_levels(levels[, level, inplace, . . .])	Set new levels on MultiIndex.
set_codes(codes[, level, inplace, . . .])	Set new codes on MultiIndex.
to_frame([index, name])	Create a DataFrame with the levels of the MultiIndex as columns.
to_flat_index()	Convert a MultiIndex to an Index of Tuples containing the level values.
is_lexsorted()	Return True if the codes are lexicographically sorted
sortlevel([level, ascending, sort_remaining])	Sort MultiIndex at the requested level.
droplevel([level])	Return index with requested level(s) removed.
swaplevel([i, j])	Swap level i with level j.
reorder_levels(order)	Rearrange levels using input order.
remove_unused_levels()	Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.

pandas.MultiIndex.from_arrays

classmethod MultiIndex.**from_arrays**(*arrays*, *sortorder=None*, *names=None*)
Convert arrays to MultiIndex.

Parameters

arrays [list / sequence of array-likes] Each array-like gives one level's value for each data point. len(arrays) is the number of levels.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Returns

index [MultiIndex]

See also:

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex(levels=[[1, 2], ['blue', 'red']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['number', 'color'])
```

pandas.MultiIndex.from_tuples

classmethod MultiIndex.**from_tuples**(*tuples*, *sortorder=None*, *names=None*)
Convert list of tuples to MultiIndex.

Parameters

tuples [list / sequence of tuple-likes] Each tuple is the index of one row/column.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Returns

index [MultiIndex]

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> tuples = [(1, u'red'), (1, u'blue'),
...           (2, u'red'), (2, u'blue')]
>>> pd.MultiIndex.from_tuples(tuples, names=('number', 'color'))
MultiIndex(levels=[[1, 2], ['blue', 'red']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['number', 'color'])
```

pandas.MultiIndex.from_product

classmethod MultiIndex.**from_product**(*iterables*, *sortorder=None*, *names=None*)

Make a MultiIndex from the cartesian product of multiple iterables.

Parameters

iterables [list / sequence of iterables] Each iterable has unique labels for each level of the index.

sortorder [int or None] Level of sortedness (must be lexicographically sorted by that level).

names [list / sequence of str, optional] Names for the levels in the index.

Returns

index [MultiIndex]

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_frame Make a MultiIndex from a DataFrame.

Examples

```
>>> numbers = [0, 1, 2]
>>> colors = ['green', 'purple']
>>> pd.MultiIndex.from_product([numbers, colors],
...                            names=['number', 'color'])
MultiIndex(levels=[[0, 1, 2], ['green', 'purple']],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
           names=['number', 'color'])
```

pandas.MultiIndex.from_frame

classmethod MultiIndex.**from_frame**(*df*, *sortorder=None*, *names=None*)

Make a MultiIndex from a DataFrame.

New in version 0.24.0.

Parameters

df [DataFrame] DataFrame to be converted to MultiIndex.

sortorder [int, optional] Level of sortedness (must be lexicographically sorted by that level).

names [list-like, optional] If no names are provided, use the column names, or tuple of column names if the columns is a MultiIndex. If a sequence, overwrite names with the given sequence.

Returns

MultiIndex The MultiIndex representation of the given DataFrame.

See also:

MultiIndex.from_arrays Convert list of arrays to MultiIndex.

MultiIndex.from_tuples Convert list of tuples to MultiIndex.

MultiIndex.from_product Make a MultiIndex from cartesian product of iterables.

Examples

```
>>> df = pd.DataFrame([['HI', 'Temp'], ['HI', 'Precip'],
...                    ['NJ', 'Temp'], ['NJ', 'Precip']],
...                   columns=['a', 'b'])
>>> df
    a       b
0  HI    Temp
1  HI  Precip
2  NJ    Temp
3  NJ  Precip
```

```
>>> pd.MultiIndex.from_frame(df)
MultiIndex(levels=[['HI', 'NJ'], ['Precip', 'Temp']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['a', 'b'])
```

Using explicit names, instead of the column names

```
>>> pd.MultiIndex.from_frame(df, names=['state', 'observation'])
MultiIndex(levels=[['HI', 'NJ'], ['Precip', 'Temp']],
           codes=[[0, 0, 1, 1], [1, 0, 1, 0]],
           names=['state', 'observation'])
```

pandas.MultiIndex.set_levels

MultiIndex.**set_levels**(*levels*, *level=None*, *inplace=False*, *verify_integrity=True*)

Set new levels on MultiIndex. Defaults to returning new index.

Parameters

levels [sequence or list of sequence] new level(s) to apply

level [int, level name, or sequence of int/level names (default None)] level(s) to set (None for all levels)

inplace [bool] if True, mutates in place

verify_integrity [bool (default True)] if True, checks that levels and codes are compatible

Returns

new index (of same type and class. . . etc)

Examples

```
>>> idx = pd.MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                    (2, u'one'), (2, u'two')],
                                    names=['foo', 'bar'])
>>> idx.set_levels([['a','b'], [1,2]])
MultiIndex(levels=[[u'a', u'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'foo', u'bar'])
>>> idx.set_levels(['a','b'], level=0)
MultiIndex(levels=[[u'a', u'b'], [u'one', u'two']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'foo', u'bar'])
>>> idx.set_levels(['a','b'], level='bar')
MultiIndex(levels=[[1, 2], [u'a', u'b']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'foo', u'bar'])
>>> idx.set_levels([['a','b'], [1,2]], level=[0,1])
MultiIndex(levels=[[u'a', u'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[u'foo', u'bar'])
```

pandas.MultiIndex.set_codes

MultiIndex.**set_codes**(*codes*, *level=None*, *inplace=False*, *verify_integrity=True*)

Set new codes on MultiIndex. Defaults to returning new index.

New in version 0.24.0: New name for deprecated method *set_labels*.

Parameters

codes [sequence or list of sequence] new codes to apply

level [int, level name, or sequence of int/level names (default None)] level(s) to set (None for all levels)

inplace [bool] if True, mutates in place

verify_integrity [bool (default True)] if True, checks that levels and codes are compatible

Returns

new index (of same type and class. . . etc)

Examples

```
>>> idx = pd.MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                    (2, u'one'), (2, u'two')],
                                    names=['foo', 'bar'])
>>> idx.set_codes([[1,0,1,0], [0,0,1,1]])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           codes=[[1, 0, 1, 0], [0, 0, 1, 1]],
```

(continues on next page)

(continued from previous page)

```
            names=[u'foo', u'bar'])
>>> idx.set_codes([1,0,1,0], level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           codes=[[1, 0, 1, 0], [0, 1, 0, 1]],
           names=[u'foo', u'bar'])
>>> idx.set_codes([0,0,1,1], level='bar')
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           codes=[[0, 0, 1, 1], [0, 0, 1, 1]],
           names=[u'foo', u'bar'])
>>> idx.set_codes([[1,0,1,0], [0,0,1,1]], level=[0,1])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           codes=[[1, 0, 1, 0], [0, 0, 1, 1]],
           names=[u'foo', u'bar'])
```

pandas.MultiIndex.to_frame

MultiIndex.**to_frame**(*index=True*, *name=None*)

Create a DataFrame with the levels of the MultiIndex as columns.

Column ordering is determined by the DataFrame constructor with data as a dict.

New in version 0.24.0.

Parameters

index [boolean, default True] Set the index of the returned DataFrame as the original MultiIndex.

name [list / sequence of strings, optional] The passed names should substitute index level names.

Returns

DataFrame [a DataFrame containing the original MultiIndex data.]

See also:

DataFrame

pandas.MultiIndex.to_flat_index

MultiIndex.**to_flat_index**()

Convert a MultiIndex to an Index of Tuples containing the level values.

New in version 0.24.0.

Returns

pd.Index Index with the MultiIndex data represented in Tuples.

Notes

This method will simply return the caller if called by anything other than a MultiIndex.

Examples

```
>>> index = pd.MultiIndex.from_product(
...     [['foo', 'bar'], ['baz', 'qux']],
...     names=['a', 'b'])
>>> index.to_flat_index()
Index([('foo', 'baz'), ('foo', 'qux'),
       ('bar', 'baz'), ('bar', 'qux')],
      dtype='object')
```

pandas.MultiIndex.is_lexsorted

MultiIndex.**is_lexsorted**()

Return True if the codes are lexicographically sorted

pandas.MultiIndex.sortlevel

MultiIndex.**sortlevel**(*level=0*, *ascending=True*, *sort_remaining=True*)

Sort MultiIndex at the requested level. The result will respect the original ordering of the associated factor at that level.

Parameters

level [list-like, int or str, default 0] If a string is given, must be a name of the level If list-like must be names or ints of levels.

ascending [boolean, default True] False to sort in descending order Can also be a list to specify a directed ordering

sort_remaining [sort by the remaining levels after level]

Returns

sorted_index [pd.MultiIndex] Resulting index

indexer [np.ndarray] Indices of output values in original index

pandas.MultiIndex.droplevel

MultiIndex.**droplevel**(*level=0*)

Return index with requested level(s) removed.

If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

New in version 0.23.1: (support for non-MultiIndex)

Parameters

level [int, str, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

Returns

index [Index or MultiIndex]

pandas.MultiIndex.swaplevel

`MultiIndex.swaplevel(i=-2, j=-1)`

Swap level i with level j.

Calling this method does not change the ordering of the values.

Parameters

i [int, str, default -2] First level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

j [int, str, default -1] Second level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

Returns

MultiIndex A new MultiIndex

.. versionchanged:: 0.18.1 The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

See also:

Series.swaplevel Swap levels i and j in a MultiIndex.

Dataframe.swaplevel Swap levels i and j in a MultiIndex on a particular axis.

Examples

```
>>> mi = pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
...                    codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi
MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi.swaplevel(0, 1)
MultiIndex(levels=[['bb', 'aa'], ['a', 'b']],
           codes=[[0, 1, 0, 1], [0, 0, 1, 1]])
```

pandas.MultiIndex.reorder_levels

`MultiIndex.reorder_levels(order)`

Rearrange levels using input order. May not drop or duplicate levels

pandas.MultiIndex.remove_unused_levels

`MultiIndex.remove_unused_levels()`

Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.

The resulting MultiIndex will have the same outward appearance, meaning the same .values and ordering. It will also be .equals() to the original.

New in version 0.20.0.

Returns

MultiIndex

Examples

```
>>> i = pd.MultiIndex.from_product([range(2), list('ab')])
MultiIndex(levels=[[0, 1], ['a', 'b']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
>>> i[2:]
MultiIndex(levels=[[0, 1], ['a', 'b']],
           codes=[[1, 1], [0, 1]])
```

The 0 from the first level is not represented and can be removed

```
>>> i[2:].remove_unused_levels()
MultiIndex(levels=[[1], ['a', 'b']],
           codes=[[0, 0], [0, 1]])
```

IndexSlice	Create an object to more easily perform multi-index slicing

pandas.IndexSlice

pandas.**IndexSlice = <pandas.core.indexing._IndexSlice object>**

Create an object to more easily perform multi-index slicing

See also:

MultiIndex.remove_unused_levels New MultiIndex with no unused levels.

Notes

See *Defined Levels* for further info on slicing a MultiIndex.

Examples

```
>>> midx = pd.MultiIndex.from_product([['A0','A1'], ['B0','B1','B2','B3']])
>>> columns = ['foo', 'bar']
>>> dfmi = pd.DataFrame(np.arange(16).reshape((len(midx), len(columns))),
                        index=midx, columns=columns)
```

Using the default slice command:

```
>>> dfmi.loc[(slice(None), slice('B0', 'B1')), :]
           foo  bar
    A0 B0    0    1
       B1    2    3
    A1 B0    8    9
       B1   10   11
```

Using the IndexSlice class for a more intuitive command:

```
>>> idx = pd.IndexSlice
>>> dfmi.loc[idx[:, 'B0':'B1'], :]
           foo  bar
    A0 B0     0    1
       B1     2    3
    A1 B0     8    9
       B1    10   11
```

MultiIndex Constructors

MultiIndex.from_arrays(arrays[, sortorder, . . .])	Convert arrays to MultiIndex.
MultiIndex.from_tuples(tuples[, sortorder, . . .])	Convert list of tuples to MultiIndex.
MultiIndex.from_product(iterables[, . . .])	Make a MultiIndex from the cartesian product of multiple iterables.
MultiIndex.from_frame(df[, sortorder, names])	Make a MultiIndex from a DataFrame.

MultiIndex Properties

MultiIndex.names	Names of levels in MultiIndex
MultiIndex.levels	
MultiIndex.codes	
MultiIndex.nlevels	Integer number of levels in this MultiIndex.
MultiIndex.levshape	A tuple with the length of each level.

pandas.MultiIndex.levels

MultiIndex.**levels**

pandas.MultiIndex.codes

MultiIndex.**codes**

MultiIndex Components

MultiIndex.set_levels(levels[, level, . . .])	Set new levels on MultiIndex.
MultiIndex.set_codes(codes[, level, . . .])	Set new codes on MultiIndex.
MultiIndex.to_hierarchical(n_repeat[, n_shuffle])	(DEPRECATED) Return a MultiIndex reshaped to conform to the shapes given by n_repeat and n_shuffle.
MultiIndex.to_flat_index()	Convert a MultiIndex to an Index of Tuples containing the level values.
MultiIndex.to_frame([index, name])	Create a DataFrame with the levels of the MultiIndex as columns.
MultiIndex.is_lexsorted()	Return True if the codes are lexicographically sorted
MultiIndex.sortlevel([level, ascending, . . .])	Sort MultiIndex at the requested level.

Continued on next page

Table 189 – continued from previous page

MultiIndex.droplevel([level])	Return index with requested level(s) removed.
MultiIndex.swaplevel([i, j])	Swap level i with level j.
MultiIndex.reorder_levels(order)	Rearrange levels using input order.
MultiIndex.remove_unused_levels()	Create a new MultiIndex from the current that removes unused levels, meaning that they are not expressed in the labels.

pandas.MultiIndex.to_hierarchical

MultiIndex.**to_hierarchical**(*n_repeat*, *n_shuffle=1*)

Return a MultiIndex reshaped to conform to the shapes given by n_repeat and n_shuffle.

Deprecated since version 0.24.0.

Useful to replicate and rearrange a MultiIndex for combination with another Index with n_repeat items.

Parameters

n_repeat [int] Number of times to repeat the labels on self

n_shuffle [int] Controls the reordering of the labels. If the result is going to be an inner level in a MultiIndex, n_shuffle will need to be greater than one. The size of each label must divisible by n_shuffle.

Returns

MultiIndex

Examples

```
>>> idx = pd.MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                    (2, u'one'), (2, u'two')])
>>> idx.to_hierarchical(3)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
           codes=[[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
                  [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1]])
```

MultiIndex Selecting

MultiIndex.get_loc(key[, method])	Get location for a label or a tuple of labels as an integer, slice or boolean mask.
MultiIndex.get_loc_level(key[, level, . . .])	Get both the location for the requested label(s) and the resulting sliced index.
MultiIndex.get_indexer(target[, method, . . .])	Compute indexer and mask for new index given the current index.
MultiIndex.get_level_values(level)	Return vector of label values for requested level, equal to the length of the index.

pandas.MultiIndex.get_loc

MultiIndex.**get_loc**(*key*, *method=None*)

Get location for a label or a tuple of labels as an integer, slice or boolean mask.

Parameters

key [label or tuple of labels (one for each level)]

method [None]

Returns

loc [int, slice object or boolean mask] If the key is past the lexsort depth, the return may be a boolean mask array, otherwise it is always a slice or int.

See also:

Index.get_loc The get_loc method for (single-level) index.

MultiIndex.slice_locs Get slice location given start label(s) and end label(s).

MultiIndex.get_locs Get location for a label/slice/list/mask or a sequence of such.

Notes

The key cannot be a slice, list of same-level labels, a boolean mask, or a sequence of such. If you want to use those, use `MultiIndex.get_locs()` instead.

Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
```

```
>>> mi.get_loc('b')
slice(1, 3, None)
```

```
>>> mi.get_loc(('b', 'e'))
1
```

pandas.MultiIndex.get_loc_level

`MultiIndex.`**`get_loc_level`**(*key*, *level=0*, *drop_level=True*)

Get both the location for the requested label(s) and the resulting sliced index.

Parameters

key [label or sequence of labels]

level [int/level name or list thereof, optional]

drop_level [bool, default True] if `False`, the resulting index will not drop any level.

Returns

loc [A 2-tuple where the elements are:] Element 0: int, slice object or boolean array Element 1: The resulting sliced multiindex/index. If the key contains all levels, this will be `None`.

See also:

MultiIndex.get_loc Get location for a label or a tuple of labels.

MultiIndex.get_locs Get location for a label/slice/list/mask or a sequence of such.

Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')],
...                                names=['A', 'B'])
```

```
>>> mi.get_loc_level('b')
(slice(1, 3, None), Index(['e', 'f'], dtype='object', name='B'))
```

```
>>> mi.get_loc_level('e', level='B')
(array([False,  True, False], dtype=bool),
Index(['b'], dtype='object', name='A'))
```

```
>>> mi.get_loc_level(['b', 'e'])
(1, None)
```

pandas.MultiIndex.get_indexer

`MultiIndex.get_indexer`(*target*, *method=None*, *limit=None*, *tolerance=None*)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.

Parameters

target [MultiIndex or list of tuples]

method [{None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional]

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

limit [int, optional] Maximum number of consecutive labels in `target` to match for inexact matches.

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

Returns

indexer [ndarray of int] Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

pandas.MultiIndex.get_level_values

`MultiIndex.`**`get_level_values`**(*level*)

Return vector of label values for requested level, equal to the length of the index.

Parameters

level [int or str] `level` is either the integer position of the level in the MultiIndex, or the name of the level.

Returns

values [Index] `values` is a level of this MultiIndex converted to a single *`Index`* (or subclass thereof).

Examples

Create a MultiIndex:

```
>>> mi = pd.MultiIndex.from_arrays((list('abc'), list('def')))
>>> mi.names = ['level_1', 'level_2']
```

Get level values by supplying level as either integer or name:

```
>>> mi.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object', name='level_1')
>>> mi.get_level_values('level_2')
Index(['d', 'e', 'f'], dtype='object', name='level_2')
```

6.7.6 DatetimeIndex

`DatetimeIndex`	Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata such as frequency information.

pandas.DatetimeIndex

`class` `pandas.`**`DatetimeIndex`**

Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata such as frequency information.

Parameters

data [array-like (1-dimensional), optional] Optional datetime-like data to construct index with

copy [bool] Make a copy of input ndarray

freq [string or pandas offset object, optional] One of pandas date offset strings or corresponding objects. The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation

start [starting value, datetime-like, optional] If data is None, start is used as the start point in generating regular timestamp data.

Deprecated since version 0.24.0.

periods [int, optional, > 0] Number of periods to generate, if generating index. Takes precedence over end argument

Deprecated since version 0.24.0.

end [end time, datetime-like, optional] If periods is none, generated index will extend to first conforming time on or just past end argument

Deprecated since version 0.24.0.

closed [string or None, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

Deprecated since version 0.24.: 0

tz [pytz.timezone or dateutil.tz.tzfile]

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

name [object] Name to be stored in the index

dayfirst [bool, default False] If True, parse dates in *data* with the day first order

yearfirst [bool, default False] If True parse dates in *data* with the year first order

See also:

Index The base pandas Index type.

TimedeltaIndex Index of timedelta64 data.

PeriodIndex Index of Period data.

to_datetime Convert argument to datetime.

date_range Create a fixed-frequency DatetimeIndex.

Notes

To learn more about the frequency strings, please see this link.

Creating a DatetimeIndex based on *start*, *periods*, and *end* has been deprecated in favor of `date_range()`.

Attributes

year	The year of the datetime.
month	The month as January=1, December=12.
day	The days of the datetime.
hour	The hours of the datetime.
minute	The minutes of the datetime.
second	The seconds of the datetime.
microsecond	The microseconds of the datetime.
nanosecond	The nanoseconds of the datetime.
date	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).
time	Returns numpy array of datetime.time.
timetz	Returns numpy array of datetime.time also containing timezone information.
dayofyear	The ordinal day of the year.
weekofyear	The week ordinal of the year.
week	The week ordinal of the year.
dayofweek	The day of the week with Monday=0, Sunday=6.
weekday	The day of the week with Monday=0, Sunday=6.
quarter	The quarter of the date.
freq	Return the frequency object if it is set, otherwise None.
freqstr	Return the frequency object as a string if it is set, otherwise None.
is_month_start	Indicates whether the date is the first day of the month.
is_month_end	Indicates whether the date is the last day of the month.
is_quarter_start	Indicator for whether the date is the first day of a quarter.
is_quarter_end	Indicator for whether the date is the last day of a quarter.
is_year_start	Indicate whether the date is the first day of a year.
is_year_end	Indicate whether the date is the last day of the year.
is_leap_year	Boolean indicator if the date belongs to a leap year.
inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.

pandas.DatetimeIndex.year

DatetimeIndex.**year**
The year of the datetime.

pandas.DatetimeIndex.month

`DatetimeIndex.month`
The month as January=1, December=12.

pandas.DatetimeIndex.day

`DatetimeIndex.day`
The days of the datetime.

pandas.DatetimeIndex.hour

`DatetimeIndex.hour`
The hours of the datetime.

pandas.DatetimeIndex.minute

`DatetimeIndex.minute`
The minutes of the datetime.

pandas.DatetimeIndex.second

`DatetimeIndex.second`
The seconds of the datetime.

pandas.DatetimeIndex.microsecond

`DatetimeIndex.microsecond`
The microseconds of the datetime.

pandas.DatetimeIndex.nanosecond

`DatetimeIndex.nanosecond`
The nanoseconds of the datetime.

pandas.DatetimeIndex.date

`DatetimeIndex.date`
Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).

pandas.DatetimeIndex.time

`DatetimeIndex.time`
Returns numpy array of datetime.time. The time part of the Timestamps.

pandas.DatetimeIndex.timetz

DatetimeIndex.**timetz**

Returns numpy array of datetime.time also containing timezone information. The time part of the Timestamps.

pandas.DatetimeIndex.dayofyear

DatetimeIndex.**dayofyear**

The ordinal day of the year.

pandas.DatetimeIndex.weekofyear

DatetimeIndex.**weekofyear**

The week ordinal of the year.

pandas.DatetimeIndex.week

DatetimeIndex.**week**

The week ordinal of the year.

pandas.DatetimeIndex.dayofweek

DatetimeIndex.**dayofweek**

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.

Series.dt.weekday Alias.

Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
```

(continues on next page)

(continued from previous page)

```
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.DatetimeIndex.weekday

DatetimeIndex.**weekday**

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

Returns

Series or Index Containing integers indicating the day number.

See also:

Series.dt.dayofweek Alias.

Series.dt.weekday Alias.

Series.dt.day_name Returns the name of the day of the week.

Examples

```
>>> s = pd.date_range('2016-12-31', '2017-01-08', freq='D').to_series()
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
Freq: D, dtype: int64
```

pandas.DatetimeIndex.quarter

DatetimeIndex.**quarter**

The quarter of the date.

pandas.DatetimeIndex.freq

DatetimeIndex.**freq**

Return the frequency object if it is set, otherwise None.

pandas.DatetimeIndex.freqstr

`DatetimeIndex.freqstr`

Return the frequency object as a string if it is set, otherwise None.

pandas.DatetimeIndex.is_month_start

`DatetimeIndex.is_month_start`

Indicates whether the date is the first day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.

is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
dtype: bool
>>> s.dt.is_month_end
0    False
1    True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.DatetimeIndex.is_month_end

`DatetimeIndex.is_month_end`

Indicates whether the date is the last day of the month.

Returns

Series or array For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

See also:

is_month_start Return a boolean indicating whether the date is the first day of the month.

is_month_end Return a boolean indicating whether the date is the last day of the month.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> s = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0   2018-02-27
1   2018-02-28
2   2018-03-01
dtype: datetime64[ns]
>>> s.dt.is_month_start
0    False
1    False
2    True
dtype: bool
>>> s.dt.is_month_end
0    False
1    True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
array([False, False, True])
>>> idx.is_month_end
array([False, True, False])
```

pandas.DatetimeIndex.is_quarter_start

`DatetimeIndex.`**`is_quarter_start`**

Indicator for whether the date is the first day of a quarter.

Returns

is_quarter_start [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.

is_quarter_end Similar property for indicating the quarter start.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                    periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
       dates  quarter  is_quarter_start
0 2017-03-30        1             False
1 2017-03-31        1             False
2 2017-04-01        2              True
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

pandas.DatetimeIndex.is_quarter_end

DatetimeIndex.**is_quarter_end**
Indicator for whether the date is the last day of a quarter.

Returns

is_quarter_end [Series or DatetimeIndex] The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

quarter Return the quarter of the date.

is_quarter_start Similar property indicating the quarter start.

Examples

This method is available on Series with datetime values under the .dt accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                     periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
       dates  quarter    is_quarter_end
0 2017-03-30        1             False
1 2017-03-31        1              True
2 2017-04-01        2             False
3 2017-04-02        2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

pandas.DatetimeIndex.is_year_start

`DatetimeIndex.`**`is_year_start`**

Indicate whether the date is the first day of a year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

`is_year_end` Similar property indicating the last day of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2    True
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

pandas.DatetimeIndex.is_year_end

`DatetimeIndex.`**`is_year_end`**

Indicate whether the date is the last day of the year.

Returns

Series or DatetimeIndex The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

is_year_start Similar property indicating the start of the year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0   2017-12-30
1   2017-12-31
2   2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

pandas.DatetimeIndex.is_leap_year

`DatetimeIndex.`**`is_leap_year`**

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

Returns

Series or ndarray Booleans indicating if dates belong to a leap year.

Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```
>>> dates = pd.Series(idx)
>>> dates_series
0   2012-12-31
1   2013-12-31
2   2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool
```

pandas.DatetimeIndex.inferred_freq

DatetimeIndex.**inferred_freq**

Tryies to return a string representing a frequency guess, generated by infer_freq. Returns None if it can't autodetect the frequency.

tz	

Methods

normalize(*args, **kwargs)	Convert times to midnight.
strftime(*args, **kwargs)	Convert to Index using specified date_format.
snap([freq])	Snap time stamps to nearest occurring frequency
tz_convert(*args, **kwargs)	Convert tz-aware Datetime Array/Index from one time zone to another.
tz_localize(*args, **kwargs)	Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.
round(*args, **kwargs)	Perform round operation on the data to the specified *freq*.
floor(*args, **kwargs)	Perform floor operation on the data to the specified *freq*.
ceil(*args, **kwargs)	Perform ceil operation on the data to the specified *freq*.
to_period(*args, **kwargs)	Cast to PeriodArray/Index at a particular frequency.
to_perioddelta(*args, **kwargs)	Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq.
to_pydatetime(*args, **kwargs)	Return Datetime Array/Index as object ndarray of datetime.datetime objects

Continued on next page

Table 193 – continued from previous page

to_series([keep_tz, index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
to_frame([index, name])	Create a DataFrame with a column containing the Index.
month_name(*args, **kwargs)	Return the month names of the DateTimeIndex with specified locale.
day_name(*args, **kwargs)	Return the day names of the DateTimeIndex with specified locale.

pandas.DatetimeIndex.normalize

DatetimeIndex.**normalize**(**args*, ***kwargs*)

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on Datetime Array/Index.

Returns

DatetimeArray, DatetimeIndex or Series The same type as the original data. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

floor Floor the datetimes to the specified freq.

ceil Ceil the datetimes to the specified freq.

round Round the datetimes to the specified freq.

Examples

```
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
...                     periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
                dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
               dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

pandas.DatetimeIndex.strftime

DatetimeIndex.**strftime**(**args*, ***kwargs*)

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc

Parameters

date_format [str] Date format string (e.g. "%Y-%m-%d").

Returns

Index Index of formatted strings

See also:

to_datetime Convert the given argument to datetime.

DatetimeIndex.normalize Return DatetimeIndex with times to midnight.

DatetimeIndex.round Round the DatetimeIndex to the specified freq.

DatetimeIndex.floor Floor the DatetimeIndex to the specified freq.

Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

pandas.DatetimeIndex.snap

DatetimeIndex.**snap**(*freq='S'*)

Snap time stamps to nearest occurring frequency

pandas.DatetimeIndex.tz_convert

DatetimeIndex.**tz_convert**(**args*, ***kwargs*)

Convert tz-aware Datetime Array/Index from one time zone to another.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone for time. Corresponding timestamps would be converted to this time zone of the Datetime Array/Index. A *tz* of None will convert to UTC and remove the timezone information.

Returns

normalized [same type as self]

Raises

TypeError If Datetime Array/Index is tz-naive.

See also:

DatetimeIndex.tz A timezone that has a variable offset from UTC.

DatetimeIndex.tz_localize Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.

Examples

With the *tz* parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.date_range(start='2014-08-01 09:00',
...                     freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.date_range(start='2014-08-01 09:00',freq='H',
...                     periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
                dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
                dtype='datetime64[ns]', freq='H')
```

pandas.DatetimeIndex.tz_localize

`DatetimeIndex.`**`tz_localize`**`(`**args*, ***kwargs*`)`

Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.

This method takes a time zone (tz) naive Datetime Array/Index object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

Parameters

tz [string, pytz.timezone, dateutil.tz.tzfile or None] Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

ambiguous ['infer', 'NaT', bool array, default 'raise'] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time

(UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

errors [{'raise', 'coerce'}, default None]

- 'raise' will raise a NonExistentTimeError if a timestamp is not valid in the specified time zone (e.g. due to a transition from or to DST time). Use `nonexistent='raise'` instead.
- 'coerce' will return NaT if the timestamp can not be converted to the specified time zone. Use `nonexistent='NaT'` instead.

Deprecated since version 0.24.0.

Returns

result [same type as self] Array/Index converted to the specified time zone.

Raises

TypeError If the Datetime Array/Index is tz-aware and tz is not None.

See also:

`DatetimeIndex.tz_convert` Convert tz-aware DatetimeIndex from one time zone to another.

Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
               '2018-03-02 09:00:00-05:00',
               '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the `tz=None`, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
               '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time: >>> s = pd.to_datetime(pd.Series([… '2018-10-28 01:30:00', … '2018-10-28 02:00:00', … '2018-10-28 02:30:00', … '2018-10-28 02:00:00', … '2018-10-28 02:30:00', … '2018-10-28 03:00:00', … '2018-10-28 03:30:00'])) >>> s.dt.tz_localize('CET', ambiguous='infer') 2018-10-28 01:30:00+02:00 0 2018-10-28 02:00:00+02:00 1 2018-10-28 02:30:00+02:00 2 2018-10-28 02:00:00+01:00 3 2018-10-28 02:30:00+01:00 4 2018-10-28 03:00:00+01:00 5 2018-10-28 03:30:00+01:00 6 dtype: int64

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.to_datetime(pd.Series([
... '2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.dt.tz_localize('CET', ambiguous=np.array([True, True, False]))
0   2018-10-28 01:20:00+02:00
1   2018-10-28 02:36:00+02:00
2   2018-10-28 03:46:00+01:00
dtype: datetime64[ns, CET]
```

If the DST transition causes nonexistent times, you can shift these dates forward or backwards with a timedelta object or *'shift_forward'* or *'shift_backwards'*. >>> s = pd.to_datetime(pd.Series([… '2015-03-29 02:30:00', … '2015-03-29 03:30:00'])) >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_forward') 0 2015-03-29 03:00:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent='shift_backward') 0 2015-03-29 01:59:59.999999999+01:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw'] >>> s.dt.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H')) 0 2015-03-29 03:30:00+02:00 1 2015-03-29 03:30:00+02:00 dtype: datetime64[ns, 'Europe/Warsaw']

pandas.DatetimeIndex.round

`DatetimeIndex.round(*args, **kwargs)`

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
```

(continues on next page)

(continued from previous page)

```
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.DatetimeIndex.floor

`DatetimeIndex.floor(*args, **kwargs)`

Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.DatetimeIndex.ceil

DatetimeIndex.**ceil**(**args*, ***kwargs*)

Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the ‘freq‘ cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.DatetimeIndex.to_period

`DatetimeIndex.`**`to_period`**`(`**args*, ***kwargs*`)`

Cast to PeriodArray/Index at a particular frequency.

Converts DatetimeArray/Index to PeriodArray/Index.

Parameters

freq [string or Offset, optional] One of pandas’ *offset strings* or an Offset object. Will be inferred by default.

Returns

PeriodArray/Index

Raises

ValueError When converting a DatetimeArray/Index with non-regular values, so that a frequency cannot be inferred.

See also:

`PeriodIndex` Immutable ndarray holding ordinal values.

`DatetimeIndex.to_pydatetime` Return DatetimeIndex as object.

Examples

```
>>> df = pd.DataFrame({"y": [1,2,3]},
...                   index=pd.to_datetime(["2000-03-31 00:00:00",
...                                         "2000-05-31 00:00:00",
...                                         "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

pandas.DatetimeIndex.to_perioddelta

`DatetimeIndex.`**`to_perioddelta`**(**args*, ***kwargs*)

Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq. Used for vectorized offsets

Parameters

freq [Period frequency]

Returns

TimedeltaArray/Index

pandas.DatetimeIndex.to_pydatetime

`DatetimeIndex.`**`to_pydatetime`**(**args*, ***kwargs*)

Return Datetime Array/Index as object ndarray of datetime.datetime objects

Returns

datetimes [ndarray]

pandas.DatetimeIndex.to_series

`DatetimeIndex.`**`to_series`**(*keep_tz=None*, *index=None*, *name=None*)

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index

Parameters

keep_tz [optional, defaults False] Return the data keeping the timezone.

If keep_tz is True:

If the timezone is not set, the resulting Series will have a datetime64[ns] dtype.

Otherwise the Series will have an datetime64[ns, tz] dtype; the tz will be preserved.

If keep_tz is False:

Series will have a datetime64[ns] dtype. TZ aware objects will have the tz removed.

Changed in version 0.24: The default value will change to True in a future release. You can set `keep_tz=True` to already obtain the future behaviour and silence the warning.

index [Index, optional] index of resulting Series. If None, defaults to original index

name [string, optional] name of resulting Series. If None, defaults to name of original index

Returns

Series

pandas.DatetimeIndex.to_frame

`DatetimeIndex.`**`to_frame`**(*index=True*, *name=None*)

Create a DataFrame with a column containing the Index.

New in version 0.24.0.

Parameters

index [boolean, default True] Set the index of the returned DataFrame as the original Index.

name [object, default None] The passed name should substitute for the index name (if it has one).

Returns

DataFrame DataFrame containing the original Index data.

See also:

Index.to_series Convert an Index to a Series.

Series.to_frame Convert Series to DataFrame.

Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
       animal
animal
Ant       Ant
Bear     Bear
Cow       Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
    animal
0   Ant
1  Bear
2   Cow
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(index=False, name='zoo')
    zoo
0   Ant
1  Bear
2   Cow
```

pandas.DatetimeIndex.month_name

DatetimeIndex.**month_name**(**args*, ***kwargs*)

Return the month names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the month name. Default is English locale.

Returns

Index Index of month names.

Examples

```
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'],
              dtype='datetime64[ns]', freq='M')
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

pandas.DatetimeIndex.day_name

DatetimeIndex.**day_name**(**args*, ***kwargs*)

Return the day names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the day name. Default is English locale.

Returns

Index Index of day names.

Examples

```
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03'],
              dtype='datetime64[ns]', freq='D')
```

(continues on next page)

(continued from previous page)

```
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

Time/Date Components

DatetimeIndex.year	The year of the datetime.
DatetimeIndex.month	The month as January=1, December=12.
DatetimeIndex.day	The days of the datetime.
DatetimeIndex.hour	The hours of the datetime.
DatetimeIndex.minute	The minutes of the datetime.
DatetimeIndex.second	The seconds of the datetime.
DatetimeIndex.microsecond	The microseconds of the datetime.
DatetimeIndex.nanosecond	The nanoseconds of the datetime.
DatetimeIndex.date	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
DatetimeIndex.time	Returns numpy array of datetime.time.
DatetimeIndex.timetz	Returns numpy array of datetime.time also containing timezone information.
DatetimeIndex.dayofyear	The ordinal day of the year.
DatetimeIndex.weekofyear	The week ordinal of the year.
DatetimeIndex.week	The week ordinal of the year.
DatetimeIndex.dayofweek	The day of the week with Monday=0, Sunday=6.
DatetimeIndex.weekday	The day of the week with Monday=0, Sunday=6.
DatetimeIndex.quarter	The quarter of the date.
DatetimeIndex.tz	
DatetimeIndex.freq	Return the frequency object if it is set, otherwise None.
DatetimeIndex.freqstr	Return the frequency object as a string if it is set, otherwise None.
DatetimeIndex.is_month_start	Indicates whether the date is the first day of the month.
DatetimeIndex.is_month_end	Indicates whether the date is the last day of the month.
DatetimeIndex.is_quarter_start	Indicator for whether the date is the first day of a quarter.
DatetimeIndex.is_quarter_end	Indicator for whether the date is the last day of a quarter.
DatetimeIndex.is_year_start	Indicate whether the date is the first day of a year.
DatetimeIndex.is_year_end	Indicate whether the date is the last day of the year.
DatetimeIndex.is_leap_year	Boolean indicator if the date belongs to a leap year.
DatetimeIndex.inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.

pandas.DatetimeIndex.tz

DatetimeIndex.**tz**

Selecting

`DatetimeIndex.indexer_at_time`(time[, asof])	Returns index locations of index values at particular time of day (e.g.
`DatetimeIndex.indexer_between_time`(…[, …])	Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

pandas.DatetimeIndex.indexer_at_time

`DatetimeIndex.indexer_at_time`(*time*, *asof=False*)

Returns index locations of index values at particular time of day (e.g. 9:30AM).

Parameters

time [datetime.time or string] datetime.time or string in appropriate format ("%H:%M", "%H%M", "%I:%M%p", "%I%M%p", "%H:%M:%S", "%H%M%S", "%I:%M:%S%p", "%I%M%S%p").

Returns

values_at_time [array of integers]

See also:

`indexer_between_time`, `DataFrame.at_time`

pandas.DatetimeIndex.indexer_between_time

`DatetimeIndex.indexer_between_time`(*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

Parameters

start_time, end_time [datetime.time, str] datetime.time or string in appropriate format ("%H:%M", "%H%M", "%I:%M%p", "%I%M%p", "%H:%M:%S", "%H%M%S", "%I:%M:%S%p", "%I%M%S%p").

include_start [boolean, default True]

include_end [boolean, default True]

Returns

values_between_time [array of integers]

See also:

`indexer_at_time`, `DataFrame.between_time`

Time-specific operations

`DatetimeIndex.normalize`(*args, **kwargs)	Convert times to midnight.
`DatetimeIndex.strftime`(*args, **kwargs)	Convert to Index using specified date_format.
`DatetimeIndex.snap`([freq])	Snap time stamps to nearest occurring frequency
`DatetimeIndex.tz_convert`(*args, **kwargs)	Convert tz-aware Datetime Array/Index from one time zone to another.

Continued on next page

Table 196 – continued from previous page

DatetimeIndex.tz_localize(*args, **kwargs)	Localize tz-naive Datetime Array/Index to tz-aware Datetime Array/Index.
DatetimeIndex.round(*args, **kwargs)	Perform round operation on the data to the specified *freq*.
DatetimeIndex.floor(*args, **kwargs)	Perform floor operation on the data to the specified *freq*.
DatetimeIndex.ceil(*args, **kwargs)	Perform ceil operation on the data to the specified *freq*.
DatetimeIndex.month_name(*args, **kwargs)	Return the month names of the DateTimeIndex with specified locale.
DatetimeIndex.day_name(*args, **kwargs)	Return the day names of the DateTimeIndex with specified locale.

Conversion

DatetimeIndex.to_period(*args, **kwargs)	Cast to PeriodArray/Index at a particular frequency.
DatetimeIndex.to_perioddelta(*args, **kwargs)	Calculate TimedeltaArray of difference between index values and index converted to PeriodArray at specified freq.
DatetimeIndex.to_pydatetime(*args, **kwargs)	Return Datetime Array/Index as object ndarray of datetime.datetime objects
DatetimeIndex.to_series([keep_tz, index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
DatetimeIndex.to_frame([index, name])	Create a DataFrame with a column containing the Index.

6.7.7 TimedeltaIndex

TimedeltaIndex	Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects

pandas.TimedeltaIndex

class pandas.**TimedeltaIndex**

Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects

Parameters

data [array-like (1-dimensional), optional] Optional timedelta-like data to construct index with

unit [unit of the arg (D,h,m,s,ms,us,ns) denote the unit, optional] which is an integer/float number

freq [string or pandas offset object, optional] One of pandas date offset strings or corresponding objects. The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation

copy [bool] Make a copy of input ndarray

start [starting value, timedelta-like, optional] If data is None, start is used as the start point in generating regular timedelta data.

Deprecated since version 0.24.0.

periods [int, optional, > 0] Number of periods to generate, if generating index. Takes precedence over end argument

Deprecated since version 0.24.0.

end [end time, timedelta-like, optional] If periods is none, generated index will extend to first conforming time on or just past end argument

Deprecated since version 0.24.: 0

closed [string or None, default None] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

Deprecated since version 0.24.: 0

name [object] Name to be stored in the index

See also:

Index The base pandas Index type.

Timedelta Represents a duration between two dates or times.

DatetimeIndex Index of datetime64 data.

PeriodIndex Index of Period data.

timedelta_range Create a fixed-frequency TimedeltaIndex.

Notes

To learn more about the frequency strings, please see this link.

Creating a TimedeltaIndex based on *start*, *periods*, and *end* has been deprecated in favor of `timedelta_range()`.

Attributes

days	Number of days for each element.
seconds	Number of seconds (>= 0 and less than 1 day) for each element.
microseconds	Number of microseconds (>= 0 and less than 1 second) for each element.
nanoseconds	Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
components	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.
inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.

pandas.TimedeltaIndex.days

`TimedeltaIndex.days`

Number of days for each element.

pandas.TimedeltaIndex.seconds

`TimedeltaIndex.seconds`
Number of seconds (>= 0 and less than 1 day) for each element.

pandas.TimedeltaIndex.microseconds

`TimedeltaIndex.microseconds`
Number of microseconds (>= 0 and less than 1 second) for each element.

pandas.TimedeltaIndex.nanoseconds

`TimedeltaIndex.nanoseconds`
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

pandas.TimedeltaIndex.components

`TimedeltaIndex.components`
Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

Returns

a DataFrame

pandas.TimedeltaIndex.inferred_freq

`TimedeltaIndex.inferred_freq`
Tryies to return a string representing a frequency guess, generated by infer_freq. Returns None if it can't autodetect the frequency.

Methods

to_pytimedelta(*args, **kwargs)	Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.
to_series([index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
round(freq[, ambiguous, nonexistent])	Perform round operation on the data to the specified *freq*.
floor(freq[, ambiguous, nonexistent])	Perform floor operation on the data to the specified *freq*.
ceil(freq[, ambiguous, nonexistent])	Perform ceil operation on the data to the specified *freq*.
to_frame([index, name])	Create a DataFrame with a column containing the Index.

pandas.TimedeltaIndex.to_pytimedelta

`TimedeltaIndex.to_pytimedelta(*args, **kwargs)`

Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.

Returns

datetimes [ndarray]

pandas.TimedeltaIndex.to_series

`TimedeltaIndex.to_series(index=None, name=None)`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.

Parameters

index [Index, optional] index of resulting Series. If None, defaults to original index

name [string, optional] name of resulting Series. If None, defaults to name of original index

Returns

Series [dtype will be based on the type of the Index values.]

pandas.TimedeltaIndex.round

`TimedeltaIndex.round(freq, ambiguous='raise', nonexistent='raise')`

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time

- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.TimedeltaIndex.floor

`TimedeltaIndex.floor`(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.TimedeltaIndex.ceil

`TimedeltaIndex.`**`ceil`**(*freq, ambiguous='raise', nonexistent='raise'*)

Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward, 'NaT', timedelta,]

default 'raise'

A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the 'freq' cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.TimedeltaIndex.to_frame

TimedeltaIndex.**to_frame**(*index=True*, *name=None*)
Create a DataFrame with a column containing the Index.

New in version 0.24.0.

Parameters

index [boolean, default True] Set the index of the returned DataFrame as the original Index.

name [object, default None] The passed name should substitute for the index name (if it has one).

Returns

DataFrame DataFrame containing the original Index data.

See also:

Index.to_series Convert an Index to a Series.

Series.to_frame Convert Series to DataFrame.

Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
       animal
animal
Ant       Ant
Bear     Bear
Cow       Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
    animal
0   Ant
1  Bear
2   Cow
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(index=False, name='zoo')
    zoo
0   Ant
1  Bear
2   Cow
```

Components

TimedeltaIndex.days	Number of days for each element.
TimedeltaIndex.seconds	Number of seconds (>= 0 and less than 1 day) for each element.
TimedeltaIndex.microseconds	Number of microseconds (>= 0 and less than 1 second) for each element.
TimedeltaIndex.nanoseconds	Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
TimedeltaIndex.components	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.
TimedeltaIndex.inferred_freq	Tryies to return a string representing a frequency guess, generated by infer_freq.

Conversion

TimedeltaIndex.to_pytimedelta(*args, **kwargs)	Return Timedelta Array/Index as object ndarray of datetime.timedelta objects.
TimedeltaIndex.to_series([index, name])	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
TimedeltaIndex.round(freq[, ambiguous, …])	Perform round operation on the data to the specified *freq*.
TimedeltaIndex.floor(freq[, ambiguous, …])	Perform floor operation on the data to the specified *freq*.
TimedeltaIndex.ceil(freq[, ambiguous, …])	Perform ceil operation on the data to the specified *freq*.
TimedeltaIndex.to_frame([index, name])	Create a DataFrame with a column containing the Index.

6.7.8 PeriodIndex

PeriodIndex	Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.

pandas.PeriodIndex

class pandas.**PeriodIndex**

Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.

Index keys are boxed to Period objects which carries the metadata (eg, frequency information).

Parameters

data [array-like (1-dimensional), optional] Optional period-like data to construct index with

copy [bool] Make a copy of input ndarray

freq [string or period object, optional] One of pandas period strings or corresponding objects

start [starting value, period-like, optional] If data is None, used as the start point in generating regular period data.

Deprecated since version 0.24.0.

periods [int, optional, > 0] Number of periods to generate, if generating index. Takes precedence over end argument

Deprecated since version 0.24.0.

end [end value, period-like, optional] If periods is none, generated index will extend to first conforming period on or just past end argument

Deprecated since version 0.24.0.

year [int, array, or Series, default None]

month [int, array, or Series, default None]

quarter [int, array, or Series, default None]

day [int, array, or Series, default None]

hour [int, array, or Series, default None]

minute [int, array, or Series, default None]

second [int, array, or Series, default None]

tz [object, default None] Timezone for converting datetime64 data to Periods

dtype [str or PeriodDtype, default None]

See also:

Index The base pandas Index type.

Period Represents a period of time.

DatetimeIndex Index with datetime64 data.

TimedeltaIndex Index of timedelta64 data.

period_range Create a fixed-frequency PeriodIndex.

Notes

Creating a PeriodIndex based on *start*, *periods*, and *end* has been deprecated in favor of `period_range()`.

Examples

```
>>> idx = pd.PeriodIndex(year=year_arr, quarter=q_arr)
```

Attributes

day	The days of the period
dayofweek	The day of the week with Monday=0, Sunday=6
dayofyear	The ordinal day of the year
days_in_month	The number of days in the month
daysinmonth	The number of days in the month
freq	Return the frequency object if it is set, otherwise None.

Continued on next page

Table 204 – continued from previous page

freqstr	Return the frequency object as a string if it is set, otherwise None.
hour	The hour of the period
is_leap_year	Logical indicating if the date belongs to a leap year
minute	The minute of the period
month	The month as January=1, December=12
quarter	The quarter of the date
second	The second of the period
week	The week ordinal of the year
weekday	The day of the week with Monday=0, Sunday=6
weekofyear	The week ordinal of the year
year	The year of the period

pandas.PeriodIndex.day

PeriodIndex.**day**
The days of the period

pandas.PeriodIndex.dayofweek

PeriodIndex.**dayofweek**
The day of the week with Monday=0, Sunday=6

pandas.PeriodIndex.dayofyear

PeriodIndex.**dayofyear**
The ordinal day of the year

pandas.PeriodIndex.days_in_month

PeriodIndex.**days_in_month**
The number of days in the month

pandas.PeriodIndex.daysinmonth

PeriodIndex.**daysinmonth**
The number of days in the month

pandas.PeriodIndex.freq

PeriodIndex.**freq**
Return the frequency object if it is set, otherwise None.

pandas.PeriodIndex.freqstr

PeriodIndex.**freqstr**
Return the frequency object as a string if it is set, otherwise None.

pandas.PeriodIndex.hour

PeriodIndex.**hour**
The hour of the period

pandas.PeriodIndex.is_leap_year

PeriodIndex.**is_leap_year**
Logical indicating if the date belongs to a leap year

pandas.PeriodIndex.minute

PeriodIndex.**minute**
The minute of the period

pandas.PeriodIndex.month

PeriodIndex.**month**
The month as January=1, December=12

pandas.PeriodIndex.quarter

PeriodIndex.**quarter**
The quarter of the date

pandas.PeriodIndex.second

PeriodIndex.**second**
The second of the period

pandas.PeriodIndex.week

PeriodIndex.**week**
The week ordinal of the year

pandas.PeriodIndex.weekday

PeriodIndex.**weekday**
The day of the week with Monday=0, Sunday=6

pandas.PeriodIndex.weekofyear

`PeriodIndex.weekofyear`
The week ordinal of the year

pandas.PeriodIndex.year

`PeriodIndex.year`
The year of the period

end_time	
qyear	
start_time	

Methods

asfreq(*args, **kwargs)	Convert the Period Array/Index to the specified frequency *freq*.
strftime(*args, **kwargs)	Convert to Index using specified date_format.
to_timestamp(*args, **kwargs)	Cast to DatetimeArray/Index.

pandas.PeriodIndex.asfreq

`PeriodIndex.asfreq(*args, **kwargs)`
Convert the Period Array/Index to the specified frequency *freq*.

Parameters

freq [str] a frequency

how [str {'E', 'S'}] 'E', 'END', or 'FINISH' for end, 'S', 'START', or 'BEGIN' for start. Whether the elements should be aligned to the end or start within pa period. January 31st ('END') vs. January 1st ('START') for example.

Returns

new [Period Array/Index with the new frequency]

Examples

```
>>> pidx = pd.period_range('2010-01-01', '2015-01-01', freq='A')
>>> pidx
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010, ..., 2015]
Length: 6, Freq: A-DEC
```

```
>>> pidx.asfreq('M')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-12, ..., 2015-12]
Length: 6, Freq: M
```

```
>>> pidx.asfreq('M', how='S')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-01, ..., 2015-01]
Length: 6, Freq: M
```

pandas.PeriodIndex.strftime

`PeriodIndex.strftime(*args, **kwargs)`

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc

Parameters

date_format [str] Date format string (e.g. "%Y-%m-%d").

Returns

Index Index of formatted strings

See also:

to_datetime Convert the given argument to datetime.

DatetimeIndex.normalize Return DatetimeIndex with times to midnight.

DatetimeIndex.round Round the DatetimeIndex to the specified freq.

DatetimeIndex.floor Floor the DatetimeIndex to the specified freq.

Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

pandas.PeriodIndex.to_timestamp

`PeriodIndex.to_timestamp(*args, **kwargs)`

Cast to DatetimeArray/Index.

Parameters

freq [string or DateOffset, optional] Target frequency. The default is 'D' for week or longer, 'S' otherwise

how [{'s', 'e', 'start', 'end'}]

Returns

DatetimeArray/Index

Properties

PeriodIndex.day	The days of the period
PeriodIndex.dayofweek	The day of the week with Monday=0, Sunday=6
PeriodIndex.dayofyear	The ordinal day of the year
PeriodIndex.days_in_month	The number of days in the month
PeriodIndex.daysinmonth	The number of days in the month
PeriodIndex.end_time	
PeriodIndex.freq	Return the frequency object if it is set, otherwise None.
PeriodIndex.freqstr	Return the frequency object as a string if it is set, otherwise None.
PeriodIndex.hour	The hour of the period
PeriodIndex.is_leap_year	Logical indicating if the date belongs to a leap year
PeriodIndex.minute	The minute of the period
PeriodIndex.month	The month as January=1, December=12
PeriodIndex.quarter	The quarter of the date
PeriodIndex.qyear	
PeriodIndex.second	The second of the period
PeriodIndex.start_time	
PeriodIndex.week	The week ordinal of the year
PeriodIndex.weekday	The day of the week with Monday=0, Sunday=6
PeriodIndex.weekofyear	The week ordinal of the year
PeriodIndex.year	The year of the period

pandas.PeriodIndex.end_time

PeriodIndex.**end_time**

pandas.PeriodIndex.qyear

PeriodIndex.**qyear**

pandas.PeriodIndex.start_time

PeriodIndex.**start_time**

Methods

PeriodIndex.asfreq(*args, **kwargs)	Convert the Period Array/Index to the specified frequency *freq*.
PeriodIndex.strftime(*args, **kwargs)	Convert to Index using specified date_format.
PeriodIndex.to_timestamp(*args, **kwargs)	Cast to DatetimeArray/Index.

6.8 Date Offsets

6.8.1 DateOffset

`DateOffset`([n, normalize])	Standard kind of date increment used for a date range.

pandas.tseries.offsets.DateOffset

class pandas.tseries.offsets.**DateOffset**(*n=1*, *normalize=False*, ***kwds*)

Standard kind of date increment used for a date range.

Works exactly like relativedelta in terms of the keyword args you pass in, use of the keyword n is discouraged– you would be better off specifying n in the keywords you use, but regardless it is there for you. n is needed for DateOffset subclasses.

DateOffets work as follows. Each offset specify a set of dates that conform to the DateOffset. For example, Bday defines this set to be the set of dates that are weekdays (M-F). To test if a date is in the set of a DateOffset dateOffset we can use the onOffset method: dateOffset.onOffset(date).

If a date is not on a valid date, the rollback and rollforward methods can be used to roll the date to the nearest valid date before/after the date.

DateOffsets can be created to move dates forward a given number of valid dates. For example, Bday(2) can be added to a date to move it two business days forward. If the date does not start on a valid date, first it is moved to a valid date. Thus pseudo code is:

def __add__(date): date = rollback(date) # does nothing if date is valid return date + <n number of periods>

When a date offset is created for a negative number of periods, the date is first rolled forward. The pseudo code is:

def __add__(date): date = rollforward(date) # does nothing is date is valid return date + <n number of periods>

Zero presents a problem. Should it roll forward or back? We arbitrarily have it rollforward:

date + BDay(0) == BDay.rollforward(date)

Since 0 is a bit weird, we suggest avoiding its use.

Parameters

n [int, default 1] The number of time periods the offset represents.

normalize [bool, default False] Whether to round the result of a DateOffset addition down to the previous midnight.

****kwds** Temporal parameter that add to or replace the offset value.

Parameters that **add** to the offset (like Timedelta):

- years
- months
- weeks
- days
- hours
- minutes
- seconds

- microseconds
- nanoseconds

Parameters that **replace** the offset value:

- year
- month
- day
- weekday
- hour
- minute
- second
- microsecond
- nanosecond

See also:

dateutil.relativedelta.relativedelta

Examples

```
>>> ts = pd.Timestamp('2017-01-01 09:10:11')
>>> ts + DateOffset(months=3)
Timestamp('2017-04-01 09:10:11')
```

```
>>> ts = pd.Timestamp('2017-01-01 09:10:11')
>>> ts + DateOffset(month=3)
Timestamp('2017-03-01 09:10:11')
```

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.DateOffset.base

DateOffset.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.DateOffset.apply_index

DateOffset.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.DateOffset.rollback

DateOffset.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.DateOffset.rollforward

DateOffset.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

DateOffset.freqstr
DateOffset.kwds
DateOffset.name
DateOffset.nanos
DateOffset.normalize
DateOffset.rule_code

pandas.tseries.offsets.DateOffset.freqstr

`DateOffset.freqstr`

pandas.tseries.offsets.DateOffset.kwds

`DateOffset.kwds`

pandas.tseries.offsets.DateOffset.name

`DateOffset.name`

pandas.tseries.offsets.DateOffset.nanos

`DateOffset.nanos`

pandas.tseries.offsets.DateOffset.normalize

`DateOffset.normalize = False`

pandas.tseries.offsets.DateOffset.rule_code

`DateOffset.rule_code`

Methods

DateOffset.apply(other)
DateOffset.copy
DateOffset.isAnchored()
DateOffset.onOffset(dt)

pandas.tseries.offsets.DateOffset.apply

`DateOffset.apply(other)`

pandas.tseries.offsets.DateOffset.copy

`DateOffset.copy`

pandas.tseries.offsets.DateOffset.isAnchored

`DateOffset.isAnchored()`

pandas.tseries.offsets.DateOffset.onOffset

`DateOffset.onOffset(dt)`

6.8.2 BusinessDay

BusinessDay([n, normalize, offset])	DateOffset subclass representing possibly n business days.

pandas.tseries.offsets.BusinessDay

class `pandas.tseries.offsets.BusinessDay(n=1, normalize=False, offset=datetime.timedelta(0))`

DateOffset subclass representing possibly n business days.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
offset	Alias for self._offset.

pandas.tseries.offsets.BusinessDay.base

`BusinessDay.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BusinessDay.offset

`BusinessDay.offset`

Alias for self._offset.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessDay.rollback

`BusinessDay.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BusinessDay.rollforward

`BusinessDay.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BusinessDay.freqstr
BusinessDay.kwds
BusinessDay.name
BusinessDay.nanos
BusinessDay.normalize
BusinessDay.rule_code

pandas.tseries.offsets.BusinessDay.freqstr

`BusinessDay.freqstr`

pandas.tseries.offsets.BusinessDay.kwds

`BusinessDay.kwds`

pandas.tseries.offsets.BusinessDay.name

`BusinessDay.name`

pandas.tseries.offsets.BusinessDay.nanos

`BusinessDay.nanos`

pandas.tseries.offsets.BusinessDay.normalize

`BusinessDay.normalize = False`

pandas.tseries.offsets.BusinessDay.rule_code

BusinessDay.**rule_code**

Methods

BusinessDay.apply(other)	
BusinessDay.apply_index	
BusinessDay.copy	
BusinessDay.isAnchored()	
BusinessDay.onOffset(dt)	

pandas.tseries.offsets.BusinessDay.apply

BusinessDay.**apply**(*other*)

pandas.tseries.offsets.BusinessDay.apply_index

BusinessDay.**apply_index**

pandas.tseries.offsets.BusinessDay.copy

BusinessDay.**copy**

pandas.tseries.offsets.BusinessDay.isAnchored

BusinessDay.**isAnchored**()

pandas.tseries.offsets.BusinessDay.onOffset

BusinessDay.**onOffset**(*dt*)

6.8.3 BusinessHour

BusinessHour([n, normalize, start, end, offset])	DateOffset subclass representing possibly n business days.

pandas.tseries.offsets.BusinessHour

class pandas.tseries.offsets.**BusinessHour**(*n=1*, *normalize=False*, *start='09:00'*, *end='17:00'*, *offset=datetime.timedelta(0)*)

DateOffset subclass representing possibly n business days.

New in version 0.16.1.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
next_bday	Used for moving to next business day.
offset	Alias for self._offset.

pandas.tseries.offsets.BusinessHour.base

BusinessHour.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BusinessHour.next_bday

BusinessHour.**next_bday**
Used for moving to next business day.

pandas.tseries.offsets.BusinessHour.offset

BusinessHour.**offset**
Alias for self._offset.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessHour.apply_index

BusinessHour.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.BusinessHour.rollback

`BusinessHour.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BusinessHour.rollforward

`BusinessHour.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

BusinessHour.freqstr
BusinessHour.kwds
BusinessHour.name
BusinessHour.nanos
BusinessHour.normalize
BusinessHour.rule_code

pandas.tseries.offsets.BusinessHour.freqstr

`BusinessHour.freqstr`

pandas.tseries.offsets.BusinessHour.kwds

`BusinessHour.kwds`

pandas.tseries.offsets.BusinessHour.name

`BusinessHour.name`

pandas.tseries.offsets.BusinessHour.nanos

`BusinessHour.nanos`

pandas.tseries.offsets.BusinessHour.normalize

`BusinessHour.normalize = False`

pandas.tseries.offsets.BusinessHour.rule_code

`BusinessHour.rule_code`

Methods

BusinessHour.apply(other)	
BusinessHour.copy	
BusinessHour.isAnchored()	
BusinessHour.onOffset(dt)	

pandas.tseries.offsets.BusinessHour.apply

`BusinessHour.apply(other)`

pandas.tseries.offsets.BusinessHour.copy

`BusinessHour.copy`

pandas.tseries.offsets.BusinessHour.isAnchored

`BusinessHour.isAnchored()`

pandas.tseries.offsets.BusinessHour.onOffset

`BusinessHour.onOffset(dt)`

6.8.4 CustomBusinessDay

CustomBusinessDay([n, normalize, weekmask, ...])	DateOffset subclass representing possibly n custom business days, excluding holidays.

pandas.tseries.offsets.CustomBusinessDay

class pandas.tseries.offsets.**CustomBusinessDay**(*n=1, normalize=False, weekmask='Mon Tue Wed Thu Fri', holidays=None, calendar=None, offset=datetime.timedelta(0)*)

DateOffset subclass representing possibly n custom business days, excluding holidays.

Parameters

n [int, default 1]

normalize [bool, default False] Normalize start/end dates to midnight before generating date range

weekmask [str, Default 'Mon Tue Wed Thu Fri'] weekmask of valid business days, passed to `numpy.busdaycalendar`

holidays [list] list/array of dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`

calendar [pd.HolidayCalendar or np.busdaycalendar]

offset [timedelta, default timedelta(0)]

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
offset	Alias for self._offset.

pandas.tseries.offsets.CustomBusinessDay.base

`CustomBusinessDay.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CustomBusinessDay.offset

`CustomBusinessDay.`**`offset`**

Alias for self._offset.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index(i)	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessDay.apply_index

`CustomBusinessDay.`**`apply_index`**(*i*)

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CustomBusinessDay.rollback

`CustomBusinessDay.`**`rollback`**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessDay.rollforward

`CustomBusinessDay.`**`rollforward`**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

CustomBusinessDay.freqstr
CustomBusinessDay.kwds
CustomBusinessDay.name
CustomBusinessDay.nanos
CustomBusinessDay.normalize
CustomBusinessDay.rule_code

pandas.tseries.offsets.CustomBusinessDay.freqstr

`CustomBusinessDay.`**`freqstr`**

pandas.tseries.offsets.CustomBusinessDay.kwds

`CustomBusinessDay.`**`kwds`**

pandas.tseries.offsets.CustomBusinessDay.name

`CustomBusinessDay.`**`name`**

pandas.tseries.offsets.CustomBusinessDay.nanos

`CustomBusinessDay.`**`nanos`**

pandas.tseries.offsets.CustomBusinessDay.normalize

`CustomBusinessDay.normalize = False`

pandas.tseries.offsets.CustomBusinessDay.rule_code

`CustomBusinessDay.rule_code`

Methods

CustomBusinessDay.apply(other)	
CustomBusinessDay.copy	
CustomBusinessDay.isAnchored()	
CustomBusinessDay.onOffset(dt)	

pandas.tseries.offsets.CustomBusinessDay.apply

`CustomBusinessDay.apply(other)`

pandas.tseries.offsets.CustomBusinessDay.copy

`CustomBusinessDay.copy`

pandas.tseries.offsets.CustomBusinessDay.isAnchored

`CustomBusinessDay.isAnchored()`

pandas.tseries.offsets.CustomBusinessDay.onOffset

`CustomBusinessDay.onOffset(dt)`

6.8.5 CustomBusinessHour

CustomBusinessHour([n, normalize, weekmask, ...])	DateOffset subclass representing possibly n custom business days.

pandas.tseries.offsets.CustomBusinessHour

`class pandas.tseries.offsets.CustomBusinessHour(n=1, normalize=False, weekmask='Mon Tue Wed Thu Fri', holidays=None, calendar=None, start='09:00', end='17:00', offset=datetime.timedelta(0))`

DateOffset subclass representing possibly n custom business days.

New in version 0.18.1.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
next_bday	Used for moving to next business day.
offset	Alias for self._offset.

pandas.tseries.offsets.CustomBusinessHour.base

`CustomBusinessHour.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CustomBusinessHour.next_bday

`CustomBusinessHour.`**`next_bday`**

Used for moving to next business day.

pandas.tseries.offsets.CustomBusinessHour.offset

`CustomBusinessHour.`**`offset`**

Alias for self._offset.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessHour.apply_index

`CustomBusinessHour.`**`apply_index`**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CustomBusinessHour.rollback

`CustomBusinessHour.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessHour.rollforward

`CustomBusinessHour.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

CustomBusinessHour.freqstr
CustomBusinessHour.kwds
CustomBusinessHour.name
CustomBusinessHour.nanos
CustomBusinessHour.normalize
CustomBusinessHour.rule_code

pandas.tseries.offsets.CustomBusinessHour.freqstr

`CustomBusinessHour.freqstr`

pandas.tseries.offsets.CustomBusinessHour.kwds

`CustomBusinessHour.kwds`

pandas.tseries.offsets.CustomBusinessHour.name

`CustomBusinessHour.name`

pandas.tseries.offsets.CustomBusinessHour.nanos

`CustomBusinessHour.nanos`

pandas.tseries.offsets.CustomBusinessHour.normalize

CustomBusinessHour.**normalize = False**

pandas.tseries.offsets.CustomBusinessHour.rule_code

CustomBusinessHour.**rule_code**

Methods

CustomBusinessHour.apply(other)
CustomBusinessHour.copy
CustomBusinessHour.isAnchored()
CustomBusinessHour.onOffset(dt)

pandas.tseries.offsets.CustomBusinessHour.apply

CustomBusinessHour.**apply**(*other*)

pandas.tseries.offsets.CustomBusinessHour.copy

CustomBusinessHour.**copy**

pandas.tseries.offsets.CustomBusinessHour.isAnchored

CustomBusinessHour.**isAnchored**()

pandas.tseries.offsets.CustomBusinessHour.onOffset

CustomBusinessHour.**onOffset**(*dt*)

6.8.6 MonthOffset

MonthOffset

Attributes

pandas.tseries.offsets.MonthOffset

class pandas.tseries.offsets.**MonthOffset**

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.MonthOffset.base

MonthOffset.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.MonthOffset.rollback

MonthOffset.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.MonthOffset.rollforward

MonthOffset.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

MonthOffset.freqstr
MonthOffset.kwds
MonthOffset.name

Continued on next page

Table 236 – continued from previous page

MonthOffset.nanos
MonthOffset.normalize
MonthOffset.rule_code

pandas.tseries.offsets.MonthOffset.freqstr

MonthOffset.**freqstr**

pandas.tseries.offsets.MonthOffset.kwds

MonthOffset.**kwds**

pandas.tseries.offsets.MonthOffset.name

MonthOffset.**name**

pandas.tseries.offsets.MonthOffset.nanos

MonthOffset.**nanos**

pandas.tseries.offsets.MonthOffset.normalize

MonthOffset.**normalize = False**

pandas.tseries.offsets.MonthOffset.rule_code

MonthOffset.**rule_code**

Methods

MonthOffset.apply(other)
MonthOffset.apply_index
MonthOffset.copy
MonthOffset.isAnchored()
MonthOffset.onOffset(dt)

pandas.tseries.offsets.MonthOffset.apply

MonthOffset.**apply**(*other*)

pandas.tseries.offsets.MonthOffset.apply_index

MonthOffset.**apply_index**

pandas.tseries.offsets.MonthOffset.copy

MonthOffset.**copy**

pandas.tseries.offsets.MonthOffset.isAnchored

MonthOffset.**isAnchored**()

pandas.tseries.offsets.MonthOffset.onOffset

MonthOffset.**onOffset**(*dt*)

6.8.7 MonthEnd

MonthEnd	DateOffset of one month end.

pandas.tseries.offsets.MonthEnd

class pandas.tseries.offsets.**MonthEnd**
DateOffset of one month end.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.MonthEnd.base

MonthEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.MonthEnd.rollback

`MonthEnd.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.MonthEnd.rollforward

`MonthEnd.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

MonthEnd.freqstr	
MonthEnd.kwds	
MonthEnd.name	
MonthEnd.nanos	
MonthEnd.normalize	
MonthEnd.rule_code	

pandas.tseries.offsets.MonthEnd.freqstr

`MonthEnd.freqstr`

pandas.tseries.offsets.MonthEnd.kwds

`MonthEnd.kwds`

pandas.tseries.offsets.MonthEnd.name

`MonthEnd.name`

pandas.tseries.offsets.MonthEnd.nanos

`MonthEnd.nanos`

pandas.tseries.offsets.MonthEnd.normalize

`MonthEnd.normalize = False`

pandas.tseries.offsets.MonthEnd.rule_code

MonthEnd.**rule_code**

Methods

MonthEnd.apply(other)	
MonthEnd.apply_index	
MonthEnd.copy	
MonthEnd.isAnchored()	
MonthEnd.onOffset(dt)	

pandas.tseries.offsets.MonthEnd.apply

MonthEnd.**apply**(*other*)

pandas.tseries.offsets.MonthEnd.apply_index

MonthEnd.**apply_index**

pandas.tseries.offsets.MonthEnd.copy

MonthEnd.**copy**

pandas.tseries.offsets.MonthEnd.isAnchored

MonthEnd.**isAnchored**()

pandas.tseries.offsets.MonthEnd.onOffset

MonthEnd.**onOffset**(*dt*)

6.8.8 MonthBegin

MonthBegin	DateOffset of one month at beginning.

pandas.tseries.offsets.MonthBegin

class pandas.tseries.offsets.**MonthBegin**

DateOffset of one month at beginning.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.MonthBegin.base

MonthBegin.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.MonthBegin.rollback

MonthBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.MonthBegin.rollforward

MonthBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

MonthBegin.freqstr
MonthBegin.kwds
MonthBegin.name
MonthBegin.nanos
MonthBegin.normalize
MonthBegin.rule_code

pandas.tseries.offsets.MonthBegin.freqstr

MonthBegin.**freqstr**

pandas.tseries.offsets.MonthBegin.kwds

MonthBegin.**kwds**

pandas.tseries.offsets.MonthBegin.name

MonthBegin.**name**

pandas.tseries.offsets.MonthBegin.nanos

MonthBegin.**nanos**

pandas.tseries.offsets.MonthBegin.normalize

MonthBegin.**normalize = False**

pandas.tseries.offsets.MonthBegin.rule_code

MonthBegin.**rule_code**

Methods

MonthBegin.apply(other)
MonthBegin.apply_index
MonthBegin.copy
MonthBegin.isAnchored()
MonthBegin.onOffset(dt)

pandas.tseries.offsets.MonthBegin.apply

MonthBegin.**apply**(*other*)

pandas.tseries.offsets.MonthBegin.apply_index

MonthBegin.**apply_index**

pandas.tseries.offsets.MonthBegin.copy

MonthBegin.**copy**

pandas.tseries.offsets.MonthBegin.isAnchored

MonthBegin.**isAnchored**()

pandas.tseries.offsets.MonthBegin.onOffset

MonthBegin.**onOffset**(*dt*)

6.8.9 BusinessMonthEnd

BusinessMonthEnd	DateOffset increments between business EOM dates.

pandas.tseries.offsets.BusinessMonthEnd

class pandas.tseries.offsets.**BusinessMonthEnd**

DateOffset increments between business EOM dates.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BusinessMonthEnd.base

BusinessMonthEnd.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessMonthEnd.rollback

BusinessMonthEnd.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BusinessMonthEnd.rollforward

BusinessMonthEnd.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BusinessMonthEnd.freqstr
BusinessMonthEnd.kwds
BusinessMonthEnd.name
BusinessMonthEnd.nanos
BusinessMonthEnd.normalize
BusinessMonthEnd.rule_code

pandas.tseries.offsets.BusinessMonthEnd.freqstr

BusinessMonthEnd.**freqstr**

pandas.tseries.offsets.BusinessMonthEnd.kwds

BusinessMonthEnd.**kwds**

pandas.tseries.offsets.BusinessMonthEnd.name

BusinessMonthEnd.**name**

pandas.tseries.offsets.BusinessMonthEnd.nanos

BusinessMonthEnd.**nanos**

pandas.tseries.offsets.BusinessMonthEnd.normalize

BusinessMonthEnd.**normalize = False**

pandas.tseries.offsets.BusinessMonthEnd.rule_code

BusinessMonthEnd.**rule_code**

Methods

BusinessMonthEnd.apply(other)	
BusinessMonthEnd.apply_index	
BusinessMonthEnd.copy	
BusinessMonthEnd.isAnchored()	
BusinessMonthEnd.onOffset(dt)	

pandas.tseries.offsets.BusinessMonthEnd.apply

BusinessMonthEnd.**apply**(*other*)

pandas.tseries.offsets.BusinessMonthEnd.apply_index

BusinessMonthEnd.**apply_index**

pandas.tseries.offsets.BusinessMonthEnd.copy

BusinessMonthEnd.**copy**

pandas.tseries.offsets.BusinessMonthEnd.isAnchored

BusinessMonthEnd.**isAnchored**()

pandas.tseries.offsets.BusinessMonthEnd.onOffset

BusinessMonthEnd.**onOffset**(*dt*)

6.8.10 BusinessMonthBegin

BusinessMonthBegin	DateOffset of one business month at beginning.

pandas.tseries.offsets.BusinessMonthBegin

class pandas.tseries.offsets.**BusinessMonthBegin**

DateOffset of one business month at beginning.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BusinessMonthBegin.base

BusinessMonthBegin.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BusinessMonthBegin.rollback

BusinessMonthBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BusinessMonthBegin.rollforward

BusinessMonthBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BusinessMonthBegin.freqstr
BusinessMonthBegin.kwds
BusinessMonthBegin.name
BusinessMonthBegin.nanos
BusinessMonthBegin.normalize
BusinessMonthBegin.rule_code

pandas.tseries.offsets.BusinessMonthBegin.freqstr

BusinessMonthBegin.**freqstr**

pandas.tseries.offsets.BusinessMonthBegin.kwds

BusinessMonthBegin.**kwds**

pandas.tseries.offsets.BusinessMonthBegin.name

BusinessMonthBegin.**name**

pandas.tseries.offsets.BusinessMonthBegin.nanos

BusinessMonthBegin.**nanos**

pandas.tseries.offsets.BusinessMonthBegin.normalize

BusinessMonthBegin.**normalize = False**

pandas.tseries.offsets.BusinessMonthBegin.rule_code

BusinessMonthBegin.**rule_code**

Methods

BusinessMonthBegin.apply(other)
BusinessMonthBegin.apply_index
BusinessMonthBegin.copy
BusinessMonthBegin.isAnchored()
BusinessMonthBegin.onOffset(dt)

pandas.tseries.offsets.BusinessMonthBegin.apply

BusinessMonthBegin.**apply**(*other*)

pandas.tseries.offsets.BusinessMonthBegin.apply_index

BusinessMonthBegin.**apply_index**

pandas.tseries.offsets.BusinessMonthBegin.copy

BusinessMonthBegin.**copy**

pandas.tseries.offsets.BusinessMonthBegin.isAnchored

BusinessMonthBegin.**isAnchored**()

pandas.tseries.offsets.BusinessMonthBegin.onOffset

`BusinessMonthBegin.onOffset(dt)`

6.8.11 CustomBusinessMonthEnd

CustomBusinessMonthEnd([n, normalize, ...])	DateOffset subclass representing one custom business month, incrementing between end of month dates.

pandas.tseries.offsets.CustomBusinessMonthEnd

class `pandas.tseries.offsets.`**`CustomBusinessMonthEnd`**(*n=1*, *normalize=False*, *weekmask='Mon Tue Wed Thu Fri'*, *holidays=None*, *calendar=None*, *offset=datetime.timedelta(0)*)

DateOffset subclass representing one custom business month, incrementing between end of month dates.

Parameters

n [int, default 1]

normalize [bool, default False] Normalize start/end dates to midnight before generating date range

weekmask [str, Default 'Mon Tue Wed Thu Fri'] weekmask of valid business days, passed to `numpy.busdaycalendar`

holidays [list] list/array of dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`

calendar [pd.HolidayCalendar or np.busdaycalendar]

offset [timedelta, default timedelta(0)]

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
cbday_roll	Define default roll function to be called in apply method.
month_roll	Define default roll function to be called in apply method.
offset	Alias for self._offset.

pandas.tseries.offsets.CustomBusinessMonthEnd.base

`CustomBusinessMonthEnd.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CustomBusinessMonthEnd.cbday_roll

`CustomBusinessMonthEnd.`**`cbday_roll`**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CustomBusinessMonthEnd.month_roll

`CustomBusinessMonthEnd.`**`month_roll`**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CustomBusinessMonthEnd.offset

`CustomBusinessMonthEnd.`**`offset`**

Alias for self._offset.

freqstr	
kwds	
m_offset	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessMonthEnd.apply_index

`CustomBusinessMonthEnd.`**`apply_index`**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CustomBusinessMonthEnd.rollback

`CustomBusinessMonthEnd.`**`rollback`**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessMonthEnd.rollforward

CustomBusinessMonthEnd.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

CustomBusinessMonthEnd.freqstr
CustomBusinessMonthEnd.kwds
CustomBusinessMonthEnd.m_offset
CustomBusinessMonthEnd.name
CustomBusinessMonthEnd.nanos
CustomBusinessMonthEnd.normalize
CustomBusinessMonthEnd.rule_code

pandas.tseries.offsets.CustomBusinessMonthEnd.freqstr

CustomBusinessMonthEnd.**freqstr**

pandas.tseries.offsets.CustomBusinessMonthEnd.kwds

CustomBusinessMonthEnd.**kwds**

pandas.tseries.offsets.CustomBusinessMonthEnd.m_offset

CustomBusinessMonthEnd.**m_offset**

pandas.tseries.offsets.CustomBusinessMonthEnd.name

CustomBusinessMonthEnd.**name**

pandas.tseries.offsets.CustomBusinessMonthEnd.nanos

CustomBusinessMonthEnd.**nanos**

pandas.tseries.offsets.CustomBusinessMonthEnd.normalize

CustomBusinessMonthEnd.**normalize = False**

pandas.tseries.offsets.CustomBusinessMonthEnd.rule_code

`CustomBusinessMonthEnd.`**`rule_code`**

Methods

CustomBusinessMonthEnd.apply(other)	
CustomBusinessMonthEnd.copy	
CustomBusinessMonthEnd.isAnchored()	
CustomBusinessMonthEnd.onOffset(dt)	

pandas.tseries.offsets.CustomBusinessMonthEnd.apply

`CustomBusinessMonthEnd.`**`apply`**`(`*other*`)`

pandas.tseries.offsets.CustomBusinessMonthEnd.copy

`CustomBusinessMonthEnd.`**`copy`**

pandas.tseries.offsets.CustomBusinessMonthEnd.isAnchored

`CustomBusinessMonthEnd.`**`isAnchored`**`()`

pandas.tseries.offsets.CustomBusinessMonthEnd.onOffset

`CustomBusinessMonthEnd.`**`onOffset`**`(`*dt*`)`

6.8.12 CustomBusinessMonthBegin

CustomBusinessMonthBegin([n, normalize, . . .])	DateOffset subclass representing one custom business month, incrementing between beginning of month dates.

pandas.tseries.offsets.CustomBusinessMonthBegin

class `pandas.tseries.offsets.`**`CustomBusinessMonthBegin`**`(`*n=1, normalize=False, weekmask='Mon Tue Wed Thu Fri', holidays=None, calendar=None, offset=datetime.timedelta(0)*`)`

DateOffset subclass representing one custom business month, incrementing between beginning of month dates.

Parameters

n [int, default 1]

normalize [bool, default False] Normalize start/end dates to midnight before generating date range

weekmask [str, Default 'Mon Tue Wed Thu Fri'] weekmask of valid business days, passed to `numpy.busdaycalendar`

holidays [list] list/array of dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`

calendar [pd.HolidayCalendar or np.busdaycalendar]

offset [timedelta, default timedelta(0)]

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
cbday_roll	Define default roll function to be called in apply method.
month_roll	Define default roll function to be called in apply method.
offset	Alias for self._offset.

pandas.tseries.offsets.CustomBusinessMonthBegin.base

`CustomBusinessMonthBegin.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CustomBusinessMonthBegin.cbday_roll

`CustomBusinessMonthBegin.`**`cbday_roll`**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CustomBusinessMonthBegin.month_roll

`CustomBusinessMonthBegin.`**`month_roll`**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CustomBusinessMonthBegin.offset

`CustomBusinessMonthBegin.`**`offset`**

Alias for self._offset.

freqstr	
kwds	
m_offset	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessMonthBegin.apply_index

CustomBusinessMonthBegin.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CustomBusinessMonthBegin.rollback

CustomBusinessMonthBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CustomBusinessMonthBegin.rollforward

CustomBusinessMonthBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

CustomBusinessMonthBegin.freqstr
CustomBusinessMonthBegin.kwds
CustomBusinessMonthBegin.m_offset
CustomBusinessMonthBegin.name
CustomBusinessMonthBegin.nanos
CustomBusinessMonthBegin.normalize
CustomBusinessMonthBegin.rule_code

pandas.tseries.offsets.CustomBusinessMonthBegin.freqstr

CustomBusinessMonthBegin.**freqstr**

pandas.tseries.offsets.CustomBusinessMonthBegin.kwds

CustomBusinessMonthBegin.**kwds**

pandas.tseries.offsets.CustomBusinessMonthBegin.m_offset

CustomBusinessMonthBegin.**m_offset**

pandas.tseries.offsets.CustomBusinessMonthBegin.name

CustomBusinessMonthBegin.**name**

pandas.tseries.offsets.CustomBusinessMonthBegin.nanos

CustomBusinessMonthBegin.**nanos**

pandas.tseries.offsets.CustomBusinessMonthBegin.normalize

CustomBusinessMonthBegin.**normalize = False**

pandas.tseries.offsets.CustomBusinessMonthBegin.rule_code

CustomBusinessMonthBegin.**rule_code**

Methods

CustomBusinessMonthBegin.apply(other)
CustomBusinessMonthBegin.copy
CustomBusinessMonthBegin.isAnchored()
CustomBusinessMonthBegin.onOffset(dt)

pandas.tseries.offsets.CustomBusinessMonthBegin.apply

CustomBusinessMonthBegin.**apply**(*other*)

pandas.tseries.offsets.CustomBusinessMonthBegin.copy

CustomBusinessMonthBegin.**copy**

pandas.tseries.offsets.CustomBusinessMonthBegin.isAnchored

CustomBusinessMonthBegin.**isAnchored**()

pandas.tseries.offsets.CustomBusinessMonthBegin.onOffset

CustomBusinessMonthBegin.**onOffset**(*dt*)

6.8.13 SemiMonthOffset

SemiMonthOffset([n, normalize, day_of_month])	

Attributes

pandas.tseries.offsets.SemiMonthOffset

class pandas.tseries.offsets.**SemiMonthOffset**(*n=1*, *normalize=False*, *day_of_month=None*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.SemiMonthOffset.base

SemiMonthOffset.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthOffset.rollback

`SemiMonthOffset.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthOffset.rollforward

`SemiMonthOffset.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

SemiMonthOffset.freqstr
SemiMonthOffset.kwds
SemiMonthOffset.name
SemiMonthOffset.nanos
SemiMonthOffset.normalize
SemiMonthOffset.rule_code

pandas.tseries.offsets.SemiMonthOffset.freqstr

`SemiMonthOffset.freqstr`

pandas.tseries.offsets.SemiMonthOffset.kwds

`SemiMonthOffset.kwds`

pandas.tseries.offsets.SemiMonthOffset.name

`SemiMonthOffset.name`

pandas.tseries.offsets.SemiMonthOffset.nanos

`SemiMonthOffset.nanos`

pandas.tseries.offsets.SemiMonthOffset.normalize

`SemiMonthOffset.normalize = False`

pandas.tseries.offsets.SemiMonthOffset.rule_code

`SemiMonthOffset.rule_code`

Methods

SemiMonthOffset.apply(other)	
SemiMonthOffset.apply_index	
SemiMonthOffset.copy	
SemiMonthOffset.isAnchored()	
SemiMonthOffset.onOffset(dt)	

pandas.tseries.offsets.SemiMonthOffset.apply

`SemiMonthOffset.apply(other)`

pandas.tseries.offsets.SemiMonthOffset.apply_index

`SemiMonthOffset.apply_index`

pandas.tseries.offsets.SemiMonthOffset.copy

`SemiMonthOffset.copy`

pandas.tseries.offsets.SemiMonthOffset.isAnchored

`SemiMonthOffset.isAnchored()`

pandas.tseries.offsets.SemiMonthOffset.onOffset

`SemiMonthOffset.onOffset(dt)`

6.8.14 SemiMonthEnd

SemiMonthEnd([n, normalize, day_of_month])	Two DateOffset's per month repeating on the last day of the month and day_of_month.

pandas.tseries.offsets.SemiMonthEnd

class `pandas.tseries.offsets.SemiMonthEnd`(*n=1*, *normalize=False*, *day_of_month=None*)

Two DateOffset's per month repeating on the last day of the month and day_of_month.

New in version 0.19.0.

Parameters

n [int]

normalize [bool, default False]

day_of_month [int, {1, 3,. . . ,27}, default 15]

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.SemiMonthEnd.base

`SemiMonthEnd.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthEnd.rollback

`SemiMonthEnd.`**`rollback`**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthEnd.rollforward

`SemiMonthEnd.`**`rollforward`**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

SemiMonthEnd.freqstr	
SemiMonthEnd.kwds	
SemiMonthEnd.name	
SemiMonthEnd.nanos	
SemiMonthEnd.normalize	
SemiMonthEnd.rule_code	

pandas.tseries.offsets.SemiMonthEnd.freqstr

SemiMonthEnd.**freqstr**

pandas.tseries.offsets.SemiMonthEnd.kwds

SemiMonthEnd.**kwds**

pandas.tseries.offsets.SemiMonthEnd.name

SemiMonthEnd.**name**

pandas.tseries.offsets.SemiMonthEnd.nanos

SemiMonthEnd.**nanos**

pandas.tseries.offsets.SemiMonthEnd.normalize

SemiMonthEnd.**normalize = False**

pandas.tseries.offsets.SemiMonthEnd.rule_code

SemiMonthEnd.**rule_code**

Methods

SemiMonthEnd.apply(other)	
SemiMonthEnd.apply_index	
SemiMonthEnd.copy	
SemiMonthEnd.isAnchored()	
SemiMonthEnd.onOffset(dt)	

pandas.tseries.offsets.SemiMonthEnd.apply

SemiMonthEnd.**apply**(*other*)

pandas.tseries.offsets.SemiMonthEnd.apply_index

SemiMonthEnd.**apply_index**

pandas.tseries.offsets.SemiMonthEnd.copy

SemiMonthEnd.**copy**

pandas.tseries.offsets.SemiMonthEnd.isAnchored

SemiMonthEnd.**isAnchored**()

pandas.tseries.offsets.SemiMonthEnd.onOffset

SemiMonthEnd.**onOffset**(*dt*)

6.8.15 SemiMonthBegin

SemiMonthBegin([n, normalize, day_of_month])	Two DateOffset's per month repeating on the first day of the month and day_of_month.

pandas.tseries.offsets.SemiMonthBegin

class pandas.tseries.offsets.**SemiMonthBegin**(*n=1*, *normalize=False*, *day_of_month=None*)

Two DateOffset's per month repeating on the first day of the month and day_of_month.

New in version 0.19.0.

Parameters

n [int]

normalize [bool, default False]

day_of_month [int, {2, 3,…,27}, default 15]

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.SemiMonthBegin.base

SemiMonthBegin.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthBegin.rollback

SemiMonthBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.SemiMonthBegin.rollforward

SemiMonthBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

SemiMonthBegin.freqstr
SemiMonthBegin.kwds
SemiMonthBegin.name
SemiMonthBegin.nanos
SemiMonthBegin.normalize
SemiMonthBegin.rule_code

pandas.tseries.offsets.SemiMonthBegin.freqstr

SemiMonthBegin.**freqstr**

pandas.tseries.offsets.SemiMonthBegin.kwds

SemiMonthBegin.**kwds**

pandas.tseries.offsets.SemiMonthBegin.name

SemiMonthBegin.**name**

pandas.tseries.offsets.SemiMonthBegin.nanos

SemiMonthBegin.**nanos**

pandas.tseries.offsets.SemiMonthBegin.normalize

SemiMonthBegin.**normalize = False**

pandas.tseries.offsets.SemiMonthBegin.rule_code

SemiMonthBegin.**rule_code**

Methods

SemiMonthBegin.apply(other)
SemiMonthBegin.apply_index
SemiMonthBegin.copy
SemiMonthBegin.isAnchored()
SemiMonthBegin.onOffset(dt)

pandas.tseries.offsets.SemiMonthBegin.apply

SemiMonthBegin.**apply**(*other*)

pandas.tseries.offsets.SemiMonthBegin.apply_index

SemiMonthBegin.**apply_index**

pandas.tseries.offsets.SemiMonthBegin.copy

SemiMonthBegin.**copy**

pandas.tseries.offsets.SemiMonthBegin.isAnchored

SemiMonthBegin.**isAnchored**()

pandas.tseries.offsets.SemiMonthBegin.onOffset

SemiMonthBegin.**onOffset**(*dt*)

6.8.16 Week

Week([n, normalize, weekday])	Weekly offset.

pandas.tseries.offsets.Week

class pandas.tseries.offsets.**Week**(*n=1*, *normalize=False*, *weekday=None*)

Weekly offset.

Parameters

weekday [int, default None] Always generate specific day of week. 0 for Monday

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Week.base

Week.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Week.rollback

Week.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Week.rollforward

Week.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

Week.freqstr
Week.kwds
Week.name
Week.nanos
Week.normalize
Week.rule_code

pandas.tseries.offsets.Week.freqstr

Week.**freqstr**

pandas.tseries.offsets.Week.kwds

Week.**kwds**

pandas.tseries.offsets.Week.name

Week.**name**

pandas.tseries.offsets.Week.nanos

Week.**nanos**

pandas.tseries.offsets.Week.normalize

Week.**normalize = False**

pandas.tseries.offsets.Week.rule_code

Week.**rule_code**

Methods

Week.apply(other)

Continued on next page

Table 287 – continued from previous page

Week.apply_index	
Week.copy	
Week.isAnchored()	
Week.onOffset(dt)	

pandas.tseries.offsets.Week.apply

Week.**apply**(*other*)

pandas.tseries.offsets.Week.apply_index

Week.**apply_index**

pandas.tseries.offsets.Week.copy

Week.**copy**

pandas.tseries.offsets.Week.isAnchored

Week.**isAnchored**()

pandas.tseries.offsets.Week.onOffset

Week.**onOffset**(*dt*)

6.8.17 WeekOfMonth

WeekOfMonth([n, normalize, week, weekday])	Describes monthly dates like "the Tuesday of the 2nd week of each month".

pandas.tseries.offsets.WeekOfMonth

class pandas.tseries.offsets.**WeekOfMonth**(*n=1*, *normalize=False*, *week=0*, *weekday=0*)

Describes monthly dates like "the Tuesday of the 2nd week of each month".

Parameters

n [int]

week [{0, 1, 2, 3, . . . }, default 0] 0 is 1st week of month, 1 2nd week, etc.

weekday [{0, 1, . . . , 6}, default 0] 0: Mondays 1: Tuesdays 2: Wednesdays 3: Thursdays 4: Fridays 5: Saturdays 6: Sundays

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.WeekOfMonth.base

WeekOfMonth.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.WeekOfMonth.apply_index

WeekOfMonth.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.WeekOfMonth.rollback

WeekOfMonth.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.WeekOfMonth.rollforward

WeekOfMonth.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

WeekOfMonth.freqstr
WeekOfMonth.kwds
WeekOfMonth.name
WeekOfMonth.nanos
WeekOfMonth.normalize
WeekOfMonth.rule_code

pandas.tseries.offsets.WeekOfMonth.freqstr

WeekOfMonth.**freqstr**

pandas.tseries.offsets.WeekOfMonth.kwds

WeekOfMonth.**kwds**

pandas.tseries.offsets.WeekOfMonth.name

WeekOfMonth.**name**

pandas.tseries.offsets.WeekOfMonth.nanos

WeekOfMonth.**nanos**

pandas.tseries.offsets.WeekOfMonth.normalize

WeekOfMonth.**normalize = False**

pandas.tseries.offsets.WeekOfMonth.rule_code

WeekOfMonth.**rule_code**

Methods

WeekOfMonth.apply(other)
WeekOfMonth.copy

Continued on next page

Table 292 – continued from previous page

WeekOfMonth.isAnchored()	
WeekOfMonth.onOffset(dt)	

pandas.tseries.offsets.WeekOfMonth.apply

WeekOfMonth.**apply**(*other*)

pandas.tseries.offsets.WeekOfMonth.copy

WeekOfMonth.**copy**

pandas.tseries.offsets.WeekOfMonth.isAnchored

WeekOfMonth.**isAnchored**()

pandas.tseries.offsets.WeekOfMonth.onOffset

WeekOfMonth.**onOffset**(*dt*)

6.8.18 LastWeekOfMonth

LastWeekOfMonth([n, normalize, weekday])	Describes monthly dates in last week of month like "the last Tuesday of each month".

pandas.tseries.offsets.LastWeekOfMonth

class pandas.tseries.offsets.**LastWeekOfMonth**(*n=1*, *normalize=False*, *weekday=0*)

Describes monthly dates in last week of month like "the last Tuesday of each month".

Parameters

n [int, default 1]

weekday [{0, 1, . . . , 6}, default 0] 0: Mondays 1: Tuesdays 2: Wednesdays 3: Thursdays 4: Fridays 5: Saturdays 6: Sundays

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.LastWeekOfMonth.base

LastWeekOfMonth.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.LastWeekOfMonth.apply_index

LastWeekOfMonth.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.LastWeekOfMonth.rollback

LastWeekOfMonth.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.LastWeekOfMonth.rollforward

LastWeekOfMonth.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

LastWeekOfMonth.freqstr
LastWeekOfMonth.kwds
LastWeekOfMonth.name
LastWeekOfMonth.nanos
LastWeekOfMonth.normalize
LastWeekOfMonth.rule_code

pandas.tseries.offsets.LastWeekOfMonth.freqstr

LastWeekOfMonth.**freqstr**

pandas.tseries.offsets.LastWeekOfMonth.kwds

LastWeekOfMonth.**kwds**

pandas.tseries.offsets.LastWeekOfMonth.name

LastWeekOfMonth.**name**

pandas.tseries.offsets.LastWeekOfMonth.nanos

LastWeekOfMonth.**nanos**

pandas.tseries.offsets.LastWeekOfMonth.normalize

LastWeekOfMonth.**normalize = False**

pandas.tseries.offsets.LastWeekOfMonth.rule_code

LastWeekOfMonth.**rule_code**

Methods

LastWeekOfMonth.apply(other)
LastWeekOfMonth.copy
LastWeekOfMonth.isAnchored()
LastWeekOfMonth.onOffset(dt)

pandas.tseries.offsets.LastWeekOfMonth.apply

LastWeekOfMonth.**apply**(*other*)

pandas.tseries.offsets.LastWeekOfMonth.copy

`LastWeekOfMonth.copy`

pandas.tseries.offsets.LastWeekOfMonth.isAnchored

`LastWeekOfMonth.isAnchored()`

pandas.tseries.offsets.LastWeekOfMonth.onOffset

`LastWeekOfMonth.onOffset(dt)`

6.8.19 QuarterOffset

QuarterOffset([n, normalize, startingMonth])	Quarter representation - doesn't call super.

pandas.tseries.offsets.QuarterOffset

class pandas.tseries.offsets.**QuarterOffset**(*n=1*, *normalize=False*, *startingMonth=None*)
Quarter representation - doesn't call super.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.QuarterOffset.base

`QuarterOffset.base`
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.QuarterOffset.rollback

QuarterOffset.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.QuarterOffset.rollforward

QuarterOffset.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

QuarterOffset.freqstr	
QuarterOffset.kwds	
QuarterOffset.name	
QuarterOffset.nanos	
QuarterOffset.normalize	
QuarterOffset.rule_code	

pandas.tseries.offsets.QuarterOffset.freqstr

QuarterOffset.**freqstr**

pandas.tseries.offsets.QuarterOffset.kwds

QuarterOffset.**kwds**

pandas.tseries.offsets.QuarterOffset.name

QuarterOffset.**name**

pandas.tseries.offsets.QuarterOffset.nanos

QuarterOffset.**nanos**

pandas.tseries.offsets.QuarterOffset.normalize

QuarterOffset.**normalize = False**

pandas.tseries.offsets.QuarterOffset.rule_code

QuarterOffset.**rule_code**

Methods

QuarterOffset.apply(other)	
QuarterOffset.apply_index	
QuarterOffset.copy	
QuarterOffset.isAnchored()	
QuarterOffset.onOffset(dt)	

pandas.tseries.offsets.QuarterOffset.apply

QuarterOffset.**apply**(*other*)

pandas.tseries.offsets.QuarterOffset.apply_index

QuarterOffset.**apply_index**

pandas.tseries.offsets.QuarterOffset.copy

QuarterOffset.**copy**

pandas.tseries.offsets.QuarterOffset.isAnchored

QuarterOffset.**isAnchored**()

pandas.tseries.offsets.QuarterOffset.onOffset

QuarterOffset.**onOffset**(*dt*)

6.8.20 BQuarterEnd

BQuarterEnd([n, normalize, startingMonth])	DateOffset increments between business Quarter dates.

pandas.tseries.offsets.BQuarterEnd

class pandas.tseries.offsets.**BQuarterEnd**(*n=1*, *normalize=False*, *startingMonth=None*)

DateOffset increments between business Quarter dates.

startingMonth = 1 corresponds to dates like 1/31/2007, 4/30/2007, … startingMonth = 2 corresponds to dates like 2/28/2007, 5/31/2007, … startingMonth = 3 corresponds to dates like 3/30/2007, 6/29/2007, …

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BQuarterEnd.base

BQuarterEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BQuarterEnd.rollback

BQuarterEnd.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BQuarterEnd.rollforward

BQuarterEnd.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BQuarterEnd.freqstr
BQuarterEnd.kwds
BQuarterEnd.name

Continued on next page

Table 306 – continued from previous page

BQuarterEnd.nanos
BQuarterEnd.normalize
BQuarterEnd.rule_code

pandas.tseries.offsets.BQuarterEnd.freqstr

BQuarterEnd.**freqstr**

pandas.tseries.offsets.BQuarterEnd.kwds

BQuarterEnd.**kwds**

pandas.tseries.offsets.BQuarterEnd.name

BQuarterEnd.**name**

pandas.tseries.offsets.BQuarterEnd.nanos

BQuarterEnd.**nanos**

pandas.tseries.offsets.BQuarterEnd.normalize

BQuarterEnd.**normalize = False**

pandas.tseries.offsets.BQuarterEnd.rule_code

BQuarterEnd.**rule_code**

Methods

BQuarterEnd.apply(other)
BQuarterEnd.apply_index
BQuarterEnd.copy
BQuarterEnd.isAnchored()
BQuarterEnd.onOffset(dt)

pandas.tseries.offsets.BQuarterEnd.apply

BQuarterEnd.**apply**(*other*)

pandas.tseries.offsets.BQuarterEnd.apply_index

BQuarterEnd.**apply_index**

pandas.tseries.offsets.BQuarterEnd.copy

`BQuarterEnd.copy`

pandas.tseries.offsets.BQuarterEnd.isAnchored

`BQuarterEnd.isAnchored()`

pandas.tseries.offsets.BQuarterEnd.onOffset

`BQuarterEnd.onOffset(dt)`

6.8.21 BQuarterBegin

BQuarterBegin([n, normalize, startingMonth])

Attributes

---

pandas.tseries.offsets.BQuarterBegin

class `pandas.tseries.offsets.BQuarterBegin`(*n=1*, *normalize=False*, *startingMonth=None*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BQuarterBegin.base

`BQuarterBegin.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.

Continued on next page

Table 310 – continued from previous page

rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BQuarterBegin.rollback

BQuarterBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BQuarterBegin.rollforward

BQuarterBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BQuarterBegin.freqstr
BQuarterBegin.kwds
BQuarterBegin.name
BQuarterBegin.nanos
BQuarterBegin.normalize
BQuarterBegin.rule_code

pandas.tseries.offsets.BQuarterBegin.freqstr

BQuarterBegin.**freqstr**

pandas.tseries.offsets.BQuarterBegin.kwds

BQuarterBegin.**kwds**

pandas.tseries.offsets.BQuarterBegin.name

BQuarterBegin.**name**

pandas.tseries.offsets.BQuarterBegin.nanos

BQuarterBegin.**nanos**

pandas.tseries.offsets.BQuarterBegin.normalize

BQuarterBegin.**normalize = False**

pandas.tseries.offsets.BQuarterBegin.rule_code

BQuarterBegin.**rule_code**

Methods

BQuarterBegin.apply(other)	
BQuarterBegin.apply_index	
BQuarterBegin.copy	
BQuarterBegin.isAnchored()	
BQuarterBegin.onOffset(dt)	

pandas.tseries.offsets.BQuarterBegin.apply

BQuarterBegin.**apply**(*other*)

pandas.tseries.offsets.BQuarterBegin.apply_index

BQuarterBegin.**apply_index**

pandas.tseries.offsets.BQuarterBegin.copy

BQuarterBegin.**copy**

pandas.tseries.offsets.BQuarterBegin.isAnchored

BQuarterBegin.**isAnchored**()

pandas.tseries.offsets.BQuarterBegin.onOffset

BQuarterBegin.**onOffset**(*dt*)

6.8.22 QuarterEnd

QuarterEnd([n, normalize, startingMonth])	DateOffset increments between business Quarter dates.

pandas.tseries.offsets.QuarterEnd

class pandas.tseries.offsets.**QuarterEnd**(*n=1*, *normalize=False*, *startingMonth=None*)

DateOffset increments between business Quarter dates.

startingMonth = 1 corresponds to dates like 1/31/2007, 4/30/2007, … startingMonth = 2 corresponds to dates like 2/28/2007, 5/31/2007, … startingMonth = 3 corresponds to dates like 3/31/2007, 6/30/2007, …

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.QuarterEnd.base

`QuarterEnd.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.QuarterEnd.rollback

`QuarterEnd.`**`rollback`**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.QuarterEnd.rollforward

`QuarterEnd.`**`rollforward`**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

QuarterEnd.freqstr
QuarterEnd.kwds
QuarterEnd.name
QuarterEnd.nanos
QuarterEnd.normalize
QuarterEnd.rule_code

pandas.tseries.offsets.QuarterEnd.freqstr

QuarterEnd.**freqstr**

pandas.tseries.offsets.QuarterEnd.kwds

QuarterEnd.**kwds**

pandas.tseries.offsets.QuarterEnd.name

QuarterEnd.**name**

pandas.tseries.offsets.QuarterEnd.nanos

QuarterEnd.**nanos**

pandas.tseries.offsets.QuarterEnd.normalize

QuarterEnd.**normalize = False**

pandas.tseries.offsets.QuarterEnd.rule_code

QuarterEnd.**rule_code**

Methods

QuarterEnd.apply(other)
QuarterEnd.apply_index
QuarterEnd.copy
QuarterEnd.isAnchored()
QuarterEnd.onOffset(dt)

pandas.tseries.offsets.QuarterEnd.apply

QuarterEnd.**apply**(*other*)

pandas.tseries.offsets.QuarterEnd.apply_index

`QuarterEnd.apply_index`

pandas.tseries.offsets.QuarterEnd.copy

`QuarterEnd.copy`

pandas.tseries.offsets.QuarterEnd.isAnchored

`QuarterEnd.isAnchored()`

pandas.tseries.offsets.QuarterEnd.onOffset

`QuarterEnd.onOffset(dt)`

6.8.23 QuarterBegin

QuarterBegin([n, normalize, startingMonth])	

Attributes

pandas.tseries.offsets.QuarterBegin

class pandas.tseries.offsets.**QuarterBegin**(*n=1*, *normalize=False*, *startingMonth=None*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.QuarterBegin.base

`QuarterBegin.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.QuarterBegin.rollback

QuarterBegin.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.QuarterBegin.rollforward

QuarterBegin.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

QuarterBegin.freqstr
QuarterBegin.kwds
QuarterBegin.name
QuarterBegin.nanos
QuarterBegin.normalize
QuarterBegin.rule_code

pandas.tseries.offsets.QuarterBegin.freqstr

QuarterBegin.**freqstr**

pandas.tseries.offsets.QuarterBegin.kwds

QuarterBegin.**kwds**

pandas.tseries.offsets.QuarterBegin.name

QuarterBegin.**name**

pandas.tseries.offsets.QuarterBegin.nanos

QuarterBegin.**nanos**

pandas.tseries.offsets.QuarterBegin.normalize

QuarterBegin.**normalize = False**

pandas.tseries.offsets.QuarterBegin.rule_code

QuarterBegin.**rule_code**

Methods

QuarterBegin.apply(other)	
QuarterBegin.apply_index	
QuarterBegin.copy	
QuarterBegin.isAnchored()	
QuarterBegin.onOffset(dt)	

pandas.tseries.offsets.QuarterBegin.apply

QuarterBegin.**apply**(*other*)

pandas.tseries.offsets.QuarterBegin.apply_index

QuarterBegin.**apply_index**

pandas.tseries.offsets.QuarterBegin.copy

QuarterBegin.**copy**

pandas.tseries.offsets.QuarterBegin.isAnchored

QuarterBegin.**isAnchored**()

pandas.tseries.offsets.QuarterBegin.onOffset

QuarterBegin.**onOffset**(*dt*)

6.8.24 YearOffset

YearOffset([n, normalize, month])	DateOffset that just needs a month.

pandas.tseries.offsets.YearOffset

class pandas.tseries.offsets.**YearOffset**(*n=1*, *normalize=False*, *month=None*)
DateOffset that just needs a month.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.YearOffset.base

YearOffset.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.YearOffset.rollback

YearOffset.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.YearOffset.rollforward

YearOffset.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

YearOffset.freqstr
YearOffset.kwds
YearOffset.name
YearOffset.nanos
YearOffset.normalize
YearOffset.rule_code

pandas.tseries.offsets.YearOffset.freqstr

YearOffset.**freqstr**

pandas.tseries.offsets.YearOffset.kwds

YearOffset.**kwds**

pandas.tseries.offsets.YearOffset.name

YearOffset.**name**

pandas.tseries.offsets.YearOffset.nanos

YearOffset.**nanos**

pandas.tseries.offsets.YearOffset.normalize

YearOffset.**normalize = False**

pandas.tseries.offsets.YearOffset.rule_code

YearOffset.**rule_code**

Methods

YearOffset.apply(other)
YearOffset.apply_index
YearOffset.copy
YearOffset.isAnchored()
YearOffset.onOffset(dt)

pandas.tseries.offsets.YearOffset.apply

YearOffset.**apply**(*other*)

pandas.tseries.offsets.YearOffset.apply_index

YearOffset.**apply_index**

pandas.tseries.offsets.YearOffset.copy

YearOffset.**copy**

pandas.tseries.offsets.YearOffset.isAnchored

YearOffset.**isAnchored**()

pandas.tseries.offsets.YearOffset.onOffset

YearOffset.**onOffset**(*dt*)

6.8.25 BYearEnd

BYearEnd([n, normalize, month])	DateOffset increments between business EOM dates.

pandas.tseries.offsets.BYearEnd

class pandas.tseries.offsets.**BYearEnd**(*n=1*, *normalize=False*, *month=None*)
DateOffset increments between business EOM dates.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BYearEnd.base

BYearEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BYearEnd.rollback

BYearEnd.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BYearEnd.rollforward

BYearEnd.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BYearEnd.freqstr
BYearEnd.kwds
BYearEnd.name
BYearEnd.nanos
BYearEnd.normalize
BYearEnd.rule_code

pandas.tseries.offsets.BYearEnd.freqstr

BYearEnd.**freqstr**

pandas.tseries.offsets.BYearEnd.kwds

BYearEnd.**kwds**

pandas.tseries.offsets.BYearEnd.name

BYearEnd.**name**

pandas.tseries.offsets.BYearEnd.nanos

BYearEnd.**nanos**

pandas.tseries.offsets.BYearEnd.normalize

BYearEnd.**normalize = False**

pandas.tseries.offsets.BYearEnd.rule_code

BYearEnd.**rule_code**

Methods

BYearEnd.apply(other)	
BYearEnd.apply_index	
BYearEnd.copy	
BYearEnd.isAnchored()	
BYearEnd.onOffset(dt)	

pandas.tseries.offsets.BYearEnd.apply

BYearEnd.**apply**(*other*)

pandas.tseries.offsets.BYearEnd.apply_index

BYearEnd.**apply_index**

pandas.tseries.offsets.BYearEnd.copy

BYearEnd.**copy**

pandas.tseries.offsets.BYearEnd.isAnchored

BYearEnd.**isAnchored**()

pandas.tseries.offsets.BYearEnd.onOffset

BYearEnd.**onOffset**(*dt*)

6.8.26 BYearBegin

BYearBegin([n, normalize, month])	DateOffset increments between business year begin dates.

pandas.tseries.offsets.BYearBegin

class pandas.tseries.offsets.**BYearBegin**(*n=1*, *normalize=False*, *month=None*)
DateOffset increments between business year begin dates.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BYearBegin.base

BYearBegin.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BYearBegin.rollback

BYearBegin.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BYearBegin.rollforward

BYearBegin.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

BYearBegin.freqstr
BYearBegin.kwds
BYearBegin.name
BYearBegin.nanos
BYearBegin.normalize
BYearBegin.rule_code

pandas.tseries.offsets.BYearBegin.freqstr

BYearBegin.**freqstr**

pandas.tseries.offsets.BYearBegin.kwds

BYearBegin.**kwds**

pandas.tseries.offsets.BYearBegin.name

BYearBegin.**name**

pandas.tseries.offsets.BYearBegin.nanos

BYearBegin.**nanos**

pandas.tseries.offsets.BYearBegin.normalize

BYearBegin.**normalize = False**

pandas.tseries.offsets.BYearBegin.rule_code

BYearBegin.**rule_code**

Methods

BYearBegin.apply(other)
BYearBegin.apply_index
BYearBegin.copy
BYearBegin.isAnchored()
BYearBegin.onOffset(dt)

pandas.tseries.offsets.BYearBegin.apply

BYearBegin.**apply**(*other*)

pandas.tseries.offsets.BYearBegin.apply_index

BYearBegin.**apply_index**

pandas.tseries.offsets.BYearBegin.copy

BYearBegin.**copy**

pandas.tseries.offsets.BYearBegin.isAnchored

BYearBegin.**isAnchored**()

pandas.tseries.offsets.BYearBegin.onOffset

BYearBegin.**onOffset**(*dt*)

6.8.27 YearEnd

YearEnd([n, normalize, month])	DateOffset increments between calendar year ends.

pandas.tseries.offsets.YearEnd

class pandas.tseries.offsets.**YearEnd**(*n=1*, *normalize=False*, *month=None*)
DateOffset increments between calendar year ends.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.YearEnd.base

YearEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.YearEnd.rollback

YearEnd.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.YearEnd.rollforward

YearEnd.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

YearEnd.freqstr
YearEnd.kwds
YearEnd.name
YearEnd.nanos
YearEnd.normalize
YearEnd.rule_code

pandas.tseries.offsets.YearEnd.freqstr

YearEnd.**freqstr**

pandas.tseries.offsets.YearEnd.kwds

YearEnd.**kwds**

pandas.tseries.offsets.YearEnd.name

YearEnd.**name**

pandas.tseries.offsets.YearEnd.nanos

YearEnd.**nanos**

pandas.tseries.offsets.YearEnd.normalize

YearEnd.**normalize = False**

pandas.tseries.offsets.YearEnd.rule_code

YearEnd.**rule_code**

Methods

YearEnd.apply(other)	
YearEnd.apply_index	
YearEnd.copy	
YearEnd.isAnchored()	
YearEnd.onOffset(dt)	

pandas.tseries.offsets.YearEnd.apply

YearEnd.**apply**(*other*)

pandas.tseries.offsets.YearEnd.apply_index

YearEnd.**apply_index**

pandas.tseries.offsets.YearEnd.copy

YearEnd.**copy**

pandas.tseries.offsets.YearEnd.isAnchored

YearEnd.**isAnchored**()

pandas.tseries.offsets.YearEnd.onOffset

YearEnd.**onOffset**(*dt*)

6.8.28 YearBegin

YearBegin([n, normalize, month])	DateOffset increments between calendar year begin dates.

pandas.tseries.offsets.YearBegin

class pandas.tseries.offsets.**YearBegin**(*n=1*, *normalize=False*, *month=None*)
DateOffset increments between calendar year begin dates.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.YearBegin.base

YearBegin.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.YearBegin.rollback

YearBegin.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.YearBegin.rollforward

YearBegin.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
apply_index	
copy	
isAnchored	
onOffset	

Properties

YearBegin.freqstr
YearBegin.kwds
YearBegin.name
YearBegin.nanos
YearBegin.normalize
YearBegin.rule_code

pandas.tseries.offsets.YearBegin.freqstr

YearBegin.**freqstr**

pandas.tseries.offsets.YearBegin.kwds

YearBegin.**kwds**

pandas.tseries.offsets.YearBegin.name

YearBegin.**name**

pandas.tseries.offsets.YearBegin.nanos

YearBegin.**nanos**

pandas.tseries.offsets.YearBegin.normalize

YearBegin.**normalize = False**

pandas.tseries.offsets.YearBegin.rule_code

YearBegin.**rule_code**

Methods

YearBegin.apply(other)
YearBegin.apply_index
YearBegin.copy
YearBegin.isAnchored()
YearBegin.onOffset(dt)

pandas.tseries.offsets.YearBegin.apply

YearBegin.**apply**(*other*)

pandas.tseries.offsets.YearBegin.apply_index

`YearBegin.apply_index`

pandas.tseries.offsets.YearBegin.copy

`YearBegin.copy`

pandas.tseries.offsets.YearBegin.isAnchored

`YearBegin.isAnchored()`

pandas.tseries.offsets.YearBegin.onOffset

`YearBegin.onOffset(dt)`

6.8.29 FY5253

FY5253([n, normalize, weekday, . . .])	Describes 52-53 week fiscal year.

pandas.tseries.offsets.FY5253

class pandas.tseries.offsets.**FY5253**(*n=1*, *normalize=False*, *weekday=0*, *startingMonth=1*, *variation='nearest'*)

Describes 52-53 week fiscal year. This is also known as a 4-4-5 calendar.

It is used by companies that desire that their fiscal year always end on the same day of the week.

It is a method of managing accounting periods. It is a common calendar structure for some industries, such as retail, manufacturing and parking industry.

For more information see: http://en.wikipedia.org/wiki/4-4-5_calendar

The year may either: - end on the last X day of the Y month. - end on the last X day closest to the last day of the Y month.

X is a specific day of the week. Y is a certain month of the year

Parameters

n [int]

weekday [{0, 1, . . . , 6}] 0: Mondays 1: Tuesdays 2: Wednesdays 3: Thursdays 4: Fridays 5: Saturdays 6: Sundays

startingMonth [The month in which fiscal years end. {1, 2, . . . 12}]

variation [str] {"nearest", "last"} for "LastOfMonth" or "NearestEndMonth"

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.FY5253.base

FY5253.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.FY5253.apply_index

FY5253.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.FY5253.rollback

FY5253.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.FY5253.rollforward

FY5253.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
get_rule_code_suffix	
get_year_end	
isAnchored	
onOffset	

Properties

FY5253.freqstr
FY5253.kwds
FY5253.name
FY5253.nanos
FY5253.normalize
FY5253.rule_code

pandas.tseries.offsets.FY5253.freqstr

FY5253.**freqstr**

pandas.tseries.offsets.FY5253.kwds

FY5253.**kwds**

pandas.tseries.offsets.FY5253.name

FY5253.**name**

pandas.tseries.offsets.FY5253.nanos

FY5253.**nanos**

pandas.tseries.offsets.FY5253.normalize

FY5253.**normalize = False**

pandas.tseries.offsets.FY5253.rule_code

FY5253.**rule_code**

Methods

FY5253.apply(other)	
FY5253.copy	
FY5253.get_rule_code_suffix()	
FY5253.get_year_end(dt)	
FY5253.isAnchored()	
FY5253.onOffset(dt)	

pandas.tseries.offsets.FY5253.apply

FY5253.**apply**(*other*)

pandas.tseries.offsets.FY5253.copy

FY5253.**copy**

pandas.tseries.offsets.FY5253.get_rule_code_suffix

FY5253.**get_rule_code_suffix**()

pandas.tseries.offsets.FY5253.get_year_end

FY5253.**get_year_end**(*dt*)

pandas.tseries.offsets.FY5253.isAnchored

FY5253.**isAnchored**()

pandas.tseries.offsets.FY5253.onOffset

FY5253.**onOffset**(*dt*)

6.8.30 FY5253Quarter

FY5253Quarter([n, normalize, weekday, . . .])	DateOffset increments between business quarter dates for 52-53 week fiscal year (also known as a 4-4-5 calendar).

pandas.tseries.offsets.FY5253Quarter

class pandas.tseries.offsets.**FY5253Quarter**(*n=1*, *normalize=False*, *weekday=0*, *startingMonth=1*, *qtr_with_extra_week=1*, *variation='nearest'*)

DateOffset increments between business quarter dates for 52-53 week fiscal year (also known as a 4-4-5 calendar).

It is used by companies that desire that their fiscal year always end on the same day of the week.

It is a method of managing accounting periods. It is a common calendar structure for some industries, such as retail, manufacturing and parking industry.

For more information see: http://en.wikipedia.org/wiki/4-4-5_calendar

The year may either: - end on the last X day of the Y month. - end on the last X day closest to the last day of the Y month.

X is a specific day of the week. Y is a certain month of the year

startingMonth = 1 corresponds to dates like 1/31/2007, 4/30/2007, ... startingMonth = 2 corresponds to dates like 2/28/2007, 5/31/2007, ... startingMonth = 3 corresponds to dates like 3/30/2007, 6/29/2007, ...

Parameters

n [int]

weekday [{0, 1, ..., 6}] 0: Mondays 1: Tuesdays 2: Wednesdays 3: Thursdays 4: Fridays 5: Saturdays 6: Sundays

startingMonth [The month in which fiscal years end. {1, 2, ... 12}]

qtr_with_extra_week [The quarter number that has the leap] or 14 week when needed. {1,2,3,4}

variation [str] {"nearest", "last"} for "LastOfMonth" or "NearestEndMonth"

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.FY5253Quarter.base

FY5253Quarter.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.FY5253Quarter.apply_index

`FY5253Quarter.apply_index`

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.FY5253Quarter.rollback

`FY5253Quarter.rollback(dt)`

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.FY5253Quarter.rollforward

`FY5253Quarter.rollforward(dt)`

Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
get_weeks	
isAnchored	
onOffset	
year_has_extra_week	

Properties

FY5253Quarter.freqstr
FY5253Quarter.kwds
FY5253Quarter.name
FY5253Quarter.nanos
FY5253Quarter.normalize
FY5253Quarter.rule_code

pandas.tseries.offsets.FY5253Quarter.freqstr

`FY5253Quarter.freqstr`

pandas.tseries.offsets.FY5253Quarter.kwds

`FY5253Quarter.kwds`

pandas.tseries.offsets.FY5253Quarter.name

FY5253Quarter.**name**

pandas.tseries.offsets.FY5253Quarter.nanos

FY5253Quarter.**nanos**

pandas.tseries.offsets.FY5253Quarter.normalize

FY5253Quarter.**normalize = False**

pandas.tseries.offsets.FY5253Quarter.rule_code

FY5253Quarter.**rule_code**

Methods

FY5253Quarter.apply(other)
FY5253Quarter.copy
FY5253Quarter.get_weeks(dt)
FY5253Quarter.isAnchored()
FY5253Quarter.onOffset(dt)
FY5253Quarter.year_has_extra_week(dt)

pandas.tseries.offsets.FY5253Quarter.apply

FY5253Quarter.**apply**(*other*)

pandas.tseries.offsets.FY5253Quarter.copy

FY5253Quarter.**copy**

pandas.tseries.offsets.FY5253Quarter.get_weeks

FY5253Quarter.**get_weeks**(*dt*)

pandas.tseries.offsets.FY5253Quarter.isAnchored

FY5253Quarter.**isAnchored**()

pandas.tseries.offsets.FY5253Quarter.onOffset

FY5253Quarter.**onOffset**(*dt*)

pandas.tseries.offsets.FY5253Quarter.year_has_extra_week

FY5253Quarter.**year_has_extra_week**(*dt*)

6.8.31 Easter

Easter	DateOffset for the Easter holiday using logic defined in dateutil.

pandas.tseries.offsets.Easter

class pandas.tseries.offsets.**Easter**

DateOffset for the Easter holiday using logic defined in dateutil.

Right now uses the revised method which is valid in years 1583-4099.

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Easter.base

Easter.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Easter.apply_index

Easter.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Easter.rollback

`Easter.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Easter.rollforward

`Easter.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
apply	
copy	
isAnchored	
onOffset	

Properties

Easter.freqstr
Easter.kwds
Easter.name
Easter.nanos
Easter.normalize
Easter.rule_code

pandas.tseries.offsets.Easter.freqstr

`Easter.freqstr`

pandas.tseries.offsets.Easter.kwds

`Easter.kwds`

pandas.tseries.offsets.Easter.name

`Easter.name`

pandas.tseries.offsets.Easter.nanos

`Easter.nanos`

pandas.tseries.offsets.Easter.normalize

`Easter.normalize = False`

pandas.tseries.offsets.Easter.rule_code

`Easter.rule_code`

Methods

Easter.apply(other)	
Easter.copy	
Easter.isAnchored()	
Easter.onOffset(dt)	

pandas.tseries.offsets.Easter.apply

`Easter.apply(other)`

pandas.tseries.offsets.Easter.copy

`Easter.copy`

pandas.tseries.offsets.Easter.isAnchored

`Easter.isAnchored()`

pandas.tseries.offsets.Easter.onOffset

`Easter.onOffset(dt)`

6.8.32 Tick

Tick([n, normalize])	

Attributes

pandas.tseries.offsets.Tick

`class pandas.tseries.offsets.Tick(n=1, normalize=False)`

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Tick.base

`Tick.base`
Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Tick.apply

`Tick.apply(other)`

pandas.tseries.offsets.Tick.apply_index

`Tick.apply_index`
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Tick.rollback

`Tick.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Tick.rollforward

`Tick.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Tick.delta	
Tick.freqstr	
Tick.kwds	
Tick.name	
Tick.nanos	
Tick.normalize	
Tick.rule_code	

pandas.tseries.offsets.Tick.delta

`Tick.delta`

pandas.tseries.offsets.Tick.freqstr

`Tick.freqstr`

pandas.tseries.offsets.Tick.kwds

`Tick.kwds`

pandas.tseries.offsets.Tick.name

`Tick.name`

pandas.tseries.offsets.Tick.nanos

`Tick.nanos`

pandas.tseries.offsets.Tick.normalize

`Tick.normalize = False`

pandas.tseries.offsets.Tick.rule_code

`Tick.`**`rule_code`**

Methods

Tick.copy	
Tick.isAnchored()	
Tick.onOffset(dt)	

pandas.tseries.offsets.Tick.copy

`Tick.`**`copy`**

pandas.tseries.offsets.Tick.isAnchored

`Tick.`**`isAnchored`**`()`

pandas.tseries.offsets.Tick.onOffset

`Tick.`**`onOffset`**`(`*dt*`)`

6.8.33 Day

Day([n, normalize])	

Attributes

pandas.tseries.offsets.Day

class `pandas.tseries.offsets.`**`Day`**`(`*n=1*, *normalize=False*`)`

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Day.base

`Day.`**`base`**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Day.apply

Day.**apply**(*other*)

pandas.tseries.offsets.Day.apply_index

Day.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Day.rollback

Day.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Day.rollforward

Day.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Day.delta
Day.freqstr
Day.kwds
Day.name
Day.nanos
Day.normalize
Day.rule_code

pandas.tseries.offsets.Day.delta

Day.**delta**

pandas.tseries.offsets.Day.freqstr

Day.**freqstr**

pandas.tseries.offsets.Day.kwds

Day.**kwds**

pandas.tseries.offsets.Day.name

Day.**name**

pandas.tseries.offsets.Day.nanos

Day.**nanos**

pandas.tseries.offsets.Day.normalize

Day.**normalize = False**

pandas.tseries.offsets.Day.rule_code

Day.**rule_code**

Methods

Day.copy	
Day.isAnchored()	
Day.onOffset(dt)	

pandas.tseries.offsets.Day.copy

`Day.copy`

pandas.tseries.offsets.Day.isAnchored

`Day.isAnchored()`

pandas.tseries.offsets.Day.onOffset

`Day.onOffset(dt)`

6.8.34 Hour

Hour([n, normalize])	

Attributes

pandas.tseries.offsets.Hour

class `pandas.tseries.offsets.Hour(n=1, normalize=False)`

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Hour.base

`Hour.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Hour.apply

Hour.**apply**(*other*)

pandas.tseries.offsets.Hour.apply_index

Hour.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Hour.rollback

Hour.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Hour.rollforward

Hour.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Hour.delta
Hour.freqstr

Continued on next page

Table 376 – continued from previous page

Hour.kwds
Hour.name
Hour.nanos
Hour.normalize
Hour.rule_code

pandas.tseries.offsets.Hour.delta

Hour.**delta**

pandas.tseries.offsets.Hour.freqstr

Hour.**freqstr**

pandas.tseries.offsets.Hour.kwds

Hour.**kwds**

pandas.tseries.offsets.Hour.name

Hour.**name**

pandas.tseries.offsets.Hour.nanos

Hour.**nanos**

pandas.tseries.offsets.Hour.normalize

Hour.**normalize = False**

pandas.tseries.offsets.Hour.rule_code

Hour.**rule_code**

Methods

Hour.copy
Hour.isAnchored()
Hour.onOffset(dt)

pandas.tseries.offsets.Hour.copy

Hour.**copy**

pandas.tseries.offsets.Hour.isAnchored

`Hour.isAnchored()`

pandas.tseries.offsets.Hour.onOffset

`Hour.onOffset(dt)`

6.8.35 Minute

Minute([n, normalize])	

Attributes

pandas.tseries.offsets.Minute

`class pandas.tseries.offsets.Minute(n=1, normalize=False)`

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Minute.base

`Minute.base`

Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.

Continued on next page

Table 380 – continued from previous page

rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Minute.apply

Minute.**apply**(*other*)

pandas.tseries.offsets.Minute.apply_index

Minute.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Minute.rollback

Minute.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Minute.rollforward

Minute.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Minute.delta
Minute.freqstr
Minute.kwds
Minute.name
Minute.nanos
Minute.normalize
Minute.rule_code

pandas.tseries.offsets.Minute.delta

Minute.**delta**

pandas.tseries.offsets.Minute.freqstr

`Minute.freqstr`

pandas.tseries.offsets.Minute.kwds

`Minute.kwds`

pandas.tseries.offsets.Minute.name

`Minute.name`

pandas.tseries.offsets.Minute.nanos

`Minute.nanos`

pandas.tseries.offsets.Minute.normalize

`Minute.normalize = False`

pandas.tseries.offsets.Minute.rule_code

`Minute.rule_code`

Methods

Minute.copy
Minute.isAnchored()
Minute.onOffset(dt)

pandas.tseries.offsets.Minute.copy

`Minute.copy`

pandas.tseries.offsets.Minute.isAnchored

`Minute.isAnchored()`

pandas.tseries.offsets.Minute.onOffset

`Minute.onOffset(dt)`

6.8.36 Second

Second([n, normalize])

Attributes

---

pandas.tseries.offsets.Second

class pandas.tseries.offsets.**Second**(*n=1*, *normalize=False*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Second.base

Second.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Second.apply

Second.**apply**(*other*)

pandas.tseries.offsets.Second.apply_index

Second.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Second.rollback

Second.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Second.rollforward

Second.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Second.delta	
Second.freqstr	
Second.kwds	
Second.name	
Second.nanos	
Second.normalize	
Second.rule_code	

pandas.tseries.offsets.Second.delta

Second.**delta**

pandas.tseries.offsets.Second.freqstr

Second.**freqstr**

pandas.tseries.offsets.Second.kwds

`Second.kwds`

pandas.tseries.offsets.Second.name

`Second.name`

pandas.tseries.offsets.Second.nanos

`Second.nanos`

pandas.tseries.offsets.Second.normalize

`Second.normalize = False`

pandas.tseries.offsets.Second.rule_code

`Second.rule_code`

Methods

Second.copy
Second.isAnchored()
Second.onOffset(dt)

pandas.tseries.offsets.Second.copy

`Second.copy`

pandas.tseries.offsets.Second.isAnchored

`Second.isAnchored()`

pandas.tseries.offsets.Second.onOffset

`Second.onOffset(dt)`

6.8.37 Milli

Milli([n, normalize])	

Attributes

pandas.tseries.offsets.Milli

class pandas.tseries.offsets.**Milli**(*n=1*, *normalize=False*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Milli.base

Milli.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Milli.apply

Milli.**apply**(*other*)

pandas.tseries.offsets.Milli.apply_index

Milli.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Milli.rollback

`Milli.rollback(dt)`
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Milli.rollforward

`Milli.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Milli.delta
Milli.freqstr
Milli.kwds
Milli.name
Milli.nanos
Milli.normalize
Milli.rule_code

pandas.tseries.offsets.Milli.delta

`Milli.delta`

pandas.tseries.offsets.Milli.freqstr

`Milli.freqstr`

pandas.tseries.offsets.Milli.kwds

`Milli.kwds`

pandas.tseries.offsets.Milli.name

`Milli.name`

pandas.tseries.offsets.Milli.nanos

`Milli.nanos`

pandas.tseries.offsets.Milli.normalize

`Milli.normalize = False`

pandas.tseries.offsets.Milli.rule_code

`Milli.rule_code`

Methods

Milli.copy	
Milli.isAnchored()	
Milli.onOffset(dt)	

pandas.tseries.offsets.Milli.copy

`Milli.copy`

pandas.tseries.offsets.Milli.isAnchored

`Milli.isAnchored()`

pandas.tseries.offsets.Milli.onOffset

`Milli.onOffset(dt)`

6.8.38 Micro

Micro([n, normalize])

Attributes

pandas.tseries.offsets.Micro

`class pandas.tseries.offsets.Micro(n=1, normalize=False)`

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Micro.base

Micro.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Micro.apply

Micro.**apply**(*other*)

pandas.tseries.offsets.Micro.apply_index

Micro.**apply_index**
Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Micro.rollback

Micro.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Micro.rollforward

`Micro.rollforward(dt)`
Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Micro.delta
Micro.freqstr
Micro.kwds
Micro.name
Micro.nanos
Micro.normalize
Micro.rule_code

pandas.tseries.offsets.Micro.delta

`Micro.delta`

pandas.tseries.offsets.Micro.freqstr

`Micro.freqstr`

pandas.tseries.offsets.Micro.kwds

`Micro.kwds`

pandas.tseries.offsets.Micro.name

`Micro.name`

pandas.tseries.offsets.Micro.nanos

`Micro.nanos`

pandas.tseries.offsets.Micro.normalize

`Micro.normalize = False`

pandas.tseries.offsets.Micro.rule_code

Micro.**rule_code**

Methods

Micro.copy	
Micro.isAnchored()	
Micro.onOffset(dt)	

pandas.tseries.offsets.Micro.copy

Micro.**copy**

pandas.tseries.offsets.Micro.isAnchored

Micro.**isAnchored**()

pandas.tseries.offsets.Micro.onOffset

Micro.**onOffset**(*dt*)

6.8.39 Nano

Nano([n, normalize])

Attributes

pandas.tseries.offsets.Nano

class pandas.tseries.offsets.**Nano**(*n=1*, *normalize=False*)

Attributes

base	Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.Nano.base

Nano.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

delta	
freqstr	
kwds	
name	
nanos	
rule_code	

Methods

apply(other)	
apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
rollback(dt)	Roll provided date backward to next offset only if not on offset.
rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.Nano.apply

Nano.**apply**(*other*)

pandas.tseries.offsets.Nano.apply_index

Nano.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.Nano.rollback

Nano.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.Nano.rollforward

Nano.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

__call__	
copy	
isAnchored	
onOffset	

Properties

Nano.delta
Nano.freqstr
Nano.kwds
Nano.name
Nano.nanos
Nano.normalize
Nano.rule_code

pandas.tseries.offsets.Nano.delta

Nano.**delta**

pandas.tseries.offsets.Nano.freqstr

Nano.**freqstr**

pandas.tseries.offsets.Nano.kwds

Nano.**kwds**

pandas.tseries.offsets.Nano.name

Nano.**name**

pandas.tseries.offsets.Nano.nanos

Nano.**nanos**

pandas.tseries.offsets.Nano.normalize

Nano.**normalize = False**

pandas.tseries.offsets.Nano.rule_code

Nano.**rule_code**

Methods

Nano.copy	
Nano.isAnchored()	
Nano.onOffset(dt)	

pandas.tseries.offsets.Nano.copy

Nano.**copy**

pandas.tseries.offsets.Nano.isAnchored

Nano.**isAnchored**()

pandas.tseries.offsets.Nano.onOffset

Nano.**onOffset**(*dt*)

6.8.40 BDay

BDay	alias of *pandas.tseries.offsets.BusinessDay*

pandas.tseries.offsets.BDay

pandas.tseries.offsets.**BDay**
alias of *pandas.tseries.offsets.BusinessDay*

Properties

BDay.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
BDay.freqstr	
BDay.kwds	
BDay.name	
BDay.nanos	
BDay.normalize	
BDay.offset	Alias for self._offset.
BDay.rule_code	

pandas.tseries.offsets.BDay.base

BDay.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BDay.freqstr

BDay.**freqstr**

pandas.tseries.offsets.BDay.kwds

BDay.**kwds**

pandas.tseries.offsets.BDay.name

BDay.**name**

pandas.tseries.offsets.BDay.nanos

BDay.**nanos**

pandas.tseries.offsets.BDay.normalize

BDay.**normalize = False**

pandas.tseries.offsets.BDay.offset

BDay.**offset**

Alias for self._offset.

pandas.tseries.offsets.BDay.rule_code

BDay.**rule_code**

Methods

BDay.apply(other)	
BDay.apply_index	
BDay.copy	
BDay.isAnchored()	
BDay.onOffset(dt)	
BDay.rollback(dt)	Roll provided date backward to next offset only if not on offset.
BDay.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BDay.apply

BDay.**apply**(*other*)

pandas.tseries.offsets.BDay.apply_index

BDay.**apply_index**

pandas.tseries.offsets.BDay.copy

BDay.**copy**

pandas.tseries.offsets.BDay.isAnchored

BDay.**isAnchored**()

pandas.tseries.offsets.BDay.onOffset

BDay.**onOffset**(*dt*)

pandas.tseries.offsets.BDay.rollback

BDay.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BDay.rollforward

BDay.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

6.8.41 BMonthEnd

BMonthEnd	alias of *pandas.tseries.offsets.BusinessMonthEnd*

pandas.tseries.offsets.BMonthEnd

pandas.tseries.offsets.**BMonthEnd**
alias of *pandas.tseries.offsets.BusinessMonthEnd*

Properties

BMonthEnd.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
BMonthEnd.freqstr	
BMonthEnd.kwds	
BMonthEnd.name	
BMonthEnd.nanos	

Continued on next page

Table 407 – continued from previous page

BMonthEnd.normalize	
BMonthEnd.rule_code	

pandas.tseries.offsets.BMonthEnd.base

BMonthEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BMonthEnd.freqstr

BMonthEnd.**freqstr**

pandas.tseries.offsets.BMonthEnd.kwds

BMonthEnd.**kwds**

pandas.tseries.offsets.BMonthEnd.name

BMonthEnd.**name**

pandas.tseries.offsets.BMonthEnd.nanos

BMonthEnd.**nanos**

pandas.tseries.offsets.BMonthEnd.normalize

BMonthEnd.**normalize = False**

pandas.tseries.offsets.BMonthEnd.rule_code

BMonthEnd.**rule_code**

Methods

BMonthEnd.apply(other)	
BMonthEnd.apply_index	
BMonthEnd.copy	
BMonthEnd.isAnchored()	
BMonthEnd.onOffset(dt)	
BMonthEnd.rollback(dt)	Roll provided date backward to next offset only if not on offset.
BMonthEnd.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BMonthEnd.apply

BMonthEnd.**apply**(*other*)

pandas.tseries.offsets.BMonthEnd.apply_index

BMonthEnd.**apply_index**

pandas.tseries.offsets.BMonthEnd.copy

BMonthEnd.**copy**

pandas.tseries.offsets.BMonthEnd.isAnchored

BMonthEnd.**isAnchored**()

pandas.tseries.offsets.BMonthEnd.onOffset

BMonthEnd.**onOffset**(*dt*)

pandas.tseries.offsets.BMonthEnd.rollback

BMonthEnd.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BMonthEnd.rollforward

BMonthEnd.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

6.8.42 BMonthBegin

BMonthBegin	alias of *pandas.tseries.offsets.BusinessMonthBegin*

pandas.tseries.offsets.BMonthBegin

pandas.tseries.offsets.**BMonthBegin**

alias of *pandas.tseries.offsets.BusinessMonthBegin*

Properties

BMonthBegin.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
BMonthBegin.freqstr	
BMonthBegin.kwds	
BMonthBegin.name	
BMonthBegin.nanos	
BMonthBegin.normalize	
BMonthBegin.rule_code	

pandas.tseries.offsets.BMonthBegin.base

BMonthBegin.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.BMonthBegin.freqstr

BMonthBegin.**freqstr**

pandas.tseries.offsets.BMonthBegin.kwds

BMonthBegin.**kwds**

pandas.tseries.offsets.BMonthBegin.name

BMonthBegin.**name**

pandas.tseries.offsets.BMonthBegin.nanos

BMonthBegin.**nanos**

pandas.tseries.offsets.BMonthBegin.normalize

BMonthBegin.**normalize = False**

pandas.tseries.offsets.BMonthBegin.rule_code

BMonthBegin.**rule_code**

Methods

BMonthBegin.apply(other)
BMonthBegin.apply_index
BMonthBegin.copy
BMonthBegin.isAnchored()

Continued on next page

Table 411 – continued from previous page

BMonthBegin.onOffset(dt)	
BMonthBegin.rollback(dt)	Roll provided date backward to next offset only if not on offset.
BMonthBegin.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.BMonthBegin.apply

BMonthBegin.**apply**(*other*)

pandas.tseries.offsets.BMonthBegin.apply_index

BMonthBegin.**apply_index**

pandas.tseries.offsets.BMonthBegin.copy

BMonthBegin.**copy**

pandas.tseries.offsets.BMonthBegin.isAnchored

BMonthBegin.**isAnchored**()

pandas.tseries.offsets.BMonthBegin.onOffset

BMonthBegin.**onOffset**(*dt*)

pandas.tseries.offsets.BMonthBegin.rollback

BMonthBegin.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.BMonthBegin.rollforward

BMonthBegin.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

6.8.43 CBMonthEnd

CBMonthEnd	alias of *pandas.tseries.offsets.CustomBusinessMonthEnd*

pandas.tseries.offsets.CBMonthEnd

pandas.tseries.offsets.**CBMonthEnd**
alias of *pandas.tseries.offsets.CustomBusinessMonthEnd*

Properties

CBMonthEnd.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
CBMonthEnd.cbday_roll	Define default roll function to be called in apply method.
CBMonthEnd.freqstr	
CBMonthEnd.kwds	
CBMonthEnd.m_offset	
CBMonthEnd.month_roll	Define default roll function to be called in apply method.
CBMonthEnd.name	
CBMonthEnd.nanos	
CBMonthEnd.normalize	
CBMonthEnd.offset	Alias for self._offset.
CBMonthEnd.rule_code	

pandas.tseries.offsets.CBMonthEnd.base

CBMonthEnd.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CBMonthEnd.cbday_roll

CBMonthEnd.**cbday_roll**
Define default roll function to be called in apply method.

pandas.tseries.offsets.CBMonthEnd.freqstr

CBMonthEnd.**freqstr**

pandas.tseries.offsets.CBMonthEnd.kwds

CBMonthEnd.**kwds**

pandas.tseries.offsets.CBMonthEnd.m_offset

CBMonthEnd.**m_offset**

pandas.tseries.offsets.CBMonthEnd.month_roll

CBMonthEnd.**month_roll**
Define default roll function to be called in apply method.

pandas.tseries.offsets.CBMonthEnd.name

CBMonthEnd.**name**

pandas.tseries.offsets.CBMonthEnd.nanos

CBMonthEnd.**nanos**

pandas.tseries.offsets.CBMonthEnd.normalize

CBMonthEnd.**normalize = False**

pandas.tseries.offsets.CBMonthEnd.offset

CBMonthEnd.**offset**

Alias for self._offset.

pandas.tseries.offsets.CBMonthEnd.rule_code

CBMonthEnd.**rule_code**

Methods

CBMonthEnd.apply(other)	
CBMonthEnd.apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
CBMonthEnd.copy	
CBMonthEnd.isAnchored()	
CBMonthEnd.onOffset(dt)	
CBMonthEnd.rollback(dt)	Roll provided date backward to next offset only if not on offset.
CBMonthEnd.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CBMonthEnd.apply

CBMonthEnd.**apply**(*other*)

pandas.tseries.offsets.CBMonthEnd.apply_index

CBMonthEnd.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CBMonthEnd.copy

CBMonthEnd.**copy**

pandas.tseries.offsets.CBMonthEnd.isAnchored

CBMonthEnd.**isAnchored**()

pandas.tseries.offsets.CBMonthEnd.onOffset

CBMonthEnd.**onOffset**(*dt*)

pandas.tseries.offsets.CBMonthEnd.rollback

CBMonthEnd.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CBMonthEnd.rollforward

CBMonthEnd.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

6.8.44 CBMonthBegin

CBMonthBegin	alias of *pandas.tseries.offsets.CustomBusinessMonthBegin*

pandas.tseries.offsets.CBMonthBegin

pandas.tseries.offsets.**CBMonthBegin**

alias of *pandas.tseries.offsets.CustomBusinessMonthBegin*

Properties

CBMonthBegin.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
CBMonthBegin.cbday_roll	Define default roll function to be called in apply method.
CBMonthBegin.freqstr	
CBMonthBegin.kwds	

Continued on next page

Table 416 – continued from previous page

CBMonthBegin.m_offset	
CBMonthBegin.month_roll	Define default roll function to be called in apply method.
CBMonthBegin.name	
CBMonthBegin.nanos	
CBMonthBegin.normalize	
CBMonthBegin.offset	Alias for self._offset.
CBMonthBegin.rule_code	

pandas.tseries.offsets.CBMonthBegin.base

CBMonthBegin.**base**

Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CBMonthBegin.cbday_roll

CBMonthBegin.**cbday_roll**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CBMonthBegin.freqstr

CBMonthBegin.**freqstr**

pandas.tseries.offsets.CBMonthBegin.kwds

CBMonthBegin.**kwds**

pandas.tseries.offsets.CBMonthBegin.m_offset

CBMonthBegin.**m_offset**

pandas.tseries.offsets.CBMonthBegin.month_roll

CBMonthBegin.**month_roll**

Define default roll function to be called in apply method.

pandas.tseries.offsets.CBMonthBegin.name

CBMonthBegin.**name**

pandas.tseries.offsets.CBMonthBegin.nanos

CBMonthBegin.**nanos**

pandas.tseries.offsets.CBMonthBegin.normalize

CBMonthBegin.**normalize = False**

pandas.tseries.offsets.CBMonthBegin.offset

CBMonthBegin.**offset**

Alias for self._offset.

pandas.tseries.offsets.CBMonthBegin.rule_code

CBMonthBegin.**rule_code**

Methods

CBMonthBegin.apply(other)	
CBMonthBegin.apply_index	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
CBMonthBegin.copy	
CBMonthBegin.isAnchored()	
CBMonthBegin.onOffset(dt)	
CBMonthBegin.rollback(dt)	Roll provided date backward to next offset only if not on offset.
CBMonthBegin.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CBMonthBegin.apply

CBMonthBegin.**apply**(*other*)

pandas.tseries.offsets.CBMonthBegin.apply_index

CBMonthBegin.**apply_index**

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CBMonthBegin.copy

CBMonthBegin.**copy**

pandas.tseries.offsets.CBMonthBegin.isAnchored

CBMonthBegin.**isAnchored**()

pandas.tseries.offsets.CBMonthBegin.onOffset

CBMonthBegin.**onOffset**(*dt*)

pandas.tseries.offsets.CBMonthBegin.rollback

CBMonthBegin.**rollback**(*dt*)
Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CBMonthBegin.rollforward

CBMonthBegin.**rollforward**(*dt*)
Roll provided date forward to next offset only if not on offset.

6.8.45 CDay

CDay	alias of *pandas.tseries.offsets.CustomBusinessDay*

pandas.tseries.offsets.CDay

pandas.tseries.offsets.**CDay**
alias of *pandas.tseries.offsets.CustomBusinessDay*

Properties

CDay.base	Returns a copy of the calling offset object with n=1 and all other attributes equal.
CDay.freqstr	
CDay.kwds	
CDay.name	
CDay.nanos	
CDay.normalize	
CDay.offset	Alias for self._offset.
CDay.rule_code	

pandas.tseries.offsets.CDay.base

CDay.**base**
Returns a copy of the calling offset object with n=1 and all other attributes equal.

pandas.tseries.offsets.CDay.freqstr

CDay.**freqstr**

pandas.tseries.offsets.CDay.kwds

CDay.**kwds**

pandas.tseries.offsets.CDay.name

CDay.**name**

pandas.tseries.offsets.CDay.nanos

CDay.**nanos**

pandas.tseries.offsets.CDay.normalize

CDay.**normalize = False**

pandas.tseries.offsets.CDay.offset

CDay.**offset**

Alias for self._offset.

pandas.tseries.offsets.CDay.rule_code

CDay.**rule_code**

Methods

CDay.apply(other)	
CDay.apply_index(i)	Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.
CDay.copy	
CDay.isAnchored()	
CDay.onOffset(dt)	
CDay.rollback(dt)	Roll provided date backward to next offset only if not on offset.
CDay.rollforward(dt)	Roll provided date forward to next offset only if not on offset.

pandas.tseries.offsets.CDay.apply

CDay.**apply**(*other*)

pandas.tseries.offsets.CDay.apply_index

CDay.**apply_index**(*i*)

Vectorized apply of DateOffset to DatetimeIndex, raises NotImplentedError for offsets without a vectorized implementation.

Parameters

i [DatetimeIndex]

Returns

y [DatetimeIndex]

pandas.tseries.offsets.CDay.copy

CDay.**copy**

pandas.tseries.offsets.CDay.isAnchored

CDay.**isAnchored**()

pandas.tseries.offsets.CDay.onOffset

CDay.**onOffset**(*dt*)

pandas.tseries.offsets.CDay.rollback

CDay.**rollback**(*dt*)

Roll provided date backward to next offset only if not on offset.

pandas.tseries.offsets.CDay.rollforward

CDay.**rollforward**(*dt*)

Roll provided date forward to next offset only if not on offset.

6.9 Frequencies

to_offset(freq)	Return DateOffset object from string or tuple representation or datetime.timedelta object

6.9.1 pandas.tseries.frequencies.to_offset

`pandas.tseries.frequencies.to_offset(freq)`

Return DateOffset object from string or tuple representation or datetime.timedelta object

Parameters

freq [str, tuple, datetime.timedelta, DateOffset or None]

Returns

delta [DateOffset] None if freq is None

Raises

ValueError If freq is an invalid frequency

See also:

`pandas.DateOffset`

Examples

```
>>> to_offset('5min')
<5 * Minutes>
```

```
>>> to_offset('1D1H')
<25 * Hours>
```

```
>>> to_offset(('W', 2))
<2 * Weeks: weekday=6>
```

```
>>> to_offset((2, 'B'))
<2 * BusinessDays>
```

```
>>> to_offset(datetime.timedelta(days=1))
<Day>
```

```
>>> to_offset(Hour())
<Hour>
```

6.10 Window

Rolling objects are returned by `.rolling` calls: *pandas.DataFrame.rolling()*, *pandas.Series.rolling()*, etc. Expanding objects are returned by `.expanding` calls: *pandas.DataFrame.expanding()*, *pandas.Series.expanding()*, etc. EWM objects are returned by `.ewm` calls: *pandas.DataFrame.ewm()*, *pandas.Series.ewm()*, etc.

6.10.1 Standard moving window functions

Rolling.count()	The rolling count of any non-NaN observations inside the window.
Rolling.sum(*args, **kwargs)	Calculate rolling sum of given DataFrame or Series.
Rolling.mean(*args, **kwargs)	Calculate the rolling mean of the values.
Rolling.median(**kwargs)	Calculate the rolling median.
Rolling.var([ddof])	Calculate unbiased rolling variance.
Rolling.std([ddof])	Calculate rolling standard deviation.
Rolling.min(*args, **kwargs)	Calculate the rolling minimum.
Rolling.max(*args, **kwargs)	Calculate the rolling maximum.
Rolling.corr([other, pairwise])	Calculate rolling correlation.
Rolling.cov([other, pairwise, ddof])	Calculate the rolling sample covariance.
Rolling.skew(**kwargs)	Unbiased rolling skewness.
Rolling.kurt(**kwargs)	Calculate unbiased rolling kurtosis.
Rolling.apply(func[, raw, args, kwargs])	The rolling function's apply function.
Rolling.aggregate(arg, *args, **kwargs)	Aggregate using one or more operations over the specified axis.
Rolling.quantile(quantile[, interpolation])	Calculate the rolling quantile.
Window.mean(*args, **kwargs)	Calculate the window mean of the values.
Window.sum(*args, **kwargs)	Calculate window sum of given DataFrame or Series.

pandas.core.window.Rolling.count

Rolling.**count**()

The rolling count of any non-NaN observations inside the window.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

pandas.Series.rolling Calling object with Series data.

pandas.DataFrame.rolling Calling object with DataFrames.

pandas.DataFrame.count Count of the full DataFrame.

Examples

```
>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
```

(continues on next page)

(continued from previous page)

```
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.Rolling.sum

`Rolling.sum(*args, **kwargs)`

Calculate rolling sum of given DataFrame or Series.

Parameters

***args, **kwargs** For compatibility with other rolling methods. Has no effect on the computed value.

Returns

Series or DataFrame Same type as the input, with the same index, containing the rolling sum.

See also:

Series.sum Reducing sum for Series.

DataFrame.sum Reducing sum for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0     NaN
1     NaN
2     6.0
3    10.0
4    15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0     NaN
1     6.0
2     9.0
3    12.0
4     NaN
dtype: float64
```

For DataFrame, each rolling sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A   B
0  1   1
1  2   4
2  3   9
3  4  16
4  5  25
```

```
>>> df.rolling(3).sum()
      A     B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3   9.0  29.0
4  12.0  50.0
```

pandas.core.window.Rolling.mean

`Rolling.mean(*args, **kwargs)`
Calculate the rolling mean of the values.

Parameters

***args** Under Review.

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

Series.rolling Calling object with Series data.

DataFrame.rolling Calling object with DataFrames.

Series.mean Equivalent method for Series.

DataFrame.mean Equivalent method for DataFrame.

Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.Rolling.median

`Rolling.median(**kwargs)`
Calculate the rolling median.

Parameters

****kwargs** For compatibility with other rolling methods. Has no effect on the computed median.

Returns

Series or DataFrame Returned type is the same as the original object.

See also:

`Series.rolling` Calling object with Series data.

`DataFrame.rolling` Calling object with DataFrames.

`Series.median` Equivalent method for Series.

`DataFrame.median` Equivalent method for DataFrame.

Examples

Compute the rolling median of a series with a window size of 3.

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
dtype: float64
```

pandas.core.window.Rolling.var

`Rolling.var(ddof=1, *args, **kwargs)`
Calculate unbiased rolling variance.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the rolling calculation.

See also:

Series.rolling Calling object with Series data.

DataFrame.rolling Calling object with DataFrames.

Series.var Equivalent method for Series.

DataFrame.var Equivalent method for DataFrame.

numpy.var Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0         NaN
1         NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0         NaN
1         NaN
2    0.333333
3    0.916667
4    0.800000
5    0.700000
6    0.619048
dtype: float64
```

pandas.core.window.Rolling.std

`Rolling.std(ddof=1, *args, **kwargs)`
Calculate rolling standard deviation.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the rolling calculation.

See also:

`Series.rolling` Calling object with Series data.

`DataFrame.rolling` Calling object with DataFrames.

`Series.std` Equivalent method for Series.

`DataFrame.std` Equivalent method for DataFrame.

`numpy.std` Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in Series.std is different than the default *ddof* of 0 in numpy.std.

A minimum of one period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0         NaN
1         NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0         NaN
1         NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

pandas.core.window.Rolling.min

`Rolling.min(*args, **kwargs)`

Calculate the rolling minimum.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

Series.rolling Calling object with a Series.

DataFrame.rolling Calling object with a DataFrame.

Series.min Similar method for Series.

DataFrame.min Similar method for DataFrame.

Examples

Performing a rolling minimum with a window size of 3.

```
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0    NaN
1    NaN
2    3.0
3    2.0
4    2.0
dtype: float64
```

pandas.core.window.Rolling.max

Rolling.**max**(**args*, ***kwargs*)
Calculate the rolling maximum.

Parameters

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.rolling Series rolling.

DataFrame.rolling DataFrame rolling.

pandas.core.window.Rolling.corr

Rolling.**corr**(*other=None*, *pairwise=None*, ***kwargs*)
Calculate rolling correlation.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self.

pairwise [bool, default None] Calculate pairwise combinations of columns within a DataFrame. If *other* is not specified, defaults to *True*, otherwise defaults to *False*. Not relevant for `Series`.

****kwargs** Unused.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

Series.rolling Calling object with Series data.

DataFrame.rolling Calling object with DataFrames.

Series.corr Equivalent method for Series.

DataFrame.corr Equivalent method for DataFrame.

rolling.cov Similar method to calculate covariance.

numpy.corrcoef NumPy Pearson's correlation calculation.

Notes

This function uses Pearson's definition of correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

When *other* is not specified, the output will be self correlation (e.g. all 1's), except for `DataFrame` inputs with *pairwise* set to *True*.

Function will return `NaN` for correlations of equal valued sequences; this is the result of a 0/0 division error.

When *pairwise* is set to *False*, only matching columns between *self* and *other* will be used.

When *pairwise* is set to *True*, the output will be a MultiIndex DataFrame with the original index on the first level, and the *other* DataFrame columns on the second level.

In the case of missing elements, only complete pairwise observations will be used.

Examples

The below example shows a rolling calculation with a window size of four matching the equivalent function call using `numpy.corrcoef()`.

```
>>> v1 = [3, 3, 3, 5, 8]
>>> v2 = [3, 4, 4, 4, 8]
>>> fmt = "{0:.6f}"  # limit the printed precision to 6 digits
>>> # numpy returns a 2X2 array, the correlation coefficient
>>> # is the number at entry [0][1]
>>> print(fmt.format(np.corrcoef(v1[:-1], v2[:-1])[0][1]))
0.333333
>>> print(fmt.format(np.corrcoef(v1[1:], v2[1:])[0][1]))
0.916949
>>> s1 = pd.Series(v1)
>>> s2 = pd.Series(v2)
>>> s1.rolling(4).corr(s2)
0         NaN
```

(continues on next page)

(continued from previous page)

```
1          NaN
2          NaN
3     0.333333
4     0.916949
dtype: float64
```

The below example shows a similar rolling calculation on a DataFrame using the pairwise option.

```
>>> matrix = np.array([[51., 35.], [49., 30.], [47., 32.],    [46., 31.], [50.,␣
↪36.]])
>>> print(np.corrcoef(matrix[:-1,0], matrix[:-1,1]).round(7))
[[1.         0.6263001]
 [0.6263001  1.       ]]
>>> print(np.corrcoef(matrix[1:,0], matrix[1:,1]).round(7))
[[1.         0.5553681]
 [0.5553681  1.       ]]
>>> df = pd.DataFrame(matrix, columns=['X','Y'])
>>> df
      X     Y
0  51.0  35.0
1  49.0  30.0
2  47.0  32.0
3  46.0  31.0
4  50.0  36.0
>>> df.rolling(4).corr(pairwise=True)
            X         Y
0 X       NaN       NaN
  Y       NaN       NaN
1 X       NaN       NaN
  Y       NaN       NaN
2 X       NaN       NaN
  Y       NaN       NaN
3 X  1.000000  0.626300
  Y  0.626300  1.000000
4 X  1.000000  0.555368
  Y  0.555368  1.000000
```

pandas.core.window.Rolling.cov

`Rolling.cov`(*other=None*, *pairwise=None*, *ddof=1*, ***kwargs*)

Calculate the rolling sample covariance.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.rolling Series rolling.

DataFrame.rolling DataFrame rolling.

pandas.core.window.Rolling.skew

Rolling.**skew**(***kwargs*)

Unbiased rolling skewness.

Parameters

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.rolling Series rolling.

DataFrame.rolling DataFrame rolling.

pandas.core.window.Rolling.kurt

Rolling.**kurt**(***kwargs*)

Calculate unbiased rolling kurtosis.

This function uses Fisher's definition of kurtosis without bias.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation

See also:

Series.rolling Calling object with Series data.

DataFrame.rolling Calling object with DataFrames.

Series.kurt Equivalent method for Series.

DataFrame.kurt Equivalent method for DataFrame.

scipy.stats.skew Third moment of a probability density.

scipy.stats.kurtosis Reference SciPy method.

Notes

A minimum of 4 periods is required for the rolling calculation.

Examples

The example below will show a rolling calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```
>>> arr = [1, 2, 3, 4, 999]
>>> fmt = "{0:.6f}"  # limit the printed precision to 6 digits
>>> import scipy.stats
>>> print(fmt.format(scipy.stats.kurtosis(arr[:-1], bias=False)))
-1.200000
>>> print(fmt.format(scipy.stats.kurtosis(arr[1:], bias=False)))
3.999946
>>> s = pd.Series(arr)
>>> s.rolling(4).kurt()
0         NaN
1         NaN
2         NaN
3   -1.200000
4    3.999946
dtype: float64
```

pandas.core.window.Rolling.apply

`Rolling.apply`(*func*, *raw=None*, *args=()*, *kwargs={}*)

The rolling function's apply function.

Parameters

func [function] Must produce a single value from an ndarray input if `raw=True` or a Series if `raw=False`.

raw [bool, default None]

- `False` : passes each row or column as a Series to the function.
- `True` or `None` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

The *raw* parameter is required and will show a FutureWarning if not passed. In the future *raw* will default to False.

New in version 0.23.0.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.rolling` Series rolling.

`DataFrame.rolling` DataFrame rolling.

pandas.core.window.Rolling.aggregate

`Rolling.aggregate`(*arg*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

pandas.Series.rolling, *pandas.DataFrame.rolling*

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.004295  0.905829 -0.954544
2  0.735167 -0.165272 -1.619346
3 -0.702657 -1.340923 -0.706334
4 -0.246845  0.211596 -0.901819
5  2.463718  3.157577 -1.380906
6 -1.142255  2.340594 -0.039875
7  1.396598 -1.647453  1.677227
8 -0.543425  1.761277 -0.220481
9 -0.640505  0.289374 -1.550670
```

```
>>> df.rolling(3).sum()
          A         B         C
0       NaN       NaN       NaN
1       NaN       NaN       NaN
2 -2.655105  0.637799 -2.135068
3 -0.971785 -0.600366 -3.280224
4 -0.214334 -1.294599 -3.227500
```

(continues on next page)

(continued from previous page)

```
5  1.514216  2.028250 -2.989060
6  1.074618  5.709767 -2.322600
7  2.718061  3.850718  0.256446
8 -0.289082  2.454418  1.416871
9  0.212668  0.403198 -0.093924
```

```
>>> df.rolling(3).agg({'A':'sum', 'B':'min'})
          A         B
0       NaN       NaN
1       NaN       NaN
2 -2.655105 -0.165272
3 -0.971785 -1.340923
4 -0.214334 -1.340923
5  1.514216 -1.340923
6  1.074618  0.211596
7  2.718061 -1.647453
8 -0.289082 -1.647453
9  0.212668 -1.647453
```

pandas.core.window.Rolling.quantile

Rolling.**quantile**(*quantile*, *interpolation='linear'*, ***kwargs*)

Calculate the rolling quantile.

Parameters

quantile [float] Quantile to compute. 0 <= quantile <= 1.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

****kwargs:** For compatibility with other rolling methods. Has no effect on the result.

Returns

Series or DataFrame Returned object type is determined by the caller of the rolling calculation.

See also:

pandas.Series.quantile Computes value at the given quantile over all data in Series.

pandas.DataFrame.quantile Computes values at the given quantile over requested axis in DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0    NaN
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

pandas.core.window.Window.mean

`Window.mean(*args, **kwargs)`
Calculate the window mean of the values.

Parameters

***args** Under Review.

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the window calculation.

See also:

`Series.window` Calling object with Series data.

`DataFrame.window` Calling object with DataFrames.

`Series.mean` Equivalent method for Series.

`DataFrame.mean` Equivalent method for DataFrame.

Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.Window.sum

`Window.sum(*args, **kwargs)`
Calculate window sum of given DataFrame or Series.

Parameters

***args, **kwargs** For compatibility with other window methods. Has no effect on the computed value.

Returns

Series or DataFrame Same type as the input, with the same index, containing the window sum.

See also:

Series.sum Reducing sum for Series.

DataFrame.sum Reducing sum for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0     NaN
1     NaN
2     6.0
3    10.0
4    15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0     NaN
1     6.0
2     9.0
3    12.0
4     NaN
dtype: float64
```

For DataFrame, each window sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A   B
0  1   1
1  2   4
2  3   9
3  4  16
4  5  25
```

```
>>> df.rolling(3).sum()
      A     B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3   9.0  29.0
4  12.0  50.0
```

6.10.2 Standard expanding window functions

Expanding.count(**kwargs)	The expanding count of any non-NaN observations inside the window.
Expanding.sum(*args, **kwargs)	Calculate expanding sum of given DataFrame or Series.
Expanding.mean(*args, **kwargs)	Calculate the expanding mean of the values.
Expanding.median(**kwargs)	Calculate the expanding median.
Expanding.var([ddof])	Calculate unbiased expanding variance.
Expanding.std([ddof])	Calculate expanding standard deviation.
Expanding.min(*args, **kwargs)	Calculate the expanding minimum.
Expanding.max(*args, **kwargs)	Calculate the expanding maximum.
Expanding.corr([other, pairwise])	Calculate expanding correlation.
Expanding.cov([other, pairwise, ddof])	Calculate the expanding sample covariance.
Expanding.skew(**kwargs)	Unbiased expanding skewness.
Expanding.kurt(**kwargs)	Calculate unbiased expanding kurtosis.
Expanding.apply(func[, raw, args, kwargs])	The expanding function's apply function.
Expanding.aggregate(arg, *args, **kwargs)	Aggregate using one or more operations over the specified axis.
Expanding.quantile(quantile[, interpolation])	Calculate the expanding quantile.

pandas.core.window.Expanding.count

Expanding.**count**(***kwargs*)

The expanding count of any non-NaN observations inside the window.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

pandas.Series.expanding Calling object with Series data.

pandas.DataFrame.expanding Calling object with DataFrames.

pandas.DataFrame.count Count of the full DataFrame.

Examples

```
>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.Expanding.sum

`Expanding.`**`sum`**`(`**args*, ***kwargs*`)`

Calculate expanding sum of given DataFrame or Series.

Parameters

***args, **kwargs** For compatibility with other expanding methods. Has no effect on the computed value.

Returns

Series or DataFrame Same type as the input, with the same index, containing the expanding sum.

See also:

Series.sum Reducing sum for Series.

DataFrame.sum Reducing sum for DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0     NaN
1     NaN
2     6.0
3     9.0
4    12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0     NaN
1     NaN
2     6.0
3    10.0
4    15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0     NaN
1     6.0
2     9.0
3    12.0
4     NaN
dtype: float64
```

For DataFrame, each expanding sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A   B
0  1   1
1  2   4
2  3   9
3  4  16
4  5  25
```

```
>>> df.rolling(3).sum()
      A     B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3   9.0  29.0
4  12.0  50.0
```

pandas.core.window.Expanding.mean

`Expanding.mean(*args, **kwargs)`
Calculate the expanding mean of the values.

Parameters

***args** Under Review.

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.

DataFrame.expanding Calling object with DataFrames.

Series.mean Equivalent method for Series.

DataFrame.mean Equivalent method for DataFrame.

Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```

pandas.core.window.Expanding.median

`Expanding.median(**kwargs)`
Calculate the expanding median.

Parameters

****kwargs** For compatibility with other expanding methods. Has no effect on the computed median.

Returns

Series or DataFrame Returned type is the same as the original object.

See also:

Series.expanding Calling object with Series data.

DataFrame.expanding Calling object with DataFrames.

Series.median Equivalent method for Series.

DataFrame.median Equivalent method for DataFrame.

Examples

Compute the rolling median of a series with a window size of 3.

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
dtype: float64
```

pandas.core.window.Expanding.var

`Expanding.var`(*ddof=1*, **args*, ***kwargs*)

Calculate unbiased expanding variance.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.

DataFrame.expanding Calling object with DataFrames.

Series.var Equivalent method for Series.

DataFrame.var Equivalent method for DataFrame.

numpy.var Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0         NaN
1         NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0         NaN
1         NaN
2    0.333333
3    0.916667
4    0.800000
5    0.700000
6    0.619048
dtype: float64
```

pandas.core.window.Expanding.std

`Expanding.std`(*ddof=1*, **args*, ***kwargs*)

Calculate expanding standard deviation.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

Parameters

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

***args, **kwargs** For NumPy compatibility. No additional arguments are used.

Returns

Series or DataFrame Returns the same object type as the caller of the expanding calculation.

See also:

`Series.expanding` Calling object with Series data.

`DataFrame.expanding` Calling object with DataFrames.

`Series.std` Equivalent method for Series.

`DataFrame.std` Equivalent method for DataFrame.

`numpy.std` Equivalent method for Numpy array.

Notes

The default *ddof* of 1 used in Series.std is different than the default *ddof* of 0 in numpy.std.

A minimum of one period is required for the rolling calculation.

Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0         NaN
1         NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0         NaN
1         NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

pandas.core.window.Expanding.min

Expanding.**min**(**args*, ***kwargs*)

Calculate the expanding minimum.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with a Series.

DataFrame.expanding Calling object with a DataFrame.

Series.min Similar method for Series.

DataFrame.min Similar method for DataFrame.

Examples

Performing a rolling minimum with a window size of 3.

```
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0    NaN
1    NaN
2    3.0
3    2.0
```

(continues on next page)

(continued from previous page)

```
4    2.0
dtype: float64
```

pandas.core.window.Expanding.max

Expanding.**max**(**args*, ***kwargs*)

Calculate the expanding maximum.

Parameters

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.

DataFrame.expanding DataFrame expanding.

pandas.core.window.Expanding.corr

Expanding.**corr**(*other=None*, *pairwise=None*, ***kwargs*)

Calculate expanding correlation.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self.

pairwise [bool, default None] Calculate pairwise combinations of columns within a DataFrame. If *other* is not specified, defaults to *True*, otherwise defaults to *False*. Not relevant for `Series`.

****kwargs** Unused.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

Series.expanding Calling object with Series data.

DataFrame.expanding Calling object with DataFrames.

Series.corr Equivalent method for Series.

DataFrame.corr Equivalent method for DataFrame.

expanding.cov Similar method to calculate covariance.

numpy.corrcoef NumPy Pearson's correlation calculation.

Notes

This function uses Pearson's definition of correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

When *other* is not specified, the output will be self correlation (e.g. all 1's), except for `DataFrame` inputs with *pairwise* set to *True*.

Function will return `NaN` for correlations of equal valued sequences; this is the result of a 0/0 division error.

When *pairwise* is set to *False*, only matching columns between *self* and *other* will be used.

When *pairwise* is set to *True*, the output will be a MultiIndex DataFrame with the original index on the first level, and the *other* DataFrame columns on the second level.

In the case of missing elements, only complete pairwise observations will be used.

Examples

The below example shows a rolling calculation with a window size of four matching the equivalent function call using `numpy.corrcoef()`.

```
>>> v1 = [3, 3, 3, 5, 8]
>>> v2 = [3, 4, 4, 4, 8]
>>> fmt = "{0:.6f}"  # limit the printed precision to 6 digits
>>> # numpy returns a 2X2 array, the correlation coefficient
>>> # is the number at entry [0][1]
>>> print(fmt.format(np.corrcoef(v1[:-1], v2[:-1])[0][1]))
0.333333
>>> print(fmt.format(np.corrcoef(v1[1:], v2[1:])[0][1]))
0.916949
>>> s1 = pd.Series(v1)
>>> s2 = pd.Series(v2)
>>> s1.rolling(4).corr(s2)
0         NaN
1         NaN
2         NaN
3    0.333333
4    0.916949
dtype: float64
```

The below example shows a similar rolling calculation on a DataFrame using the pairwise option.

```
>>> matrix = np.array([[51., 35.], [49., 30.], [47., 32.],    [46., 31.], [50.,␣
↪36.]])
>>> print(np.corrcoef(matrix[:-1,0], matrix[:-1,1]).round(7))
[[1.         0.6263001]
 [0.6263001  1.       ]]
>>> print(np.corrcoef(matrix[1:,0], matrix[1:,1]).round(7))
[[1.         0.5553681]
 [0.5553681  1.        ]]
>>> df = pd.DataFrame(matrix, columns=['X','Y'])
>>> df
      X     Y
0  51.0  35.0
1  49.0  30.0
2  47.0  32.0
3  46.0  31.0
```

(continues on next page)

(continued from previous page)

```
4  50.0  36.0
>>> df.rolling(4).corr(pairwise=True)
            X         Y
0 X       NaN       NaN
  Y       NaN       NaN
1 X       NaN       NaN
  Y       NaN       NaN
2 X       NaN       NaN
  Y       NaN       NaN
3 X  1.000000  0.626300
  Y  0.626300  1.000000
4 X  1.000000  0.555368
  Y  0.555368  1.000000
```

pandas.core.window.Expanding.cov

Expanding.**cov**(*other=None*, *pairwise=None*, *ddof=1*, ***kwargs*)

Calculate the expanding sample covariance.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.

DataFrame.expanding DataFrame expanding.

pandas.core.window.Expanding.skew

Expanding.**skew**(***kwargs*)

Unbiased expanding skewness.

Parameters

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.expanding Series expanding.

DataFrame.expanding DataFrame expanding.

pandas.core.window.Expanding.kurt

Expanding.**kurt**(***kwargs*)

Calculate unbiased expanding kurtosis.

This function uses Fisher's definition of kurtosis without bias.

Parameters

****kwargs** Under Review.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation

See also:

Series.expanding Calling object with Series data.

DataFrame.expanding Calling object with DataFrames.

Series.kurt Equivalent method for Series.

DataFrame.kurt Equivalent method for DataFrame.

scipy.stats.skew Third moment of a probability density.

scipy.stats.kurtosis Reference SciPy method.

Notes

A minimum of 4 periods is required for the expanding calculation.

Examples

The example below will show an expanding calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```
>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats
>>> fmt = "{0:.6f}"  # limit the printed precision to 6 digits
>>> print(fmt.format(scipy.stats.kurtosis(arr[:-1], bias=False)))
-1.200000
>>> print(fmt.format(scipy.stats.kurtosis(arr, bias=False)))
4.999874
>>> s = pd.Series(arr)
>>> s.expanding(4).kurt()
0         NaN
1         NaN
2         NaN
3   -1.200000
4    4.999874
dtype: float64
```

pandas.core.window.Expanding.apply

`Expanding.apply(func, raw=None, args=(), kwargs={})`

The expanding function's apply function.

Parameters

func [function] Must produce a single value from an ndarray input if `raw=True` or a Series if `raw=False`.

raw [bool, default None]

- `False` : passes each row or column as a Series to the function.
- `True` or `None` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

The *raw* parameter is required and will show a FutureWarning if not passed. In the future *raw* will default to False.

New in version 0.23.0.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.expanding` Series expanding.

`DataFrame.expanding` DataFrame expanding.

pandas.core.window.Expanding.aggregate

`Expanding.aggregate(arg, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series/Dataframe or when passed to Series/Dataframe.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

`pandas.DataFrame.expanding.aggregate`, `pandas.DataFrame.rolling.aggregate`, *`pandas.DataFrame.aggregate`*

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.004295  0.905829 -0.954544
2  0.735167 -0.165272 -1.619346
3 -0.702657 -1.340923 -0.706334
4 -0.246845  0.211596 -0.901819
5  2.463718  3.157577 -1.380906
6 -1.142255  2.340594 -0.039875
7  1.396598 -1.647453  1.677227
8 -0.543425  1.761277 -0.220481
9 -0.640505  0.289374 -1.550670
```

```
>>> df.ewm(alpha=0.5).mean()
          A         B         C
0 -2.385977 -0.102758  0.438822
1 -1.464856  0.569633 -0.490089
2 -0.207700  0.149687 -1.135379
3 -0.471677 -0.645305 -0.906555
4 -0.355635 -0.203033 -0.904111
5  1.076417  1.503943 -1.146293
6 -0.041654  1.925562 -0.588728
7  0.680292  0.132049  0.548693
8  0.067236  0.948257  0.163353
9 -0.286980  0.618493 -0.694496
```

pandas.core.window.Expanding.quantile

`Expanding.`**`quantile`**(*quantile*, *interpolation='linear'*, ***kwargs*)

Calculate the expanding quantile.

Parameters

quantile [float] Quantile to compute. 0 <= quantile <= 1.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.

- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

****kwargs:** For compatibility with other expanding methods. Has no effect on the result.

Returns

Series or DataFrame Returned object type is determined by the caller of the expanding calculation.

See also:

pandas.Series.quantile Computes value at the given quantile over all data in Series.

pandas.DataFrame.quantile Computes values at the given quantile over requested axis in DataFrame.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0    NaN
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

6.10.3 Exponentially-weighted moving window functions

EWM.mean(*args, **kwargs)	Exponential weighted moving average.
EWM.std([bias])	Exponential weighted moving stddev.
EWM.var([bias])	Exponential weighted moving variance.
EWM.corr([other, pairwise])	Exponential weighted sample correlation.
EWM.cov([other, pairwise, bias])	Exponential weighted sample covariance.

pandas.core.window.EWM.mean

EWM.**mean**(**args*, ***kwargs*)

Exponential weighted moving average.

Parameters

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.

DataFrame.ewm DataFrame ewm.

pandas.core.window.EWM.std

EWM.**std**(*bias=False*, **args*, ***kwargs*)
Exponential weighted moving stddev.

Parameters

bias [bool, default False] Use a standard estimation bias correction.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.

DataFrame.ewm DataFrame ewm.

pandas.core.window.EWM.var

EWM.**var**(*bias=False*, **args*, ***kwargs*)
Exponential weighted moving variance.

Parameters

bias [bool, default False] Use a standard estimation bias correction.

***args, **kwargs** Arguments and keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

Series.ewm Series ewm.

DataFrame.ewm DataFrame ewm.

pandas.core.window.EWM.corr

EWM.**corr**(*other=None*, *pairwise=None*, ***kwargs*)
Exponential weighted sample correlation.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

bias [bool, default False] Use a standard estimation bias correction.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.ewm` Series ewm.

`DataFrame.ewm` DataFrame ewm.

pandas.core.window.EWM.cov

`EWM.cov`(*other=None*, *pairwise=None*, *bias=False*, ***kwargs*)

Exponential weighted sample covariance.

Parameters

other [Series, DataFrame, or ndarray, optional] If not supplied then will default to self and produce pairwise output.

pairwise [bool, default None] If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

bias [bool, default False] Use a standard estimation bias correction.

****kwargs** Keyword arguments to be passed into func.

Returns

Series or DataFrame Return type is determined by the caller.

See also:

`Series.ewm` Series ewm.

`DataFrame.ewm` DataFrame ewm.

6.11 GroupBy

GroupBy objects are returned by groupby calls: *`pandas.DataFrame.groupby()`*, *`pandas.Series.groupby()`*, etc.

6.11.1 Indexing, iteration

GroupBy.__iter__()	Groupby iterator.
GroupBy.groups	Dict {group name -> group labels}.
GroupBy.indices	Dict {group name -> group indices}.
GroupBy.get_group(name[, obj])	Constructs NDFrame from group with provided name.

pandas.core.groupby.GroupBy.__iter__

GroupBy.**__iter__**()

Groupby iterator.

Returns

Generator yielding sequence of (name, subsetted object)

for each group

pandas.core.groupby.GroupBy.groups

GroupBy.**groups**

Dict {group name -> group labels}.

pandas.core.groupby.GroupBy.indices

GroupBy.**indices**

Dict {group name -> group indices}.

pandas.core.groupby.GroupBy.get_group

GroupBy.**get_group**(*name*, *obj=None*)

Constructs NDFrame from group with provided name.

Parameters

name [object] the name of the group to get as a DataFrame

obj [NDFrame, default None] the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns

group [same type as obj]

Grouper([key, level, freq, axis, sort])	A Grouper allows the user to specify a groupby instruction for a target object

pandas.Grouper

class pandas.**Grouper**(*key=None*, *level=None*, *freq=None*, *axis=0*, *sort=False*)

A Grouper allows the user to specify a groupby instruction for a target object

This specification will select a column via the key parameter, or if the level and/or axis parameters are given, a level of the index of the target object.

These are local specifications and will override 'global' settings, that is the parameters axis and level which are passed to the groupby itself.

Parameters

key [string, defaults to None] groupby key, which selects the grouping column of the target

level [name/number, defaults to None] the level for the target index

freq [string / frequency object, defaults to None] This will groupby the specified frequency if the target selection (via key or level) is a datetime-like object. For full specification of available frequencies, please see here.

axis [number/name of the axis, defaults to 0]

sort [boolean, default to False] whether to sort the resulting labels

additional kwargs to control time-like groupers (when 'freq' is passed)

closed [closed end of interval; 'left' or 'right']

label [interval boundary to use for labeling; 'left' or 'right']

convention [{'start', 'end', 'e', 's'}] If grouper is PeriodIndex

base, loffset

Returns

A specification for a groupby instruction

Examples

Syntactic sugar for `df.groupby('A')`

```
>>> df.groupby(Grouper(key='A'))
```

Specify a resample operation on the column 'date'

```
>>> df.groupby(Grouper(key='date', freq='60s'))
```

Specify a resample operation on the level 'date' on the columns axis with a frequency of 60s

```
>>> df.groupby(Grouper(level='date', freq='60s', axis=1))
```

Attributes

ax	
groups	

6.11.2 Function application

GroupBy.apply(func, *args, **kwargs)	Apply function *func* group-wise and combine the results together.
GroupBy.agg(func, *args, **kwargs)	
GroupBy.aggregate(func, *args, **kwargs)	
GroupBy.transform(func, *args, **kwargs)	

Continued on next page

Table 427 – continued from previous page

GroupBy.pipe(func, *args, **kwargs)	Apply a function *func* with arguments to this GroupBy object and return the function's result.

pandas.core.groupby.GroupBy.apply

GroupBy.**apply**(*func*, **args*, ***kwargs*)

Apply function *func* group-wise and combine the results together.

The function passed to *apply* must take a dataframe as its first argument and return a DataFrame, Series or scalar. *apply* will then take care of combining the results back together into a single dataframe or series. *apply* is therefore a highly flexible grouping method.

While *apply* is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods like *agg* or *transform*. Pandas offers a wide range of method that will be much faster than using *apply* for their specific purposes, so try to use them before reaching for *apply*.

Parameters

func [callable] A callable that takes a dataframe as its first argument, and returns a dataframe, a series or a scalar. In addition the callable may take positional and keyword arguments.

args, kwargs [tuple and dict] Optional positional and keyword arguments to pass to *func*.

Returns

applied [Series or DataFrame]

See also:

pipe Apply function to the full GroupBy object instead of to each group.

aggregate Apply aggregate function to the GroupBy object.

transform Apply function column-by-column to the GroupBy object.

Series.apply Apply a function to a Series.

DataFrame.apply Apply a function to each row or column of a DataFrame.

pandas.core.groupby.GroupBy.agg

GroupBy.**agg**(*func*, **args*, ***kwargs*)

pandas.core.groupby.GroupBy.aggregate

GroupBy.**aggregate**(*func*, **args*, ***kwargs*)

pandas.core.groupby.GroupBy.transform

GroupBy.**transform**(*func*, **args*, ***kwargs*)

pandas.core.groupby.GroupBy.pipe

GroupBy.**pipe**(*func*, **args*, ***kwargs*)

Apply a function *func* with arguments to this GroupBy object and return the function's result.

New in version 0.21.0.

Use *.pipe* when you want to improve readability by chaining together functions that expect Series, DataFrames, GroupBy or Resampler objects. Instead of writing

```
>>> h(g(f(df.groupby('group')), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.groupby('group')
...     .pipe(f)
...     .pipe(g, arg1=a)
...     .pipe(h, arg2=b, arg3=c))
```

which is much more readable.

Parameters

func [callable or tuple of (callable, string)] Function to apply to this GroupBy object or, alternatively, a *(callable, data_keyword)* tuple where *data_keyword* is a string indicating the keyword of *callable* that expects the GroupBy object.

args [iterable, optional] positional arguments passed into *func*.

kwargs [dict, optional] a dictionary of keyword arguments passed into *func*.

Returns

object [the return type of *func*.]

See also:

pandas.Series.pipe Apply a function with arguments to a series.

pandas.DataFrame.pipe Apply a function with arguments to a dataframe.

apply Apply function to each group instead of to the full GroupBy object.

Notes

See more here

Examples

```
>>> df = pd.DataFrame({'A': 'a b a b'.split(), 'B': [1, 2, 3, 4]})
>>> df
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
```

To get the difference between each groups maximum and minimum value in one pass, you can do

```
>>> df.groupby('A').pipe(lambda x: x.max() - x.min())
   B
A
a  2
b  2
```

6.11.3 Computations / Descriptive Stats

GroupBy.all([skipna])	Returns True if all values in the group are truthful, else False.
GroupBy.any([skipna])	Returns True if any value in the group is truthful, else False.
GroupBy.bfill([limit])	Backward fill the values.
GroupBy.count()	Compute count of group, excluding missing values.
GroupBy.cumcount([ascending])	Number each item in each group from 0 to the length of that group - 1.
GroupBy.ffill([limit])	Forward fill the values.
GroupBy.first(**kwargs)	Compute first of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
GroupBy.head([n])	Returns first n rows of each group.
GroupBy.last(**kwargs)	Compute last of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
GroupBy.max(**kwargs)	Compute max of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
GroupBy.mean(*args, **kwargs)	Compute mean of groups, excluding missing values.
GroupBy.median(**kwargs)	Compute median of groups, excluding missing values.
GroupBy.min(**kwargs)	Compute min of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
GroupBy.ngroup([ascending])	Number each group from 0 to the number of groups - 1.
GroupBy.nth(n[, dropna])	Take the nth row from each group if n is an int, or a subset of rows if n is a list of ints.
GroupBy.ohlc()	Compute sum of values, excluding missing values.
GroupBy.prod(**kwargs)	Compute prod of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
GroupBy.rank([method, ascending, na_option, . . .])	Provides the rank of values within each group.
GroupBy.pct_change([periods, fill_method, . . .])	Calculate pct_change of each value to previous entry in group.
GroupBy.size()	Compute group sizes.
GroupBy.sem([ddof])	Compute standard error of the mean of groups, excluding missing values.
GroupBy.std([ddof])	Compute standard deviation of groups, excluding missing values.
GroupBy.sum(**kwargs)	Compute sum of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

Continued on next page

Table 428 – continued from previous page

GroupBy.var([ddof])	Compute variance of groups, excluding missing values.
GroupBy.tail([n])	Returns last n rows of each group.

pandas.core.groupby.GroupBy.all

GroupBy.**all**(*skipna=True*)
Returns True if all values in the group are truthful, else False.

Parameters

skipna [bool, default True] Flag to ignore nan values during truth testing

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.any

GroupBy.**any**(*skipna=True*)
Returns True if any value in the group is truthful, else False.

Parameters

skipna [bool, default True] Flag to ignore nan values during truth testing

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.bfill

GroupBy.**bfill**(*limit=None*)
Backward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

See also:

Series.backfill, DataFrame.backfill, Series.fillna, DataFrame.fillna

pandas.core.groupby.GroupBy.count

GroupBy.**count**()
Compute count of group, excluding missing values.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.cumcount

GroupBy.**cumcount**(*ascending=True*)
Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
>>> self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))
```

Parameters

ascending [bool, default True] If False, number in reverse, from length of group - 1 to 0.

See also:

ngroup Number the groups themselves.

Examples

```
>>> df = pd.DataFrame([['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
...                   columns=['A'])
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').cumcount()
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
>>> df.groupby('A').cumcount(ascending=False)
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

pandas.core.groupby.GroupBy.ffill

GroupBy.**ffill**(*limit=None*)

Forward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

See also:

Series.pad, DataFrame.pad, Series.fillna, DataFrame.fillna

pandas.core.groupby.GroupBy.first

GroupBy.**first**(***kwargs*)

Compute first of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.head

GroupBy.**head**(*n=5*)

Returns first n rows of each group.

Essentially equivalent to `.apply(lambda x: x.head(n))`, except ignores as_index flag.

See also:

`pandas.Series.`, `pandas.DataFrame.`, `pandas.Panel.`

Examples

```
>>> df = pd.DataFrame([[1, 2], [1, 4], [5, 6]],
                      columns=['A', 'B'])
>>> df.groupby('A', as_index=False).head(1)
   A  B
0  1  2
2  5  6
>>> df.groupby('A').head(1)
   A  B
0  1  2
2  5  6
```

pandas.core.groupby.GroupBy.last

GroupBy.**last**(***kwargs*)

Compute last of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.max

GroupBy.**max**(***kwargs*)

Compute max of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.mean

GroupBy.**mean**(**args*, ***kwargs*)

Compute mean of groups, excluding missing values.

Returns

pandas.Series or pandas.DataFrame

See also:

`pandas.Series.`, `pandas.DataFrame.`, `pandas.Panel.`

Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```
>>> df.groupby('A').mean()
>>>
     B         C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```
>>> df.groupby(['A', 'B']).mean()
>>>
       C
A B
1 2.0  2
  4.0  1
2 3.0  1
  5.0  2
```

Groupby one column and return the mean of only particular column in the group.

```
>>> df.groupby('A')['B'].mean()
>>>
A
1    3.0
2    4.0
Name: B, dtype: float64
```

pandas.core.groupby.GroupBy.median

`GroupBy.`**`median`**`(`*`**kwargs`*`)`

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

See also:

`pandas.Series.groupby`, *`pandas.DataFrame.groupby`*, *`pandas.Panel.groupby`*

pandas.core.groupby.GroupBy.min

`GroupBy.`**`min`**`(`*`**kwargs`*`)`

Compute min of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.ngroup

GroupBy.**ngroup**(*ascending=True*)

Number each group from 0 to the number of groups - 1.

This is the enumerative complement of cumcount. Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

New in version 0.20.2.

Parameters

ascending [bool, default True] If False, number in reverse, from number of group - 1 to 0.

See also:

cumcount Number the rows in each group.

Examples

```
>>> df = pd.DataFrame({"A": list("aaabba")})
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').ngroup()
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64
>>> df.groupby('A').ngroup(ascending=False)
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
>>> df.groupby(["A", [1,1,2,3,2,1]]).ngroup()
0    0
1    0
2    1
3    3
4    2
5    0
dtype: int64
```

pandas.core.groupby.GroupBy.nth

`GroupBy.nth`(*n*, *dropna=None*)

Take the nth row from each group if n is an int, or a subset of rows if n is a list of ints.

If dropna, will take the nth non-null row, dropna is either Truthy (if a Series) or 'all', 'any' (if a DataFrame); this is equivalent to calling dropna(how=dropna) before the groupby.

Parameters

n [int or list of ints] a single nth value for the row or a list of nth values

dropna [None or str, optional] apply the specified dropna operation before counting which row is the nth row. Needs to be None, 'any' or 'all'

See also:

`pandas.Series.`, `pandas.DataFrame.`, `pandas.Panel.`

Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5]}, columns=['A', 'B'])
>>> g = df.groupby('A')
>>> g.nth(0)
     B
A
1  NaN
2  3.0
>>> g.nth(1)
     B
A
1  2.0
2  5.0
>>> g.nth(-1)
     B
A
1  4.0
2  5.0
>>> g.nth([0, 1])
     B
A
1  NaN
1  2.0
2  3.0
2  5.0
```

Specifying *dropna* allows count ignoring `NaN`

```
>>> g.nth(0, dropna='any')
     B
A
1  2.0
2  3.0
```

NaNs denote group exhausted when using dropna

```
>>> g.nth(3, dropna='any')
    B
A
1 NaN
2 NaN
```

Specifying *as_index=False* in *groupby* keeps the original index.

```
>>> df.groupby('A', as_index=False).nth(1)
   A    B
1  1  2.0
4  2  5.0
```

pandas.core.groupby.GroupBy.ohlc

GroupBy.**ohlc**()

Compute sum of values, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.prod

GroupBy.**prod**(***kwargs*)

Compute prod of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.rank

GroupBy.**rank**(*method='average'*, *ascending=True*, *na_option='keep'*, *pct=False*, *axis=0*)

Provides the rank of values within each group.

Parameters

method [{'average', 'min', 'max', 'first', 'dense'}, default 'average']

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

ascending [boolean, default True] False for ranks by high (1) to low (N)

na_option [{'keep', 'top', 'bottom'}, default 'keep']

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

pct [boolean, default False] Compute percentage rank of data within each group

axis [int, default 0] The axis of the object over which to compute the rank.

Returns

—

DataFrame with ranking of values within each group

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.pct_change

GroupBy.**pct_change**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, *axis=0*)

Calculate pct_change of each value to previous entry in group.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.size

GroupBy.**size**()

Compute group sizes.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.sem

GroupBy.**sem**(*ddof=1*)

Compute standard error of the mean of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof [integer, default 1] degrees of freedom

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.std

GroupBy.**std**(*ddof=1*, **args*, ***kwargs*)

Compute standard deviation of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof [integer, default 1] degrees of freedom

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.sum

`GroupBy.sum(**kwargs)`

Compute sum of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.groupby.GroupBy.var

`GroupBy.var(ddof=1, *args, **kwargs)`

Compute variance of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof [integer, default 1] degrees of freedom

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.GroupBy.tail

`GroupBy.tail(n=5)`

Returns last n rows of each group.

Essentially equivalent to `.apply(lambda x: x.tail(n))`, except ignores as_index flag.

See also:

pandas.Series., pandas.DataFrame., pandas.Panel.

Examples

```
>>> df = pd.DataFrame([['a', 1], ['a', 2], ['b', 1], ['b', 2]],
                      columns=['A', 'B'])
>>> df.groupby('A').tail(1)
   A  B
1  a  2
3  b  2
>>> df.groupby('A').head(1)
   A  B
0  a  1
2  b  1
```

The following methods are available in both `SeriesGroupBy` and `DataFrameGroupBy` objects, but may differ slightly, usually in that the `DataFrameGroupBy` version usually permits the specification of an axis argument, and often an argument indicating whether to restrict application to columns of a specific data type.

DataFrameGroupBy.all([skipna])	Returns True if all values in the group are truthful, else False.
DataFrameGroupBy.any([skipna])	Returns True if any value in the group is truthful, else False.
DataFrameGroupBy.bfill([limit])	Backward fill the values.

Continued on next page

Table 429 – continued from previous page

DataFrameGroupBy.corr	Compute pairwise correlation of columns, excluding NA/null values.
DataFrameGroupBy.count()	Compute count of group, excluding missing values
DataFrameGroupBy.cov	Compute pairwise covariance of columns, excluding NA/null values.
DataFrameGroupBy.cummax([axis])	Cumulative max for each group.
DataFrameGroupBy.cummin([axis])	Cumulative min for each group.
DataFrameGroupBy.cumprod([axis])	Cumulative product for each group.
DataFrameGroupBy.cumsum([axis])	Cumulative sum for each group.
DataFrameGroupBy.describe(**kwargs)	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
DataFrameGroupBy.diff	First discrete difference of element.
DataFrameGroupBy.ffill([limit])	Forward fill the values.
DataFrameGroupBy.fillna	Fill NA/NaN values using the specified method.
DataFrameGroupBy.filter(func[, dropna])	Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.
DataFrameGroupBy.hist	Make a histogram of the DataFrame's.
DataFrameGroupBy.idxmax	Return index of first occurrence of maximum over requested axis.
DataFrameGroupBy.idxmin	Return index of first occurrence of minimum over requested axis.
DataFrameGroupBy.mad	Return the mean absolute deviation of the values for the requested axis.
DataFrameGroupBy.pct_change([periods, ...])	Calculate pct_change of each value to previous entry in group.
DataFrameGroupBy.plot	Class implementing the .plot attribute for groupby objects.
DataFrameGroupBy.quantile	Return values at the given quantile over requested axis.
DataFrameGroupBy.rank([method, ascending, ...])	Provides the rank of values within each group.
DataFrameGroupBy.resample(rule, *args, **kwargs)	Provide resampling when using a TimeGrouper.
DataFrameGroupBy.shift([periods, freq, ...])	Shift each group by periods observations.
DataFrameGroupBy.size()	Compute group sizes.
DataFrameGroupBy.skew	Return unbiased skew over requested axis Normalized by N-1.
DataFrameGroupBy.take	Return the elements in the given *positional* indices along an axis.
DataFrameGroupBy.tshift	Shift the time index, using the index's frequency if available.

pandas.core.groupby.DataFrameGroupBy.all

`DataFrameGroupBy.`**`all`**(*skipna=True*)

Returns True if all values in the group are truthful, else False.

Parameters

skipna [bool, default True] Flag to ignore nan values during truth testing

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.any

`DataFrameGroupBy.`**`any`**(*skipna=True*)

Returns True if any value in the group is truthful, else False.

Parameters

skipna [bool, default True] Flag to ignore nan values during truth testing

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.bfill

`DataFrameGroupBy.`**`bfill`**(*limit=None*)

Backward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

See also:

`Series.backfill`, `DataFrame.backfill`, `Series.fillna`, `DataFrame.fillna`

pandas.core.groupby.DataFrameGroupBy.corr

`DataFrameGroupBy.`**`corr`**

Compute pairwise correlation of columns, excluding NA/null values.

Parameters

method [{'pearson', 'kendall', 'spearman'} or callable]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays** and returning a float .. version-added:: 0.24.0

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

Returns

y [DataFrame]

See also:

`DataFrame.corrwith`, `Series.corr`

Examples

```
>>> histogram_intersection = lambda a, b: np.minimum(a, b
... ).sum().round(decimals=1)
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

pandas.core.groupby.DataFrameGroupBy.count

`DataFrameGroupBy.count()`
Compute count of group, excluding missing values

pandas.core.groupby.DataFrameGroupBy.cov

`DataFrameGroupBy.cov`
Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

Returns

DataFrame The covariance matrix of the series of the DataFrame.

See also:

pandas.Series.cov Compute covariance with another Series.

pandas.core.window.EWM.cov Exponential weighted sample covariance.

pandas.core.window.Expanding.cov Expanding sample covariance.

pandas.core.window.Rolling.cov Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

pandas.core.groupby.DataFrameGroupBy.cummax

`DataFrameGroupBy.`**`cummax`**(*axis=0*, ***kwargs*)

Cumulative max for each group.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.cummin

`DataFrameGroupBy.`**`cummin`**(*axis=0*, ***kwargs*)

Cumulative min for each group.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.cumprod

DataFrameGroupBy.**cumprod**(*axis=0*, **args*, ***kwargs*)

Cumulative product for each group.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.cumsum

DataFrameGroupBy.**cumsum**(*axis=0*, **args*, ***kwargs*)

Cumulative sum for each group.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.describe

DataFrameGroupBy.**describe**(***kwargs*)

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`

- None (default) : The result will exclude nothing.

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

DataFrame.count Count number of non-NA/null observations.

DataFrame.max Maximum of the values in the object.

DataFrame.min Minimum of the values in the object.

DataFrame.mean Mean of the values.

DataFrame.std Standard deviation of the obersvations.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       categorical  numeric
count            3      3.0
unique           3      NaN
```

(continues on next page)

(continued from previous page)

```
top               f       NaN
freq              1       NaN
mean            NaN       2.0
std             NaN       1.0
min             NaN       1.0
25%             NaN       1.5
50%             NaN       2.0
75%             NaN       2.5
max             NaN       3.0
```

pandas.core.groupby.DataFrameGroupBy.diff

DataFrameGroupBy.**diff**

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

Parameters

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

Returns

diffed [DataFrame]

See also:

Series.diff First discrete difference for a Series.

DataFrame.pct_change Percent change over given number of periods.

DataFrame.shift Shift index by desired number of periods with an optional time freq.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
```

(continues on next page)

(continued from previous page)

```
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

pandas.core.groupby.DataFrameGroupBy.ffill

`DataFrameGroupBy.`**`ffill`**(*limit=None*)

Forward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

See also:

`Series.pad`, `DataFrame.pad`, `Series.fillna`, `DataFrame.fillna`

pandas.core.groupby.DataFrameGroupBy.fillna

`DataFrameGroupBy.`**`fillna`**

Fill NA/NaN values using the specified method.

Parameters

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis [{0 or 'index', 1 or 'columns'}]

inplace [boolean, default False] If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns

filled [DataFrame]

See also:

`interpolate` Fill NaN values using interpolation.

`reindex`, `asfreq`

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A    B    C    D
0   NaN  2.0  NaN  0
1   3.0  4.0  NaN  1
2   3.0  4.0  NaN  5
3   3.0  3.0  NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A    B    C    D
0   0.0  2.0  2.0  0
1   3.0  4.0  2.0  1
2   0.0  1.0  2.0  5
3   0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A    B    C    D
0   0.0  2.0  2.0  0
1   3.0  4.0  NaN  1
2   NaN  1.0  NaN  5
3   NaN  3.0  NaN  4
```

pandas.core.groupby.DataFrameGroupBy.filter

`DataFrameGroupBy.filter` (*func*, *dropna=True*, **args*, ***kwargs*)

Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.

Parameters

f [function] Function to apply to each subframe. Should return True or False.

dropna [Drop groups that do not pass the filter. True by default;] if False, groups that evaluate False are filled with NaNs.

Returns

filtered [DataFrame]

Notes

Each subframe is endowed the attribute 'name' in case you need to know which group you are working on.

Examples

```
>>> df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...                           'foo', 'bar'],
...                    'B' : [1, 2, 3, 4, 5, 6],
...                    'C' : [2.0, 5., 8., 1., 2., 9.]})
```

(continues on next page)

(continued from previous page)

```
>>> grouped = df.groupby('A')
>>> grouped.filter(lambda x: x['B'].mean() > 3.)
     A  B    C
1  bar  2  5.0
3  bar  4  1.0
5  bar  6  9.0
```

pandas.core.groupby.DataFrameGroupBy.hist

`DataFrameGroupBy.hist`

Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

Parameters

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

Returns

axes [matplotlib.AxesSubplot or numpy.ndarray of them]

See also:

matplotlib.pyplot.hist Plot a histogram using matplotlib.

Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index= ['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```

pandas.core.groupby.DataFrameGroupBy.idxmax

`DataFrameGroupBy.`**`idxmax`**

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmax [Series]

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmax`

Notes

This method is the DataFrame version of `ndarray.argmax`.

pandas.core.groupby.DataFrameGroupBy.idxmin

`DataFrameGroupBy.`**`idxmin`**

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns

idxmin [Series]

Raises

ValueError

- If the row/column is empty

See also:

`Series.idxmin`

Notes

This method is the DataFrame version of `ndarray.argmin`.

pandas.core.groupby.DataFrameGroupBy.mad

`DataFrameGroupBy.`**`mad`**

Return the mean absolute deviation of the values for the requested axis.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

mad [Series or DataFrame (if level specified)]

pandas.core.groupby.DataFrameGroupBy.pct_change

`DataFrameGroupBy.`**`pct_change`**(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, *axis=0*)

Calculate pct_change of each value to previous entry in group.

See also:

`pandas.Series.groupby`, *`pandas.DataFrame.groupby`*, *`pandas.Panel.groupby`*

pandas.core.groupby.DataFrameGroupBy.plot

`DataFrameGroupBy.`**`plot`**

Class implementing the .plot attribute for groupby objects.

pandas.core.groupby.DataFrameGroupBy.quantile

`DataFrameGroupBy.quantile`

Return values at the given quantile over requested axis.

Parameters

q [float or array-like, default 0.5 (50% quantile)] Value between 0 <= q <= 1, the quantile(s) to compute.

axis [{0, 1, 'index', 'columns'} (default 0)] Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

numeric_only [bool, default True] If False, the quantile of datetime and timedelta data will be computed as well.

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

New in version 0.18.0.

Returns

quantiles [Series or DataFrame]

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

See also:

`core.window.Rolling.quantile` Rolling quantile.

`numpy.percentile` Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.groupby.DataFrameGroupBy.rank

`DataFrameGroupBy.`**`rank`**(*method='average'*, *ascending=True*, *na_option='keep'*, *pct=False*, *axis=0*)

Provides the rank of values within each group.

Parameters

method [{'average', 'min', 'max', 'first', 'dense'}, default 'average']

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

ascending [boolean, default True] False for ranks by high (1) to low (N)

na_option [{'keep', 'top', 'bottom'}, default 'keep']

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

pct [boolean, default False] Compute percentage rank of data within each group

axis [int, default 0] The axis of the object over which to compute the rank.

Returns

—

DataFrame with ranking of values within each group

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.resample

`DataFrameGroupBy.`**`resample`**(*rule*, **args*, ***kwargs*)

Provide resampling when using a TimeGrouper.

Given a grouper, the function resamples it according to a string "string" -> "frequency".

See the *frequency aliases* documentation for more details.

Parameters

rule [str or DateOffset] The offset string or object representing target grouper conversion.

***args, **kwargs** Possible arguments are *how*, *fill_method*, *limit*, *kind* and *on*, and other arguments of *TimeGrouper*.

Returns

Grouper Return a new grouper with our resampler appended.

See also:

pandas.Grouper Specify a frequency to resample with when grouping by a key.

DatetimeIndex.resample Frequency conversion and resampling of time series.

Examples

```
>>> idx = pd.date_range('1/1/2000', periods=4, freq='T')
>>> df = pd.DataFrame(data=4 * [range(2)],
...                   index=idx,
...                   columns=['a', 'b'])
>>> df.iloc[2, 0] = 5
>>> df
                     a  b
2000-01-01 00:00:00  0  1
2000-01-01 00:01:00  0  1
2000-01-01 00:02:00  5  1
2000-01-01 00:03:00  0  1
```

Downsample the DataFrame into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> df.groupby('a').resample('3T').sum()
                         a  b
a
0   2000-01-01 00:00:00  0  2
    2000-01-01 00:03:00  0  1
5   2000-01-01 00:00:00  5  1
```

Upsample the series into 30 second bins.

```
>>> df.groupby('a').resample('30S').sum()
                    a  b
a
0   2000-01-01 00:00:00  0  1
    2000-01-01 00:00:30  0  0
    2000-01-01 00:01:00  0  1
    2000-01-01 00:01:30  0  0
    2000-01-01 00:02:00  0  0
    2000-01-01 00:02:30  0  0
    2000-01-01 00:03:00  0  1
5   2000-01-01 00:02:00  5  1
```

Resample by month. Values are assigned to the month of the period.

```
>>> df.groupby('a').resample('M').sum()
            a  b
```

(continues on next page)

(continued from previous page)

```
a
0   2000-01-31  0  3
5   2000-01-31  5  1
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> df.groupby('a').resample('3T', closed='right').sum()
                         a  b
a
0   1999-12-31 23:57:00  0  1
    2000-01-01 00:00:00  0  2
5   2000-01-01 00:00:00  5  1
```

Downsample the series into 3 minute bins and close the right side of the bin interval, but label each bin using the right edge instead of the left.

```
>>> df.groupby('a').resample('3T', closed='right', label='right').sum()
                         a  b
a
0   2000-01-01 00:00:00  0  1
    2000-01-01 00:03:00  0  2
5   2000-01-01 00:03:00  5  1
```

Add an offset of twenty seconds.

```
>>> df.groupby('a').resample('3T', loffset='20s').sum()
                       a  b
a
0   2000-01-01 00:00:20  0  2
    2000-01-01 00:03:20  0  1
5   2000-01-01 00:00:20  5  1
```

pandas.core.groupby.DataFrameGroupBy.shift

`DataFrameGroupBy.shift`(*periods=1*, *freq=None*, *axis=0*, *fill_value=None*)

Shift each group by periods observations.

Parameters

periods [integer, default 1] number of periods to shift

freq [frequency string]

axis [axis to shift, default 0]

fill_value [optional] New in version 0.24.0.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.size

`DataFrameGroupBy.size`()

Compute group sizes.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.groupby.DataFrameGroupBy.skew

`DataFrameGroupBy.`**`skew`**

Return unbiased skew over requested axis Normalized by N-1.

Parameters

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

****kwargs** Additional keyword arguments to be passed to the function.

Returns

skew [Series or DataFrame (if level specified)]

pandas.core.groupby.DataFrameGroupBy.take

`DataFrameGroupBy.`**`take`**

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

Returns

taken [same type as caller] An array-like containing the elements taken from the object.

See also:

`DataFrame.loc` Select a subset of a DataFrame by labels.

`DataFrame.iloc` Select a subset of a DataFrame by positions.

`numpy.take` Take elements from an array along an axis.

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=['name', 'class', 'max_speed'],
...                   index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

pandas.core.groupby.DataFrameGroupBy.tshift

`DataFrameGroupBy.`**`tshift`**

Shift the time index, using the index's frequency if available.

Parameters

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

Returns

shifted [NDFrame]

Notes

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

The following methods are available only for `SeriesGroupBy` objects.

SeriesGroupBy.nlargest	Return the largest *n* elements.
SeriesGroupBy.nsmallest	Return the smallest *n* elements.
SeriesGroupBy.nunique([dropna])	Returns number of unique elements in the group
SeriesGroupBy.unique	Return unique values of Series object.
SeriesGroupBy.value_counts([normalize, . . .])	
SeriesGroupBy.is_monotonic_increasing	Return boolean if values in the object are monotonic_increasing.
SeriesGroupBy.is_monotonic_decreasing	Return boolean if values in the object are monotonic_decreasing.

pandas.core.groupby.SeriesGroupBy.nlargest

`SeriesGroupBy.`**`nlargest`**

Return the largest *n* elements.

Parameters

n [int, default 5] Return this many descending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- `first` : take the first occurrences based on the index order
- `last` : take the last occurrences based on the index order
- **`all`** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* largest values in the Series, sorted in decreasing order.

See also:

`Series.nsmallest` Get the *n* smallest elements.

`Series.sort_values` Sort Series by values.

`Series.head` Return the first *n* rows.

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Malta": 434000, "Maldives": 434000,
...                         "Brunei": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Malta         434000
Maldives      434000
Brunei        434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Monserat        5200
dtype: int64
```

The *n* largest elements where `n=5` by default.

```
>>> s.nlargest()
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3`. Default *keep* value is 'first' so Malta will be kept.

```
>>> s.nlargest(3)
France    65000000
Italy     59000000
Malta       434000
dtype: int64
```

The *n* largest elements where `n=3` and keeping the last duplicates. Brunei will be kept since it is the last with value 434000 based on the index order.

```
>>> s.nlargest(3, keep='last')
France      65000000
Italy       59000000
Brunei        434000
dtype: int64
```

The *n* largest elements where `n=3` with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```
>>> s.nlargest(3, keep='all')
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

pandas.core.groupby.SeriesGroupBy.nsmallest

`SeriesGroupBy.`**`nsmallest`**

Return the smallest *n* elements.

Parameters

n [int, default 5] Return this many ascending sorted values.

keep [{'first', 'last', 'all'}, default 'first'] When there are duplicate values that cannot all fit in a Series of *n* elements:

- `first` : take the first occurrences based on the index order
- `last` : take the last occurrences based on the index order
- **`all`** [keep all occurrences. This can result in a Series of] size larger than *n*.

Returns

Series The *n* smallest values in the Series, sorted in increasing order.

See also:

`Series.nlargest` Get the *n* largest elements.

`Series.sort_values` Sort Series by values.

`Series.head` Return the first *n* rows.

Notes

Faster than `.sort_values().head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Brunei": 434000, "Malta": 434000,
...                         "Maldives": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Brunei        434000
Malta         434000
Maldives      434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Monserat        5200
dtype: int64
```

The *n* largest elements where `n=5` by default.

```
>>> s.nsmallest()
Monserat      5200
Nauru        11300
Tuvalu       11300
Anguilla     11300
Iceland     337000
dtype: int64
```

The *n* smallest elements where `n=3`. Default *keep* value is 'first' so Nauru and Tuvalu will be kept.

```
>>> s.nsmallest(3)
Monserat     5200
Nauru       11300
Tuvalu      11300
dtype: int64
```

The *n* smallest elements where `n=3` and keeping the last duplicates. Anguilla and Tuvalu will be kept since they are the last with value 11300 based on the index order.

```
>>> s.nsmallest(3, keep='last')
Monserat     5200
Anguilla    11300
Tuvalu      11300
dtype: int64
```

The *n* smallest elements where `n=3` with all duplicates kept. Note that the returned Series has four elements due to the three duplicates.

```
>>> s.nsmallest(3, keep='all')
Monserat     5200
Nauru       11300
Tuvalu      11300
Anguilla    11300
dtype: int64
```

pandas.core.groupby.SeriesGroupBy.nunique

`SeriesGroupBy.`**`nunique`**(*dropna=True*)

Returns number of unique elements in the group

pandas.core.groupby.SeriesGroupBy.unique

`SeriesGroupBy.`**`unique`**

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

Returns

ndarray or ExtensionArray The unique values returned as a NumPy array. In case of an extension-array backed Series, a new `ExtensionArray` of that type with just the unique values is returned. This includes

- Categorical
- Period

- Datetime with Timezone
- Interval
- Sparse
- IntegerNA

See also:

unique Top-level unique method for any 1-d array-like object.

Index.unique Return Index with unique values from an Index object.

Examples

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...                for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                          ordered=True)).unique()
[b, a, c]
Categories (3, object): [a < b < c]
```

pandas.core.groupby.SeriesGroupBy.value_counts

SeriesGroupBy.**value_counts**(*normalize=False*, *sort=True*, *ascending=False*, *bins=None*, *dropna=True*)

pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing

SeriesGroupBy.**is_monotonic_increasing**

Return boolean if values in the object are monotonic_increasing.

New in version 0.19.0.

Returns

is_monotonic [boolean]

pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing

`SeriesGroupBy.`**`is_monotonic_decreasing`**

Return boolean if values in the object are monotonic_decreasing.

New in version 0.19.0.

Returns

is_monotonic_decreasing [boolean]

The following methods are available only for `DataFrameGroupBy` objects.

DataFrameGroupBy.corrwith	Compute pairwise correlation between rows or columns of DataFrame with rows or columns of Series or DataFrame.
DataFrameGroupBy.boxplot([subplots, column, …])	Make box plots from DataFrameGroupBy data.

pandas.core.groupby.DataFrameGroupBy.corrwith

`DataFrameGroupBy.`**`corrwith`**

Compute pairwise correlation between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

Parameters

other [DataFrame, Series]

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result

method [{'pearson', 'kendall', 'spearman'} or callable]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation
- **callable: callable with input two 1d ndarrays** and returning a float

New in version 0.24.0.

Returns

correls [Series]

See Also

——-

DataFrame.corr

pandas.core.groupby.DataFrameGroupBy.boxplot

`DataFrameGroupBy.`**`boxplot`**(*subplots=True, column=None, fontsize=None, rot=0, grid=True, ax=None, figsize=None, layout=None, sharex=False, sharey=True, **kwds*)

Make box plots from DataFrameGroupBy data.

Parameters

grouped [Grouped DataFrame]

subplots :

- `False` - no subplots will be used
- `True` - create a subplot for each group

column [column name or list of names, or vector] Can be any valid input to groupby

fontsize [int or string]

rot [label rotation angle]

grid [Setting this to True will show the grid]

ax [Matplotlib axis object, default None]

figsize [A tuple (width, height) in inches]

layout [tuple (optional)] (rows, columns) for the layout of the plot

sharex [bool, default False] Whether x-axes will be shared among subplots

New in version 0.23.1.

sharey [bool, default True] Whether y-axes will be shared among subplots

New in version 0.23.1.

'kwds'** [Keyword Arguments] All other plotting keyword arguments to be passed to matplotlib's boxplot function

Returns

dict of key/value = group key/DataFrame.boxplot return value

or DataFrame.boxplot return value in case subplots=figures=False

Examples

```
>>> import itertools
>>> tuples = [t for t in itertools.product(range(1000), range(4))]
>>> index = pd.MultiIndex.from_tuples(tuples, names=['lvl0', 'lvl1'])
>>> data = np.random.randn(len(index),4)
>>> df = pd.DataFrame(data, columns=list('ABCD'), index=index)
>>>
>>> grouped = df.groupby(level='lvl1')
>>> boxplot_frame_groupby(grouped)
>>>
>>> grouped = df.unstack(level='lvl1').groupby(level=0, axis=1)
>>> boxplot_frame_groupby(grouped, subplots=False)
```

6.12 Resampling

Resampler objects are returned by resample calls: *pandas.DataFrame.resample()*, *pandas.Series.resample()*.

6.12.1 Indexing, iteration

Resampler.__iter__()	Resampler iterator.
Resampler.groups	Dict {group name -> group labels}.
Resampler.indices	Dict {group name -> group indices}.
Resampler.get_group(name[, obj])	Constructs NDFrame from group with provided name.

pandas.core.resample.Resampler.__iter__

Resampler.**__iter__**()

Resampler iterator.

Returns

Generator yielding sequence of (name, subsetted object)

for each group

See also:

`GroupBy.__iter__`

pandas.core.resample.Resampler.groups

Resampler.**groups**

Dict {group name -> group labels}.

pandas.core.resample.Resampler.indices

Resampler.**indices**

Dict {group name -> group indices}.

pandas.core.resample.Resampler.get_group

Resampler.**get_group**(*name*, *obj=None*)

Constructs NDFrame from group with provided name.

Parameters

name [object] the name of the group to get as a DataFrame

obj [NDFrame, default None] the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns

group [same type as obj]

6.12.2 Function application

Resampler.apply(func, *args, **kwargs)	Aggregate using one or more operations over the specified axis.

Continued on next page

Table 433 – continued from previous page

Resampler.aggregate(func, *args, **kwargs)	Aggregate using one or more operations over the specified axis.
Resampler.transform(arg, *args, **kwargs)	Call function producing a like-indexed Series on each group and return a Series with the transformed values.
Resampler.pipe(func, *args, **kwargs)	Apply a function *func* with arguments to this Resampler object and return the function's result.

pandas.core.resample.Resampler.apply

Resampler.**apply**(*func*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

`pandas.DataFrame.groupby.aggregate`, `pandas.DataFrame.resample.transform`, *pandas.DataFrame.aggregate*

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> s = pd.Series([1,2,3,4,5],
                  index=pd.date_range('20130101', periods=5,freq='s'))
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
```

(continues on next page)

(continued from previous page)

```
2013-01-01 00:00:04    5
Freq: S, dtype: int64
```

```
>>> r = s.resample('2s')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left,
                        label=left, convention=start, base=0]
```

```
>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64
```

```
>>> r.agg(['sum','mean','max'])
                     sum  mean  max
2013-01-01 00:00:00    3   1.5    2
2013-01-01 00:00:02    7   3.5    4
2013-01-01 00:00:04    5   5.0    5
```

```
>>> r.agg({'result' : lambda x: x.mean() / x.std(),
           'total' : np.sum})
                     total    result
2013-01-01 00:00:00      3  2.121320
2013-01-01 00:00:02      7  4.949747
2013-01-01 00:00:04      5       NaN
```

pandas.core.resample.Resampler.aggregate

Resampler.**aggregate**(*func*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

func [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. `[np.sum, 'mean']`
- dict of axis labels -> functions, function names or list of such.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

`pandas.DataFrame.groupby.aggregate`, `pandas.DataFrame.resample.transform`, *`pandas.DataFrame.aggregate`*

Notes

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> s = pd.Series([1,2,3,4,5],
                  index=pd.date_range('20130101', periods=5,freq='s'))
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
2013-01-01 00:00:04    5
Freq: S, dtype: int64
```

```
>>> r = s.resample('2s')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left,
                        label=left, convention=start, base=0]
```

```
>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64
```

```
>>> r.agg(['sum','mean','max'])
                     sum  mean  max
2013-01-01 00:00:00    3   1.5    2
2013-01-01 00:00:02    7   3.5    4
2013-01-01 00:00:04    5   5.0    5
```

```
>>> r.agg({'result' : lambda x: x.mean() / x.std(),
           'total' : np.sum})
                     total    result
2013-01-01 00:00:00      3  2.121320
2013-01-01 00:00:02      7  4.949747
2013-01-01 00:00:04      5       NaN
```

pandas.core.resample.Resampler.transform

`Resampler.`**`transform`**(*arg*, **args*, ***kwargs*)

Call function producing a like-indexed Series on each group and return a Series with the transformed values.

Parameters

func [function] To apply to each group. Should return a Series with the same index

Returns

transformed [Series]

Examples

```
>>> resampled.transform(lambda x: (x - x.mean()) / x.std())
```

pandas.core.resample.Resampler.pipe

`Resampler.pipe`(*func*, **args*, ***kwargs*)

Apply a function *func* with arguments to this Resampler object and return the function's result.

New in version 0.23.0.

Use *.pipe* when you want to improve readability by chaining together functions that expect Series, DataFrames, GroupBy or Resampler objects. Instead of writing

```
>>> h(g(f(df.groupby('group')), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.groupby('group')
...     .pipe(f)
...     .pipe(g, arg1=a)
...     .pipe(h, arg2=b, arg3=c))
```

which is much more readable.

Parameters

func [callable or tuple of (callable, string)] Function to apply to this Resampler object or, alternatively, a *(callable, data_keyword)* tuple where *data_keyword* is a string indicating the keyword of *callable* that expects the Resampler object.

args [iterable, optional] positional arguments passed into *func*.

kwargs [dict, optional] a dictionary of keyword arguments passed into *func*.

Returns

object [the return type of *func*.]

See also:

pandas.Series.pipe Apply a function with arguments to a series.

pandas.DataFrame.pipe Apply a function with arguments to a dataframe.

apply Apply function to each group instead of to the full Resampler object.

Notes

See more here

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4]},
...                   index=pd.date_range('2012-08-02', periods=4))
>>> df
            A
2012-08-02  1
2012-08-03  2
2012-08-04  3
2012-08-05  4
```

To get the difference between each 2-day period's maximum and minimum value in one pass, you can do

```
>>> df.resample('2D').pipe(lambda x: x.max() - x.min())
            A
2012-08-02  1
2012-08-04  1
```

6.12.3 Upsampling

Resampler.ffill([limit])	Forward fill the values.
Resampler.backfill([limit])	Backward fill the new missing values in the resampled data.
Resampler.bfill([limit])	Backward fill the new missing values in the resampled data.
Resampler.pad([limit])	Forward fill the values.
Resampler.nearest([limit])	Resample by using the nearest value.
Resampler.fillna(method[, limit])	Fill missing values introduced by upsampling.
Resampler.asfreq([fill_value])	Return the values at the new freq, essentially a reindex.
Resampler.interpolate([method, axis, limit, ...])	Interpolate values according to different methods.

pandas.core.resample.Resampler.ffill

Resampler.**ffill**(*limit=None*)

Forward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

Returns

an upsampled Series

See also:

Series.fillna, DataFrame.fillna

pandas.core.resample.Resampler.backfill

Resampler.**backfill**(*limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [**?**]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency).

The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

Parameters

limit [integer, optional] Limit of how many values to fill.

Returns

Series, DataFrame An upsampled Series or DataFrame with backward filled NaN values.

See also:

bfill Alias of backfill.

fillna Fill NaN values using the specified method, which can be 'backfill'.

nearest Fill NaN values with nearest neighbor starting from center.

pad Forward fill NaN values.

pandas.Series.fillna Fill NaN values in the Series using the specified method, which can be 'backfill'.

pandas.DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'backfill'.

References

[?]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

```
>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
```

(continues on next page)

(continued from previous page)

```
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').backfill()
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('15min').backfill(limit=2)
                       a    b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  3.0
2018-01-01 00:45:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0
```

pandas.core.resample.Resampler.bfill

`Resampler.bfill`(*limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [**?**]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

Parameters

limit [integer, optional] Limit of how many values to fill.

Returns

Series, DataFrame An upsampled Series or DataFrame with backward filled NaN values.

See also:

bfill Alias of backfill.

fillna Fill NaN values using the specified method, which can be 'backfill'.

nearest Fill NaN values with nearest neighbor starting from center.

pad Forward fill NaN values.

pandas.Series.fillna Fill NaN values in the Series using the specified method, which can be 'backfill'.

pandas.DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'backfill'.

References

[?]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

```
>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').backfill()
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('15min').backfill(limit=2)
                       a    b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  3.0
2018-01-01 00:45:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0
```

pandas.core.resample.Resampler.pad

`Resampler.pad(limit=None)`
Forward fill the values.

Parameters

limit [integer, optional] limit of how many values to fill

Returns

an upsampled Series

See also:

`Series.fillna`, `DataFrame.fillna`

pandas.core.resample.Resampler.nearest

`Resampler.nearest(limit=None)`
Resample by using the nearest value.

When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The *nearest* method will replace `NaN` values that appeared in the resampled data with the value from the nearest member of the sequence, based on the index value. Missing values that existed in the original data will not be modified. If *limit* is given, fill only this many values in each direction for each of the original values.

Parameters

limit [int, optional] Limit of how many values to fill.

New in version 0.21.0.

Returns

Series or DataFrame An upsampled Series or DataFrame with `NaN` values filled with their nearest value.

See also:

backfill Backward fill the new missing values in the resampled data.

pad Forward fill NaN values.

Examples

```
>>> s = pd.Series([1, 2],
...               index=pd.date_range('20180101',
...                                   periods=2,
...                                   freq='1h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
Freq: H, dtype: int64
```

```
>>> s.resample('15min').nearest()
2018-01-01 00:00:00    1
2018-01-01 00:15:00    1
2018-01-01 00:30:00    2
2018-01-01 00:45:00    2
2018-01-01 01:00:00    2
Freq: 15T, dtype: int64
```

Limit the number of upsampled values imputed by the nearest:

```
>>> s.resample('15min').nearest(limit=1)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
Freq: 15T, dtype: float64
```

pandas.core.resample.Resampler.fillna

`Resampler.`**`fillna`**(*method*, *limit=None*)

Fill missing values introduced by upsampling.

In statistics, imputation is the process of replacing missing data with substituted values [**?**]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency).

Missing values that existed in the original data will not be modified.

Parameters

method [{'pad', 'backfill', 'ffill', 'bfill', 'nearest'}] Method to use for filling holes in resampled data

- 'pad' or 'ffill': use previous valid observation to fill gap (forward fill).
- 'backfill' or 'bfill': use next valid observation to fill gap.
- 'nearest': use nearest valid observation to fill gap.

limit [integer, optional] Limit of how many consecutive missing values to fill.

Returns

Series or DataFrame An upsampled Series or DataFrame with missing values filled.

See also:

backfill Backward fill NaN values in the resampled data.

pad Forward fill NaN values in the resampled data.

nearest Fill NaN values in the resampled data with nearest neighbor starting from center.

interpolate Fill NaN values using interpolation.

pandas.Series.fillna Fill NaN values in the Series using the specified method, which can be 'bfill' and 'ffill'.

pandas.DataFrame.fillna Fill NaN values in the DataFrame using the specified method, which can be 'bfill' and 'ffill'.

References

[**?**]

Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

Without filling the missing values you get:

```
>>> s.resample("30min").asfreq()
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    2.0
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> s.resample('30min').fillna("backfill")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').fillna("backfill", limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
```

(continues on next page)

(continued from previous page)

```
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

```
>>> s.resample('30min').fillna("pad")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    1
2018-01-01 01:00:00    2
2018-01-01 01:30:00    2
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('30min').fillna("nearest")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

Missing values present before the upsampling are not affected.

```
>>> sm = pd.Series([1, None, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> sm
2018-01-01 00:00:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 02:00:00    3.0
Freq: H, dtype: float64
```

```
>>> sm.resample('30min').fillna('backfill')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('pad')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('nearest')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
```

(continues on next page)

(continued from previous page)

```
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

DataFrame resampling is done column-wise. All the same options are available.

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                   index=pd.date_range('20180101', periods=3,
...                                       freq='h'))
>>> df
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').fillna("bfill")
                       a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

pandas.core.resample.Resampler.asfreq

`Resampler.`**`asfreq`**(*fill_value=None*)

Return the values at the new freq, essentially a reindex.

Parameters

fill_value [scalar, optional] Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

See also:

`Series.asfreq`, `DataFrame.asfreq`

pandas.core.resample.Resampler.interpolate

`Resampler.`**`interpolate`**(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

New in version 0.18.1.

Please note that only `method='linear'` is supported for DataFrame/Series with a MultiIndex.

Parameters

method [str, default 'linear'] Interpolation technique to use. One of:

- 'linear': Ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes.
- 'time': Works on daily and higher resolution data to interpolate given length of interval.

- 'index', 'values': use the actual numerical values of the index.
- 'pad': Fill in NaNs using existing values.
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline', 'barycentric', 'polynomial': Passed to *scipy.interpolate.interp1d*. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima': Wrappers around the SciPy interpolation methods of similar names. See *Notes*.
- 'from_derivatives': Refers to *scipy.interpolate.BPoly.from_derivatives* which replaces 'piecewise_polynomial' interpolation method in scipy 0.18.

New in version 0.18.1: Added support for the 'akima' method. Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in SciPy 0.18; backwards-compatible with SciPy < 0.18

axis [{0 or 'index', 1 or 'columns', None}, default None] Axis to interpolate along.

limit [int, optional] Maximum number of consecutive NaNs to fill. Must be greater than 0.

inplace [bool, default False] Update the data in place if possible.

limit_direction [{'forward', 'backward', 'both'}, default 'forward'] If limit is specified, consecutive NaNs will be filled in this direction.

limit_area [{*None*, 'inside', 'outside'}, default None] If limit is specified, consecutive NaNs will be filled with this restriction.

- `None`: No fill restriction.
- 'inside': Only fill NaNs surrounded by valid values (interpolate).
- 'outside': Only fill NaNs outside valid values (extrapolate).

New in version 0.21.0.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

****kwargs** Keyword arguments to pass on to the interpolating function.

Returns

Series or DataFrame Returns the same object type as the caller, interpolated at some or all `NaN` values

See also:

fillna Fill missing values using different methods.

scipy.interpolate.Akima1DInterpolator Piecewise cubic polynomials (Akima interpolator).

scipy.interpolate.BPoly.from_derivatives Piecewise polynomial in the Bernstein basis.

scipy.interpolate.interp1d Interpolate a 1-D function.

scipy.interpolate.KroghInterpolator Interpolate polynomial (Krogh interpolator).

scipy.interpolate.PchipInterpolator PCHIP 1-d monotonic cubic interpolation.

scipy.interpolate.CubicSpline Cubic spline data interpolator.

Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' methods are wrappers around the respective SciPy implementations of similar names. These use the actual numerical values of the index. For more information on their behavior, see the SciPy documentation and SciPy tutorial.

Examples

Filling in NaN in a *Series* via linear interpolation.

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0    0.0
1    1.0
2    NaN
3    3.0
dtype: float64
>>> s.interpolate()
0    0.0
1    1.0
2    2.0
3    3.0
dtype: float64
```

Filling in NaN in a Series by padding, but filling at most two consecutive NaN at a time.

```
>>> s = pd.Series([np.nan, "single_one", np.nan,
...                "fill_two_more", np.nan, np.nan, np.nan,
...                4.71, np.nan])
>>> s
0              NaN
1       single_one
2              NaN
3    fill_two_more
4              NaN
5              NaN
6              NaN
7             4.71
8              NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0              NaN
1       single_one
2       single_one
3    fill_two_more
4    fill_two_more
5    fill_two_more
6              NaN
7             4.71
8             4.71
dtype: object
```

Filling in NaN in a Series via polynomial interpolation or splines: Both 'polynomial' and 'spline' methods require that you also specify an order (int).

```
>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0    0.000000
1    2.000000
2    4.666667
3    8.000000
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation. Note how the first entry in column 'b' remains NaN, because there is no entry befofe it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0,  np.nan, -1.0, 1.0),
...                    (np.nan, 2.0, np.nan, np.nan),
...                    (2.0, 3.0, np.nan, 9.0),
...                    (np.nan, 4.0, -4.0, 16.0)],
...                   columns=list('abcd'))
>>> df
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  NaN  2.0  NaN   NaN
2  2.0  3.0  NaN   9.0
3  NaN  4.0 -4.0  16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
     a    b    c     d
0  0.0  NaN -1.0   1.0
1  1.0  2.0 -2.0   5.0
2  2.0  3.0 -3.0   9.0
3  2.0  4.0 -4.0  16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0     1.0
1     4.0
2     9.0
3    16.0
Name: d, dtype: float64
```

6.12.4 Computations / Descriptive Stats

Resampler.count([_method])	Compute count of group, excluding missing values.
Resampler.nunique([_method])	Returns number of unique elements in the group
Resampler.first([_method])	Compute first of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
Resampler.last([_method])	Compute last of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
Resampler.max([_method])	Compute max of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

Continued on next page

Table 435 – continued from previous page

Resampler.mean([_method])	Compute mean of groups, excluding missing values.
Resampler.median([_method])	Compute median of groups, excluding missing values.
Resampler.min([_method])	Compute min of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
Resampler.ohlc([_method])	Compute sum of values, excluding missing values.
Resampler.prod([_method, min_count])	Compute prod of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
Resampler.size()	Compute group sizes.
Resampler.sem([_method])	Compute standard error of the mean of groups, excluding missing values.
Resampler.std([ddof])	Compute standard deviation of groups, excluding missing values.
Resampler.sum([_method, min_count])	Compute sum of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby
Resampler.var([ddof])	Compute variance of groups, excluding missing values.
Resampler.quantile([q])	Return value at the given quantile.

pandas.core.resample.Resampler.count

Resampler.**count**(*_method='count'*)

Compute count of group, excluding missing values.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.resample.Resampler.nunique

Resampler.**nunique**(*_method='nunique'*)

Returns number of unique elements in the group

pandas.core.resample.Resampler.first

Resampler.**first**(*_method='first'*, **args*, ***kwargs*)

Compute first of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.last

Resampler.**last**(*_method='last'*, **args*, ***kwargs*)

Compute last of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.max

Resampler.**max**(*_method='max'*, **args*, ***kwargs*)

Compute max of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.mean

`Resampler.mean(_method='mean', *args, **kwargs)`

Compute mean of groups, excluding missing values.

Returns

pandas.Series or pandas.DataFrame

See also:

`pandas.Series.`, `pandas.DataFrame.`, `pandas.Panel.`

Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```
>>> df.groupby('A').mean()
>>>
     B         C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```
>>> df.groupby(['A', 'B']).mean()
>>>
       C
A B
1 2.0  2
  4.0  1
2 3.0  1
  5.0  2
```

Groupby one column and return the mean of only particular column in the group.

```
>>> df.groupby('A')['B'].mean()
>>>
A
1    3.0
2    4.0
Name: B, dtype: float64
```

pandas.core.resample.Resampler.median

`Resampler.median(_method='median', *args, **kwargs)`

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.resample.Resampler.min

`Resampler.min(_method='min', *args, **kwargs)`

Compute min of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.ohlc

`Resampler.ohlc(_method='ohlc', *args, **kwargs)`

Compute sum of values, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.resample.Resampler.prod

`Resampler.prod(_method='prod', min_count=0, *args, **kwargs)`

Compute prod of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.size

`Resampler.size()`

Compute group sizes.

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.resample.Resampler.sem

`Resampler.sem(_method='sem', *args, **kwargs)`

Compute standard error of the mean of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex.

Parameters

ddof [integer, default 1] degrees of freedom

See also:

pandas.Series.groupby, *pandas.DataFrame.groupby*, *pandas.Panel.groupby*

pandas.core.resample.Resampler.std

`Resampler.std(ddof=1, *args, **kwargs)`

Compute standard deviation of groups, excluding missing values.

Parameters

ddof [integer, default 1]

degrees of freedom

pandas.core.resample.Resampler.sum

Resampler.**sum**(*_method='sum'*, *min_count=0*, **args*, ***kwargs*)

Compute sum of group values See Also ——— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

pandas.core.resample.Resampler.var

Resampler.**var**(*ddof=1*, **args*, ***kwargs*)

Compute variance of groups, excluding missing values.

Parameters

ddof [integer, default 1] degrees of freedom

pandas.core.resample.Resampler.quantile

Resampler.**quantile**(*q=0.5*, ***kwargs*)

Return value at the given quantile.

New in version 0.24.0.

Parameters

q [float or array-like, default 0.5 (50% quantile)]

See also:

Series.quantile, DataFrame.quantile, DataFrameGroupBy.quantile

6.13 Style

Styler objects are returned by *pandas.DataFrame.style*.

6.13.1 Styler Constructor

Styler(data[, precision, table_styles, . . .])	Helps style a DataFrame or Series according to the data with HTML and CSS.
Styler.from_custom_template(searchpath, name)	Factory function for creating a subclass of Styler with a custom template and Jinja environment.

pandas.io.formats.style.Styler

class pandas.io.formats.style.**Styler**(*data*, *precision=None*, *table_styles=None*, *uuid=None*, *caption=None*, *table_attributes=None*, *cell_ids=True*)

Helps style a DataFrame or Series according to the data with HTML and CSS.

Parameters

data [Series or DataFrame]

precision [int] precision to round floats to, defaults to pd.options.display.precision

table_styles [list-like, default None] list of {selector: (attr, value)} dicts; see Notes

uuid [str, default None] a unique identifier to avoid CSS collisions; generated automatically

caption [str, default None] caption to attach to the table

cell_ids [bool, default True] If True, each cell will have an `id` attribute in their HTML tag. The `id` takes the form `T_<uuid>_row<num_row>_col<num_col>` where `<uuid>` is the unique identifier, `<num_row>` is the row number and `<num_col>` is the column number.

See also:

pandas.DataFrame.style

Notes

Most styling will be done by passing style functions into `Styler.apply` or `Styler.applymap`. Style functions should return values with strings containing CSS `'attr: value'` that will be applied to the indicated cells.

If using in the Jupyter notebook, Styler has defined a `_repr_html_` to automatically render itself. Otherwise call Styler.render to get the generated HTML.

CSS classes are attached to the generated HTML

- Index and Column names include `index_name` and `level<k>` where *k* is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `row<n>` where *n* is the numeric position of the row
  - `level<k>` where *k* is the level in a MultiIndex
- Column label cells include * `col_heading` * `col<n>` where *n* is the numeric position of the column * `evel<k>` where *k* is the level in a MultiIndex
- Blank cells include `blank`
- Data cells include `data`

Attributes

env	(Jinja2 Environment)
template	(Jinja2 Template)
loader	(Jinja2 Loader)

Methods

apply(func[, axis, subset])	Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.
applymap(func[, subset])	Apply a function elementwise, updating the HTML representation with the result.
background_gradient([cmap, low, high, axis, . . .])	Color the background in a gradient according to the data in each column (optionally row).

Continued on next page

Table 437 – continued from previous page

bar([subset, axis, color, width, align, ...])	Draw bar chart in the cell backgrounds.
clear()	Reset the styler, removing any previously applied styles.
export()	Export the styles to applied to the current Styler.
format(formatter[, subset])	Format the text display value of cells.
from_custom_template(searchpath, name)	Factory function for creating a subclass of `Styler` with a custom template and Jinja environment.
hide_columns(subset)	Hide columns from rendering.
hide_index()	Hide any indices from rendering.
highlight_max([subset, color, axis])	Highlight the maximum by shading the background.
highlight_min([subset, color, axis])	Highlight the minimum by shading the background.
highlight_null([null_color])	Shade the background `null_color` for missing values.
pipe(func, *args, **kwargs)	Apply `func(self, *args, **kwargs)`, and return the result.
render(**kwargs)	Render the built up styles to HTML.
set_caption(caption)	Set the caption on a Styler
set_precision(precision)	Set the precision used to render.
set_properties([subset])	Convenience method for setting one or more non-data dependent properties or each cell.
set_table_attributes(attributes)	Set the table attributes.
set_table_styles(table_styles)	Set the table styles on a Styler.
set_uuid(uuid)	Set the uuid for a Styler.
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write Styler to an Excel sheet.
use(styles)	Set the styles on the current Styler, possibly using styles from `Styler.export`.
where(cond, value[, other, subset])	Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.

pandas.io.formats.style.Styler.apply

`Styler.`**`apply`**(*func*, *axis=0*, *subset=None*, ***kwargs*)

Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.

Parameters

func [function] `func` should take a Series or DataFrame (depending on `axis`), and return an object with the same shape. Must return a DataFrame with identical index and column labels when `axis=None`

axis [int, str or None] apply to each column (`axis=0` or `'index'`) or to each row (`axis=1` or `'columns'`) or to the entire DataFrame at once with `axis=None`

subset [IndexSlice] a valid indexer to limit `data` to *before* applying the function. Consider using a pandas.IndexSlice

kwargs [dict] pass along to `func`

Returns

self [Styler]

Notes

The output shape of `func` should match the input, i.e. if `x` is the input row, column, or table (depending on `axis`), then `func(x).shape == x.shape` should be true.

This is similar to `DataFrame.apply`, except that `axis=None` applies the function to the entire DataFrame at once, rather than column-wise or row-wise.

Examples

```
>>> def highlight_max(x):
...     return ['background-color: yellow' if v == x.max() else ''
                for v in x]
...
>>> df = pd.DataFrame(np.random.randn(5, 2))
>>> df.style.apply(highlight_max)
```

pandas.io.formats.style.Styler.applymap

`Styler.`**`applymap`**(*func*, *subset=None*, ***kwargs*)

Apply a function elementwise, updating the HTML representation with the result.

Parameters

func [function] `func` should take a scalar and return a scalar

subset [IndexSlice] a valid indexer to limit `data` to *before* applying the function. Consider using a pandas.IndexSlice

kwargs [dict] pass along to `func`

Returns

self [Styler]

See also:

`Styler.where`

pandas.io.formats.style.Styler.background_gradient

`Styler.`**`background_gradient`**(*cmap='PuBu'*, *low=0*, *high=0*, *axis=0*, *subset=None*, *text_color_threshold=0.408*)

Color the background in a gradient according to the data in each column (optionally row).

Requires matplotlib.

Parameters

cmap [str or colormap] matplotlib colormap

low, high [float] compress the range by these values.

axis [int or str] 1 or 'columns' for columnwise, 0 or 'index' for rowwise

subset [IndexSlice] a valid slice for `data` to limit the style application to

text_color_threshold [float or int] luminance threshold for determining text color. Facilitates text visibility across varying background colors. From 0 to 1. 0 = all text is dark colored, 1 = all text is light colored.

New in version 0.24.0.

Returns

self [Styler]

Raises

ValueError If `text_color_threshold` is not a value from 0 to 1.

Notes

Set `text_color_threshold` or tune `low` and `high` to keep the text legible by not using the entire range of the color map. The range of the data is extended by `low * (x.max() - x.min())` and `high * (x.max() - x.min())` before normalizing.

pandas.io.formats.style.Styler.bar

`Styler.bar`(*subset=None*, *axis=0*, *color='#d65f5f'*, *width=100*, *align='left'*, *vmin=None*, *vmax=None*)

Draw bar chart in the cell backgrounds.

Parameters

subset [IndexSlice, optional] A valid slice for *data* to limit the style application to.

axis [int, str or None, default 0] Apply to each column (*axis=0* or *'index'*) or to each row (*axis=1* or *'columns'*) or to the entire DataFrame at once with *axis=None*.

color [str or 2-tuple/list] If a str is passed, the color is the same for both negative and positive numbers. If 2-tuple/list is used, the first element is the color_negative and the second is the color_positive (eg: ['#d65f5f', '#5fba7d']).

width [float, default 100] A number between 0 or 100. The largest value will cover *width* percent of the cell's width.

align [{'left', 'zero',' mid'}, default 'left'] How to align the bars with the cells.

- 'left' : the min value starts at the left of the cell.
- 'zero' : a value of zero is located at the center of the cell.
- 'mid' : the center of the cell is at (max-min)/2, or if values are all negative (positive) the zero is aligned at the right (left) of the cell.

  New in version 0.20.0.

vmin [float, optional] Minimum bar value, defining the left hand limit of the bar drawing range, lower values are clipped to *vmin*. When None (default): the minimum value of the data will be used.

New in version 0.24.0.

vmax [float, optional] Maximum bar value, defining the right hand limit of the bar drawing range, higher values are clipped to *vmax*. When None (default): the maximum value of the data will be used.

New in version 0.24.0.

Returns

self [Styler]

pandas.io.formats.style.Styler.clear

`Styler.clear()`

Reset the styler, removing any previously applied styles. Returns None.

pandas.io.formats.style.Styler.export

`Styler.export()`

Export the styles to applied to the current Styler.

Can be applied to a second style with `Styler.use`.

Returns

styles [list]

See also:

`Styler.use`

pandas.io.formats.style.Styler.format

`Styler.format(formatter, subset=None)`

Format the text display value of cells.

New in version 0.18.0.

Parameters

formatter [str, callable, or dict]

subset [IndexSlice] An argument to `DataFrame.loc` that restricts which elements `formatter` is applied to.

Returns

self [Styler]

Notes

`formatter` is either an `a` or a dict `{column name: a}` where `a` is one of

- str: this will be wrapped in: `a.format(x)`
- callable: called with the value of an individual cell

The default display value for numeric values is the "general" (`g`) format with `pd.options.display.precision` precision.

Examples

```
>>> df = pd.DataFrame(np.random.randn(4, 2), columns=['a', 'b'])
>>> df.style.format("{:.2%}")
>>> df['c'] = ['a', 'b', 'c', 'd']
>>> df.style.format({'c': str.upper})
```

pandas.io.formats.style.Styler.from_custom_template

classmethod `Styler.`**from_custom_template**(*searchpath*, *name*)

Factory function for creating a subclass of `Styler` with a custom template and Jinja environment.

Parameters

searchpath [str or list] Path or paths of directories containing the templates

name [str] Name of your custom template to use for rendering

Returns

MyStyler [subclass of Styler] has the correct `env` and `template` class attributes set.

pandas.io.formats.style.Styler.hide_columns

`Styler.`**hide_columns**(*subset*)

Hide columns from rendering.

New in version 0.23.0.

Parameters

subset [IndexSlice] An argument to `DataFrame.loc` that identifies which columns are hidden.

Returns

self [Styler]

pandas.io.formats.style.Styler.hide_index

`Styler.`**hide_index**()

Hide any indices from rendering.

New in version 0.23.0.

Returns

self [Styler]

pandas.io.formats.style.Styler.highlight_max

`Styler.`**highlight_max**(*subset=None*, *color='yellow'*, *axis=0*)

Highlight the maximum by shading the background.

Parameters

subset [IndexSlice, default None] a valid slice for `data` to limit the style application to

color [str, default 'yellow']

axis [int, str, or None; default 0] 0 or 'index' for columnwise (default), 1 or 'columns' for rowwise, or `None` for tablewise

Returns

self [Styler]

pandas.io.formats.style.Styler.highlight_min

`Styler.`**`highlight_min`**(*subset=None*, *color='yellow'*, *axis=0*)

Highlight the minimum by shading the background.

Parameters

subset [IndexSlice, default None] a valid slice for `data` to limit the style application to

color [str, default 'yellow']

axis [int, str, or None; default 0] 0 or 'index' for columnwise (default), 1 or 'columns' for rowwise, or `None` for tablewise

Returns

self [Styler]

pandas.io.formats.style.Styler.highlight_null

`Styler.`**`highlight_null`**(*null_color='red'*)

Shade the background `null_color` for missing values.

Parameters

null_color [str]

Returns

self [Styler]

pandas.io.formats.style.Styler.pipe

`Styler.`**`pipe`**(*func*, **args*, ***kwargs*)

Apply `func(self, *args, **kwargs)`, and return the result.

New in version 0.24.0.

Parameters

func [function] Function to apply to the Styler. Alternatively, a `(callable, keyword)` tuple where `keyword` is a string indicating the keyword of `callable` that expects the Styler.

***args, **kwargs :** Arguments passed to *func*.

Returns

object : The value returned by `func`.

See also:

`DataFrame.pipe` Analogous method for DataFrame.

Styler.apply Apply a function row-wise, column-wise, or table-wise to modify the dataframe's styling.

Notes

Like `DataFrame.pipe()`, this method can simplify the application of several user-defined functions to a styler. Instead of writing:

```
f(g(df.style.set_precision(3), arg1=a), arg2=b, arg3=c)
```

users can write:

```
(df.style.set_precision(3)
   .pipe(g, arg1=a)
   .pipe(f, arg2=b, arg3=c))
```

In particular, this allows users to define functions that take a styler object, along with other parameters, and return the styler after making styling changes (such as calling *Styler.apply()* or *Styler.set_properties()*). Using `.pipe`, these user-defined style "transformations" can be interleaved with calls to the built-in Styler interface.

Examples

```
>>> def format_conversion(styler):
...     return (styler.set_properties(**{'text-align': 'right'})
...                   .format({'conversion': '{:.1%}'}))
```

The user-defined `format_conversion` function above can be called within a sequence of other style modifications:

```
>>> df = pd.DataFrame({'trial': list(range(5)),
...                    'conversion': [0.75, 0.85, np.nan, 0.7, 0.72]})
>>> (df.style
...     .highlight_min(subset=['conversion'], color='yellow')
...     .pipe(format_conversion)
...     .set_caption("Results with minimum conversion highlighted."))
```

pandas.io.formats.style.Styler.render

`Styler.`**`render`**`(`***kwargs*`)`

Render the built up styles to HTML.

Parameters

'kwargs'** [Any additional keyword arguments are passed through]

to ''self.template.render''. This is useful when you need to provide

additional variables for a custom template. New in version 0.20.

Returns

rendered [str] the rendered HTML

Notes

`Styler` objects have defined the `_repr_html_` method which automatically calls `self.render()` when it's the last item in a Notebook cell. When calling `Styler.render()` directly, wrap the result in `IPython.display.HTML` to view the rendered HTML in the notebook.

Pandas uses the following keys in render. Arguments passed in `**kwargs` take precedence, so think carefully if you want to override them:

- head
- cellstyle
- body
- uuid
- precision
- table_styles
- caption
- table_attributes

pandas.io.formats.style.Styler.set_caption

`Styler.`**`set_caption`**`(`*caption*`)`

Set the caption on a Styler

Parameters

caption [str]

Returns

self [Styler]

pandas.io.formats.style.Styler.set_precision

`Styler.`**`set_precision`**`(`*precision*`)`

Set the precision used to render.

Parameters

precision [int]

Returns

self [Styler]

pandas.io.formats.style.Styler.set_properties

`Styler.`**`set_properties`**`(`*subset=None*, ***kwargs*`)`

Convenience method for setting one or more non-data dependent properties or each cell.

Parameters

subset [IndexSlice] a valid slice for `data` to limit the style application to

kwargs [dict] property: value pairs to be set for each cell

Returns

self [Styler]

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_properties(color="white", align="right")
>>> df.style.set_properties(**{'background-color': 'yellow'})
```

pandas.io.formats.style.Styler.set_table_attributes

`Styler.`**`set_table_attributes`**(*attributes*)

Set the table attributes.

These are the items that show up in the opening `<table>` tag in addition to to automatic (by default) id.

Parameters

attributes [string]

Returns

self [Styler]

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_attributes('class="pure-table"')
# ... <table class="pure-table"> ...
```

pandas.io.formats.style.Styler.set_table_styles

`Styler.`**`set_table_styles`**(*table_styles*)

Set the table styles on a Styler.

These are placed in a `<style>` tag before the generated HTML table.

Parameters

table_styles [list] Each individual table_style should be a dictionary with `selector` and `props` keys. `selector` should be a CSS selector that the style will be applied to (automatically prefixed by the table's UUID) and `props` should be a list of tuples with `(attribute, value)`.

Returns

self [Styler]

Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_styles(
...     [{'selector': 'tr:hover',
...       'props': [('background-color', 'yellow')]}]
... )
```

pandas.io.formats.style.Styler.set_uuid

`Styler.set_uuid(uuid)`

Set the uuid for a Styler.

Parameters

uuid [str]

Returns

self [Styler]

pandas.io.formats.style.Styler.to_excel

`Styler.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None)`

Write Styler to an Excel sheet.

To write a single Styler to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

Parameters

excel_writer [str or ExcelWriter object] File path or existing ExcelWriter.

sheet_name [str, default 'Sheet1'] Name of sheet which will contain DataFrame.

na_rep [str, default ''] Missing data representation.

float_format [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.

columns [sequence or list of str, optional] Columns to write.

header [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

index [bool, default True] Write row names (index).

index_label [str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow [int, default 0] Upper left cell row to dump data frame.

startcol [int, default 0] Upper left cell column to dump data frame.

engine [str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

verbose [bool, default True] Display more information in the error logs.

freeze_panes [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

New in version 0.20.0..

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

`ExcelWriter` Class for writing DataFrame objects into excel sheets.

`read_excel` Read an Excel file into a pandas DataFrame.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

Notes

For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                    index=['row 1', 'row 2'],
...                    columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")  # doctest: +SKIP
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')  # doctest: +SKIP
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer:  # doctest: +SKIP
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')  # doctest: +SKIP
```

pandas.io.formats.style.Styler.use

`Styler.use(styles)`

Set the styles on the current Styler, possibly using styles from `Styler.export`.

Parameters

styles [list] list of style functions

Returns

self [Styler]

See also:

Styler.export

pandas.io.formats.style.Styler.where

`Styler.where(cond, value, other=None, subset=None, **kwargs)`

Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.

New in version 0.21.0.

Parameters

cond [callable] `cond` should take a scalar and return a boolean

value [str] applied when `cond` returns true

other [str] applied when `cond` returns false

subset [IndexSlice] a valid indexer to limit `data` to *before* applying the function. Consider using a pandas.IndexSlice

kwargs [dict] pass along to `cond`

Returns

self [Styler]

See also:

Styler.applymap

6.13.2 Styler Properties

Styler.env
Styler.template
Styler.loader

pandas.io.formats.style.Styler.env

Styler.**env = <jinja2.environment.Environment object>**

pandas.io.formats.style.Styler.template

Styler.**template = <Template 'html.tpl'>**

pandas.io.formats.style.Styler.loader

Styler.**loader = <jinja2.loaders.PackageLoader object>**

6.13.3 Style Application

Styler.apply(func[, axis, subset])	Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.
Styler.applymap(func[, subset])	Apply a function elementwise, updating the HTML representation with the result.
Styler.where(cond, value[, other, subset])	Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.
Styler.format(formatter[, subset])	Format the text display value of cells.
Styler.set_precision(precision)	Set the precision used to render.
Styler.set_table_styles(table_styles)	Set the table styles on a Styler.
Styler.set_table_attributes(attributes)	Set the table attributes.
Styler.set_caption(caption)	Set the caption on a Styler
Styler.set_properties([subset])	Convenience method for setting one or more non-data dependent properties or each cell.
Styler.set_uuid(uuid)	Set the uuid for a Styler.
Styler.clear()	Reset the styler, removing any previously applied styles.
Styler.pipe(func, *args, **kwargs)	Apply `func(self, *args, **kwargs)`, and return the result.

6.13.4 Builtin Styles

Styler.highlight_max([subset, color, axis])	Highlight the maximum by shading the background.
Styler.highlight_min([subset, color, axis])	Highlight the minimum by shading the background.
Styler.highlight_null([null_color])	Shade the background `null_color` for missing values.
Styler.background_gradient([cmap, low, …])	Color the background in a gradient according to the data in each column (optionally row).
Styler.bar([subset, axis, color, width, …])	Draw bar chart in the cell backgrounds.

6.13.5 Style Export and Import

Styler.render(**kwargs)	Render the built up styles to HTML.
Styler.export()	Export the styles to applied to the current Styler.

Continued on next page

Table 441 – continued from previous page

Styler.use(styles)	Set the styles on the current Styler, possibly using styles from `Styler.export`.
Styler.to_excel(excel_writer[, sheet_name, . . .])	Write Styler to an Excel sheet.

6.14 Plotting

The following functions are contained in the *pandas.plotting* module.

andrews_curves(frame, class_column[, ax, . . .])	Generates a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.
bootstrap_plot(series[, fig, size, samples])	Bootstrap plot on mean, median and mid-range statistics.
deregister_matplotlib_converters()	Remove pandas' formatters and converters
lag_plot(series[, lag, ax])	Lag plot for time series.
parallel_coordinates(frame, class_column[, . . .])	Parallel coordinates plotting.
radviz(frame, class_column[, ax, color, . . .])	Plot a multidimensional dataset in 2D.
register_matplotlib_converters([explicit])	Register Pandas Formatters and Converters with matplotlib
scatter_matrix(frame[, alpha, figsize, ax, . . .])	Draw a matrix of scatter plots.

6.14.1 pandas.plotting.andrews_curves

`pandas.plotting.`**`andrews_curves`** (*frame*, *class_column*, *ax=None*, *samples=200*, *color=None*, *colormap=None*, ***kwds*)

Generates a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.

Andrews curves have the functional form:

f(t) = x_1/sqrt(2) + x_2 sin(t) + x_3 cos(t) + x_4 sin(2t) + x_5 cos(2t) + . . .

Where x coefficients correspond to the values of each dimension and t is linearly spaced between -pi and +pi. Each row of frame then corresponds to a single curve.

Parameters

frame [DataFrame] Data to be plotted, preferably normalized to (0.0, 1.0)

class_column [Name of the column containing class names]

ax [matplotlib axes object, default None]

samples [Number of points to plot in each curve]

color [list or tuple, optional] Colors to use for the different classes

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

kwds [keywords] Options to pass to matplotlib plotting method

Returns

ax [Matplotlib axis object]

6.14.2 pandas.plotting.bootstrap_plot

`pandas.plotting.bootstrap_plot(series, fig=None, size=50, samples=500, **kwds)`

Bootstrap plot on mean, median and mid-range statistics.

The bootstrap plot is used to estimate the uncertainty of a statistic by relaying on random sampling with replacement [?]. This function will generate bootstrapping plots for mean, median and mid-range statistics for the given number of samples of the given size.

Parameters

series [pandas.Series] Pandas Series from where to get the samplings for the bootstrapping.

fig [matplotlib.figure.Figure, default None] If given, it will use the *fig* reference for plotting instead of creating a new one with default parameters.

size [int, default 50] Number of data points to consider during each sampling. It must be greater or equal than the length of the *series*.

samples [int, default 500] Number of times the bootstrap procedure is performed.

****kwds :** Options to pass to matplotlib plotting method.

Returns

fig [matplotlib.figure.Figure] Matplotlib figure

See also:

pandas.DataFrame.plot Basic plotting for DataFrame objects.

pandas.Series.plot Basic plotting for Series objects.

Examples

```
>>> s = pd.Series(np.random.uniform(size=100))
>>> fig = pd.plotting.bootstrap_plot(s)  # doctest: +SKIP
```

6.14.3 pandas.plotting.deregister_matplotlib_converters

`pandas.plotting.deregister_matplotlib_converters()`

Remove pandas' formatters and converters

Removes the custom converters added by `register()`. This attempts to set the state of the registry back to the state before pandas registered its own units. Converters for pandas' own types like Timestamp and Period are removed completely. Converters for types pandas overwrites, like `datetime.datetime`, are restored to their original value.

See also:

deregister_matplotlib_converters

6.14.4 pandas.plotting.lag_plot

`pandas.plotting.lag_plot(series, lag=1, ax=None, **kwds)`

Lag plot for time series.

Parameters

series [Time series]

lag [lag of the scatter plot, default 1]

ax [Matplotlib axis object, optional]

kwds [Matplotlib scatter method keyword arguments, optional]

Returns

ax: Matplotlib axis object

6.14.5 pandas.plotting.parallel_coordinates

`pandas.plotting.parallel_coordinates`(*frame*, *class_column*, *cols=None*, *ax=None*, *color=None*, *use_columns=False*, *xticks=None*, *colormap=None*, *axvlines=True*, *axvlines_kwds=None*, *sort_labels=False*, ***kwds*)

Parallel coordinates plotting.

Parameters

frame [DataFrame]

class_column [str] Column name containing class names

cols [list, optional] A list of column names to use

ax [matplotlib.axis, optional] matplotlib axis object

color [list or tuple, optional] Colors to use for the different classes

use_columns [bool, optional] If true, columns will be used as xticks

xticks [list or tuple, optional] A list of values to use for xticks

colormap [str or matplotlib colormap, default None] Colormap to use for line colors.

axvlines [bool, optional] If true, vertical lines will be added at each xtick

axvlines_kwds [keywords, optional] Options to be passed to axvline method for vertical lines

sort_labels [bool, False] Sort class_column labels, useful when assigning colors

New in version 0.20.0.

kwds [keywords] Options to pass to matplotlib plotting method

Returns

ax: matplotlib axis object

Examples

```
>>> from matplotlib import pyplot as plt
>>> df = pd.read_csv('https://raw.github.com/pandas-dev/pandas/master'
                     '/pandas/tests/data/iris.csv')
>>> pd.plotting.parallel_coordinates(
        df, 'Name',
        color=('#556270', '#4ECDC4', '#C7F464'))
>>> plt.show()
```

6.14.6 pandas.plotting.radviz

pandas.plotting.**radviz**(*frame*, *class_column*, *ax=None*, *color=None*, *colormap=None*, ***kwds*)

Plot a multidimensional dataset in 2D.

Each Series in the DataFrame is represented as a evenly distributed slice on a circle. Each data point is rendered in the circle according to the value on each Series. Highly correlated *Series* in the *DataFrame* are placed closer on the unit circle.

RadViz allow to project a N-dimensional data set into a 2D space where the influence of each dimension can be interpreted as a balance between the influence of all dimensions.

More info available at the original article describing RadViz.

Parameters

frame [*DataFrame*] Pandas object holding the data.

class_column [str] Column name containing the name of the data point category.

ax [`matplotlib.axes.Axes`, optional] A plot instance to which to add the information.

color [list[str] or tuple[str], optional] Assign a color to each category. Example: ['blue', 'green'].

colormap [str or `matplotlib.colors.Colormap`, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

kwds [optional] Options to pass to matplotlib scatter plotting method.

Returns

axes [`matplotlib.axes.Axes`]

See also:

`pandas.plotting.andrews_curves` Plot clustering visualization.

Examples

```
>>> df = pd.DataFrame({
...         'SepalLength': [6.5, 7.7, 5.1, 5.8, 7.6, 5.0, 5.4, 4.6,
...                         6.7, 4.6],
...         'SepalWidth': [3.0, 3.8, 3.8, 2.7, 3.0, 2.3, 3.0, 3.2,
...                        3.3, 3.6],
...         'PetalLength': [5.5, 6.7, 1.9, 5.1, 6.6, 3.3, 4.5, 1.4,
...                         5.7, 1.0],
...         'PetalWidth': [1.8, 2.2, 0.4, 1.9, 2.1, 1.0, 1.5, 0.2,
...                        2.1, 0.2],
...         'Category': ['virginica', 'virginica', 'setosa',
...                      'virginica', 'virginica', 'versicolor',
...                      'versicolor', 'setosa', 'virginica',
...                      'setosa']
...     })
>>> rad_viz = pd.plotting.radviz(df, 'Category')  # doctest: +SKIP
```

6.14.7 pandas.plotting.register_matplotlib_converters

pandas.plotting.**register_matplotlib_converters**(*explicit=True*)

Register Pandas Formatters and Converters with matplotlib

This function modifies the global `matplotlib.units.registry` dictionary. Pandas adds custom converters for

- pd.Timestamp
- pd.Period
- np.datetime64
- datetime.datetime
- datetime.date
- datetime.time

See also:

`deregister_matplotlib_converter`

6.14.8 pandas.plotting.scatter_matrix

pandas.plotting.**scatter_matrix**(*frame*, *alpha=0.5*, *figsize=None*, *ax=None*, *grid=False*, *diagonal='hist'*, *marker='.'*, *density_kwds=None*, *hist_kwds=None*, *range_padding=0.05*, ***kwds*)

Draw a matrix of scatter plots.

Parameters

frame [DataFrame]

alpha [float, optional] amount of transparency applied

figsize [(float,float), optional] a tuple (width, height) in inches

ax [Matplotlib axis object, optional]

grid [bool, optional] setting this to True will show the grid

diagonal [{'hist', 'kde'}] pick between 'kde' and 'hist' for either Kernel Density Estimation or Histogram plot in the diagonal

marker [str, optional] Matplotlib marker type, default '.'

hist_kwds [other plotting keyword arguments] To be passed to hist function

density_kwds [other plotting keyword arguments] To be passed to kernel density estimate plot

range_padding [float, optional] relative extension of axis range in x and y with respect to (x_max - x_min) or (y_max - y_min), default 0.05

kwds [other plotting keyword arguments] To be passed to scatter function

Examples

```
>>> df = pd.DataFrame(np.random.randn(1000, 4), columns=['A','B','C','D'])
>>> scatter_matrix(df, alpha=0.2)
```

6.15 General utility functions

6.15.1 Working with options

describe_option(pat[, _print_desc])	Prints the description for one or more registered options.
reset_option(pat)	Reset one or more options to their default value.
get_option(pat)	Retrieves the value of the specified option.
set_option(pat, value)	Sets the value of the specified option.
option_context(*args)	Context manager to temporarily set options in the *with* statement context.

pandas.describe_option

pandas.**describe_option**(*pat*, *_print_desc=False*) **= <pandas.core.config.CallableDynamicDoc object>**

Prints the description for one or more registered options.

Call with not arguments to get a listing for all registered options.

Available options:

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]
- plotting.matplotlib.[register_converters]

Parameters

pat [str] Regexp pattern. All matching keys will have their description displayed.

_print_desc [bool, default True] If True (default) the description(s) will be printed to stdout. Otherwise, the description(s) will be returned as a unicode string (for testing).

Returns

None by default, the description(s) as a unicode string if _print_desc is False

Notes

The available options with its descriptions:

compute.use_bottleneck [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

compute.use_numexpr [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

display.chop_threshold [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

display.colheader_justify ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

display.column_space No description available. [default: 12] [currently: 12]

display.date_dayfirst [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

display.date_yearfirst [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

display.encoding [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

display.expand_frame_repr [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, *max_columns* is still respected, but the output will wrap-around across multiple "pages" if its width exceeds *display.width*. [default: True] [currently: True]

display.float_format [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

display.html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

display.large_repr ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

display.latex.longtable :bool This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multi-columns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]

display.latex.multirow [bool] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]

display.latex.repr [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype "category". [default: 8] [currently: 8]

display.max_columns [int] If max_cols is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "…" placeholder is embedded in the output. [default: 50] [currently: 50]

display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

display.max_seq_items [int or None] when pretty-printing a long sequence, no more then *max_seq_items* will be printed. If items are omitted, they will be denoted by the addition of "…" to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,'deep' [default: True] [currently: True]

display.multi_sparse [boolean] "sparsify" MultiIndex display (don't display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]

display.show_dimensions [boolean or 'truncate'] Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]

html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use *display.html.border* instead.)

io.excel.xls.writer [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]

io.excel.xlsm.writer [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]

io.excel.xlsx.writer [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]

io.hdf.default_format [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]

io.parquet.engine [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]

mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use *use_inf_as_na* instead. [default: False] [currently: False] (Deprecated, use *mode.use_inf_as_na* instead.)

plotting.matplotlib.register_converters [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

pandas.reset_option

`pandas.reset_option(pat) = <pandas.core.config.CallableDynamicDoc object>`

Reset one or more options to their default value.

Pass "all" as argument to reset all options.

Available options:

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]
- plotting.matplotlib.[register_converters]

Parameters

pat [str/regex] If specified only options matching *prefix** will be reset. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

Returns

None

Notes

The available options with its descriptions:

compute.use_bottleneck [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

compute.use_numexpr [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

display.chop_threshold [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

display.colheader_justify ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

display.column_space No description available. [default: 12] [currently: 12]

display.date_dayfirst [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

display.date_yearfirst [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

display.encoding [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

display.expand_frame_repr [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, *max_columns* is still respected, but the output will wrap-around across multiple "pages" if its width exceeds *display.width*. [default: True] [currently: True]

display.float_format [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

display.html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

display.large_repr ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

display.latex.longtable :bool This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]

display.latex.multirow [bool] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]

display.latex.repr [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype "category". [default: 8] [currently: 8]

display.max_columns [int] If max_cols is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "…" placeholder is embedded in the output. [default: 50] [currently: 50]

display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

display.max_seq_items [int or None] when pretty-printing a long sequence, no more then *max_seq_items* will be printed. If items are omitted, they will be denoted by the addition of "…" to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,'deep' [default: True] [currently: True]

display.multi_sparse [boolean] "sparsify" MultiIndex display (don't display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]

display.show_dimensions [boolean or 'truncate'] Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]

html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use *display.html.border* instead.)

io.excel.xls.writer [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]

io.excel.xlsm.writer [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]

io.excel.xlsx.writer [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]

io.hdf.default_format [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]

io.parquet.engine [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]

mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use *use_inf_as_na* instead. [default: False] [currently: False] (Deprecated, use *mode.use_inf_as_na* instead.)

plotting.matplotlib.register_converters [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

pandas.get_option

`pandas.get_option(pat) = <pandas.core.config.CallableDynamicDoc object>`

Retrieves the value of the specified option.

Available options:

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]

- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]
- plotting.matplotlib.[register_converters]

Parameters

pat [str] Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

Returns

result [the value of the option]

Raises

OptionError [if no such option exists]

Notes

The available options with its descriptions:

compute.use_bottleneck [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

compute.use_numexpr [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

display.chop_threshold [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

display.colheader_justify ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

display.column_space No description available. [default: 12] [currently: 12]

display.date_dayfirst [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

display.date_yearfirst [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

display.encoding [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

display.expand_frame_repr [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, *max_columns* is still respected, but the output will wrap-around across multiple "pages" if its width exceeds *display.width*. [default: True] [currently: True]

display.float_format [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

display.html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

display.large_repr ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

display.latex.longtable :bool This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]

display.latex.multirow [bool] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]

display.latex.repr [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype "category". [default: 8] [currently: 8]

display.max_columns [int] If max_cols is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "..." placeholder is embedded in the output. [default: 50] [currently: 50]

display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

display.max_seq_items [int or None] when pretty-printing a long sequence, no more then *max_seq_items* will be printed. If items are omitted, they will be denoted by the addition of "..." to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,'deep' [default: True] [currently: True]

display.multi_sparse [boolean] "sparsify" MultiIndex display (don't display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]

display.show_dimensions [boolean or 'truncate'] Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]

html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use *display.html.border* instead.)

io.excel.xls.writer [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]

io.excel.xlsm.writer [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]

io.excel.xlsx.writer [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]

io.hdf.default_format [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]

io.parquet.engine [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]

mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use *use_inf_as_na* instead. [default: False] [currently: False] (Deprecated, use *mode.use_inf_as_na* instead.)

plotting.matplotlib.register_converters [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

pandas.set_option

pandas.**set_option**(*pat*, *value*) = <pandas.core.config.CallableDynamicDoc object>

Sets the value of the specified option.

Available options:

- compute.[use_bottleneck, use_numexpr]
- display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]
- display.html.[border, table_schema, use_mathjax]
- display.[large_repr]
- display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]
- display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]
- display.unicode.[ambiguous_as_wide, east_asian_width]
- display.[width]
- html.[border]
- io.excel.xls.[writer]
- io.excel.xlsm.[writer]
- io.excel.xlsx.[writer]
- io.hdf.[default_format, dropna_table]
- io.parquet.[engine]
- mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]
- plotting.matplotlib.[register_converters]

Parameters

pat [str] Regexp which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

value : new value of option.

Returns

None

Raises

OptionError if no such option exists

Notes

The available options with its descriptions:

compute.use_bottleneck [bool] Use the bottleneck library to accelerate if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

compute.use_numexpr [bool] Use the numexpr library to accelerate computation if it is installed, the default is True Valid values: False,True [default: True] [currently: True]

display.chop_threshold [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

display.colheader_justify ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

display.column_space No description available. [default: 12] [currently: 12]

display.date_dayfirst [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

display.date_yearfirst [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

display.encoding [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to_string, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

display.expand_frame_repr [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, *max_columns* is still respected, but the output will wrap-around across multiple "pages" if its width exceeds *display.width*. [default: True] [currently: True]

display.float_format [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

display.html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

display.html.table_schema [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

display.html.use_mathjax [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

display.large_repr ['truncate'/'info'] For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

display.latex.escape [bool] This specifies if the to_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

display.latex.longtable :bool This specifies if the to_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

display.latex.multicolumn [bool] This specifies if the to_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

display.latex.multicolumn_format [bool] This specifies if the to_latex method of a Dataframe uses multi-columns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]

display.latex.multirow [bool] This specifies if the to_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]

display.latex.repr [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

display.max_categories [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype "category". [default: 8] [currently: 8]

display.max_columns [int] If max_cols is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

display.max_colwidth [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "…" placeholder is embedded in the output. [default: 50] [currently: 50]

display.max_info_columns [int] max_info_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

display.max_info_rows [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max_info_rows and max_info_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

display.max_rows [int] If max_rows is exceeded, switch to truncate view. Depending on *large_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.

In case python/IPython is running in a terminal and *large_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

display.max_seq_items [int or None] when pretty-printing a long sequence, no more then *max_seq_items* will be printed. If items are omitted, they will be denoted by the addition of "…" to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

display.memory_usage [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,'deep' [default: True] [currently: True]

display.multi_sparse [boolean] "sparsify" MultiIndex display (don't display repeated elements in outer levels within groups) [default: True] [currently: True]

display.notebook_repr_html [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

display.pprint_nest_depth [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

display.precision [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]

display.show_dimensions [boolean or 'truncate'] Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

display.unicode.ambiguous_as_wide [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.unicode.east_asian_width [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

display.width [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]

html.border [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use *display.html.border* instead.)

io.excel.xls.writer [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]

io.excel.xlsm.writer [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]

io.excel.xlsx.writer [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]

io.hdf.default_format [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]

io.hdf.dropna_table [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]

io.parquet.engine [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]

mode.chained_assignment [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]

mode.sim_interactive [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]

mode.use_inf_as_na [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]

mode.use_inf_as_null [boolean] use_inf_as_null had been deprecated and will be removed in a future version. Use *use_inf_as_na* instead. [default: False] [currently: False] (Deprecated, use *mode.use_inf_as_na* instead.)

plotting.matplotlib.register_converters [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

pandas.option_context

class `pandas.option_context`(**args*)

Context manager to temporarily set options in the *with* statement context.

You need to invoke as `option_context(pat, val, [(pat, val), ...])`.

Examples

```
>>> with option_context('display.max_rows', 10, 'display.max_columns', 5):
...     ...
```

6.15.2 Testing functions

testing.assert_frame_equal(left, right[, ...])	Check that left and right DataFrame are equal.
testing.assert_series_equal(left, right[, ...])	Check that left and right Series are equal.
testing.assert_index_equal(left, right[, ...])	Check that left and right Index are equal.

pandas.testing.assert_frame_equal

pandas.testing.**assert_frame_equal**(*left, right, check_dtype=True, check_index_type='equiv', check_column_type='equiv', check_frame_type=True, check_less_precise=False, check_names=True, by_blocks=False, check_exact=False, check_datetimelike_compat=False, check_categorical=True, check_like=False, obj='DataFrame'*)

Check that left and right DataFrame are equal.

This function is intended to compare two DataFrames and output any differences. Is is mostly intended for use in unit tests. Additional parameters allow varying the strictness of the equality checks performed.

Parameters

left [DataFrame] First DataFrame to compare.

right [DataFrame] Second DataFrame to compare.

check_dtype [bool, default True] Whether to check the DataFrame dtype is identical.

check_index_type [bool / string {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and inferred_type are identical.

check_column_type [bool / string {'equiv'}, default 'equiv'] Whether to check the columns class, dtype and inferred_type are identical. Is passed as the `exact` argument of *assert_index_equal()*.

check_frame_type [bool, default True] Whether to check the DataFrame class is identical.

check_less_precise [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare.

check_names [bool, default True] Whether to check that the *names* attribute for both the *index* and *column* attributes of the DataFrame is identical, i.e.

- left.index.names == right.index.names
- left.columns.names == right.columns.names

by_blocks [bool, default False] Specify how to compare internal data. If False, compare by columns. If True, compare by blocks.

check_exact [bool, default False] Whether to compare number exactly.

check_datetimelike_compat [bool, default False] Compare datetime-like which is comparable ignoring dtype.

check_categorical [bool, default True] Whether to compare internal Categorical exactly.

check_like [bool, default False] If True, ignore the order of index & columns. Note: index labels must match their respective rows (same as in columns) - same labels must be with the same data.

obj [str, default 'DataFrame'] Specify object name being compared, internally used to show appropriate assertion message.

See also:

assert_series_equal Equivalent method for asserting Series equality.

DataFrame.equals Check DataFrame equality.

Examples

This example shows comparing two DataFrames that are equal but with columns of differing dtypes.

```
>>> from pandas.util.testing import assert_frame_equal
>>> df1 = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
>>> df2 = pd.DataFrame({'a': [1, 2], 'b': [3.0, 4.0]})
```

df1 equals itself. >>> assert_frame_equal(df1, df1)

df1 differs from df2 as column 'b' is of a different type. >>> assert_frame_equal(df1, df2) Traceback (most recent call last): AssertionError: Attributes are different

Attribute "dtype" are different [left]: int64 [right]: float64

Ignore differing dtypes in columns with check_dtype. >>> assert_frame_equal(df1, df2, check_dtype=False)

pandas.testing.assert_series_equal

pandas.testing.**assert_series_equal**(*left*, *right*, *check_dtype=True*, *check_index_type='equiv'*, *check_series_type=True*, *check_less_precise=False*, *check_names=True*, *check_exact=False*, *check_datetimelike_compat=False*, *check_categorical=True*, *obj='Series'*)

Check that left and right Series are equal.

Parameters

left [Series]

right [Series]

check_dtype [bool, default True] Whether to check the Series dtype is identical.

check_index_type [bool / string {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and inferred_type are identical.

check_series_type [bool, default True] Whether to check the Series class is identical.

check_less_precise [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare.

check_names [bool, default True] Whether to check the Series and Index names attribute.

check_exact [bool, default False] Whether to compare number exactly.

check_datetimelike_compat [bool, default False] Compare datetime-like which is comparable ignoring dtype.

check_categorical [bool, default True] Whether to compare internal Categorical exactly.

obj [str, default 'Series'] Specify object name being compared, internally used to show appropriate assertion message.

pandas.testing.assert_index_equal

`pandas.testing.assert_index_equal`(*left*, *right*, *exact='equiv'*, *check_names=True*, *check_less_precise=False*, *check_exact=True*, *check_categorical=True*, *obj='Index'*)

Check that left and right Index are equal.

Parameters

left [Index]

right [Index]

exact [bool / string {'equiv'}, default 'equiv'] Whether to check the Index class, dtype and inferred_type are identical. If 'equiv', then RangeIndex can be substituted for Int64Index as well.

check_names [bool, default True] Whether to check the names attribute.

check_less_precise [bool or int, default False] Specify comparison precision. Only used when check_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare

check_exact [bool, default True] Whether to compare number exactly.

check_categorical [bool, default True] Whether to compare internal Categorical exactly.

obj [str, default 'Index'] Specify object name being compared, internally used to show appropriate assertion message

6.15.3 Exceptions and warnings

`errors.DtypeWarning`	Warning raised when reading different dtypes in a column from a file.
`errors.EmptyDataError`	Exception that is thrown in *pd.read_csv* (by both the C and Python engines) when empty data or header is encountered.
`errors.OutOfBoundsDatetime`	
`errors.ParserError`	Exception that is raised by an error encountered in *pd.read_csv*.
`errors.ParserWarning`	Warning raised when reading a file that doesn't use the default 'c' parser.

Continued on next page

Table 445 – continued from previous page

errors.PerformanceWarning	Warning raised when there is a possible performance impact.
errors.UnsortedIndexError	Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted.
errors.UnsupportedFunctionCall	Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g.

pandas.errors.DtypeWarning

exception pandas.errors.**DtypeWarning**

Warning raised when reading different dtypes in a column from a file.

Raised for a dtype incompatibility. This can happen whenever *read_csv* or *read_table* encounter non-uniform dtypes in a column(s) of a given CSV file.

See also:

pandas.read_csv Read CSV (comma-separated) file into a DataFrame.

pandas.read_table Read general delimited file into a DataFrame.

Notes

This warning is issued when dealing with larger files because the dtype checking happens per chunk read.

Despite the warning, the CSV file is read with mixed types in a single column which will be an object type. See the examples below to better understand this issue.

Examples

This example creates and reads a large CSV file with a column that contains *int* and *str*.

```
>>> df = pd.DataFrame({'a': (['1'] * 100000 + ['X'] * 100000 +
...                          ['1'] * 100000),
...                    'b': ['b'] * 300000})
>>> df.to_csv('test.csv', index=False)
>>> df2 = pd.read_csv('test.csv')
... # DtypeWarning: Columns (0) have mixed types
```

Important to notice that `df2` will contain both *str* and *int* for the same input, '1'.

```
>>> df2.iloc[262140, 0]
'1'
>>> type(df2.iloc[262140, 0])
<class 'str'>
>>> df2.iloc[262150, 0]
1
>>> type(df2.iloc[262150, 0])
<class 'int'>
```

One way to solve this issue is using the *dtype* parameter in the *read_csv* and *read_table* functions to explicit the conversion:

```
>>> df2 = pd.read_csv('test.csv', sep=',', dtype={'a': str})
```

No warning was issued.

```
>>> import os
>>> os.remove('test.csv')
```

pandas.errors.EmptyDataError

exception pandas.errors.**EmptyDataError**

Exception that is thrown in *pd.read_csv* (by both the C and Python engines) when empty data or header is encountered.

pandas.errors.OutOfBoundsDatetime

exception pandas.errors.**OutOfBoundsDatetime**

pandas.errors.ParserError

exception pandas.errors.**ParserError**

Exception that is raised by an error encountered in *pd.read_csv*.

pandas.errors.ParserWarning

exception pandas.errors.**ParserWarning**

Warning raised when reading a file that doesn't use the default 'c' parser.

Raised by *pd.read_csv* and *pd.read_table* when it is necessary to change parsers, generally from the default 'c' parser to 'python'.

It happens due to a lack of support or functionality for parsing a particular attribute of a CSV file with the requested engine.

Currently, 'c' unsupported options include the following parameters:

1. *sep* other than a single character (e.g. regex separators)
2. *skipfooter* higher than 0
3. *sep=None* with *delim_whitespace=False*

The warning can be avoided by adding *engine='python'* as a parameter in *pd.read_csv* and *pd.read_table* methods.

See also:

pd.read_csv Read CSV (comma-separated) file into DataFrame.

pd.read_table Read general delimited file into DataFrame.

Examples

Using a *sep* in *pd.read_csv* other than a single character:

```
>>> import io
>>> csv = u'''a;b;c
...           1;1,8
...           1;2,1'''
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]')  # doctest: +SKIP
... # ParserWarning: Falling back to the 'python' engine...
```

Adding *engine='python'* to *pd.read_csv* removes the Warning:

```
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]', engine='python')
```

pandas.errors.PerformanceWarning

exception pandas.errors.**PerformanceWarning**

Warning raised when there is a possible performance impact.

pandas.errors.UnsortedIndexError

exception pandas.errors.**UnsortedIndexError**

Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted. Subclass of *KeyError*.

New in version 0.20.0.

pandas.errors.UnsupportedFunctionCall

exception pandas.errors.**UnsupportedFunctionCall**

Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g. `np.cumsum(groupby_object)`.

6.15.4 Data types related functionality

api.types.union_categoricals(to_union[, ...])	Combine list-like of Categorical-like, unioning categories.
api.types.infer_dtype	Efficiently infer the type of a passed val, or list-like array of values.
api.types.pandas_dtype(dtype)	Converts input into a pandas only dtype object or a numpy dtype object.

pandas.api.types.union_categoricals

pandas.api.types.**union_categoricals**(*to_union*, *sort_categories=False*, *ignore_order=False*)

Combine list-like of Categorical-like, unioning categories. All categories must have the same dtype.

New in version 0.19.0.

Parameters

to_union [list-like of Categorical, CategoricalIndex,] or Series with dtype='category'

sort_categories [boolean, default False] If true, resulting categories will be lexsorted, otherwise they will be ordered as they appear in the data.

ignore_order [boolean, default False] If true, the ordered attribute of the Categoricals will be ignored. Results in an unordered categorical.

New in version 0.20.0.

Returns

result [Categorical]

Raises

TypeError

- all inputs do not have the same dtype
- all inputs do not have the same ordered property
- all inputs are ordered and their categories are not identical
- sort_categories=True and Categoricals are ordered

ValueError Empty list of categoricals passed

Notes

To learn more about categories, see link

Examples

```
>>> from pandas.api.types import union_categoricals
```

If you want to combine categoricals that do not necessarily have the same categories, *union_categoricals* will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
>>> a = pd.Categorical(["b", "c"])
>>> b = pd.Categorical(["a", "b"])
>>> union_categoricals([a, b])
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the *categories* of the data. If you want the categories to be lexsorted, use *sort_categories=True* argument.

```
>>> union_categoricals([a, b], sort_categories=True)
[b, c, a, b]
Categories (3, object): [a, b, c]
```

union_categoricals also works with the case of combining two categoricals of the same categories and order information (e.g. what you could also *append* for).

```
>>> a = pd.Categorical(["a", "b"], ordered=True)
>>> b = pd.Categorical(["a", "b", "a"], ordered=True)
>>> union_categoricals([a, b])
[a, b, a, b, a]
Categories (2, object): [a < b]
```

Raises *TypeError* because the categories are ordered and not identical.

```
>>> a = pd.Categorical(["a", "b"], ordered=True)
>>> b = pd.Categorical(["a", "b", "c"], ordered=True)
>>> union_categoricals([a, b])
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0

Ordered categoricals with different categories or orderings can be combined by using the *ignore_ordered=True* argument.

```
>>> a = pd.Categorical(["a", "b", "c"], ordered=True)
>>> b = pd.Categorical(["c", "b", "a"], ordered=True)
>>> union_categoricals([a, b], ignore_order=True)
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

union_categoricals also works with a *CategoricalIndex*, or *Series* containing categorical data, but note that the resulting array will always be a plain *Categorical*

```
>>> a = pd.Series(["b", "c"], dtype='category')
>>> b = pd.Series(["a", "b"], dtype='category')
>>> union_categoricals([a, b])
[b, c, a, b]
Categories (3, object): [b, c, a]
```

pandas.api.types.infer_dtype

`pandas.api.types.infer_dtype()`

Efficiently infer the type of a passed val, or list-like array of values. Return a string describing the type.

Parameters

value [scalar, list, ndarray, or pandas type]

skipna [bool, default False] Ignore NaN values when inferring the type.

New in version 0.21.0.

Returns

string describing the common type of the input data.

Results can include:

- string

- unicode

- bytes

- floating

- integer

- mixed-integer

- mixed-integer-float

- decimal

- complex

- categorical

- boolean

- datetime64

- datetime

- date

- timedelta64

- timedelta

- time

- period

- mixed

Raises

TypeError if ndarray-like but cannot infer the dtype

Notes

- 'mixed' is the catchall for anything that is not otherwise specialized
- 'mixed-integer-float' are floats and integers
- 'mixed-integer' are integers mixed with non-integers

Examples

```
>>> infer_dtype(['foo', 'bar'])
'string'
```

```
>>> infer_dtype(['a', np.nan, 'b'], skipna=True)
'string'
```

```
>>> infer_dtype(['a', np.nan, 'b'], skipna=False)
'mixed'
```

```
>>> infer_dtype([b'foo', b'bar'])
'bytes'
```

```
>>> infer_dtype([1, 2, 3])
'integer'
```

```
>>> infer_dtype([1, 2, 3.5])
'mixed-integer-float'
```

```
>>> infer_dtype([1.0, 2.0, 3.5])
'floating'
```

```
>>> infer_dtype(['a', 1])
'mixed-integer'
```

```
>>> infer_dtype([Decimal(1), Decimal(2.0)])
'decimal'
```

```
>>> infer_dtype([True, False])
'boolean'
```

```
>>> infer_dtype([True, False, np.nan])
'mixed'
```

```
>>> infer_dtype([pd.Timestamp('20130101')])
'datetime'
```

```
>>> infer_dtype([datetime.date(2013, 1, 1)])
'date'
```

```
>>> infer_dtype([np.datetime64('2013-01-01')])
'datetime64'
```

```
>>> infer_dtype([datetime.timedelta(0, 1, 1)])
'timedelta'
```

```
>>> infer_dtype(pd.Series(list('aabc')).astype('category'))
'categorical'
```

pandas.api.types.pandas_dtype

pandas.api.types.**pandas_dtype**(*dtype*)

Converts input into a pandas only dtype object or a numpy dtype object.

Parameters

dtype [object to be converted]

Returns

np.dtype or a pandas dtype

Raises

TypeError if not a dtype

Dtype introspection

api.types.is_bool_dtype(arr_or_dtype)	Check whether the provided array or dtype is of a boolean dtype.
api.types.is_categorical_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the Categorical dtype.
api.types.is_complex_dtype(arr_or_dtype)	Check whether the provided array or dtype is of a complex dtype.
api.types.is_datetime64_any_dtype(arr_or_dtype)	Check whether the provided array or dtype is of the datetime64 dtype.

Continued on next page

Table 447 – continued from previous page

api.types.is_datetime64_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the datetime64 dtype.
api.types.is_datetime64_ns_dtype(arr_or_dtype)	Check whether the provided array or dtype is of the datetime64[ns] dtype.
api.types.is_datetime64tz_dtype(arr_or_dtype)	Check whether an array-like or dtype is of a DatetimeTZDtype dtype.
api.types.is_extension_type(arr)	Check whether an array-like is of a pandas extension class instance.
api.types.is_extension_array_dtype(arr_or_dtype)	Check if an object is a pandas extension array type.
api.types.is_float_dtype(arr_or_dtype)	Check whether the provided array or dtype is of a float dtype.
api.types.is_int64_dtype(arr_or_dtype)	Check whether the provided array or dtype is of the int64 dtype.
api.types.is_integer_dtype(arr_or_dtype)	Check whether the provided array or dtype is of an integer dtype.
api.types.is_interval_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the Interval dtype.
api.types.is_numeric_dtype(arr_or_dtype)	Check whether the provided array or dtype is of a numeric dtype.
api.types.is_object_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the object dtype.
api.types.is_period_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the Period dtype.
api.types.is_signed_integer_dtype(arr_or_dtype)	Check whether the provided array or dtype is of a signed integer dtype.
api.types.is_string_dtype(arr_or_dtype)	Check whether the provided array or dtype is of the string dtype.
api.types.is_timedelta64_dtype(arr_or_dtype)	Check whether an array-like or dtype is of the timedelta64 dtype.
api.types.is_timedelta64_ns_dtype(arr_or_dtype)	Check whether the provided array or dtype is of the timedelta64[ns] dtype.
api.types.is_unsigned_integer_dtype(arr_or_dtype)	Check whether the provided array or dtype is of an unsigned integer dtype.
api.types.is_sparse(arr)	Check whether an array-like is a 1-D pandas sparse array.

pandas.api.types.is_bool_dtype

pandas.api.types.**is_bool_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of a boolean dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of a boolean dtype.]

Notes

An ExtensionArray is considered boolean when the `_is_boolean` attribute is set to True.

Examples

```
>>> is_bool_dtype(str)
False
>>> is_bool_dtype(int)
False
>>> is_bool_dtype(bool)
True
>>> is_bool_dtype(np.bool)
True
>>> is_bool_dtype(np.array(['a', 'b']))
False
>>> is_bool_dtype(pd.Series([1, 2]))
False
>>> is_bool_dtype(np.array([True, False]))
True
>>> is_bool_dtype(pd.Categorical([True, False]))
True
>>> is_bool_dtype(pd.SparseArray([True, False]))
True
```

pandas.api.types.is_categorical_dtype

pandas.api.types.**is_categorical_dtype**(*arr_or_dtype*)

Check whether an array-like or dtype is of the Categorical dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is] of the Categorical dtype.

Examples

```
>>> is_categorical_dtype(object)
False
>>> is_categorical_dtype(CategoricalDtype())
True
>>> is_categorical_dtype([1, 2, 3])
False
>>> is_categorical_dtype(pd.Categorical([1, 2, 3]))
True
>>> is_categorical_dtype(pd.CategoricalIndex([1, 2, 3]))
True
```

pandas.api.types.is_complex_dtype

pandas.api.types.**is_complex_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of a complex dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of a compex dtype.]

Examples

```
>>> is_complex_dtype(str)
False
>>> is_complex_dtype(int)
False
>>> is_complex_dtype(np.complex)
True
>>> is_complex_dtype(np.array(['a', 'b']))
False
>>> is_complex_dtype(pd.Series([1, 2]))
False
>>> is_complex_dtype(np.array([1 + 1j, 5]))
True
```

pandas.api.types.is_datetime64_any_dtype

`pandas.api.types.is_datetime64_any_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of the datetime64 dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of the datetime64 dtype.]

Examples

```
>>> is_datetime64_any_dtype(str)
False
>>> is_datetime64_any_dtype(int)
False
>>> is_datetime64_any_dtype(np.datetime64)  # can be tz-naive
True
>>> is_datetime64_any_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_any_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_any_dtype(np.array([1, 2]))
False
>>> is_datetime64_any_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_any_dtype(pd.DatetimeIndex([1, 2, 3],
                                dtype=np.datetime64))
True
```

pandas.api.types.is_datetime64_dtype

pandas.api.types.**is_datetime64_dtype**(*arr_or_dtype*)
Check whether an array-like or dtype is of the datetime64 dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is of] the datetime64 dtype.

Examples

```
>>> is_datetime64_dtype(object)
False
>>> is_datetime64_dtype(np.datetime64)
True
>>> is_datetime64_dtype(np.array([], dtype=int))
False
>>> is_datetime64_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_dtype([1, 2, 3])
False
```

pandas.api.types.is_datetime64_ns_dtype

pandas.api.types.**is_datetime64_ns_dtype**(*arr_or_dtype*)
Check whether the provided array or dtype is of the datetime64[ns] dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of the datetime64[ns] dtype.]

Examples

```
>>> is_datetime64_ns_dtype(str)
False
>>> is_datetime64_ns_dtype(int)
False
>>> is_datetime64_ns_dtype(np.datetime64)  # no unit
False
>>> is_datetime64_ns_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_ns_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_ns_dtype(np.array([1, 2]))
False
>>> is_datetime64_ns_dtype(np.array([], dtype=np.datetime64))  # no unit
False
```

(continues on next page)

(continued from previous page)

```
>>> is_datetime64_ns_dtype(np.array([],
                           dtype="datetime64[ps]"))  # wrong unit
False
>>> is_datetime64_ns_dtype(pd.DatetimeIndex([1, 2, 3],
                           dtype=np.datetime64))  # has 'ns' unit
True
```

pandas.api.types.is_datetime64tz_dtype

pandas.api.types.**is_datetime64tz_dtype**(*arr_or_dtype*)

Check whether an array-like or dtype is of a DatetimeTZDtype dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is of] a DatetimeTZDtype dtype.

Examples

```
>>> is_datetime64tz_dtype(object)
False
>>> is_datetime64tz_dtype([1, 2, 3])
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3]))  # tz-naive
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

```
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_datetime64tz_dtype(dtype)
True
>>> is_datetime64tz_dtype(s)
True
```

pandas.api.types.is_extension_type

pandas.api.types.**is_extension_type**(*arr*)

Check whether an array-like is of a pandas extension class instance.

Extension classes include categoricals, pandas sparse objects (i.e. classes represented within the pandas library and not ones external to it like scipy sparse matrices), and datetime-like arrays.

Parameters

arr [array-like] The array-like to check.

Returns

boolean [Whether or not the array-like is of a pandas] extension class instance.

Examples

```
>>> is_extension_type([1, 2, 3])
False
>>> is_extension_type(np.array([1, 2, 3]))
False
>>>
>>> cat = pd.Categorical([1, 2, 3])
>>>
>>> is_extension_type(cat)
True
>>> is_extension_type(pd.Series(cat))
True
>>> is_extension_type(pd.SparseArray([1, 2, 3]))
True
>>> is_extension_type(pd.SparseSeries([1, 2, 3]))
True
>>>
>>> from scipy.sparse import bsr_matrix
>>> is_extension_type(bsr_matrix([1, 2, 3]))
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
>>>
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_extension_type(s)
True
```

pandas.api.types.is_extension_array_dtype

pandas.api.types.**is_extension_array_dtype**(*arr_or_dtype*)

Check if an object is a pandas extension array type.

See the *Use Guide* for more.

Parameters

arr_or_dtype [object] For array-like input, the `.dtype` attribute will be extracted.

Returns

bool Whether the *arr_or_dtype* is an extension array type.

Notes

This checks whether an object implements the pandas extension array interface. In pandas, this includes:

- Categorical
- Sparse
- Interval
- Period
- DatetimeArray

- TimedeltaArray

Third-party libraries may implement arrays or types satisfying this interface as well.

Examples

```
>>> from pandas.api.types import is_extension_array_dtype
>>> arr = pd.Categorical(['a', 'b'])
>>> is_extension_array_dtype(arr)
True
>>> is_extension_array_dtype(arr.dtype)
True
```

```
>>> arr = np.array(['a', 'b'])
>>> is_extension_array_dtype(arr.dtype)
False
```

pandas.api.types.is_float_dtype

pandas.api.types.**is_float_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of a float dtype.

This function is internal and should not be exposed in the public API.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of a float dtype.]

Examples

```
>>> is_float_dtype(str)
False
>>> is_float_dtype(int)
False
>>> is_float_dtype(float)
True
>>> is_float_dtype(np.array(['a', 'b']))
False
>>> is_float_dtype(pd.Series([1, 2]))
False
>>> is_float_dtype(pd.Index([1, 2.]))
True
```

pandas.api.types.is_int64_dtype

pandas.api.types.**is_int64_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of the int64 dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of the int64 dtype.]

Notes

Depending on system architecture, the return value of *is_int64_dtype(int)* will be True if the OS uses 64-bit integers and False if the OS uses 32-bit integers.

Examples

```
>>> is_int64_dtype(str)
False
>>> is_int64_dtype(np.int32)
False
>>> is_int64_dtype(np.int64)
True
>>> is_int64_dtype('int8')
False
>>> is_int64_dtype('Int8')
False
>>> is_int64_dtype(pd.Int64Dtype)
True
>>> is_int64_dtype(float)
False
>>> is_int64_dtype(np.uint64)  # unsigned
False
>>> is_int64_dtype(np.array(['a', 'b']))
False
>>> is_int64_dtype(np.array([1, 2], dtype=np.int64))
True
>>> is_int64_dtype(pd.Index([1, 2.]))  # float
False
>>> is_int64_dtype(np.array([1, 2], dtype=np.uint32))  # unsigned
False
```

pandas.api.types.is_integer_dtype

pandas.api.types.**is_integer_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of an integer dtype.

Unlike in *in_any_int_dtype*, timedelta64 instances will return False.

Changed in version 0.24.0: The nullable Integer dtypes (e.g. pandas.Int64Dtype) are also considered as integer by this function.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of an integer dtype] and not an instance of timedelta64.

Examples

```
>>> is_integer_dtype(str)
False
>>> is_integer_dtype(int)
True
>>> is_integer_dtype(float)
False
>>> is_integer_dtype(np.uint64)
True
>>> is_integer_dtype('int8')
True
>>> is_integer_dtype('Int8')
True
>>> is_integer_dtype(pd.Int8Dtype)
True
>>> is_integer_dtype(np.datetime64)
False
>>> is_integer_dtype(np.timedelta64)
False
>>> is_integer_dtype(np.array(['a', 'b']))
False
>>> is_integer_dtype(pd.Series([1, 2]))
True
>>> is_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_integer_dtype(pd.Index([1, 2.]))  # float
False
```

pandas.api.types.is_interval_dtype

`pandas.api.types.is_interval_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the Interval dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is] of the Interval dtype.

Examples

```
>>> is_interval_dtype(object)
False
>>> is_interval_dtype(IntervalDtype())
True
>>> is_interval_dtype([1, 2, 3])
False
>>>
>>> interval = pd.Interval(1, 2, closed="right")
>>> is_interval_dtype(interval)
False
>>> is_interval_dtype(pd.IntervalIndex([interval]))
True
```

pandas.api.types.is_numeric_dtype

pandas.api.types.**is_numeric_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of a numeric dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of a numeric dtype.]

Examples

```
>>> is_numeric_dtype(str)
False
>>> is_numeric_dtype(int)
True
>>> is_numeric_dtype(float)
True
>>> is_numeric_dtype(np.uint64)
True
>>> is_numeric_dtype(np.datetime64)
False
>>> is_numeric_dtype(np.timedelta64)
False
>>> is_numeric_dtype(np.array(['a', 'b']))
False
>>> is_numeric_dtype(pd.Series([1, 2]))
True
>>> is_numeric_dtype(pd.Index([1, 2.]))
True
>>> is_numeric_dtype(np.array([], dtype=np.timedelta64))
False
```

pandas.api.types.is_object_dtype

pandas.api.types.**is_object_dtype**(*arr_or_dtype*)

Check whether an array-like or dtype is of the object dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is of the object dtype.]

Examples

```
>>> is_object_dtype(object)
True
>>> is_object_dtype(int)
False
```

(continues on next page)

(continued from previous page)

```
>>> is_object_dtype(np.array([], dtype=object))
True
>>> is_object_dtype(np.array([], dtype=int))
False
>>> is_object_dtype([1, 2, 3])
False
```

pandas.api.types.is_period_dtype

pandas.api.types.**is_period_dtype**(*arr_or_dtype*)

Check whether an array-like or dtype is of the Period dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is of the Period dtype.]

Examples

```
>>> is_period_dtype(object)
False
>>> is_period_dtype(PeriodDtype(freq="D"))
True
>>> is_period_dtype([1, 2, 3])
False
>>> is_period_dtype(pd.Period("2017-01-01"))
False
>>> is_period_dtype(pd.PeriodIndex([], freq="A"))
True
```

pandas.api.types.is_signed_integer_dtype

pandas.api.types.**is_signed_integer_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of a signed integer dtype.

Unlike in *in_any_int_dtype*, timedelta64 instances will return False.

Changed in version 0.24.0: The nullable Integer dtypes (e.g. pandas.Int64Dtype) are also considered as integer by this function.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of a signed integer dtype] and not an instance of timedelta64.

Examples

```
>>> is_signed_integer_dtype(str)
False
>>> is_signed_integer_dtype(int)
True
>>> is_signed_integer_dtype(float)
False
>>> is_signed_integer_dtype(np.uint64)  # unsigned
False
>>> is_signed_integer_dtype('int8')
True
>>> is_signed_integer_dtype('Int8')
True
>>> is_signed_dtype(pd.Int8Dtype)
True
>>> is_signed_integer_dtype(np.datetime64)
False
>>> is_signed_integer_dtype(np.timedelta64)
False
>>> is_signed_integer_dtype(np.array(['a', 'b']))
False
>>> is_signed_integer_dtype(pd.Series([1, 2]))
True
>>> is_signed_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_signed_integer_dtype(pd.Index([1, 2.]))  # float
False
>>> is_signed_integer_dtype(np.array([1, 2], dtype=np.uint32))  # unsigned
False
```

pandas.api.types.is_string_dtype

pandas.api.types.**is_string_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of the string dtype.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of the string dtype.]

Examples

```
>>> is_string_dtype(str)
True
>>> is_string_dtype(object)
True
>>> is_string_dtype(int)
False
>>>
>>> is_string_dtype(np.array(['a', 'b']))
True
```

(continues on next page)

(continued from previous page)

```
>>> is_string_dtype(pd.Series([1, 2]))
False
```

pandas.api.types.is_timedelta64_dtype

pandas.api.types.**is_timedelta64_dtype**(*arr_or_dtype*)

Check whether an array-like or dtype is of the timedelta64 dtype.

Parameters

arr_or_dtype [array-like] The array-like or dtype to check.

Returns

boolean [Whether or not the array-like or dtype is] of the timedelta64 dtype.

Examples

```
>>> is_timedelta64_dtype(object)
False
>>> is_timedelta64_dtype(np.timedelta64)
True
>>> is_timedelta64_dtype([1, 2, 3])
False
>>> is_timedelta64_dtype(pd.Series([], dtype="timedelta64[ns]"))
True
>>> is_timedelta64_dtype('0 days')
False
```

pandas.api.types.is_timedelta64_ns_dtype

pandas.api.types.**is_timedelta64_ns_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of the timedelta64[ns] dtype.

This is a very specific dtype, so generic ones like *np.timedelta64* will return False if passed into this function.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of the] timedelta64[ns] dtype.

Examples

```
>>> is_timedelta64_ns_dtype(np.dtype('m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.dtype('m8[ps]'))  # Wrong frequency
False
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype='m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype=np.timedelta64))
False
```

pandas.api.types.is_unsigned_integer_dtype

pandas.api.types.**is_unsigned_integer_dtype**(*arr_or_dtype*)

Check whether the provided array or dtype is of an unsigned integer dtype.

Changed in version 0.24.0: The nullable Integer dtypes (e.g. pandas.UInt64Dtype) are also considered as integer by this function.

Parameters

arr_or_dtype [array-like] The array or dtype to check.

Returns

boolean [Whether or not the array or dtype is of an] unsigned integer dtype.

Examples

```
>>> is_unsigned_integer_dtype(str)
False
>>> is_unsigned_integer_dtype(int)  # signed
False
>>> is_unsigned_integer_dtype(float)
False
>>> is_unsigned_integer_dtype(np.uint64)
True
>>> is_unsigned_integer_dtype('uint8')
True
>>> is_unsigned_integer_dtype('UInt8')
True
>>> is_unsigned_integer_dtype(pd.UInt8Dtype)
True
>>> is_unsigned_integer_dtype(np.array(['a', 'b']))
False
>>> is_unsigned_integer_dtype(pd.Series([1, 2]))  # signed
False
>>> is_unsigned_integer_dtype(pd.Index([1, 2.]))  # float
False
>>> is_unsigned_integer_dtype(np.array([1, 2], dtype=np.uint32))
True
```

pandas.api.types.is_sparse

pandas.api.types.**is_sparse**(*arr*)

Check whether an array-like is a 1-D pandas sparse array.

Check that the one-dimensional array-like is a pandas sparse array. Returns True if it is a pandas sparse array, not another type of sparse array.

Parameters

arr [array-like] Array-like to check.

Returns

bool Whether or not the array-like is a pandas sparse array.

See also:

DataFrame.to_sparse Convert DataFrame to a SparseDataFrame.

Series.to_sparse Convert Series to SparseSeries.

Series.to_dense Return dense representation of a Series.

Examples

Returns *True* if the parameter is a 1-D pandas sparse array.

```
>>> is_sparse(pd.SparseArray([0, 0, 1, 0]))
True
>>> is_sparse(pd.SparseSeries([0, 0, 1, 0]))
True
```

Returns *False* if the parameter is not sparse.

```
>>> is_sparse(np.array([0, 0, 1, 0]))
False
>>> is_sparse(pd.Series([0, 1, 0, 0]))
False
```

Returns *False* if the parameter is not a pandas sparse array.

```
>>> from scipy.sparse import bsr_matrix
>>> is_sparse(bsr_matrix([0, 1, 0, 0]))
False
```

Returns *False* if the parameter has more than one dimension.

```
>>> df = pd.SparseDataFrame([389., 24., 80.5, np.nan],
                            columns=['max_speed'],
                            index=['falcon', 'parrot', 'lion', 'monkey'])
>>> is_sparse(df)
False
>>> is_sparse(df.max_speed)
True
```

Iterable introspection

api.types.is_dict_like(obj)	Check if the object is dict-like.
api.types.is_file_like(obj)	Check if the object is a file-like object.
api.types.is_list_like(obj[, allow_sets])	Check if the object is list-like.
api.types.is_named_tuple(obj)	Check if the object is a named tuple.
api.types.is_iterator(obj)	Check if the object is an iterator.

pandas.api.types.is_dict_like

pandas.api.types.**is_dict_like**(*obj*)

Check if the object is dict-like.

Parameters

obj [The object to check]

Returns

is_dict_like [bool] Whether *obj* has dict-like properties.

Examples

```
>>> is_dict_like({1: 2})
True
>>> is_dict_like([1, 2, 3])
False
```

pandas.api.types.is_file_like

`pandas.api.types.is_file_like(obj)`

Check if the object is a file-like object.

For objects to be considered file-like, they must be an iterator AND have either a *read* and/or *write* method as an attribute.

Note: file-like objects must be iterable, but iterable objects need not be file-like.

New in version 0.20.0.

Parameters

obj [The object to check]

Returns

is_file_like [bool] Whether *obj* has file-like properties.

Examples

```
>>> buffer(StringIO("data"))
>>> is_file_like(buffer)
True
>>> is_file_like([1, 2, 3])
False
```

pandas.api.types.is_list_like

`pandas.api.types.is_list_like(obj, allow_sets=True)`

Check if the object is list-like.

Objects that are considered list-like are for example Python lists, tuples, sets, NumPy arrays, and Pandas Series.

Strings and datetime objects, however, are not considered list-like.

Parameters

obj [The object to check]

allow_sets [boolean, default True] If this parameter is False, sets will not be considered list-like

New in version 0.24.0.

Returns

is_list_like [bool] Whether *obj* has list-like properties.

Examples

```
>>> is_list_like([1, 2, 3])
True
>>> is_list_like({1, 2, 3})
True
>>> is_list_like(datetime(2017, 1, 1))
False
>>> is_list_like("foo")
False
>>> is_list_like(1)
False
>>> is_list_like(np.array([2]))
True
>>> is_list_like(np.array(2)))
False
```

pandas.api.types.is_named_tuple

pandas.api.types.**is_named_tuple**(*obj*)
Check if the object is a named tuple.

Parameters

obj [The object to check]

Returns

is_named_tuple [bool] Whether *obj* is a named tuple.

Examples

```
>>> Point = namedtuple("Point", ["x", "y"])
>>> p = Point(1, 2)
>>>
>>> is_named_tuple(p)
True
>>> is_named_tuple((1, 2))
False
```

pandas.api.types.is_iterator

pandas.api.types.**is_iterator**(*obj*)
Check if the object is an iterator.

For example, lists are considered iterators but not strings or datetime objects.

Parameters

obj [The object to check]

Returns

is_iter [bool] Whether *obj* is an iterator.

Examples

```
>>> is_iterator([1, 2, 3])
True
>>> is_iterator(datetime(2017, 1, 1))
False
>>> is_iterator("foo")
False
>>> is_iterator(1)
False
```

Scalar introspection

api.types.is_bool	
api.types.is_categorical(arr)	Check whether an array-like is a Categorical instance.
api.types.is_complex	
api.types.is_datetimetz(arr)	(DEPRECATED) Check whether an array-like is a datetime array-like with a timezone component in its dtype.
api.types.is_float	
api.types.is_hashable(obj)	Return True if hash(obj) will succeed, False otherwise.
api.types.is_integer	
api.types.is_interval	
api.types.is_number(obj)	Check if the object is a number.
api.types.is_period(arr)	(DEPRECATED) Check whether an array-like is a periodical index.
api.types.is_re(obj)	Check if the object is a regex pattern instance.
api.types.is_re_compilable(obj)	Check if the object can be compiled into a regex pattern instance.
api.types.is_scalar	Return True if given value is scalar.

pandas.api.types.is_bool

pandas.api.types.**is_bool**()

pandas.api.types.is_categorical

pandas.api.types.**is_categorical**(*arr*)

Check whether an array-like is a Categorical instance.

Parameters

arr [array-like] The array-like to check.

Returns

boolean [Whether or not the array-like is of a Categorical instance.]

Examples

```
>>> is_categorical([1, 2, 3])
False
```

Categoricals, Series Categoricals, and CategoricalIndex will return True.

```
>>> cat = pd.Categorical([1, 2, 3])
>>> is_categorical(cat)
True
>>> is_categorical(pd.Series(cat))
True
>>> is_categorical(pd.CategoricalIndex([1, 2, 3]))
True
```

pandas.api.types.is_complex

pandas.api.types.**is_complex**()

pandas.api.types.is_datetimetz

pandas.api.types.**is_datetimetz**(*arr*)

Check whether an array-like is a datetime array-like with a timezone component in its dtype.

Deprecated since version 0.24.0.

Parameters

arr [array-like] The array-like to check.

Returns

boolean [Whether or not the array-like is a datetime array-like with] a timezone component in its dtype.

Examples

```
>>> is_datetimetz([1, 2, 3])
False
```

Although the following examples are both DatetimeIndex objects, the first one returns False because it has no timezone component unlike the second one, which returns True.

```
>>> is_datetimetz(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_datetimetz(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

The object need not be a DatetimeIndex object. It just needs to have a dtype which has a timezone component.

```
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_datetimetz(s)
True
```

pandas.api.types.is_float

pandas.api.types.**is_float**()

pandas.api.types.is_hashable

pandas.api.types.**is_hashable**(*obj*)

Return True if hash(obj) will succeed, False otherwise.

Some types will pass a test against collections.Hashable but fail when they are actually hashed with hash().

Distinguish between these and other types by trying the call to hash() and seeing if they raise TypeError.

Examples

```
>>> a = ([],)
>>> isinstance(a, collections.Hashable)
True
>>> is_hashable(a)
False
```

pandas.api.types.is_integer

pandas.api.types.**is_integer**()

pandas.api.types.is_interval

pandas.api.types.**is_interval**()

pandas.api.types.is_number

pandas.api.types.**is_number**(*obj*)

Check if the object is a number.

Returns True when the object is a number, and False if is not.

Parameters

obj [any type] The object to check if is a number.

Returns

is_number [bool] Whether *obj* is a number or not.

See also:

pandas.api.types.is_integer Checks a subgroup of numbers.

Examples

```
>>> pd.api.types.is_number(1)
True
>>> pd.api.types.is_number(7.15)
True
```

Booleans are valid because they are int subclass.

```
>>> pd.api.types.is_number(False)
True
```

```
>>> pd.api.types.is_number("foo")
False
>>> pd.api.types.is_number("5")
False
```

pandas.api.types.is_period

pandas.api.types.**is_period**(*arr*)

Check whether an array-like is a periodical index.

Deprecated since version 0.24.0.

Parameters

arr [array-like] The array-like to check.

Returns

boolean [Whether or not the array-like is a periodical index.]

Examples

```
>>> is_period([1, 2, 3])
False
>>> is_period(pd.Index([1, 2, 3]))
False
>>> is_period(pd.PeriodIndex(["2017-01-01"], freq="D"))
True
```

pandas.api.types.is_re

pandas.api.types.**is_re**(*obj*)

Check if the object is a regex pattern instance.

Parameters

obj [The object to check]

Returns

is_regex [bool] Whether *obj* is a regex pattern.

Examples

```
>>> is_re(re.compile(".*"))
True
>>> is_re("foo")
False
```

pandas.api.types.is_re_compilable

`pandas.api.types.is_re_compilable(obj)`
Check if the object can be compiled into a regex pattern instance.

Parameters

obj [The object to check]

Returns

is_regex_compilable [bool] Whether *obj* can be compiled as a regex pattern.

Examples

```
>>> is_re_compilable(".*")
True
>>> is_re_compilable(1)
False
```

pandas.api.types.is_scalar

`pandas.api.types.is_scalar()`
Return True if given value is scalar.

Parameters

val [object] This includes:

- numpy array scalar (e.g. np.int64)
- Python builtin numerics
- Python builtin byte arrays and strings
- None
- datetime.datetime
- datetime.timedelta
- Period
- decimal.Decimal
- Interval
- DateOffset
- Fraction
- Number

Returns

bool Return True if given object is scalar, False otherwise

Examples

```
>>> dt = pd.datetime.datetime(2018, 10, 3)
>>> pd.is_scalar(dt)
True
```

```
>>> pd.api.types.is_scalar([2, 3])
False
```

```
>>> pd.api.types.is_scalar({0: 1, 2: 3})
False
```

```
>>> pd.api.types.is_scalar((0, 2))
False
```

pandas supports PEP 3141 numbers:

```
>>> from fractions import Fraction
>>> pd.api.types.is_scalar(Fraction(3, 5))
True
```

6.16 Extensions

These are primarily intended for library authors looking to extend pandas objects.

api.extensions.register_extension_dtype(cls)	Class decorator to register an ExtensionType with pandas.
api.extensions.register_dataframe_accessor(name)	Register a custom accessor on DataFrame objects.
api.extensions.register_series_accessor(name)	Register a custom accessor on Series objects.
api.extensions.register_index_accessor(name)	Register a custom accessor on Index objects.
api.extensions.ExtensionDtype	A custom data type, to be paired with an ExtensionArray.
api.extensions.ExtensionArray	Abstract base class for custom 1-D array types.
arrays.PandasArray(values[, copy])	A pandas ExtensionArray for NumPy data.

6.16.1 pandas.api.extensions.register_extension_dtype

pandas.api.extensions.**register_extension_dtype**(*cls*)

Class decorator to register an ExtensionType with pandas.

New in version 0.24.0.

This enables operations like `.astype(name)` for the name of the ExtensionDtype.

Examples

```
>>> from pandas.api.extensions import register_extension_dtype
>>> from pandas.api.extensions import ExtensionDtype
>>> @register_extension_dtype
... class MyExtensionDtype(ExtensionDtype):
...     pass
```

6.16.2 pandas.api.extensions.register_dataframe_accessor

pandas.api.extensions.**register_dataframe_accessor**(*name*)

Register a custom accessor on DataFrame objects.

Parameters

name [str] Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

See also:

register_series_accessor, *register_index_accessor*

Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):  # noqa: E999
    ...
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
```

(continues on next page)

(continued from previous page)

```
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Back in an interactive IPython session:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

6.16.3 pandas.api.extensions.register_series_accessor

pandas.api.extensions.**register_series_accessor**(*name*)

Register a custom accessor on Series objects.

Parameters

name [str] Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

See also:

register_dataframe_accessor, *register_index_accessor*

Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):  # noqa: E999
    ...
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Back in an interactive IPython session:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

6.16.4 pandas.api.extensions.register_index_accessor

pandas.api.extensions.**register_index_accessor**(*name*)

Register a custom accessor on Index objects.

Parameters

name [str] Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

See also:

register_dataframe_accessor, *register_series_accessor*

Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):  # noqa: E999
    ...
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Back in an interactive IPython session:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

6.16.5 pandas.api.extensions.ExtensionDtype

class pandas.api.extensions.**ExtensionDtype**

A custom data type, to be paired with an ExtensionArray.

New in version 0.23.0.

See also:

pandas.api.extensions.register_extension_dtype, *pandas.api.extensions.ExtensionArray*

Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- type
- name
- construct_from_string

The following attributes influence the behavior of the dtype in pandas operations

- _is_numeric
- _is_boolean

Optionally one can override construct_array_type for construction with the name of this dtype via the Registry. See *pandas.api.extensions.register_extension_dtype()*.

- construct_array_type

The *na_value* class attribute can be used to set the default NA value for this type. `numpy.nan` is used by default.

ExtensionDtypes are required to be hashable. The base class provides a default implementation, which relies on the `_metadata` class attribute. `_metadata` should be a tuple containing the strings that define your data type. For example, with `PeriodDtype` that's the `freq` attribute.

If you have a parametrized dtype you should set the ``_metadata`` class property.

Ideally, the attributes in `_metadata` will match the parameters to your `ExtensionDtype.__init__` (if any). If any of the attributes in `_metadata` don't implement the standard `__eq__` or `__hash__`, the default implementations here will not work.

Changed in version 0.24.0: Added `_metadata`, `__hash__`, and changed the default definition of `__eq__`.

This class does not inherit from 'abc.ABCMeta' for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

Attributes

kind	A character code (one of 'biufcmMOSUV'), default 'O'
name	A string identifying the data type.
names	Ordered list of field names, or None if there are no fields.
type	The scalar type for the array, e.g.

pandas.api.extensions.ExtensionDtype.kind

`ExtensionDtype.`**`kind`**

A character code (one of 'biufcmMOSUV'), default 'O'

This should match the NumPy dtype used when the array is converted to an ndarray, which is probably 'O' for object if the extension type cannot be represented as a built-in NumPy type.

See also:

`numpy.dtype.kind`

pandas.api.extensions.ExtensionDtype.name

`ExtensionDtype.`**`name`**

A string identifying the data type.

Will be used for display in, e.g. `Series.dtype`

pandas.api.extensions.ExtensionDtype.names

`ExtensionDtype.`**`names`**

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.api.extensions.ExtensionDtype.type

`ExtensionDtype.type`
The scalar type for the array, e.g. `int`

It's expected `ExtensionArray[item]` returns an instance of `ExtensionDtype.type` for scalar `item`, assuming that value is valid (not NA). NA values do not need to be instances of *type*.

Methods

construct_array_type()	Return the array type associated with this dtype
construct_from_string(string)	Attempt to construct this type from a string.
is_dtype(dtype)	Check if we match 'dtype'.

pandas.api.extensions.ExtensionDtype.construct_array_type

classmethod `ExtensionDtype.`**construct_array_type**()
Return the array type associated with this dtype

Returns

type

pandas.api.extensions.ExtensionDtype.construct_from_string

classmethod `ExtensionDtype.`**construct_from_string**(*string*)
Attempt to construct this type from a string.

Parameters

string [str]

Returns

self [instance of 'cls']

Raises

TypeError If a class cannot be constructed from this 'string'.

Examples

If the extension dtype can be constructed without any arguments, the following may be an adequate implementation.

```
>>> @classmethod
... def construct_from_string(cls, string)
...     if string == cls.name:
...         return cls()
...     else:
...         raise TypeError("Cannot construct a '{}' from "
...                         "'{}'".format(cls, string))
```

pandas.api.extensions.ExtensionDtype.is_dtype

classmethod `ExtensionDtype.is_dtype(dtype)`

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

6.16.6 pandas.api.extensions.ExtensionArray

class `pandas.api.extensions.ExtensionArray`

Abstract base class for custom 1-D array types.

pandas will recognize instances of this class as proper arrays with a custom type and will not attempt to coerce them to objects. They may be stored directly inside a `DataFrame` or `Series`.

New in version 0.23.0.

Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- _from_sequence
- _from_factorized
- __getitem__
- __len__
- dtype
- nbytes
- isna
- take
- copy
- _concat_same_type

A default repr displaying the type, (truncated) data, length, and dtype is provided. It can be customized or replaced by by overriding:

- __repr__ : A default repr for the ExtensionArray.
- _formatter : Print scalars inside a Series or DataFrame.

Some methods require casting the ExtensionArray to an ndarray of Python objects with `self.astype(object)`, which may be expensive. When performance is a concern, we highly recommend overriding the following methods:

- fillna
- dropna
- unique
- factorize / _values_for_factorize
- argsort / _values_for_argsort
- searchsorted

The remaining methods implemented on this class should be performant, as they only compose abstract methods. Still, a more efficient implementation may be available, and these methods can be overridden.

One can implement methods to handle array reductions.

- _reduce

One can implement methods to handle parsing from strings that will be used in methods such as `pandas.io.parsers.read_csv`.

- _from_sequence_of_strings

This class does not inherit from 'abc.ABCMeta' for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

ExtensionArrays are limited to 1 dimension.

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 address may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists. Pandas makes no assumptions on how the data are stored, just that it can be converted to a NumPy array. The ExtensionArray interface does not impose any rules on how this data is stored. However, currently, the backing data cannot be stored in attributes called `.values` or `._values` to ensure full compatibility with pandas internals. But other names as `.data`, `._data`, `._items`, `...` can be freely used.

Attributes

dtype	An instance of 'ExtensionDtype'.
nbytes	The number of bytes needed to store this object in memory.
ndim	Extension Arrays are only allowed to be 1-dimensional.
shape	Return a tuple of the array dimensions.

pandas.api.extensions.ExtensionArray.dtype

`ExtensionArray.`**`dtype`**

An instance of 'ExtensionDtype'.

pandas.api.extensions.ExtensionArray.nbytes

`ExtensionArray.nbytes`
The number of bytes needed to store this object in memory.

pandas.api.extensions.ExtensionArray.ndim

`ExtensionArray.ndim`
Extension Arrays are only allowed to be 1-dimensional.

pandas.api.extensions.ExtensionArray.shape

`ExtensionArray.shape`
Return a tuple of the array dimensions.

Methods

argsort([ascending, kind])	Return the indices that would sort this array.
astype(dtype[, copy])	Cast to a NumPy array with 'dtype'.
copy([deep])	Return a copy of the array.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill NA/NaN values using the specified method.
isna()	A 1-D array indicating if each value is missing.
repeat(repeats[, axis])	Repeat elements of a ExtensionArray.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
take(indices[, allow_fill, fill_value])	Take elements from an array.
unique()	Compute the ExtensionArray of unique values.

pandas.api.extensions.ExtensionArray.argsort

`ExtensionArray.argsort`(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)
Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

`numpy.argsort` Sorting implementation used internally.

pandas.api.extensions.ExtensionArray.astype

`ExtensionArray.astype(dtype, copy=True)`
Cast to a NumPy array with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray] NumPy ndarray with 'dtype' for its dtype.

pandas.api.extensions.ExtensionArray.copy

`ExtensionArray.copy(deep=False)`
Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray.dropna

`ExtensionArray.dropna()`
Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.api.extensions.ExtensionArray.factorize

`ExtensionArray.factorize(na_sentinel=-1)`
Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

Note: uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a *sort* keyword as well.

pandas.api.extensions.ExtensionArray.fillna

ExtensionArray.**fillna**(*value=None*, *method=None*, *limit=None*)

Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

filled [ExtensionArray with NA/NaN filled]

pandas.api.extensions.ExtensionArray.isna

ExtensionArray.**isna**()

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.api.extensions.ExtensionArray.repeat

ExtensionArray.**repeat**(*repeats*, *axis=None*)

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_array [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

Series.repeat Equivalent function for Series.

Index.repeat Equivalent function for Index.

numpy.repeat Similar method for `numpy.ndarray`.

ExtensionArray.take Take arbitrary positions.

Examples

```
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

pandas.api.extensions.ExtensionArray.searchsorted

ExtensionArray.**searchsorted**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array *self* (a) such that, if the corresponding elements in *v* were inserted before the indices, the order of *self* would be preserved.

Assuming that *a* is sorted:

side	returned index *i* satisfies
left	`self[i-1] < v <= self[i]`
right	`self[i-1] <= v < self[i]`

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

See also:

`numpy.searchsorted` Similar method from NumPy.

pandas.api.extensions.ExtensionArray.shift

`ExtensionArray.shift(periods=1, fill_value=None)`

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.api.extensions.ExtensionArray.take

`ExtensionArray.take(indices, allow_fill=False, fill_value=None)`

Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.api.extensions.ExtensionArray.unique

`ExtensionArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

6.16.7 pandas.arrays.PandasArray

`class pandas.arrays.PandasArray(values, copy=False)`

A pandas ExtensionArray for NumPy data.

New in version 0.24.0.

This is mostly for internal compatibility, and is not especially useful on its own.

Parameters

values [ndarray] The NumPy ndarray to wrap. Must be 1-dimensional.

copy [bool, default False] Whether to copy *values*.

Notes

Operations like + and applying ufuncs requires NumPy>=1.13.

Attributes

dtype	An instance of 'ExtensionDtype'.
nbytes	The number of bytes needed to store this object in memory.
ndim	Extension Arrays are only allowed to be 1-dimensional.
shape	Return a tuple of the array dimensions.

pandas.arrays.PandasArray.dtype

`PandasArray.dtype`

An instance of 'ExtensionDtype'.

pandas.arrays.PandasArray.nbytes

`PandasArray.nbytes`

The number of bytes needed to store this object in memory.

pandas.arrays.PandasArray.ndim

`PandasArray.ndim`

Extension Arrays are only allowed to be 1-dimensional.

pandas.arrays.PandasArray.shape

PandasArray.**shape**
Return a tuple of the array dimensions.

Methods

argsort([ascending, kind])	Return the indices that would sort this array.
astype(dtype[, copy])	Cast to a NumPy array with 'dtype'.
copy([deep])	Return a copy of the array.
dropna()	Return ExtensionArray without NA values
factorize([na_sentinel])	Encode the extension array as an enumerated type.
fillna([value, method, limit])	Fill NA/NaN values using the specified method.
isna()	A 1-D array indicating if each value is missing.
repeat(repeats[, axis])	Repeat elements of a ExtensionArray.
searchsorted(value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift([periods, fill_value])	Shift values by desired number.
take(indices[, allow_fill, fill_value])	Take elements from an array.
to_numpy([dtype, copy])	Convert the PandasArray to a `numpy.ndarray`.
unique()	Compute the ExtensionArray of unique values.

pandas.arrays.PandasArray.argsort

PandasArray.**argsort**(*ascending=True*, *kind='quicksort'*, **args*, ***kwargs*)
Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

index_array [ndarray] Array of indices that sort `self`.

See also:

numpy.argsort Sorting implementation used internally.

pandas.arrays.PandasArray.astype

PandasArray.**astype**(*dtype*, *copy=True*)
Cast to a NumPy array with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray] NumPy ndarray with 'dtype' for its dtype.

pandas.arrays.PandasArray.copy

`PandasArray.copy(deep=False)`

Return a copy of the array.

Parameters

deep [bool, default False] Also copy the underlying data backing this array.

Returns

ExtensionArray

pandas.arrays.PandasArray.dropna

`PandasArray.dropna()`

Return ExtensionArray without NA values

Returns

valid [ExtensionArray]

pandas.arrays.PandasArray.factorize

`PandasArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *labels* array to indicate missing values.

Returns

labels [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

> **Note:** uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

pandas.factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a *sort* keyword as well.

pandas.arrays.PandasArray.fillna

`PandasArray.fillna(value=None, method=None, limit=None)`

Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

filled [ExtensionArray with NA/NaN filled]

pandas.arrays.PandasArray.isna

`PandasArray.isna()`

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement `ExtensionArray._reduce()`
- `na_values.any` and `na_values.all` should be implemented

pandas.arrays.PandasArray.repeat

`PandasArray.repeat(repeats, axis=None)`

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_array [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

`Series.repeat` Equivalent function for Series.

`Index.repeat` Equivalent function for Index.

`numpy.repeat` Similar method for `numpy.ndarray`.

`ExtensionArray.take` Take arbitrary positions.

Examples

```
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

pandas.arrays.PandasArray.searchsorted

`PandasArray.`**`searchsorted`**(*value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array *self* (a) such that, if the corresponding elements in *v* were inserted before the indices, the order of *self* would be preserved.

Assuming that *a* is sorted:

side	returned index *i* satisfies
left	`self[i-1] < v <= self[i]`
right	`self[i-1] <= v < self[i]`

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

Returns

indices [array of ints] Array of insertion points with the same shape as *value*.

See also:

`numpy.searchsorted` Similar method from NumPy.

pandas.arrays.PandasArray.shift

`PandasArray.shift`(*periods=1*, *fill_value=None*)
Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`

New in version 0.24.0.

Returns

shifted [ExtensionArray]

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.arrays.PandasArray.take

`PandasArray.take`(*indices*, *allow_fill=False*, *fill_value=None*)
Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.arrays.PandasArray.to_numpy

`PandasArray.`**`to_numpy`**(*dtype=None*, *copy=False*)

Convert the PandasArray to a `numpy.ndarray`.

By default, this requires no coercion or copying of data.

Parameters

dtype [numpy.dtype] The NumPy dtype to pass to `numpy.asarray()`.

copy [bool, default False] Whether to copy the underlying data.

Returns

ndarray

pandas.arrays.PandasArray.unique

`PandasArray.unique()`

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

all	
any	
kurt	
max	
mean	
median	
min	
prod	
sem	
skew	
std	
sum	
var	

6.17 pandas.Index.asi8

`Index.asi8 = None`

6.18 pandas.Index.holds_integer

`Index.holds_integer()`

6.19 pandas.Index.is_type_compatible

`Index.is_type_compatible(kind)`

6.20 pandas.Index.nlevels

`Index.nlevels`

6.21 pandas.Index.sort

`Index.sort(*args, **kwargs)`

6.22 pandas.Panel.agg

`Panel.agg(func, *args, **kwargs)`

6.23 pandas.Panel.aggregate

`Panel.aggregate(func, *args, **kwargs)`

6.24 pandas.api.extensions.ExtensionDtype.na_value

`ExtensionDtype.na_value = nan`

CHAPTER

SEVEN

DEVELOPMENT

7.1 Contributing to pandas

Table of contents:

7.1.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are brand new to pandas or open-source development, we recommend going through the GitHub "issues" tab to find issues that interest you. There are a number of issues listed under Docs and good first issue where you could start out. Once you've found an interesting issue, you can return here to get your development environment setup.

Feel free to ask questions on the mailing list or on Gitter.

7.1.2 Bug reports and enhancement requests

Bug reports are an important part of making *pandas* more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing. See this stackoverflow article and this blogpost for tips on writing a good bug report.

Trying the bug-producing code out on the *master* branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self-contained Python snippet reproducing the problem. You can format the code nicely by using GitHub Flavored Markdown:

````
```python
>>> from pandas import DataFrame
>>> df = DataFrame(...)
...
```
````

2. Include the full version string of *pandas* and its dependencies. You can use the built in function:

```
>>> import pandas as pd
>>> pd.show_versions()
```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the *pandas* community and be open to comments/ideas from others.

7.1.3 Working with the code

Now that you have an issue you want to fix, enhancement to add, or documentation to improve, you need to learn how to work with GitHub and the *pandas* code base.

Version control, Git, and GitHub

To the new user, working with Git is one of the more daunting aspects of contributing to *pandas*. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- the GitHub help pages.
- the NumPy's documentation.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the pandas project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/pandas.git pandas-yourname
cd pandas-yourname
git remote add upstream https://github.com/pandas-dev/pandas.git
```

This creates the directory *pandas-yourname* and connects your repository to the upstream (main project) *pandas* repository.

Creating a development environment

To test out code changes, you'll need to build pandas from source, which requires a C compiler and Python environment. If you're making documentation changes, you can skip to *Contributing to the documentation* but you won't be able to build the documentation locally before pushing your changes.

Installing a C Compiler

Pandas uses C extensions (mostly written using Cython) to speed up certain operations. To install pandas from source, you need to compile these C extensions, which means you need a C compiler. This process depends on which platform you're using. Follow the CPython contributing guide for getting a compiler installed. You don't need to do any of the `./configure` or `make` steps; you only need to install the compiler.

For Windows developers, when using Python 3.5 and later, it is sufficient to install Visual Studio 2017 with the **Python development workload** and the **Python native development tools** option. Otherwise, the following links may be helpful.

- https://blogs.msdn.microsoft.com/pythonengineering/2017/03/07/python-support-in-vs2017/
- https://blogs.msdn.microsoft.com/pythonengineering/2016/04/11/unable-to-find-vcvarsall-bat/
- https://github.com/conda/conda-recipes/wiki/Building-from-Source-on-Windows-32-bit-and-64-bit
- https://cowboyprogrammer.org/building-python-wheels-for-windows/
- https://blog.ionelmc.ro/2014/12/21/compiling-python-extensions-on-windows/
- https://support.enthought.com/hc/en-us/articles/204469260-Building-Python-extensions-with-Canopy

Let us know if you have any difficulties by opening an issue or reaching out on Gitter.

Creating a Python Environment

Now that you have a C compiler, create an isolated pandas development environment:

- Install either Anaconda or miniconda
- Make sure your conda is up to date (`conda update conda`)
- Make sure that you have *cloned the repository*
- `cd` to the *pandas* source directory

We'll now kick off a three-step process:

1. Install the build dependencies
2. Build and install pandas
3. Install the optional dependencies

```
# Create and activate the build environment
conda env create -f environment.yml
conda activate pandas-dev

# or with older versions of Anaconda:
source activate pandas-dev

# Build and install pandas
python setup.py build_ext --inplace -j 4
python -m pip install -e .
```

At this point you should be able to import pandas from your locally built version:

```
$ python  # start an interpreter
>>> import pandas
>>> print(pandas.__version__)
0.22.0.dev0+29.g4ad6d4d74
```

This will create the new environment, and not touch any of your existing environments, nor any existing Python installation.

To view your environments:

```
conda info -e
```

To return to your root environment:

```
conda deactivate
```

See the full conda docs here.

Creating a Python Environment (pip)

If you aren't using conda for you development environment, follow these instructions. You'll need to have at least python3.5 installed on your system.

```
# Create a virtual environment
# Use an ENV_DIR of your choice. We'll use ~/virtualenvs/pandas-dev
# Any parent directories should already exist
python3 -m venv ~/virtualenvs/pandas-dev
# Activate the virtulaenv
. ~/virtualenvs/pandas-dev/bin/activate

# Install the build dependencies
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext --inplace -j 4
python -m pip install -e .
```

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *pandas*. You can have many shiny-new-features and switch in between them using the git checkout command.

When creating this branch, make sure your master branch is up to date with the latest upstream master version. To update your local master branch, you can do:

```
git checkout master
git pull upstream master --ff-only
```

When you want to update the feature branch with changes in master after you created the branch, check the section on *updating a PR*.

7.1.4 Contributing to the documentation

Contributing to the documentation benefits everyone who uses *pandas*. We encourage you to help us improve the documentation, and you don't have to be an expert on *pandas* to do so! In fact, there are sections of the docs that are worse off after being written by experts. If something in the docs doesn't make sense to you, updating the relevant section after you figure it out is a great way to ensure it will help the next person.

Documentation:

About the *pandas* documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The *pandas* documentation consists of two parts: the docstrings in the code itself and the docs in this folder `pandas/doc/`.

  The docstrings provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what's new, installation, etc).

- The docstrings follow a pandas convention, based on the **Numpy Docstring Standard**. Follow the *pandas docstring guide* for detailed instructions on how to write a correct docstring.

pandas docstring guide

About docstrings and standards

A Python docstring is a string used to document a Python module, class, function or method, so programmers can understand what it does without having to read the details of the implementation.

Also, it is a common practice to generate online (html) documentation automatically from docstrings. Sphinx serves this purpose.

Next example gives an idea on how a docstring looks like:

```
def add(num1, num2):
"""
Add up two integer numbers.

This function simply wraps the `+` operator, and does not
do anything interesting, except for illustrating what is
the docstring of a very simple function.

Parameters
----------
num1 : int
    First number to add
num2 : int
    Second number to add

Returns
-------
int
    The sum of `num1` and `num2`

See Also
--------
subtract : Subtract one integer from another

Examples
--------
>>> add(2, 2)
4
>>> add(25, 0)
25
>>> add(10, -10)
0
"""
return num1 + num2
```

Some standards exist about docstrings, so they are easier to read, and they can be exported to other formats such as html or pdf.

The first conventions every Python docstring should follow are defined in PEP-257.

As PEP-257 is quite open, and some other standards exist on top of it. In the case of pandas, the numpy docstring convention is followed. The conventions is explained in this document:

– numpydoc docstring guide (which is based in the original Guide to NumPy/SciPy documentation)

numpydoc is a Sphinx extension to support the numpy docstring convention.

The standard uses reStructuredText (reST). reStructuredText is a markup language that allows encoding styles in plain text files. Documentation about reStructuredText can be found in:

– Sphinx reStructuredText primer

– Quick reStructuredText reference

– Full reStructuredText specification

Pandas has some helpers for sharing docstrings between related classes, see *Sharing Docstrings*.

The rest of this document will summarize all the above guides, and will provide additional convention specific to the pandas project.

Writing a docstring

General rules

Docstrings must be defined with three double-quotes. No blank lines should be left before or after the docstring. The text starts in the next line after the opening quotes. The closing quotes have their own line (meaning that they are not at the end of the last sentence).

In rare occasions reST styles like bold text or italics will be used in docstrings, but is it common to have inline code, which is presented between backticks. It is considered inline code:

- The name of a parameter
- Python code, a module, function, built-in, type, literal... (e.g. `os`, `list`, `numpy.abs`, `datetime.date`, `True`)
- A pandas class (in the form ``:class:`pandas.Series` ``)
- A pandas method (in the form ``:meth:`pandas.Series.sum` ``)
- A pandas function (in the form ``:func:`pandas.to_datetime` ``)

Note: To display only the last component of the linked class, method or function, prefix it with ~. For example, ``:class:`~pandas.Series` `` will link to `pandas.Series` but only display the last part, `Series` as the link text. See Sphinx cross-referencing syntax for details.

Good:

```
def add_values(arr):
    """
    Add the values in `arr`.

    This is equivalent to Python `sum` of :meth:`pandas.Series.sum`.

    Some sections are omitted here for simplicity.
    """
    return sum(arr)
```

Bad:

```
def func():

    """Some function.

    With several mistakes in the docstring.

    It has a blank like after the signature `def func():`.

    The text 'Some function' should go in the line after the
    opening quotes of the docstring, not in the same line.

    There is a blank line between the docstring and the first line
    of code `foo = 1`.
```

(continues on next page)

(continued from previous page)

```
    The closing quotes should be in the next line, not in this one."""

    foo = 1
    bar = 2
    return foo + bar
```

Section 1: Short summary

The short summary is a single sentence that expresses what the function does in a concise way.

The short summary must start with a capital letter, end with a dot, and fit in a single line. It needs to express what the object does without providing details. For functions and methods, the short summary must start with an infinitive verb.

Good:

```
def astype(dtype):
    """
    Cast Series type.

    This section will provide further details.
    """
    pass
```

Bad:

```
def astype(dtype):
    """
    Casts Series type.

    Verb in third-person of the present simple, should be infinitive.
    """
    pass
```

```
def astype(dtype):
    """
    Method to cast Series type.

    Does not start with verb.
    """
    pass
```

```
def astype(dtype):
    """
    Cast Series type

    Missing dot at the end.
    """
    pass
```

```
def astype(dtype):
    """
    Cast Series type from its current type to the new type defined in
```

(continues on next page)

(continued from previous page)

```
    the parameter dtype.

    Summary is too verbose and doesn't fit in a single line.
    """
    pass
```

Section 2: Extended summary

The extended summary provides details on what the function does. It should not go into the details of the parameters, or discuss implementation notes, which go in other sections.

A blank line is left between the short summary and the extended summary. And every paragraph in the extended summary is finished by a dot.

The extended summary should provide details on why the function is useful and their use cases, if it is not too generic.

```
def unstack():
    """
    Pivot a row index to columns.

    When using a MultiIndex, a level can be pivoted so each value in
    the index becomes a column. This is especially useful when a subindex
    is repeated for the main index, and data is easier to visualize as a
    pivot table.

    The index level will be automatically removed from the index when added
    as columns.
    """
    pass
```

Section 3: Parameters

The details of the parameters will be added in this section. This section has the title "Parameters", followed by a line with a hyphen under each letter of the word "Parameters". A blank line is left before the section title, but not after, and not between the line with the word "Parameters" and the one with the hyphens.

After the title, each parameter in the signature must be documented, including **args* and ***kwargs*, but not *self*.

The parameters are defined by their name, followed by a space, a colon, another space, and the type (or types). Note that the space between the name and the colon is important. Types are not defined for **args* and ***kwargs*, but must be defined for all other parameters. After the parameter definition, it is required to have a line with the parameter description, which is indented, and can have multiple lines. The description must start with a capital letter, and finish with a dot.

For keyword arguments with a default value, the default will be listed after a comma at the end of the type. The exact form of the type in this case will be "int, default 0". In some cases it may be useful to explain what the default argument means, which can be added after a comma "int, default -1, meaning all cpus".

In cases where the default value is *None*, meaning that the value will not be used. Instead of "str, default None", it is preferred to write "str, optional". When *None* is a value being used, we will keep the form "str, default None". For example, in *df.to_csv(compression=None)*, *None* is not a value being used, but means that compression is optional, and no compression is being used if not provided. In this case we will use *str, optional*.

Only in cases like *func(value=None)* and *None* is being used in the same way as *0* or *foo* would be used, then we will specify "str, int or None, default None".

Good:

```
class Series:
    def plot(self, kind, color='blue', **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the
        specified kind.

        Parameters
        ----------
        kind : str
            Kind of matplotlib plot.
        color : str, default 'blue'
            Color name or rgb code.
        **kwargs
            These parameters will be passed to the matplotlib plotting
            function.
        """
        pass
```

Bad:

```
class Series:
    def plot(self, kind, **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the
        specified kind.

        Note the blank line between the parameters title and the first
        parameter. Also, note that after the name of the parameter `kind`
        and before the colon, a space is missing.

        Also, note that the parameter descriptions do not start with a
        capital letter, and do not finish with a dot.

        Finally, the `**kwargs` parameter is missing.

        Parameters
        ----------

        kind: str
            kind of matplotlib plot
        """
        pass
```

Parameter types

When specifying the parameter types, Python built-in data types can be used directly (the Python type is preferred to the more verbose string, integer, boolean, etc):

- int

- float
- str
- bool

For complex types, define the subtypes. For *dict* and *tuple*, as more than one type is present, we use the brackets to help read the type (curly brackets for *dict* and normal brackets for *tuple*):

- list of int
- dict of {str : int}
- tuple of (str, int, int)
- tuple of (str,)
- set of str

In case where there are just a set of values allowed, list them in curly brackets and separated by commas (followed by a space). If the values are ordinal and they have an order, list them in this order. Otherwise, list the default value first, if there is one:

- {0, 10, 25}
- {'simple', 'advanced'}
- {'low', 'medium', 'high'}
- {'cat', 'dog', 'bird'}

If the type is defined in a Python module, the module must be specified:

- datetime.date
- datetime.datetime
- decimal.Decimal

If the type is in a package, the module must be also specified:

- numpy.ndarray
- scipy.sparse.coo_matrix

If the type is a pandas type, also specify pandas except for Series and DataFrame:

- Series
- DataFrame
- pandas.Index
- pandas.Categorical
- pandas.SparseArray

If the exact type is not relevant, but must be compatible with a numpy array, array-like can be specified. If Any type that can be iterated is accepted, iterable can be used:

- array-like
- iterable

If more than one type is accepted, separate them by commas, except the last two types, that need to be separated by the word 'or':

- int or float
- float, decimal.Decimal or None

- str or list of str

If None is one of the accepted values, it always needs to be the last in the list.

For axis, the convention is to use something like:

- axis : {0 or 'index', 1 or 'columns', None}, default None

Section 4: Returns or Yields

If the method returns a value, it will be documented in this section. Also if the method yields its output.

The title of the section will be defined in the same way as the "Parameters". With the names "Returns" or "Yields" followed by a line with as many hyphens as the letters in the preceding word.

The documentation of the return is also similar to the parameters. But in this case, no name will be provided, unless the method returns or yields more than one value (a tuple of values).

The types for "Returns" and "Yields" are the same as the ones for the "Parameters". Also, the description must finish with a dot.

For example, with a single value:

```
def sample():
    """
    Generate and return a random number.

    The value is sampled from a continuous uniform distribution between
    0 and 1.

    Returns
    -------
    float
        Random number generated.
    """
    return np.random.random()
```

With more than one value:

```
import string

def random_letters():
    """
    Generate and return a sequence of random letters.

    The length of the returned string is also random, and is also
    returned.

    Returns
    -------
    length : int
        Length of the returned string.
    letters : str
        String of random letters.
    """
    length = np.random.randint(1, 10)
    letters = ''.join(np.random.choice(string.ascii_lowercase)
                      for i in range(length))
    return length, letters
```

If the method yields its value:

```
def sample_values():
    """
    Generate an infinite sequence of random numbers.

    The values are sampled from a continuous uniform distribution between
    0 and 1.

    Yields
    ------
    float
        Random number generated.
    """
    while True:
        yield np.random.random()
```

Section 5: See Also

This section is used to let users know about pandas functionality related to the one being documented. In rare cases, if no related methods or functions can be found at all, this section can be skipped.

An obvious example would be the *head()* and *tail()* methods. As *tail()* does the equivalent as *head()* but at the end of the *Series* or *DataFrame* instead of at the beginning, it is good to let the users know about it.

To give an intuition on what can be considered related, here there are some examples:

- `loc` and `iloc`, as they do the same, but in one case providing indices and in the other positions
- `max` and `min`, as they do the opposite
- `iterrows`, `itertuples` and `iteritems`, as it is easy that a user looking for the method to iterate over columns ends up in the method to iterate over rows, and vice-versa
- `fillna` and `dropna`, as both methods are used to handle missing values
- `read_csv` and `to_csv`, as they are complementary
- `merge` and `join`, as one is a generalization of the other
- `astype` and `pandas.to_datetime`, as users may be reading the documentation of `astype` to know how to cast as a date, and the way to do it is with `pandas.to_datetime`
- `where` is related to `numpy.where`, as its functionality is based on it

When deciding what is related, you should mainly use your common sense and think about what can be useful for the users reading the documentation, especially the less experienced ones.

When relating to other libraries (mainly `numpy`), use the name of the module first (not an alias like `np`). If the function is in a module which is not the main one, like `scipy.sparse`, list the full module (e.g. `scipy.sparse.coo_matrix`).

This section, as the previous, also has a header, "See Also" (note the capital S and A). Also followed by the line with hyphens, and preceded by a blank line.

After the header, we will add a line for each related method or function, followed by a space, a colon, another space, and a short description that illustrated what this method or function does, why is it relevant in this context, and what are the key differences between the documented function and the one referencing. The description must also finish with a dot.

Note that in "Returns" and "Yields", the description is located in the following line than the type. But in this section it is located in the same line, with a colon in between. If the description does not fit in the same line, it can continue in the next ones, but it has to be indented in them.

For example:

```
class Series:
    def head(self):
        """
        Return the first 5 elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Returns
        -------
        Series
            Subset of the original series with the 5 first values.

        See Also
        --------
        Series.tail : Return the last 5 elements of the Series.
        Series.iloc : Return a slice of the elements in the Series,
            which can also be used to return the first or last n.
        """
        return self.iloc[:5]
```

Section 6: Notes

This is an optional section used for notes about the implementation of the algorithm. Or to document technical aspects of the function behavior.

Feel free to skip it, unless you are familiar with the implementation of the algorithm, or you discover some counter-intuitive behavior while writing the examples for the function.

This section follows the same format as the extended summary section.

Section 7: Examples

This is one of the most important sections of a docstring, even if it is placed in the last position. As often, people understand concepts better with examples, than with accurate explanations.

Examples in docstrings, besides illustrating the usage of the function or method, must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users.

They are presented as a session in the Python terminal. `>>>` is used to present code. `...` is used for code continuing from the previous line. Output is presented immediately after the last line of code generating the output (no blank lines in between). Comments describing the examples can be added with blank lines before and after them.

The way to present examples is as follows:

1. Import required libraries (except `numpy` and `pandas`)
2. Create the data required for the example
3. Show a very basic example that gives an idea of the most common use case

4. Add examples with explanations that illustrate how the parameters can be used for extended functionality

A simple example could be:

```python
class Series:

    def head(self, n=5):
        """
        Return the first elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Parameters
        ----------
        n : int
            Number of values to return.

        Return
        ------
        pandas.Series
            Subset of the original series with the n first values.

        See Also
        --------
        tail : Return the last n elements of the Series.

        Examples
        --------
        >>> s = pd.Series(['Ant', 'Bear', 'Cow', 'Dog', 'Falcon',
        ...                'Lion', 'Monkey', 'Rabbit', 'Zebra'])
        >>> s.head()
        0   Ant
        1   Bear
        2   Cow
        3   Dog
        4   Falcon
        dtype: object

        With the `n` parameter, we can change the number of returned rows:

        >>> s.head(n=3)
        0   Ant
        1   Bear
        2   Cow
        dtype: object
        """
        return self.iloc[:n]
```

The examples should be as concise as possible. In cases where the complexity of the function requires long examples, is recommended to use blocks with headers in bold. Use double star `**` to make a text bold, like in `**this example**`.

Conventions for the examples

Code in examples is assumed to always start with these two lines which are not shown:

```
import numpy as np
import pandas as pd
```

Any other module used in the examples must be explicitly imported, one per line (as recommended in **PEP 8#imports**) and avoiding aliases. Avoid excessive imports, but if needed, imports from the standard library go first, followed by third-party libraries (like matplotlib).

When illustrating examples with a single `Series` use the name `s`, and if illustrating with a single `DataFrame` use the name `df`. For indices, `idx` is the preferred name. If a set of homogeneous `Series` or `DataFrame` is used, name them `s1`, `s2`, `s3`... or `df1`, `df2`, `df3`... If the data is not homogeneous, and more than one structure is needed, name them with something meaningful, for example `df_main` and `df_to_join`.

Data used in the example should be as compact as possible. The number of rows is recommended to be around 4, but make it a number that makes sense for the specific example. For example in the `head` method, it requires to be higher than 5, to show the example with the default values. If doing the `mean`, we could use something like `[1, 2, 3]`, so it is easy to see that the value returned is the mean.

For more complex examples (grouping for example), avoid using data without interpretation, like a matrix of random numbers with columns A, B, C, D... And instead use a meaningful example, which makes it easier to understand the concept. Unless required by the example, use names of animals, to keep examples consistent. And numerical properties of them.

When calling the method, keywords arguments `head(n=3)` are preferred to positional arguments `head(3)`.

Good:

```
class Series:

    def mean(self):
        """
        Compute the mean of the input.

        Examples
        --------
        >>> s = pd.Series([1, 2, 3])
        >>> s.mean()
        2
        """
        pass


    def fillna(self, value):
        """
        Replace missing values by `value`.

        Examples
        --------
        >>> s = pd.Series([1, np.nan, 3])
        >>> s.fillna(0)
        [1, 0, 3]
        """
        pass

    def groupby_mean(self):
        """
        Group by index and return mean.

        Examples
```

(continues on next page)

(continued from previous page)

```
        --------
        >>> s = pd.Series([380., 370., 24., 26],
        ...               name='max_speed',
        ...               index=['falcon', 'falcon', 'parrot', 'parrot'])
        >>> s.groupby_mean()
        index
        falcon    375.0
        parrot     25.0
        Name: max_speed, dtype: float64
        """
        pass

    def contains(self, pattern, case_sensitive=True, na=numpy.nan):
        """
        Return whether each value contains `pattern`.

        In this case, we are illustrating how to use sections, even
        if the example is simple enough and does not require them.

        Examples
        --------
        >>> s = pd.Series('Antelope', 'Lion', 'Zebra', np.nan)
        >>> s.contains(pattern='a')
        0    False
        1    False
        2     True
        3      NaN
        dtype: bool

        **Case sensitivity**

        With `case_sensitive` set to `False` we can match `a` with both
        `a` and `A`:

        >>> s.contains(pattern='a', case_sensitive=False)
        0     True
        1    False
        2     True
        3      NaN
        dtype: bool

        **Missing values**

        We can fill missing values in the output using the `na` parameter:

        >>> s.contains(pattern='a', na=False)
        0    False
        1    False
        2     True
        3    False
        dtype: bool
        """
        pass
```

Bad:

```
def method(foo=None, bar=None):
    """
    A sample DataFrame method.

    Do not import numpy and pandas.

    Try to use meaningful data, when it makes the example easier
    to understand.

    Try to avoid positional arguments like in `df.method(1)`. They
    can be all right if previously defined with a meaningful name,
    like in `present_value(interest_rate)`, but avoid them otherwise.

    When presenting the behavior with different parameters, do not place
    all the calls one next to the other. Instead, add a short sentence
    explaining what the example shows.

    Examples
    --------
    >>> import numpy as np
    >>> import pandas as pd
    >>> df = pd.DataFrame(np.random.randn(3, 3),
    ...                   columns=('a', 'b', 'c'))
    >>> df.method(1)
    21
    >>> df.method(bar=14)
    123
    """
    pass
```

Tips for getting your examples pass the doctests

Getting the examples pass the doctests in the validation script can sometimes be tricky. Here are some attention points:

- Import all needed libraries (except for pandas and numpy, those are already imported as `import pandas as pd` and `import numpy as np`) and define all variables you use in the example.
- Try to avoid using random data. However random data might be OK in some cases, like if the function you are documenting deals with probability distributions, or if the amount of data needed to make the function result meaningful is too much, such that creating it manually is very cumbersome. In those cases, always use a fixed random seed to make the generated examples predictable. Example:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame({'normal': np.random.normal(100, 5, 20)})
```

- If you have a code snippet that wraps multiple lines, you need to use '…' on the continued lines:

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], index=['a', 'b', 'c'],
...                   columns=['A', 'B'])
```

- If you want to show a case where an exception is raised, you can do:

```
>>> pd.to_datetime(["712-01-01"])
Traceback (most recent call last):
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 712-01-01 00:00:00
```

It is essential to include the "Traceback (most recent call last):", but for the actual error only the error name is sufficient.

– If there is a small part of the result that can vary (e.g. a hash in an object representation), you can use `...` to represent this part.

If you want to show that `s.plot()` returns a matplotlib AxesSubplot object, this will fail the doctest

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at 0x7efd0c0b0690>
```

However, you can do (notice the comment that needs to be added)

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at ...>
```

Plots in examples

There are some methods in pandas returning plots. To render the plots generated by the examples in the documentation, the `.. plot::` directive exists.

To use it, place the next code after the "Examples" header as shown below. The plot will be generated automatically when building the documentation.

```
class Series:
    def plot(self):
        """
        Generate a plot with the `Series` data.

        Examples
        --------

        .. plot::
            :context: close-figs

            >>> s = pd.Series([1, 2, 3])
            >>> s.plot()
        """
        pass
```

Sharing Docstrings

Pandas has a system for sharing docstrings, with slight variations, between classes. This helps us keep docstrings consistent, while keeping things clear for the user reading. It comes at the cost of some complexity when writing.

Each shared docstring will have a base template with variables, like `%(klass)s`. The variables filled in later on using the `Substitution` decorator. Finally, docstrings can be appended to with the `Appender` decorator.

In this example, we'll create a parent docstring normally (this is like `pandas.core.generic.NDFrame`. Then we'll have two children (like `pandas.core.series.Series` and `pandas.core.frame.DataFrame`). We'll substitute the children's class names in this docstring.

```
class Parent:
    def my_function(self):
        """Apply my function to %(klass)s."""
```

(continues on next page)

(continued from previous page)

```
        ...


class ChildA(Parent):
    @Substitution(klass="ChildA")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
        ...


class ChildB(Parent):
    @Substitution(klass="ChildB")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
        ...
```

The resulting docstrings are

```
>>> print(Parent.my_function.__doc__)
Apply my function to %(klass)s.
>>> print(ChildA.my_function.__doc__)
Apply my function to ChildA.
>>> print(ChildB.my_function.__doc__)
Apply my function to ChildB.
```

Notice two things:

1. We "append" the parent docstring to the children docstrings, which are initially empty.
2. Python decorators are applied inside out. So the order is Append then Substitution, even though Substitution comes first in the file.

Our files will often contain a module-level `_shared_doc_kwargs` with some common substitution values (things like `klass`, `axes`, etc).

You can substitute and append in one shot with something like

```
@Appender(template % _shared_doc_kwargs)
def my_function(self):
    ...
```

where `template` may come from a module-level `_shared_docs` dictionary mapping function names to docstrings. Wherever possible, we prefer using `Appender` and `Substitution`, since the docstring-writing processes is slightly closer to normal.

See `pandas.core.generic.NDFrame.fillna` for an example template, and `pandas.core.series.Series.fillna` and `pandas.core.generic.frame.fillna` for the filled versions.

- The tutorials make heavy use of the ipython directive sphinx extension. This directive lets you put code in the documentation which will be run during the doc build. For example:

```
.. ipython:: python

    x = 2
    x**3
```

will be rendered as:

```
In [1]: x = 2

In [2]: x**3
Out[2]: 8
```

Almost all code examples in the docs are run (and the output saved) during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

- Our API documentation in `doc/source/api.rst` houses the auto-generated documentation from the docstrings. For classes, there are a few subtleties around controlling which methods and attributes have pages auto-generated.

  We have two autosummary templates for classes.

  1. `_templates/autosummary/class.rst`. Use this when you want to automatically generate a page for every public method and attribute on the class. The `Attributes` and `Methods` sections will be automatically added to the class' rendered documentation by numpydoc. See `DataFrame` for an example.
  2. `_templates/autosummary/class_without_autosummary`. Use this when you want to pick a subset of methods / attributes to auto-generate pages for. When using this template, you should include an `Attributes` and `Methods` section in the class docstring. See `CategoricalIndex` for an example.

  Every method should be included in a `toctree` in `api.rst`, else Sphinx will emit a warning.

Note: The `.rst` files are used to automatically generate Markdown and HTML versions of the docs. For this reason, please do not edit `CONTRIBUTING.md` directly, but instead make any changes to `doc/source/contributing.rst`. Then, to generate `CONTRIBUTING.md`, use pandoc with the following command:

```
pandoc doc/source/contributing.rst -t markdown_github > CONTRIBUTING.md
```

The utility script `scripts/validate_docstrings.py` can be used to get a csv summary of the API documentation. And also validate common errors in the docstring of a specific class, function or method. The summary also compares the list of methods documented in `doc/source/api.rst` (which is used to generate the API Reference page) and the actual public methods. This will identify methods documented in `doc/source/api.rst` that are not actually class methods, and existing methods that are not documented in `doc/source/api.rst`.

Updating a *pandas* docstring

When improving a single function or method's docstring, it is not necessarily needed to build the full documentation (see next section). However, there is a script that checks a docstring (for example for the `DataFrame.mean` method):

```
python scripts/validate_docstrings.py pandas.DataFrame.mean
```

This script will indicate some formatting errors if present, and will also run and test the examples included in the docstring. Check the *pandas docstring guide* for a detailed guide on how to format the docstring.

The examples in the docstring ('doctests') must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users. This can be checked with the script above, and is also tested on Travis. A failing doctest will be a blocker for merging a PR. Check the *examples* section in the docstring guide for some tips and tricks to get the doctests passing.

When doing a PR with a docstring update, it is good to post the output of the validation script in a comment on github.

How to build the *pandas* documentation

Requirements

First, you need to have a development environment to be able to build pandas (see the docs on *creating a development environment above*).

Building the documentation

So how do you build the docs? Navigate to your local `pandas/doc/` directory in the console and run:

```
python make.py html
```

Then you can find the HTML output in the folder `pandas/doc/build/html/`.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```
python make.py clean
python make.py html
```

You can tell `make.py` to compile only a single section of the docs, greatly reducing the turn-around time for checking your changes.

```
# omit autosummary and API section
python make.py clean
python make.py --no-api

# compile the docs with only a single
# section, that which is in indexing.rst
python make.py clean
python make.py --single indexing

# compile the reference docs for a single function
python make.py clean
python make.py --single DataFrame.join
```

For comparison, a full documentation build may take 15 minutes, but a single section may take 15 seconds. Subsequent builds, which only process portions you have changed, will be faster.

You can also specify to use multiple cores to speed up the documentation build:

```
python make.py html --num-jobs 4
```

Open the following file in a web browser to see the full documentation you just built:

```
pandas/docs/build/html/index.html
```

And you'll have the satisfaction of seeing your new and improved documentation!

Building master branch documentation

When pull requests are merged into the *pandas* `master` branch, the main parts of the documentation are also built by Travis-CI. These docs are then hosted here, see also the *Continuous Integration* section.

7.1.5 Contributing to the code base

Code Base:

- *Code standards*
  - *C (cpplint)*
  - *Python (PEP8)*
  - *Import Formatting*
  - *Backwards Compatibility*
- *Testing With Continuous Integration*
- *Test-driven development/code writing*
  - *Writing tests*
  - *Transitioning to* `pytest`
  - *Using* `pytest`
  - *Using* `hypothesis`
  - *Testing Warnings*
- *Running the test suite*
- *Running the performance test suite*
- *Documenting your code*

Code standards

Writing good code is not just about what you write. It is also about *how* you write it. During *Continuous Integration* testing, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to *pandas*.

There is a tool in pandas to help contributors verify their changes before contributing them to the project:

```
./ci/code_checks.sh
```

The script verify the linting of code files, it looks for common mistake patterns (like missing spaces around sphinx directives that make the documentation not being rendered properly) and it also validates the doctests. It is possible to run the checks independently by using the parameters `lint`, `patterns` and `doctests` (e.g. `./ci/code_checks.sh lint`).

In addition, because a lot of people use our library, it is important that we do not make sudden changes to the code that could have the potential to break a lot of user code as a result, that is, we need it to be as *backwards compatible* as possible to avoid mass breakages.

Additional standards are outlined on the code style wiki page.

C (cpplint)

pandas uses the Google standard. Google provides an open source style checker called `cpplint`, but we use a fork of it that can be found here. Here are *some* of the more common `cpplint` issues:

- we restrict line-length to 80 characters to promote readability
- every header file must include a header guard to avoid name collisions if re-included

Continuous Integration will run the cpplint tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-
↪build/include_subdir modified-c-file
```

You can also run this command on an entire directory if necessary:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-
↪build/include_subdir --recursive modified-c-directory
```

To make your commits compliant with this standard, you can install the ClangFormat tool, which can be downloaded here. To configure, in your home directory, run the following command:

```
clang-format style=google -dump-config  > .clang-format
```

Then modify the file to ensure that any indentation width parameters are at least four. Once configured, you can run the tool as follows:

```
clang-format modified-c-file
```

This will output what your file will look like if the changes are made, and to apply them, run the following command:

```
clang-format -i modified-c-file
```

To run the tool on an entire directory, you can run the following analogous commands:

```
clang-format modified-c-directory/*.c modified-c-directory/*.h
clang-format -i modified-c-directory/*.c modified-c-directory/*.h
```

Do note that this tool is best-effort, meaning that it will try to correct as many errors as possible, but it may not correct *all* of them. Thus, it is recommended that you run `cpplint` to double check and make any other style fixes manually.

Python (PEP8)

pandas uses the PEP8 standard. There are several tools to ensure you abide by this standard. Here are *some* of the more common `PEP8` issues:

- we restrict line-length to 79 characters to promote readability
- passing arguments should have spaces after commas, e.g. `foo(arg1, arg2, kw1='bar')`

Continuous Integration will run the flake8 tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself on the diff:

```
git diff upstream/master -u -- "*.py" | flake8 --diff
```

This command will catch any stylistic errors in your changes specifically, but be beware it may not catch all of them. For example, if you delete the only usage of an imported function, it is stylistically incorrect to import an unused function. However, style-checking the diff will not catch this because the actual import is not part of the diff. Thus, for completeness, you should run this command, though it will take longer:

```
git diff upstream/master --name-only -- "*.py" | xargs -r flake8
```

Note that on OSX, the -r flag is not available, so you have to omit it and run this slightly modified command:

```
git diff upstream/master --name-only -- "*.py" | xargs flake8
```

Windows does not support the xargs command (unless installed for example via the MinGW toolchain), but one can imitate the behaviour as follows:

```
for /f %i in ('git diff upstream/master --name-only -- "*.py"') do flake8 %i
```

This will get all the files being changed by the PR (and ending with .py), and run flake8 on them, one after the other.

Import Formatting

pandas uses isort to standardise import formatting across the codebase.

A guide to import layout as per pep8 can be found here.

A summary of our current import sections (in order):

- Future
- Python Standard Library
- Third Party
- pandas._libs, pandas.compat, pandas.util._*, pandas.errors (largely not dependent on pandas.core)
- pandas.core.dtypes (largely not dependent on the rest of pandas.core)
- Rest of pandas.core.*
- Non-core pandas.io, pandas.plotting, pandas.tseries
- Local application/library specific imports

Imports are alphabetically sorted within these sections.

As part of *Continuous Integration* checks we run:

```
isort --recursive --check-only pandas
```

to check that imports are correctly formatted as per the *setup.cfg*.

If you see output like the below in *Continuous Integration* checks:

```
Check import format using isort
ERROR: /home/travis/build/pandas-dev/pandas/pandas/io/pytables.py Imports are␣
↪incorrectly sorted
Check import format using isort DONE
The command "ci/code_checks.sh" exited with 1
```

You should run:

```
isort pandas/io/pytables.py
```

to automatically format imports correctly. This will modify your local copy of the files.

The *–recursive* flag can be passed to sort all files in a directory.

You can then verify the changes look ok, then git *commit* and *push*.

Backwards Compatibility

Please try to maintain backward compatibility. *pandas* has lots of users with lots of existing code, so don't break it if at all possible. If you think breakage is required, clearly state why as part of the pull request. Also, be careful when changing method signatures and add deprecation warnings where needed. Also, add the deprecated sphinx directive to the deprecated functions or methods.

If a function with the same arguments as the one being deprecated exist, you can use the `pandas.util._decorators.deprecate`:

```
from pandas.util._decorators import deprecate

deprecate('old_func', 'new_func', '0.21.0')
```

Otherwise, you need to do it manually:

```
import warnings


def old_func():
    """Summary of the function.

    .. deprecated:: 0.21.0
       Use new_func instead.
    """
    warnings.warn('Use new_func instead.', FutureWarning, stacklevel=2)
    new_func()


def new_func():
    pass
```

You'll also need to

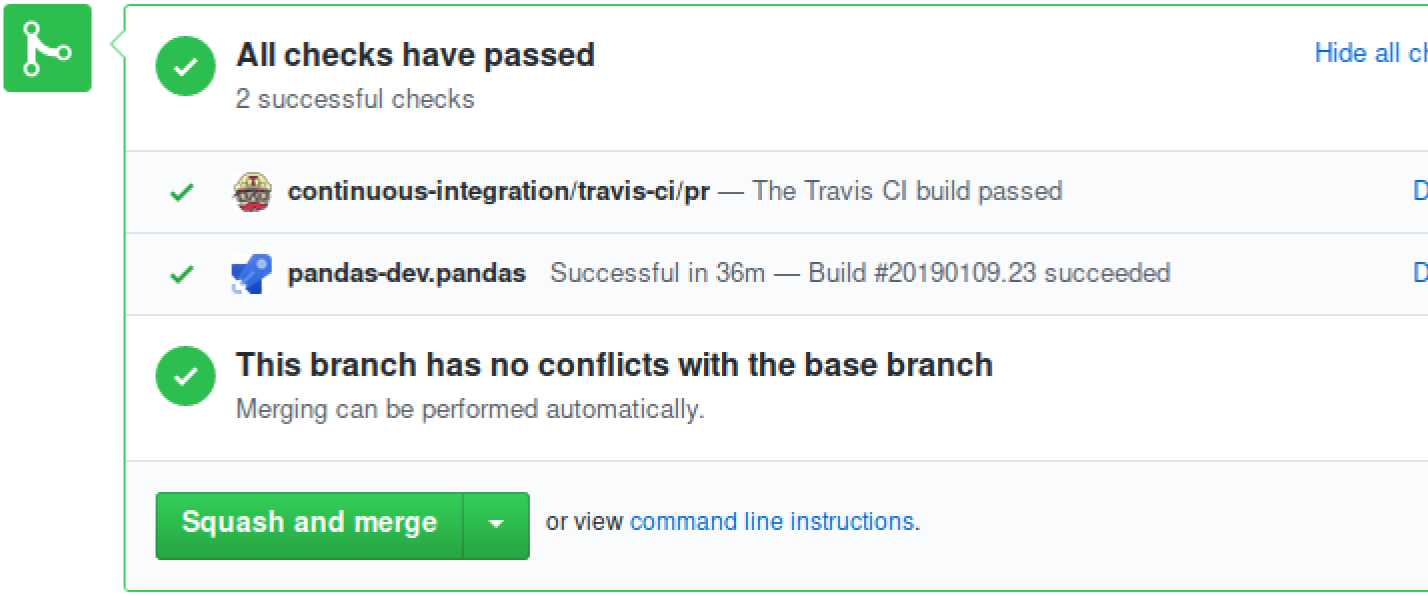
1. write a new test that asserts a warning is issued when calling with the deprecated argument
2. Update all of pandas existing tests and code to use the new argument

See *Testing Warnings* for more.

Testing With Continuous Integration

The *pandas* test suite will run automatically on Travis-CI and Azure Pipelines continuous integration services, once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then the continuous integration services need to be hooked to your GitHub repository. Instructions are here for Travis-CI and Azure Pipelines.

A pull-request will be considered for merging when you have an all 'green' build. If any tests are failing, then you will get a red 'X', where you can click through to see the individual failed tests. This is an example of a green build.



Note: Each time you push to *your* fork, a *new* run of the tests will be triggered on the CI. You can enable the auto-cancel feature, which removes any non-currently-running tests for that same pull-request, for Travis-CI here.

Test-driven development/code writing

pandas is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *pandas*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

Like many packages, *pandas* uses pytest and the convenient extensions in numpy.testing.

Note: The earliest supported pytest version is 3.6.0.

Writing tests

All tests should go into the `tests` subdirectory of the specific package. This folder contains many current examples of tests, and we suggest looking to these for inspiration. If your test requires working with files or network connectivity, there is more information on the testing page of the wiki.

The `pandas.util.testing` module has many special `assert` functions that make it easier to make statements about whether Series or DataFrame objects are equivalent. The easiest way to verify that your code is correct is to explicitly construct the result you expect, then compare the actual result to the expected correct result:

```
def test_pivot(self):
    data = {
        'index' : ['A', 'B', 'C', 'C', 'B', 'A'],
        'columns' : ['One', 'One', 'One', 'Two', 'Two', 'Two'],
        'values' : [1., 2., 3., 3., 2., 1.]
    }

    frame = DataFrame(data)
    pivoted = frame.pivot(index='index', columns='columns', values='values')

    expected = DataFrame({
        'One' : {'A' : 1., 'B' : 2., 'C' : 3.},
        'Two' : {'A' : 1., 'B' : 2., 'C' : 3.}
    })

    assert_frame_equal(pivoted, expected)
```

Transitioning to `pytest`

pandas existing test structure is *mostly* classed based, meaning that you will typically find tests wrapped in a class.

```
class TestReallyCoolFeature(object):
    pass
```

Going forward, we are moving to a more *functional* style using the pytest framework, which offers a richer testing framework that will facilitate testing and developing. Thus, instead of writing test classes, we will write test functions like this:

```
def test_really_cool_feature():
    pass
```

Using `pytest`

Here is an example of a self-contained set of tests that illustrate multiple features that we like to use.

- functional style: tests are like `test_*` and *only* take arguments that are either fixtures or parameters
- `pytest.mark` can be used to set metadata on test functions, e.g. `skip` or `xfail`.
- using `parametrize`: allow testing of multiple cases
- to set a mark on a parameter, `pytest.param(..., marks=...)` syntax should be used
- `fixture`, code for object construction, on a per-test basis
- using bare `assert` for scalars and truth-testing
- `tm.assert_series_equal` (and its counter part `tm.assert_frame_equal`), for pandas object comparisons.
- the typical pattern of constructing an `expected` and comparing versus the `result`

We would name this file `test_cool_feature.py` and put in an appropriate place in the `pandas/tests/` structure.

```python
import pytest
import numpy as np
import pandas as pd


@pytest.mark.parametrize('dtype', ['int8', 'int16', 'int32', 'int64'])
def test_dtypes(dtype):
    assert str(np.dtype(dtype)) == dtype


@pytest.mark.parametrize(
    'dtype', ['float32', pytest.param('int16', marks=pytest.mark.skip),
              pytest.param('int32', marks=pytest.mark.xfail(
                  reason='to show how it works'))])
def test_mark(dtype):
    assert str(np.dtype(dtype)) == 'float32'


@pytest.fixture
def series():
    return pd.Series([1, 2, 3])


@pytest.fixture(params=['int8', 'int16', 'int32', 'int64'])
def dtype(request):
    return request.param


def test_series(series, dtype):
    result = series.astype(dtype)
    assert result.dtype == dtype

    expected = pd.Series([1, 2, 3], dtype=dtype)
    tm.assert_series_equal(result, expected)
```

A test run of this yields

```
((pandas) bash-3.2$ pytest  test_cool_feature.py  -v
=========================== test session starts ===========================
platform darwin -- Python 3.6.2, pytest-3.6.0, py-1.4.31, pluggy-0.4.0
collected 11 items

tester.py::test_dtypes[int8] PASSED
tester.py::test_dtypes[int16] PASSED
tester.py::test_dtypes[int32] PASSED
tester.py::test_dtypes[int64] PASSED
tester.py::test_mark[float32] PASSED
tester.py::test_mark[int16] SKIPPED
tester.py::test_mark[int32] xfail
tester.py::test_series[int8] PASSED
tester.py::test_series[int16] PASSED
tester.py::test_series[int32] PASSED
tester.py::test_series[int64] PASSED
```

Tests that we have `parametrized` are now accessible via the test name, for example we could run these with `-k int8` to sub-select *only* those tests which match `int8`.

```
((pandas) bash-3.2$ pytest  test_cool_feature.py  -v -k int8
=========================== test session starts ===========================
platform darwin -- Python 3.6.2, pytest-3.6.0, py-1.4.31, pluggy-0.4.0
collected 11 items

test_cool_feature.py::test_dtypes[int8] PASSED
test_cool_feature.py::test_series[int8] PASSED
```

Using `hypothesis`

Hypothesis is a library for property-based testing. Instead of explicitly parametrizing a test, you can describe *all* valid inputs and let Hypothesis try to find a failing input. Even better, no matter how many random examples it tries, Hypothesis always reports a single minimal counterexample to your assertions - often an example that you would never have thought to test.

See Getting Started with Hypothesis for more of an introduction, then refer to the Hypothesis documentation for details.

```
import json
from hypothesis import given, strategies as st

any_json_value = st.deferred(lambda: st.one_of(
    st.none(), st.booleans(), st.floats(allow_nan=False), st.text(),
    st.lists(any_json_value), st.dictionaries(st.text(), any_json_value)
))


@given(value=any_json_value)
def test_json_roundtrip(value):
    result = json.loads(json.dumps(value))
    assert value == result
```

This test shows off several useful features of Hypothesis, as well as demonstrating a good use-case: checking properties that should hold over a large or complicated domain of inputs.

To keep the Pandas test suite running quickly, parametrized tests are preferred if the inputs or logic are simple, with Hypothesis tests reserved for cases with complex logic or where there are too many combinations of options or subtle interactions to test (or think of!) all of them.

Testing Warnings

By default, one of pandas CI workers will fail if any unhandled warnings are emitted.

If your change involves checking that a warning is actually emitted, use `tm.assert_produces_warning(ExpectedWarning)`.

```
import pandas.util.testing as tm


df = pd.DataFrame()
with tm.assert_produces_warning(FutureWarning):
    df.some_operation()
```

We prefer this to the `pytest.warns` context manager because ours checks that the warning's stacklevel is set correctly. The stacklevel is what ensure the *user's* file name and line number is printed in the warning, rather than something internal to pandas. It represents the number of function calls from user code (e.g. `df.some_operation()`) to the function that actually emits the warning. Our linter will fail the build if you use `pytest.warns` in a test.

If you have a test that would emit a warning, but you aren't actually testing the warning itself (say because it's going to be removed in the future, or because we're matching a 3rd-party library's behavior), then use `pytest.mark.filterwarnings` to ignore the error.

```
@pytest.mark.filterwarnings("ignore:msg:category")
def test_thing(self):
    ...
```

If the test generates a warning of class `category` whose message starts with `msg`, the warning will be ignored and the test will pass.

If you need finer-grained control, you can use Python's usual warnings module to control whether a warning is ignored / raised at different places within a single test.

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore", FutureWarning)
    # Or use warnings.filterwarnings(...)
```

Alternatively, consider breaking up the unit test.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *pandas*) by typing:

```
pytest pandas
```

The tests suite is exhaustive and takes around 20 minutes to run. Often it is worth running only a subset of tests first around your changes before running the entire suite.

The easiest way to do this is with:

```
pytest pandas/path/to/test.py -k regex_matching_test_name
```

Or with one of the following constructs:

```
pytest pandas/tests/[test-module].py
pytest pandas/tests/[test-module].py::[TestClass]
pytest pandas/tests/[test-module].py::[TestClass]::[test_method]
```

Using pytest-xdist, one can speed up local testing on multicore machines. To use this feature, you will need to install *pytest-xdist* via:

```
pip install pytest-xdist
```

Two scripts are provided to assist with this. These scripts distribute testing across 4 threads.

On Unix variants, one can type:

```
test_fast.sh
```

On Windows, one can type:

```
test_fast.bat
```

This can significantly reduce the time it takes to locally run tests before submitting a pull request.

For more, see the pytest documentation.

New in version 0.20.0.

Furthermore one can run

```
pd.test()
```

with an imported pandas to run tests similarly.

Running the performance test suite

Performance matters and it is worth considering whether your code has introduced performance regressions. *pandas* is in the process of migrating to asv benchmarks to enable easy monitoring of the performance of critical *pandas* operations. These benchmarks are all found in the `pandas/asv_bench` directory. asv supports both python2 and python3.

To use all features of asv, you will need either `conda` or `virtualenv`. For more details please check the asv installation webpage.

To install asv:

```
pip install git+https://github.com/spacetelescope/asv
```

If you need to run a benchmark, change your directory to `asv_bench/` and run:

```
asv continuous -f 1.1 upstream/master HEAD
```

You can replace `HEAD` with the name of the branch you are working on, and report benchmarks that changed by more than 10%. The command uses `conda` by default for creating the benchmark environments. If you want to use virtualenv instead, write:

```
asv continuous -f 1.1 -E virtualenv upstream/master HEAD
```

The `-E virtualenv` option should be added to all `asv` commands that run benchmarks. The default value is defined in `asv.conf.json`.

Running the full test suite can take up to one hour and use up to 3GB of RAM. Usually it is sufficient to paste only a subset of the results into the pull request to show that the committed changes do not cause unexpected performance regressions. You can run specific benchmarks using the `-b` flag, which takes a regular expression. For example, this will only run tests from a `pandas/asv_bench/benchmarks/groupby.py` file:

```
asv continuous -f 1.1 upstream/master HEAD -b ^groupby
```

If you want to only run a specific group of tests from a file, you can do it using `.` as a separator. For example:

```
asv continuous -f 1.1 upstream/master HEAD -b groupby.GroupByMethods
```

will only run the `GroupByMethods` benchmark defined in `groupby.py`.

You can also run the benchmark suite using the version of `pandas` already installed in your current Python environment. This can be useful if you do not have virtualenv or conda, or are using the `setup.py develop` approach discussed above; for the in-place build you need to set `PYTHONPATH`, e.g. `PYTHONPATH="$PWD/.." asv [remaining arguments]`. You can run benchmarks using an existing Python environment by:

```
asv run -e -E existing
```

or, to use a specific Python interpreter,:

```
asv run -e -E existing:python3.5
```

This will display stderr from the benchmarks, and use your local `python` that comes from your `$PATH`.

Information on how to write a benchmark and how to use asv can be found in the asv documentation.

Documenting your code

Changes should be reflected in the release notes located in `doc/source/whatsnew/vx.y.z.rst`. This file contains an ongoing change log for each release. Add an entry to this file to document your fix, enhancement or (unavoidable) breaking change. Make sure to include the GitHub issue number when adding your entry (using ``:issue:`1234` `` where `1234` is the issue/pull request number).

If your code is an enhancement, it is most likely necessary to add usage examples to the existing documentation. This can be done following the section regarding documentation *above*. Further, to let users know when this feature was added, the `versionadded` directive is used. The sphinx syntax for that is:

```
.. versionadded:: 0.21.0
```

This will put the text *New in version 0.21.0* wherever you put the sphinx directive. This should also be put in the docstring when adding a new function or method (example) or a new keyword argument (example).

7.1.6 Contributing your changes to *pandas*

Committing your code

Keep style fixes to a separate commit to make your pull request more readable.

Once you've made changes, you can see them by typing:

```
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```
git add path/to/file-to-be-added.py
```

Doing 'git status' again should give something like:

```
# On branch shiny-new-feature
#
#       modified:   /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message. *Pandas* uses a convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- ENH: Enhancement, new functionality
- BUG: Bug fix
- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts

- PERF: Performance improvement
- CLN: Code cleanup

The following defines how a commit message should be structured. Please reference the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < *80* chars.
- One blank line.
- Optionally, a commit message body.

Now you can commit your changes in your local repository:

```
git commit -m
```

Pushing your changes

When you want your changes to appear publicly on your GitHub page, push your forked feature branch's commits:

```
git push origin shiny-new-feature
```

Here `origin` is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:yourname/pandas.git (fetch)
origin  git@github.com:yourname/pandas.git (push)
upstream        git://github.com/pandas-dev/pandas.git (fetch)
upstream        git://github.com/pandas-dev/pandas.git (push)
```

Now your code is on GitHub, but it is not yet a part of the *pandas* project. For that to happen, a pull request needs to be submitted on GitHub.

Review your code

When you're ready to ask for a code review, file a pull request. Before you do, once again make sure that you have followed all the guidelines outlined in this document regarding code style, tests, performance tests, and documentation. You should also double check your branch changes against the branch it was based on:

1. Navigate to your repository on GitHub – https://github.com/your-user-name/pandas
2. Click on `Branches`
3. Click on the `Compare` button for your feature branch
4. Select the `base` and `compare` branches, if necessary. This will be `master` and `shiny-new-feature`, respectively.

Finally, make the pull request

If everything looks good, you are ready to make a pull request. A pull request is how code from a local repository becomes available to the GitHub community and can be looked at and eventually merged into the master version. This pull request and its associated changes will eventually be committed to the master branch and available in the next release. To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the `Pull Request` button
3. You can then click on `Commits` and `Files Changed` to make sure everything looks okay one last time
4. Write a description of your changes in the `Preview Discussion` tab
5. Click `Send Pull Request`.

This request then goes to the repository maintainers, and they will review the code.

Updating your pull request

Based on the review you get on your pull request, you will probably need to make some changes to the code. In that case, you can make them in your branch, add a new commit to that branch, push it to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the *Continuous Integration* tests.

Another reason you might need to update your pull request is to solve conflicts with changes that have been merged into the master branch since you opened your pull request.

To do this, you need to "merge upstream master" in your branch:

```
git checkout shiny-new-feature
git fetch upstream
git merge upstream/master
```

If there are no conflicts (or they could be fixed automatically), a file with a default commit message will open, and you can simply save and quit this file.

If there are merge conflicts, you need to solve those conflicts. See for example at https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/ for an explanation on how to do this. Once the conflicts are merged and the files where the conflicts were solved are added, you can run `git commit` to save those fixes.

If you have uncommitted changes at the moment you want to update the branch with master, you will need to `stash` them prior to updating (see the stash docs). This will effectively store your changes and they can be reapplied after updating.

After the feature branch has been update locally, you can now update your pull request by pushing to the branch on GitHub:

```
git push origin shiny-new-feature
```

Delete your merged branch (optional)

Once your feature branch is accepted into upstream, you'll probably want to get rid of the branch. First, merge upstream master into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can do:

```
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`, or else git won't warn you if your feature branch has not actually been merged.

The branch will still exist on GitHub, so to delete it there do:

```
git push origin --delete shiny-new-feature
```

7.2 Internals

This section will provide a look into some of pandas internals. It's primarily intended for developers of pandas itself.

7.2.1 Indexing

In pandas there are a few objects implemented which can serve as valid containers for the axis labels:

- `Index`: the generic "ordered set" object, an ndarray of object dtype assuming nothing about its contents. The labels must be hashable (and likely immutable) and unique. Populates a dict of label to location in Cython to do `O(1)` lookups.
- `Int64Index`: a version of `Index` highly optimized for 64-bit integer data, such as time stamps
- `Float64Index`: a version of `Index` highly optimized for 64-bit float data
- `MultiIndex`: the standard hierarchical index object
- `DatetimeIndex`: An Index object with `Timestamp` boxed elements (impl are the int64 values)
- `TimedeltaIndex`: An Index object with `Timedelta` boxed elements (impl are the in64 values)
- `PeriodIndex`: An Index object with Period elements

There are functions that make the creation of a regular index easy:

- `date_range`: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Python datetime objects
- `period_range`: fixed frequency date range generated from a time rule or DateOffset. An ndarray of `Period` objects, representing timespans

The motivation for having an `Index` class in the first place was to enable different implementations of indexing. This means that it's possible for you, the user, to implement a custom `Index` subclass that may be better suited to a particular application than the ones provided in pandas.

From an internal implementation point of view, the relevant methods that an `Index` must define are one or more of the following (depending on how incompatible the new object internals are with the `Index` functions):

- `get_loc`: returns an "indexer" (an integer, or in some cases a slice object) for a label
- `slice_locs`: returns the "range" to slice between two labels
- `get_indexer`: Computes the indexing vector for reindexing / data alignment purposes. See the source / docstrings for more on this
- `get_indexer_non_unique`: Computes the indexing vector for reindexing / data alignment purposes when the index is non-unique. See the source / docstrings for more on this
- `reindex`: Does any pre-conversion of the input index then calls `get_indexer`
- `union`, `intersection`: computes the union or intersection of two Index objects

- `insert`: Inserts a new label into an Index, yielding a new object
- `delete`: Delete a label, yielding a new object
- `drop`: Deletes a set of labels
- `take`: Analogous to ndarray.take

MultiIndex

Internally, the `MultiIndex` consists of a few things: the **levels**, the integer **codes** (until version 0.24 named *labels*), and the level **names**:

```
In [1]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']],
   ...:                                    names=['first', 'second'])
   ...:

In [2]: index
Out[2]:
MultiIndex(levels=[[0, 1, 2], ['one', 'two']],
           codes=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
           names=['first', 'second'])

In [3]: index.levels
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪FrozenList([[0, 1, 2], ['one', 'two']])

In [4]: index.codes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪FrozenList([[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])

In [5]: index.names
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪FrozenList(['first', 'second'])
```

You can probably guess that the codes determine which unique element is identified with that location at each layer of the index. It's important to note that sortedness is determined **solely** from the integer codes and does not check (or care) whether the levels themselves are sorted. Fortunately, the constructors `from_tuples` and `from_arrays` ensure that this is true, but if you compute the levels and codes yourself, please be careful.

Values

Pandas extends NumPy's type system with custom types, like `Categorical` or datetimes with a timezone, so we have multiple notions of "values". For 1-D containers (`Index` classes and `Series`) we have the following convention:

- `cls._ndarray_values` is *always* a NumPy `ndarray`. Ideally, `_ndarray_values` is cheap to compute. For example, for a `Categorical`, this returns the codes, not the array of objects.
- `cls._values` refers is the "best possible" array. This could be an `ndarray`, `ExtensionArray`, or in `Index` subclass (note: we're in the process of removing the index subclasses here so that it's always an `ndarray` or `ExtensionArray`).

So, for example, `Series[category]._values` is a `Categorical`, while `Series[category]._ndarray_values` is the underlying codes.

7.2.2 Subclassing pandas Data Structures

This section has been moved to *Subclassing pandas Data Structures*.

7.3 Extending Pandas

While pandas provides a rich set of methods, containers, and data types, your needs may not be fully satisfied. Pandas offers a few options for extending pandas.

7.3.1 Registering Custom Accessors

Libraries can use the decorators *pandas.api.extensions.register_dataframe_accessor()*, *pandas.api.extensions.register_series_accessor()*, and *pandas.api.extensions.register_index_accessor()*, to add additional "namespaces" to pandas objects. All of these follow a similar convention: you decorate a class, providing the name of attribute to add. The class's `__init__` method gets the object being decorated. For example:

```
@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._validate(pandas_obj)
        self._obj = pandas_obj

    @staticmethod
    def _validate(obj):
        if 'lat' not in obj.columns or 'lon' not in obj.columns:
            raise AttributeError("Must have 'lat' and 'lon'.")

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Now users can access your methods using the `geo` namespace:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

This can be a convenient way to extend pandas objects without subclassing them. If you write a custom accessor, make a pull request adding it to our *pandas Ecosystem* page.

We highly recommend validating the data in your accessor's *__init__*. In our `GeoAccessor`, we validate that the data contains the expected columns, raising an `AttributeError` when the validation fails. For a `Series` accessor, you should validate the `dtype` if the accessor applies only to certain dtypes.

7.3.2 Extension Types

New in version 0.23.0.

Warning: The *pandas.api.extensions.ExtensionDtype* and *pandas.api.extensions.ExtensionArray* APIs are new and experimental. They may change between versions without warning.

Pandas defines an interface for implementing data types and arrays that *extend* NumPy's type system. Pandas itself uses the extension system for some types that aren't built into NumPy (categorical, period, interval, datetime with timezone).

Libraries can define a custom array and data type. When pandas encounters these objects, they will be handled properly (i.e. not converted to an ndarray of objects). Many methods like *pandas.isna()* will dispatch to the extension type's implementation.

If you're building a library that implements the interface, please publicize it on *Extension Data Types*.

The interface consists of two classes.

ExtensionDtype

A *pandas.api.extensions.ExtensionDtype* is similar to a `numpy.dtype` object. It describes the data type. Implementors are responsible for a few unique items like the name.

One particularly important item is the `type` property. This should be the class that is the scalar type for your data. For example, if you were writing an extension array for IP Address data, this might be `ipaddress.IPv4Address`.

See the extension dtype source for interface definition.

New in version 0.24.0.

`pandas.api.extension.ExtensionDtype` can be registered to pandas to allow creation via a string dtype name. This allows one to instantiate `Series` and `.astype()` with a registered string name, for example `'category'` is a registered string accessor for the `CategoricalDtype`.

See the extension dtype dtypes for more on how to register dtypes.

ExtensionArray

This class provides all the array-like functionality. ExtensionArrays are limited to 1 dimension. An ExtensionArray is linked to an ExtensionDtype via the `dtype` attribute.

Pandas makes no restrictions on how an extension array is created via its `__new__` or `__init__`, and puts no restrictions on how you store your data. We do require that your array be convertible to a NumPy array, even if this is relatively expensive (as it is for `Categorical`).

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 addresses may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists.

See the extension array source for the interface definition. The docstrings and comments contain guidance for properly implementing the interface.

ExtensionArray Operator Support

New in version 0.24.0.

By default, there are no operators defined for the class *ExtensionArray*. There are two approaches for providing operator support for your ExtensionArray:

1. Define each of the operators on your ExtensionArray subclass.
2. Use an operator implementation from pandas that depends on operators that are already defined on the underlying elements (scalars) of the ExtensionArray.

Note: Regardless of the approach, you may want to set __array_priority__ if you want your implementation to be called when involved in binary operations with NumPy arrays.

For the first approach, you define selected operators, e.g., __add__, __le__, etc. that you want your ExtensionArray subclass to support.

The second approach assumes that the underlying elements (i.e., scalar type) of the ExtensionArray have the individual operators already defined. In other words, if your ExtensionArray named MyExtensionArray is implemented so that each element is an instance of the class MyExtensionElement, then if the operators are defined for MyExtensionElement, the second approach will automatically define the operators for MyExtensionArray.

A mixin class, ExtensionScalarOpsMixin supports this second approach. If developing an ExtensionArray subclass, for example MyExtensionArray, can simply include ExtensionScalarOpsMixin as a parent class of MyExtensionArray, and then call the methods _add_arithmetic_ops() and/or _add_comparison_ops() to hook the operators into your MyExtensionArray class, as follows:

```
from pandas.api.extensions import ExtensionArray, ExtensionScalarOpsMixin

class MyExtensionArray(ExtensionArray, ExtensionScalarOpsMixin):
    pass


MyExtensionArray._add_arithmetic_ops()
MyExtensionArray._add_comparison_ops()
```

Note: Since pandas automatically calls the underlying operator on each element one-by-one, this might not be as performant as implementing your own version of the associated operators directly on the ExtensionArray.

For arithmetic operations, this implementation will try to reconstruct a new ExtensionArray with the result of the element-wise operation. Whether or not that succeeds depends on whether the operation returns a result that's valid for the ExtensionArray. If an ExtensionArray cannot be reconstructed, an ndarray containing the scalars returned instead.

For ease of implementation and consistency with operations between pandas and NumPy ndarrays, we recommend *not* handling Series and Indexes in your binary ops. Instead, you should detect these cases and return NotImplemented. When pandas encounters an operation like op(Series, ExtensionArray), pandas will

1. unbox the array from the Series (Series.array)
2. call result = op(values, ExtensionArray)
3. re-box the result in a Series

Testing Extension Arrays

We provide a test suite for ensuring that your extension arrays satisfy the expected behavior. To use the test suite, you must provide several pytest fixtures and inherit from the base test class. The required fixtures are found in https://github.com/pandas-dev/pandas/blob/master/pandas/tests/extension/conftest.py.

To use a test, subclass it:

```
from pandas.tests.extension import base


class TestConstructors(base.BaseConstructorsTests):
    pass
```

See https://github.com/pandas-dev/pandas/blob/master/pandas/tests/extension/base/__init__.py for a list of all the tests available.

7.3.3 Subclassing pandas Data Structures

Warning: There are some easier alternatives before considering subclassing `pandas` data structures.

1. Extensible method chains with *pipe*
2. Use *composition*. See here.
3. Extending by *registering an accessor*
4. Extending by *extension type*

This section describes how to subclass `pandas` data structures to meet more specific needs. There are two points that need attention:

1. Override constructor properties.
2. Define original properties

Note: You can find a nice example in geopandas project.

Override Constructor Properties

Each data structure has several *constructor properties* for returning a new data structure as the result of an operation. By overriding these properties, you can retain subclasses through `pandas` data manipulations.

There are 3 constructor properties to be defined:

- `_constructor`: Used when a manipulation result has the same dimensions as the original.
- `_constructor_sliced`: Used when a manipulation result has one lower dimension(s) as the original, such as `DataFrame` single columns slicing.
- `_constructor_expanddim`: Used when a manipulation result has one higher dimension as the original, such as `Series.to_frame()` and `DataFrame.to_panel()`.

Following table shows how `pandas` data structures define constructor properties by default.

Property Attributes	Series	DataFrame
_constructor	Series	DataFrame
_constructor_sliced	NotImplementedError	Series
_constructor_expanddim	DataFrame	Panel

Below example shows how to define SubclassedSeries and SubclassedDataFrame overriding constructor properties.

```
class SubclassedSeries(pd.Series):

    @property
    def _constructor(self):
        return SubclassedSeries

    @property
    def _constructor_expanddim(self):
        return SubclassedDataFrame


class SubclassedDataFrame(pd.DataFrame):

    @property
    def _constructor(self):
        return SubclassedDataFrame

    @property
    def _constructor_sliced(self):
        return SubclassedSeries
```

```
>>> s = SubclassedSeries([1, 2, 3])
>>> type(s)
<class '__main__.SubclassedSeries'>

>>> to_framed = s.to_frame()
>>> type(to_framed)
<class '__main__.SubclassedDataFrame'>

>>> df = SubclassedDataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9

>>> type(df)
<class '__main__.SubclassedDataFrame'>

>>> sliced1 = df[['A', 'B']]
>>> sliced1
   A  B
0  1  4
1  2  5
2  3  6

>>> type(sliced1)
<class '__main__.SubclassedDataFrame'>
```

(continues on next page)

(continued from previous page)

```
>>> sliced2 = df['A']
>>> sliced2
0    1
1    2
2    3
Name: A, dtype: int64

>>> type(sliced2)
<class '__main__.SubclassedSeries'>
```

Define Original Properties

To let original data structures have additional properties, you should let `pandas` know what properties are added. `pandas` maps unknown properties to data names overriding `__getattribute__`. Defining original properties can be done in one of 2 ways:

1. Define `_internal_names` and `_internal_names_set` for temporary properties which WILL NOT be passed to manipulation results.
2. Define `_metadata` for normal properties which will be passed to manipulation results.

Below is an example to define two original properties, "internal_cache" as a temporary property and "added_property" as a normal property

```
class SubclassedDataFrame2(pd.DataFrame):

    # temporary properties
    _internal_names = pd.DataFrame._internal_names + ['internal_cache']
    _internal_names_set = set(_internal_names)

    # normal properties
    _metadata = ['added_property']

    @property
    def _constructor(self):
        return SubclassedDataFrame2
```

```
>>> df = SubclassedDataFrame2({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9

>>> df.internal_cache = 'cached'
>>> df.added_property = 'property'

>>> df.internal_cache
cached
>>> df.added_property
property

# properties defined in _internal_names is reset after manipulation
>>> df[['A', 'B']].internal_cache
```

(continues on next page)

(continued from previous page)

```
AttributeError: 'SubclassedDataFrame2' object has no attribute 'internal_cache'

# properties defined in _metadata are retained
>>> df[['A', 'B']].added_property
property
```

7.4 Developer

This section will focus on downstream applications of pandas.

7.4.1 Storing pandas DataFrame objects in Apache Parquet format

The Apache Parquet format provides key-value metadata at the file and column level, stored in the footer of the Parquet file:

```
5: optional list<KeyValue> key_value_metadata
```

where `KeyValue` is

```
struct KeyValue {
  1: required string key
  2: optional string value
}
```

So that a `pandas.DataFrame` can be faithfully reconstructed, we store a `pandas` metadata key in the `FileMetaData` with the value stored as :

```
{'index_columns': ['__index_level_0__', '__index_level_1__', ...],
 'column_indexes': [<ci0>, <ci1>, ..., <ciN>],
 'columns': [<c0>, <c1>, ...],
 'pandas_version': $VERSION}
```

Here, `<c0>`/`<ci0>` and so forth are dictionaries containing the metadata for each column, *including the index columns*. This has JSON form:

```
{'name': column_name,
 'field_name': parquet_column_name,
 'pandas_type': pandas_type,
 'numpy_type': numpy_type,
 'metadata': metadata}
```

Note: Every index column is stored with a name matching the pattern `__index_level_\d+__` and its corresponding column information is can be found with the following code snippet.

Following this naming convention isn't strictly necessary, but strongly suggested for compatibility with Arrow.

Here's an example of how the index metadata is structured in pyarrow:

```
# assuming there's at least 3 levels in the index
index_columns = metadata['index_columns']  # noqa: F821
columns = metadata['columns']  # noqa: F821
ith_index = 2
```

(continues on next page)

(continued from previous page)

```
assert index_columns[ith_index] == '__index_level_2__'
ith_index_info = columns[-len(index_columns):][ith_index]
ith_index_level_name = ith_index_info['name']
```

`pandas_type` is the logical type of the column, and is one of:

- Boolean: `'bool'`
- Integers: `'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64'`
- Floats: `'float16', 'float32', 'float64'`
- Date and Time Types: `'datetime', 'datetimetz'`, `'timedelta'`
- String: `'unicode', 'bytes'`
- Categorical: `'categorical'`
- Other Python objects: `'object'`

The `numpy_type` is the physical storage type of the column, which is the result of `str(dtype)` for the underlying NumPy array that holds the data. So for `datetimetz` this is `datetime64[ns]` and for categorical, it may be any of the supported integer categorical types.

The `metadata` field is `None` except for:

- `datetimetz: {'timezone': zone, 'unit': 'ns'}`, e.g. `{'timezone', 'America/New_York', 'unit': 'ns'}`. The `'unit'` is optional, and if omitted it is assumed to be nanoseconds.
- `categorical`: `{'num_categories': K, 'ordered': is_ordered, 'type': $TYPE}`
  - Here `'type'` is optional, and can be a nested pandas type specification here (but not categorical)
- `unicode`: `{'encoding': encoding}`
  - The encoding is optional, and if not present is UTF-8
- `object: {'encoding': encoding}`. Objects can be serialized and stored in `BYTE_ARRAY` Parquet columns. The encoding can be one of:
  - `'pickle'`
  - `'msgpack'`
  - `'bson'`
  - `'json'`
- `timedelta`: `{'unit': 'ns'}`. The `'unit'` is optional, and if omitted it is assumed to be nanoseconds. This metadata is optional altogether

For types other than these, the `'metadata'` key can be omitted. Implementations can assume `None` if the key is not present.

As an example of fully-formed metadata:

```
{'index_columns': ['__index_level_0__'],
 'column_indexes': [
     {'name': None,
      'field_name': 'None',
      'pandas_type': 'unicode',
      'numpy_type': 'object',
```

(continues on next page)

(continued from previous page)

```
     'metadata': {'encoding': 'UTF-8'}}
],
'columns': [
    {'name': 'c0',
     'field_name': 'c0',
     'pandas_type': 'int8',
     'numpy_type': 'int8',
     'metadata': None},
    {'name': 'c1',
     'field_name': 'c1',
     'pandas_type': 'bytes',
     'numpy_type': 'object',
     'metadata': None},
    {'name': 'c2',
     'field_name': 'c2',
     'pandas_type': 'categorical',
     'numpy_type': 'int16',
     'metadata': {'num_categories': 1000, 'ordered': False}},
    {'name': 'c3',
     'field_name': 'c3',
     'pandas_type': 'datetimetz',
     'numpy_type': 'datetime64[ns]',
     'metadata': {'timezone': 'America/Los_Angeles'}},
    {'name': 'c4',
     'field_name': 'c4',
     'pandas_type': 'object',
     'numpy_type': 'object',
     'metadata': {'encoding': 'pickle'}},
    {'name': None,
     'field_name': '__index_level_0__',
     'pandas_type': 'int64',
     'numpy_type': 'int64',
     'metadata': None}
],
'pandas_version': '0.20.0'}
```

CHAPTER

EIGHT

RELEASE NOTES

This is the list of changes to pandas between each release. For full details, see the commit logs at http://github.com/pandas-dev/pandas. For install and upgrade instructions, see *Installation*.

8.1 Version 0.24

8.2 Version 0.23

8.2.1 What's New in 0.23.4 (August 3, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See *Plan for dropping Python 2.7* for more.

What's new in v0.23.4

- *Fixed Regressions*
- *Bug Fixes*
- *Contributors*

Fixed Regressions

- Python 3.7 with Windows gave all missing values for rolling variance calculations (GH21813)

Bug Fixes

Groupby/Resample/Rolling

- Bug where calling `DataFrameGroupBy.agg()` with a list of functions including `ohlc` as the non-initial element would raise a `ValueError` (GH21716)
- Bug in `roll_quantile` caused a memory leak when calling `.rolling(...).quantile(q)` with `q` in (0,1) (GH21965)

Missing

- Bug in `Series.clip()` and `DataFrame.clip()` cannot accept list-like threshold containing `NaN` (GH19992)

Contributors

A total of 6 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Jeff Reback
- MeeseeksMachine +
- Tom Augspurger
- chris-b1
- h-vetinari
- meeseeksdev[bot]

8.2.2 What's New in 0.23.3 (July 7, 2018)

This release fixes a build issue with the sdist for Python 3.7 (GH21785) There are no other changes.

Contributors

A total of 2 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Tom Augspurger
- meeseeksdev[bot] +

8.2.3 What's New in 0.23.2 (July 5, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Note: Pandas 0.23.2 is first pandas release that's compatible with Python 3.7 (GH20552)

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See *Plan for dropping Python 2.7* for more.

What's new in v0.23.2

- *Logical Reductions over Entire DataFrame*
- *Fixed Regressions*
- *Build Changes*

- *Bug Fixes*
- *Contributors*

Logical Reductions over Entire DataFrame

DataFrame.all() and *DataFrame.any()* now accept `axis=None` to reduce over all axes to a scalar (GH19976)

```
In [1]: df = pd.DataFrame({"A": [1, 2], "B": [True, False]})

In [2]: df.all(axis=None)
Out[2]: False
```

This also provides compatibility with NumPy 1.15, which now dispatches to `DataFrame.all`. With NumPy 1.15 and pandas 0.23.1 or earlier, `numpy.all()` will no longer reduce over every axis:

```
>>> # NumPy 1.15, pandas 0.23.1
>>> np.any(pd.DataFrame({"A": [False], "B": [False]}))
A    False
B    False
dtype: bool
```

With pandas 0.23.2, that will correctly return False, as it did with NumPy < 1.15.

```
In [3]: np.any(pd.DataFrame({"A": [False], "B": [False]}))
Out[3]: False
```

Fixed Regressions

- Fixed regression in `to_csv()` when handling file-like object incorrectly (GH21471)
- Re-allowed duplicate level names of a `MultiIndex`. Accessing a level that has a duplicate name by name still raises an error (GH19029).
- Bug in both *DataFrame.first_valid_index()* and *Series.first_valid_index()* raised for a row index having duplicate values (GH21441)
- Fixed printing of DataFrames with hierarchical columns with long names (GH21180)
- Fixed regression in *reindex()* and *groupby()* with a MultiIndex or multiple keys that contains categorical datetime-like values (GH21390).
- Fixed regression in unary negative operations with object dtype (GH21380)
- Bug in *Timestamp.ceil()* and *Timestamp.floor()* when timestamp is a multiple of the rounding frequency (GH21262)
- Fixed regression in `to_clipboard()` that defaulted to copying dataframes with space delimited instead of tab delimited (GH21104)

Build Changes

- The source and binary distributions no longer include test data files, resulting in smaller download sizes. Tests relying on these data files will be skipped when using `pandas.test()`. (GH19320)

Bug Fixes

Conversion

- Bug in constructing *Index* with an iterator or generator (GH21470)
- Bug in *Series.nlargest()* for signed and unsigned integer dtypes when the minimum value is present (GH21426)

Indexing

- Bug in *Index.get_indexer_non_unique()* with categorical key (GH21448)
- Bug in comparison operations for *MultiIndex* where error was raised on equality / inequality comparison involving a MultiIndex with `nlevels == 1` (GH21149)
- Bug in *DataFrame.drop()* behaviour is not consistent for unique and non-unique indexes (GH21494)
- Bug in *DataFrame.duplicated()* with a large number of columns causing a 'maximum recursion depth exceeded' (GH21524).

I/O

- Bug in *read_csv()* that caused it to incorrectly raise an error when `nrows=0`, `low_memory=True`, and `index_col` was not `None` (GH21141)
- Bug in `json_normalize()` when formatting the `record_prefix` with integer columns (GH21536)

Categorical

- Bug in rendering *Series* with `Categorical` dtype in rare conditions under Python 2.7 (GH21002)

Timezones

- Bug in *Timestamp* and *DatetimeIndex* where passing a *Timestamp* localized after a DST transition would return a datetime before the DST transition (GH20854)
- Bug in comparing *DataFrame* with tz-aware *DatetimeIndex* columns with a DST transition that raised a `KeyError` (GH19970)
- Bug in `DatetimeIndex.shift()` where an `AssertionError` would raise when shifting across DST (GH8616)
- Bug in *Timestamp* constructor where passing an invalid timezone offset designator (`Z`) would not raise a `ValueError` (GH8910)
- Bug in *Timestamp.replace()* where replacing at a DST boundary would retain an incorrect offset (GH7825)
- Bug in `DatetimeIndex.reindex()` when reindexing a tz-naive and tz-aware *DatetimeIndex* (GH8306)
- Bug in `DatetimeIndex.resample()` when downsampling across a DST boundary (GH8531)

Timedelta

- Bug in *Timedelta* where non-zero timedeltas shorter than 1 microsecond were considered False (GH21484)

Contributors

A total of 17 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- David Krych

- Jacopo Rota +
- Jeff Reback
- Jeremy Schendel
- Joris Van den Bossche
- Kalyan Gokhale
- Matthew Roeschke
- Michael Odintsov +
- Ming Li
- Pietro Battiston
- Tom Augspurger
- Uddeshya Singh
- Vu Le +
- alimcmaster1 +
- david-liu-brattle-1 +
- gfyoung
- jbrockmendel

8.2.4 What's New in 0.23.1 (June 12, 2018)

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

Warning: Starting January 1, 2019, pandas feature releases will support Python 3 only. See *Plan for dropping Python 2.7* for more.

What's new in v0.23.1

- *Fixed Regressions*
- *Performance Improvements*
- *Bug Fixes*
- *Contributors*

Fixed Regressions

Comparing Series with datetime.date

We've reverted a 0.23.0 change to comparing a `Series` holding datetimes and a `datetime.date` object (GH21152). In pandas 0.22 and earlier, comparing a Series holding datetimes and `datetime.date` objects would coerce the `datetime.date` to a datetime before comapring. This was inconsistent with Python, NumPy, and `DatetimeIndex`, which never consider a datetime and `datetime.date` equal.

In 0.23.0, we unified operations between DatetimeIndex and Series, and in the process changed comparisons between a Series of datetimes and `datetime.date` without warning.

We've temporarily restored the 0.22.0 behavior, so datetimes and dates may again compare equal, but restore the 0.23.0 behavior in a future release.

To summarize, here's the behavior in 0.22.0, 0.23.0, 0.23.1:

```
# 0.22.0... Silently coerce the datetime.date
>>> import datetime
>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0     True
1    False
dtype: bool

# 0.23.0... Do not coerce the datetime.date
>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0    False
1    False
dtype: bool

# 0.23.1... Coerce the datetime.date with a warning
>>> pd.Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
/bin/python:1: FutureWarning: Comparing Series of datetimes with 'datetime.date'. ␣
↪Currently, the
'datetime.date' is coerced to a datetime. In the future pandas will
not coerce, and the values not compare equal to the 'datetime.date'.
To retain the current behavior, convert the 'datetime.date' to a
datetime with 'pd.Timestamp'.
  #!/bin/python3
0     True
1    False
dtype: bool
```

In addition, ordering comparisons will raise a `TypeError` in the future.

Other Fixes

- Reverted the ability of *to_sql()* to perform multivalue inserts as this caused regression in certain cases (GH21103). In the future this will be made configurable.
- Fixed regression in the *DatetimeIndex.date* and *DatetimeIndex.time* attributes in case of timezone-aware data: *DatetimeIndex.time* returned a tz-aware time instead of tz-naive (GH21267) and *DatetimeIndex.date* returned incorrect date when the input date has a non-UTC timezone (GH21230).
- Fixed regression in *pandas.io.json.json_normalize()* when called with `None` values in nested levels in JSON, and to not drop keys with value as *None* (GH21158, GH21356).
- Bug in *to_csv()* causes encoding error when compression and encoding are specified (GH21241, GH21118)
- Bug preventing pandas from being importable with -OO optimization (GH21071)
- Bug in `Categorical.fillna()` incorrectly raising a `TypeError` when *value* the individual categories are iterable and *value* is an iterable (GH21097, GH19788)
- Fixed regression in constructors coercing NA values like `None` to strings when passing `dtype=str` (GH21083)
- Regression in *pivot_table()* where an ordered `Categorical` with missing values for the pivot's `index` would give a mis-aligned result (GH21133)
- Fixed regression in merging on boolean index/columns (GH21119).

Performance Improvements

- Improved performance of `CategoricalIndex.is_monotonic_increasing()`, `CategoricalIndex.is_monotonic_decreasing()` and `CategoricalIndex.is_monotonic()` (GH21025)
- Improved performance of `CategoricalIndex.is_unique()` (GH21107)

Bug Fixes

Groupby/Resample/Rolling

- Bug in *DataFrame.agg()* where applying multiple aggregation functions to a *DataFrame* with duplicated column names would cause a stack overflow (GH21063)
- Bug in *pandas.core.groupby.GroupBy.ffill()* and *pandas.core.groupby.GroupBy.bfill()* where the fill within a grouping would not always be applied as intended due to the implementations' use of a non-stable sort (GH21207)
- Bug in *pandas.core.groupby.GroupBy.rank()* where results did not scale to 100% when specifying `method='dense'` and `pct=True`
- Bug in *pandas.DataFrame.rolling()* and *pandas.Series.rolling()* which incorrectly accepted a 0 window size rather than raising (GH21286)

Data-type specific

- Bug in *Series.str.replace()* where the method throws *TypeError* on Python 3.5.2 (GH21078)
- Bug in *Timedelta* where passing a float with a unit would prematurely round the float precision (GH14156)
- Bug in *pandas.testing.assert_index_equal()* which raised `AssertionError` incorrectly, when comparing two *CategoricalIndex* objects with param `check_categorical=False` (GH19776)

Sparse

- Bug in *SparseArray.shape* which previously only returned the shape *SparseArray.sp_values* (GH21126)

Indexing

- Bug in *Series.reset_index()* where appropriate error was not raised with an invalid level name (GH20925)
- Bug in *interval_range()* when `start`/`periods` or `end`/`periods` are specified with float `start` or `end` (GH21161)
- Bug in `MultiIndex.set_names()` where error raised for a `MultiIndex` with `nlevels == 1` (GH21149)
- Bug in *IntervalIndex* constructors where creating an `IntervalIndex` from categorical data was not fully supported (GH21243, GH21253)
- Bug in `MultiIndex.sort_index()` which was not guaranteed to sort correctly with `level=1`; this was also causing data misalignment in particular *DataFrame.stack()* operations (GH20994, GH20945, GH21052)

Plotting

- New keywords (sharex, sharey) to turn on/off sharing of x/y-axis by subplots generated with pandas.DataFrame().groupby().boxplot() (GH20968)

I/O

- Bug in IO methods specifying `compression='zip'` which produced uncompressed zip archives (GH17778, GH21144)
- Bug in *DataFrame.to_stata()* which prevented exporting DataFrames to buffers and most file-like objects (GH21041)
- Bug in *read_stata()* and `StataReader` which did not correctly decode utf-8 strings on Python 3 from Stata 14 files (dta version 118) (GH21244)
- Bug in IO JSON *read_json()* reading empty JSON schema with `orient='table'` back to *DataFrame* caused an error (GH21287)

Reshaping

- Bug in *concat()* where error was raised in concatenating *Series* with numpy scalar and tuple names (GH21015)
- Bug in *concat()* warning message providing the wrong guidance for future behavior (GH21101)

Other

- Tab completion on *Index* in IPython no longer outputs deprecation warnings (GH21125)
- Bug preventing pandas being used on Windows without C++ redistributable installed (GH21106)

Contributors

A total of 30 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam J. Stewart
- Adam Kim +
- Aly Sivji
- Chalmer Lowe +
- Damini Satya +
- Dr. Irv
- Gabe Fernando +
- Giftlin Rajaiah
- Jeff Reback
- Jeremy Schendel +
- Joris Van den Bossche
- Kalyan Gokhale +
- Kevin Sheppard
- Matthew Roeschke
- Max Kanter +
- Ming Li
- Pyry Kovanen +
- Stefano Cianciulli

- Tom Augspurger
- Uddeshya Singh +
- Wenhuan
- William Ayd
- chris-b1
- gfyoung
- h-vetinari
- nprad +
- ssikdar1 +
- tmnhat2001
- topper-123
- zertrin +

8.2.5 What's new in 0.23.0 (May 15, 2018)

This is a major release from 0.22.0 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- *Round-trippable JSON format with 'table' orient.*
- *Instantiation from dicts respects order for Python 3.6+.*
- *Dependent column arguments for assign.*
- *Merging / sorting on a combination of columns and index levels.*
- *Extending Pandas with custom types.*
- *Excluding unobserved categories from groupby.*
- *Changes to make output shape of DataFrame.apply consistent.*

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

> **Warning:** Starting January 1, 2019, pandas feature releases will support Python 3 only. See *Plan for dropping Python 2.7* for more.

What's new in v0.23.0

- *New features*
  - *JSON read/write round-trippable with* `orient='table'`
  - `.assign()` *accepts dependent arguments*
  - *Merging on a combination of columns and index levels*
  - *Sorting by a combination of columns and index levels*

New features

JSON read/write round-trippable with `orient='table'`

A `DataFrame` can now be written to and subsequently read back via JSON while preserving metadata through usage of the `orient='table'` argument (see GH18912 and GH9146). Previously, none of the available `orient` values guaranteed the preservation of dtypes and index names, amongst other metadata.

```
In [1]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
   ...:                     'bar': ['a', 'b', 'c', 'd'],
   ...:                     'baz': pd.date_range('2018-01-01', freq='d', periods=4),
   ...:                     'qux': pd.Categorical(['a', 'b', 'c', 'c'])},
   ...:                    index=pd.Index(range(4), name='idx'))
   ...:

In [2]: df
Out[2]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

[4 rows x 4 columns]

In [3]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo               int64
bar              object
baz      datetime64[ns]
qux            category
Length: 4, dtype: object

In [4]: df.to_json('test.json', orient='table')

In [5]: new_df = pd.read_json('test.json', orient='table')
```

(continues on next page)

(continued from previous page)

```
In [6]: new_df
Out[6]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

[4 rows x 4 columns]

In [7]: new_df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo             int64
bar            object
baz    datetime64[ns]
qux          category
Length: 4, dtype: object
```

Please note that the string *index* is not supported with the round trip format, as it is used by default in `write_json` to indicate a missing index name.

```
In [8]: df.index.name = 'index'

In [9]: df.to_json('test.json', orient='table')

In [10]: new_df = pd.read_json('test.json', orient='table')

In [11]: new_df
Out[11]:
   foo bar        baz qux
0    1   a 2018-01-01   a
1    2   b 2018-01-02   b
2    3   c 2018-01-03   c
3    4   d 2018-01-04   c

[4 rows x 4 columns]

In [12]: new_df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
foo             int64
bar            object
baz    datetime64[ns]
qux          category
Length: 4, dtype: object
```

`.assign()` accepts dependent arguments

The *`DataFrame.assign()`* now accepts dependent keyword arguments for python version later than 3.6 (see also PEP 468). Later keyword arguments may now refer to earlier ones if the argument is a callable. See the *documentation here* (GH14207)

```
In [13]: df = pd.DataFrame({'A': [1, 2, 3]})

In [14]: df
Out[14]:
   A
0  1
1  2
2  3

[3 rows x 1 columns]

In [15]: df.assign(B=df.A, C=lambda x: x['A'] + x['B'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]:
   A  B  C
0  1  1  2
1  2  2  4
2  3  3  6

[3 rows x 3 columns]
```

Warning: This may subtly change the behavior of your code when you're using `.assign()` to update an existing column. Previously, callables referring to other variables being updated would get the "old" values

Previous Behavior:

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

In [3]: df.assign(A=lambda df: df.A + 1, C=lambda df: df.A * -1)
Out[3]:
   A  C
0  2 -1
1  3 -2
2  4 -3
```

New Behavior:

```
In [16]: df.assign(A=df.A + 1, C=lambda df: df.A * -1)
Out[16]:
   A  C
0  2 -2
1  3 -3
2  4 -4

[3 rows x 2 columns]
```

Merging on a combination of columns and index levels

Strings passed to *DataFrame.merge()* as the `on`, `left_on`, and `right_on` parameters may now refer to either column names or index level names. This enables merging `DataFrame` instances on a combination of index levels and columns without resetting indexes. See the *Merge on columns and levels* documentation section. (GH14355)

```
In [17]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [18]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
   ....:                      'B': ['B0', 'B1', 'B2', 'B3'],
```

(continues on next page)

(continued from previous page)

```
   ....:                          'key2': ['K0', 'K1', 'K0', 'K1']},
   ....:                         index=left_index)
   ....:

In [19]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [20]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
   ....:                           'D': ['D0', 'D1', 'D2', 'D3'],
   ....:                           'key2': ['K0', 'K0', 'K0', 'K1']},
   ....:                          index=right_index)
   ....:

In [21]: left.merge(right, on=['key1', 'key2'])
Out[21]:
       A   B key2   C   D
key1
K0    A0  B0   K0  C0  D0
K1    A2  B2   K0  C1  D1
K2    A3  B3   K1  C3  D3

[3 rows x 5 columns]
```

Sorting by a combination of columns and index levels

Strings passed to *DataFrame.sort_values()* as the by parameter may now refer to either column names or index level names. This enables sorting DataFrame instances by a combination of index levels and columns without resetting indexes. See the *Sorting by Indexes and Values* documentation section. (GH14353)

```
# Build MultiIndex
In [22]: idx = pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('a', 2),
   ....:                                  ('b', 2), ('b', 1), ('b', 1)])
   ....:

In [23]: idx.names = ['first', 'second']

# Build DataFrame
In [24]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
   ....:                         index=idx)
   ....:

In [25]: df_multi
Out[25]:
              A
first second
a     1       6
      2       5
      2       4
b     2       3
      1       2
      1       1

[6 rows x 1 columns]

# Sort by 'second' (index) and 'A' (column)
In [26]: df_multi.sort_values(by=['second', 'A'])
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
              A
first second
b     1       1
      1       2
a     1       6
b     2       3
a     2       4
      2       5

[6 rows x 1 columns]
```

Extending Pandas with Custom Types (Experimental)

Pandas now supports storing array-like objects that aren't necessarily 1-D NumPy arrays as columns in a DataFrame or values in a Series. This allows third-party libraries to implement extensions to NumPy's types, similar to how pandas implemented categoricals, datetimes with timezones, periods, and intervals.

As a demonstration, we'll use cyberpandas, which provides an `IPArray` type for storing ip addresses.

```
In [1]: from cyberpandas import IPArray

In [2]: values = IPArray([
   ...:     0,
   ...:     3232235777,
   ...:     42540766452641154071740215577757643572
   ...: ])
   ...:
   ...:
```

`IPArray` isn't a normal 1-D NumPy array, but because it's a pandas *ExtensionArray*, it can be stored properly inside pandas' containers.

```
In [3]: ser = pd.Series(values)

In [4]: ser
Out[4]:
0                       0.0.0.0
1                   192.168.1.1
2    2001:db8:85a3::8a2e:370:7334
dtype: ip
```

Notice that the dtype is `ip`. The missing value semantics of the underlying array are respected:

```
In [5]: ser.isna()
Out[5]:
0     True
1    False
2    False
dtype: bool
```

For more, see the *extension types* documentation. If you build an extension array, publicize it on our *ecosystem page*.

New `observed` keyword for excluding unobserved categories in `groupby`

Grouping by a categorical includes the unobserved categories in the output. When grouping by multiple categorical columns, this means you get the cartesian product of all the categories, including combinations where there are no observations, which can result in a large number of groups. We have added a keyword `observed` to control this behavior, it defaults to `observed=False` for backward-compatibility. (GH14942, GH8138, GH15217, GH17594, GH8669, GH20583, GH20902)

```
In [27]: cat1 = pd.Categorical(["a", "a", "b", "b"],
   ....:                       categories=["a", "b", "z"], ordered=True)
   ....:

In [28]: cat2 = pd.Categorical(["c", "d", "c", "d"],
   ....:                       categories=["c", "d", "y"], ordered=True)
   ....:

In [29]: df = pd.DataFrame({"A": cat1, "B": cat2, "values": [1, 2, 3, 4]})

In [30]: df['C'] = ['foo', 'bar'] * 2

In [31]: df
Out[31]:
   A  B  values    C
0  a  c       1  foo
1  a  d       2  bar
2  b  c       3  foo
3  b  d       4  bar

[4 rows x 4 columns]
```

To show all values, the previous behavior:

```
In [32]: df.groupby(['A', 'B', 'C'], observed=False).count()
Out[32]:
         values
A B C
a c bar     NaN
    foo     1.0
  d bar     1.0
    foo     NaN
  y bar     NaN
    foo     NaN
b c bar     NaN
...         ...
  y foo     NaN
z c bar     NaN
    foo     NaN
  d bar     NaN
    foo     NaN
  y bar     NaN
    foo     NaN

[18 rows x 1 columns]
```

To show only observed values:

```
In [33]: df.groupby(['A', 'B', 'C'], observed=True).count()
```

(continues on next page)

(continued from previous page)

```
Out[33]:
         values
A B C
a c foo       1
  d bar       1
b c foo       1
  d bar       1

[4 rows x 1 columns]
```

For pivotting operations, this behavior is *already* controlled by the `dropna` keyword:

```
In [34]: cat1 = pd.Categorical(["a", "a", "b", "b"],
   ....:                       categories=["a", "b", "z"], ordered=True)
   ....:

In [35]: cat2 = pd.Categorical(["c", "d", "c", "d"],
   ....:                       categories=["c", "d", "y"], ordered=True)
   ....:

In [36]: df = pd.DataFrame({"A": cat1, "B": cat2, "values": [1, 2, 3, 4]})

In [37]: df
Out[37]:
   A  B  values
0  a  c       1
1  a  d       2
2  b  c       3
3  b  d       4

[4 rows x 3 columns]
```

```
In [38]: pd.pivot_table(df, values='values', index=['A', 'B'],
   ....:                dropna=True)
   ....:
Out[38]:
     values
A B
a c       1
  d       2
b c       3
  d       4

[4 rows x 1 columns]

In [39]: pd.pivot_table(df, values='values', index=['A', 'B'],
   ....:                dropna=False)
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     values
A B
a c     1.0
  d     2.0
  y     NaN
b c     3.0
```

(continues on next page)

(continued from previous page)

```
  d     4.0
  y     NaN
z c     NaN
  d     NaN
  y     NaN

[9 rows x 1 columns]
```

Rolling/Expanding.apply() accepts `raw=False` to pass a `Series` to the function

Series.rolling().apply(), *DataFrame.rolling().apply()*, *Series.expanding().apply()*, and *DataFrame.expanding().apply()* have gained a `raw=None` parameter. This is similar to `DataFame.apply()`. This parameter, if `True` allows one to send a `np.ndarray` to the applied function. If `False` a `Series` will be passed. The default is `None`, which preserves backward compatibility, so this will default to `True`, sending an `np.ndarray`. In a future version the default will be changed to `False`, sending a `Series`. (GH5071, GH20584)

```
In [40]: s = pd.Series(np.arange(5), np.arange(5) + 1)

In [41]: s
Out[41]:
1    0
2    1
3    2
4    3
5    4
Length: 5, dtype: int64
```

Pass a `Series`:

```
In [42]: s.rolling(2, min_periods=1).apply(lambda x: x.iloc[-1], raw=False)
Out[42]:
1    0.0
2    1.0
3    2.0
4    3.0
5    4.0
Length: 5, dtype: float64
```

Mimic the original behavior of passing a ndarray:

```
In [43]: s.rolling(2, min_periods=1).apply(lambda x: x[-1], raw=True)
Out[43]:
1    0.0
2    1.0
3    2.0
4    3.0
5    4.0
Length: 5, dtype: float64
```

`DataFrame.interpolate` has gained the `limit_area` kwarg

DataFrame.interpolate() has gained a `limit_area` parameter to allow further control of which

NaN s are replaced. Use limit_area='inside' to fill only NaNs surrounded by valid values or use limit_area='outside' to fill only NaN s outside the existing valid values while preserving those inside. (GH16284) See the *full documentation here*.

```
In [44]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan,
   ....:                  np.nan, 13, np.nan, np.nan])
   ....:

In [45]: ser
Out[45]:
0     NaN
1     NaN
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill one consecutive inside value in both directions

```
In [46]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
Out[46]:
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill all consecutive outside values backward

```
In [47]: ser.interpolate(limit_direction='backward', limit_area='outside')
Out[47]:
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
8     NaN
Length: 9, dtype: float64
```

Fill all consecutive outside values in both directions

```
In [48]: ser.interpolate(limit_direction='both', limit_area='outside')
Out[48]:
0     5.0
1     5.0
2     5.0
```

(continues on next page)

(continued from previous page)

```
3      NaN
4      NaN
5      NaN
6     13.0
7     13.0
8     13.0
Length: 9, dtype: float64
```

get_dummies now supports dtype argument

The *get_dummies()* now accepts a dtype argument, which specifies a dtype for the new columns. The default remains uint8. (GH18330)

```
In [49]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [5, 6]})

In [50]: pd.get_dummies(df, columns=['c']).dtypes
Out[50]:
a       int64
b       int64
c_5     uint8
c_6     uint8
Length: 4, dtype: object

In [51]: pd.get_dummies(df, columns=['c'], dtype=bool).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[51]:
↪
a       int64
b       int64
c_5      bool
c_6      bool
Length: 4, dtype: object
```

Timedelta mod method

mod (%) and divmod operations are now defined on Timedelta objects when operating with either timedelta-like or with numeric arguments. See the *documentation here*. (GH19365)

```
In [52]: td = pd.Timedelta(hours=37)

In [53]: td % pd.Timedelta(minutes=45)
Out[53]: Timedelta('0 days 00:15:00')
```

.rank() handles inf values when NaN are present

In previous versions, .rank() would assign inf elements NaN as their ranks. Now ranks are calculated properly. (GH6945)

```
In [54]: s = pd.Series([-np.inf, 0, 1, np.nan, np.inf])

In [55]: s
Out[55]:
```

(continues on next page)

(continued from previous page)

```
0   -inf
1    0.0
2    1.0
3    NaN
4    inf
Length: 5, dtype: float64
```

Previous Behavior:

```
In [11]: s.rank()
Out[11]:
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
dtype: float64
```

Current Behavior:

```
In [56]: s.rank()
Out[56]:
0    1.0
1    2.0
2    3.0
3    NaN
4    4.0
Length: 5, dtype: float64
```

Furthermore, previously if you rank `inf` or `-inf` values together with `NaN` values, the calculation won't distinguish `NaN` from infinity when using 'top' or 'bottom' argument.

```
In [57]: s = pd.Series([np.nan, np.nan, -np.inf, -np.inf])

In [58]: s
Out[58]:
0    NaN
1    NaN
2   -inf
3   -inf
Length: 4, dtype: float64
```

Previous Behavior:

```
In [15]: s.rank(na_option='top')
Out[15]:
0    2.5
1    2.5
2    2.5
3    2.5
dtype: float64
```

Current Behavior:

```
In [59]: s.rank(na_option='top')
Out[59]:
```

(continues on next page)

(continued from previous page)

```
0    1.5
1    1.5
2    3.5
3    3.5
Length: 4, dtype: float64
```

These bugs were squashed:

- Bug in *DataFrame.rank()* and *Series.rank()* when `method='dense'` and `pct=True` in which percentile ranks were not being used with the number of distinct observations (GH15630)
- Bug in *Series.rank()* and *DataFrame.rank()* when `ascending='False'` failed to return correct ranks for infinity if `NaN` were present (GH19538)
- Bug in `DataFrameGroupBy.rank()` where ranks were incorrect when both infinity and `NaN` were present (GH20561)

`Series.str.cat` has gained the `join` kwarg

Previously, *Series.str.cat()* did not – in contrast to most of `pandas` – align *Series* on their index before concatenation (see GH18657). The method has now gained a keyword `join` to control the manner of alignment, see examples below and *here*.

In v.0.23 *join* will default to None (meaning no alignment), but this default will change to `'left'` in a future version of pandas.

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'])

In [61]: t = pd.Series(['b', 'd', 'e', 'c'], index=[1, 3, 4, 2])

In [62]: s.str.cat(t)
Out[62]:
0    ab
1    bd
2    ce
3    dc
Length: 4, dtype: object

In [63]: s.str.cat(t, join='left', na_rep='-')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[63]:
0    a-
1    bb
2    cc
3    dd
Length: 4, dtype: object
```

Furthermore, *Series.str.cat()* now works for `CategoricalIndex` as well (previously raised a `ValueError`; see GH20842).

`DataFrame.astype` performs column-wise conversion to `Categorical`

DataFrame.astype() can now perform column-wise conversion to `Categorical` by supplying the string `'category'` or a `CategoricalDtype`. Previously, attempting this would raise a `NotImplementedError`. See the *Object Creation* section of the documentation for more details and examples. (GH12860, GH18099)

Supplying the string 'category' performs column-wise conversion, with only labels appearing in a given column set as categories:

```
In [64]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [65]: df = df.astype('category')

In [66]: df['A'].dtype
Out[66]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)

In [67]: df['B'].dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[67]:␣
↪CategoricalDtype(categories=['b', 'c', 'd'], ordered=False)
```

Supplying a CategoricalDtype will make the categories in each column consistent with the supplied dtype:

```
In [68]: from pandas.api.types import CategoricalDtype

In [69]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [70]: cdt = CategoricalDtype(categories=list('abcd'), ordered=True)

In [71]: df = df.astype(cdt)

In [72]: df['A'].dtype
Out[72]: CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)

In [73]: df['B'].dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[73]:␣
↪CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)
```

Other Enhancements

- Unary + now permitted for Series and DataFrame as numeric operator (GH16073)
- Better support for *to_excel()* output with the xlsxwriter engine. (GH16149)
- *pandas.tseries.frequencies.to_offset()* now accepts leading '+' signs e.g. '+1h'. (GH18171)
- MultiIndex.unique() now supports the level= argument, to get unique values from a specific index level (GH17896)
- *pandas.io.formats.style.Styler* now has method hide_index() to determine whether the index will be rendered in output (GH14194)
- *pandas.io.formats.style.Styler* now has method hide_columns() to determine whether columns will be hidden in output (GH14194)
- Improved wording of ValueError raised in *to_datetime()* when unit= is passed with a non-convertible value (GH14350)
- *Series.fillna()* now accepts a Series or a dict as a value for a categorical dtype (GH17033)
- *pandas.read_clipboard()* updated to use qtpy, falling back to PyQt5 and then PyQt4, adding compatibility with Python3 and multiple python-qt bindings (GH17722)
- Improved wording of ValueError raised in *read_csv()* when the usecols argument cannot match all columns. (GH17301)

- *DataFrame.corrwith()* now silently drops non-numeric columns when passed a Series. Before, an exception was raised (GH18570).
- *IntervalIndex* now supports time zone aware `Interval` objects (GH18537, GH18538)
- *Series()* / *DataFrame()* tab completion also returns identifiers in the first level of a *MultiIndex()*. (GH16326)
- *read_excel()* has gained the `nrows` parameter (GH16645)
- *DataFrame.append()* can now in more cases preserve the type of the calling dataframe's columns (e.g. if both are `CategoricalIndex`) (GH18359)
- *DataFrame.to_json()* and *Series.to_json()* now accept an `index` argument which allows the user to exclude the index from the JSON output (GH17394)
- `IntervalIndex.to_tuples()` has gained the `na_tuple` parameter to control whether NA is returned as a tuple of NA, or NA itself (GH18756)
- `Categorical.rename_categories`, `CategoricalIndex.rename_categories` and *Series.cat.rename_categories* can now take a callable as their argument (GH18862)
- *Interval* and *IntervalIndex* have gained a `length` attribute (GH18789)
- `Resampler` objects now have a functioning *pipe* method. Previously, calls to `pipe` were diverted to the `mean` method (GH17905).
- *is_scalar()* now returns `True` for `DateOffset` objects (GH18943).
- *DataFrame.pivot()* now accepts a list for the `values=` kwarg (GH17160).
- Added *pandas.api.extensions.register_dataframe_accessor()*, *pandas.api.extensions.register_series_accessor()*, and *pandas.api.extensions.register_index_accessor()*, accessor for libraries downstream of pandas to register custom accessors like `.cat` on pandas objects. See *Registering Custom Accessors* for more (GH14781).
- `IntervalIndex.astype` now supports conversions between subtypes when passed an `IntervalDtype` (GH19197)
- *IntervalIndex* and its associated constructor methods (`from_arrays`, `from_breaks`, `from_tuples`) have gained a `dtype` parameter (GH19262)
- Added *pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing()* and *pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing()* (GH17015)
- For subclassed `DataFrames`, *DataFrame.apply()* will now preserve the `Series` subclass (if defined) when passing the data to the applied function (GH19822)
- *DataFrame.from_dict()* now accepts a `columns` argument that can be used to specify the column names when `orient='index'` is used (GH18529)
- Added option `display.html.use_mathjax` so MathJax can be disabled when rendering tables in `Jupyter` notebooks (GH19856, GH19824)
- *DataFrame.replace()* now supports the `method` parameter, which can be used to specify the replacement method when `to_replace` is a scalar, list or tuple and `value` is `None` (GH19632)
- *Timestamp.month_name()*, *DatetimeIndex.month_name()*, and *Series.dt.month_name()* are now available (GH12805)
- *Timestamp.day_name()* and *DatetimeIndex.day_name()* are now available to return day names with a specified locale (GH12806)

- *DataFrame.to_sql()* now performs a multi-value insert if the underlying connection supports itk rather than inserting row by row. `SQLAlchemy` dialects supporting multi-value inserts include: `mysql`, `postgresql`, `sqlite` and any dialect with `supports_multivalues_insert`. (GH14315, GH8953)
- *read_html()* now accepts a `displayed_only` keyword argument to controls whether or not hidden elements are parsed (`True` by default) (GH20027)
- *read_html()* now reads all `<tbody>` elements in a `<table>`, not just the first. (GH20690)
- *quantile()* and *quantile()* now accept the `interpolation` keyword, `linear` by default (GH20497)
- zip compression is supported via `compression=zip` in *DataFrame.to_pickle()*, *Series.to_pickle()*, *DataFrame.to_csv()*, *Series.to_csv()*, *DataFrame.to_json()*, *Series.to_json()*. (GH17778)
- *WeekOfMonth* constructor now supports `n=0` (GH20517).
- *DataFrame* and *Series* now support matrix multiplication (`@`) operator (GH10259) for Python>=3.5
- Updated *DataFrame.to_gbq()* and *pandas.read_gbq()* signature and documentation to reflect changes from the Pandas-GBQ library version 0.4.0. Adds intersphinx mapping to Pandas-GBQ library. (GH20564)
- Added new writer for exporting Stata dta files in version 117, `StataWriter117`. This format supports exporting strings with lengths up to 2,000,000 characters (GH16450)
- `to_hdf()` and *read_hdf()* now accept an `errors` keyword argument to control encoding error handling (GH20835)
- *cut()* has gained the `duplicates='raise'|'drop'` option to control whether to raise on duplicated edges (GH20947)
- *date_range()*, *timedelta_range()*, and *interval_range()* now return a linearly spaced index if `start`, `stop`, and `periods` are specified, but `freq` is not. (GH20808, GH20983, GH20976)

Backwards incompatible API changes

Dependencies have increased minimum versions

We have updated our minimum supported versions of dependencies (GH15184). If installed, we now require:

Package	Minimum Version	Required	Issue
python-dateutil	2.5.0	X	GH15184
openpyxl	2.4.0		GH15184
beautifulsoup4	4.2.1		GH20082
setuptools	24.2.0		GH20698

Instantiation from dicts preserves dict insertion order for python 3.6+

Until Python 3.6, dicts in Python had no formally defined ordering. For Python version 3.6 and later, dicts are ordered by insertion order, see PEP 468. Pandas will use the dict's insertion order, when creating a `Series` or `DataFrame` from a dict and you're using Python version 3.6 or higher. (GH19884)

Previous Behavior (and current behavior if on Python < 3.6):

```
In [16]: pd.Series({'Income': 2000,
   ....:            'Expenses': -1500,
   ....:            'Taxes': -200,
   ....:            'Net result': 300})
Out[16]:
Expenses     -1500
Income        2000
Net result     300
Taxes         -200
dtype: int64
```

Note the Series above is ordered alphabetically by the index values.

New Behavior (for Python >= 3.6):

```
In [74]: pd.Series({'Income': 2000,
   ....:            'Expenses': -1500,
   ....:            'Taxes': -200,
   ....:            'Net result': 300})
   ....:
Out[74]:
Income        2000
Expenses     -1500
Taxes         -200
Net result     300
Length: 4, dtype: int64
```

Notice that the Series is now ordered by insertion order. This new behavior is used for all relevant pandas types (`Series`, `DataFrame`, `SparseSeries` and `SparseDataFrame`).

If you wish to retain the old behavior while using Python >= 3.6, you can use `.sort_index()`:

```
In [75]: pd.Series({'Income': 2000,
   ....:            'Expenses': -1500,
   ....:            'Taxes': -200,
   ....:            'Net result': 300}).sort_index()
   ....:
Out[75]:
Expenses     -1500
Income        2000
Net result     300
Taxes         -200
Length: 4, dtype: int64
```

Deprecate Panel

`Panel` was deprecated in the 0.20.x release, showing as a `DeprecationWarning`. Using `Panel` will now show a `FutureWarning`. 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. For more details see *Deprecate Panel* documentation. (GH13563, GH18324).

```
In [76]: p = tm.makePanel()

In [77]: p
Out[77]:
<class 'pandas.core.panel.Panel'>
```

(continues on next page)

(continued from previous page)

```
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 [78]: p.to_frame()
Out[78]:
                     ItemA     ItemB     ItemC
major      minor
2000-01-03 A      0.469112  0.721555  0.404705
           B     -1.135632  0.271860 -1.039268
           C      0.119209  0.276232 -1.344312
           D     -2.104569  0.113648 -0.109050
2000-01-04 A     -0.282863 -0.706771  0.577046
           B      1.212112 -0.424972 -0.370647
           C     -1.044236 -1.087401  0.844885
           D     -0.494929 -1.478427  1.643563
2000-01-05 A     -1.509059 -1.039575 -1.715002
           B     -0.173215  0.567020 -1.157892
           C     -0.861849 -0.673690  1.075770
           D      1.071804  0.524988 -1.469388

[12 rows x 3 columns]
```

Convert to an xarray DataArray

```
In [79]: p.to_xarray()
Out[79]:
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[[ 0.469112, -1.135632,  0.119209, -2.104569],
        [-0.282863,  1.212112, -1.044236, -0.494929],
        [-1.509059, -0.173215, -0.861849,  1.071804]],

       [[ 0.721555,  0.27186 ,  0.276232,  0.113648],
        [-0.706771, -0.424972, -1.087401, -1.478427],
        [-1.039575,  0.56702 , -0.67369 ,  0.524988]],

       [[ 0.404705, -1.039268, -1.344312, -0.10905 ],
        [ 0.577046, -0.370647,  0.844885,  1.643563],
        [-1.715002, -1.157892,  1.07577 , -1.469388]]])
Coordinates:
  * items       (items) object 'ItemA' 'ItemB' 'ItemC'
  * major_axis  (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05
  * minor_axis  (minor_axis) object 'A' 'B' 'C' 'D'
```

pandas.core.common removals

The following error & warning messages are removed from `pandas.core.common` (GH13634, GH19769):

- `PerformanceWarning`
- `UnsupportedFunctionCall`
- `UnsortedIndexError`

- `AbstractMethodError`

These are available from import from `pandas.errors` (since 0.19.0).

Changes to make output of `DataFrame.apply` consistent

DataFrame.apply() was inconsistent when applying an arbitrary user-defined-function that returned a list-like with `axis=1`. Several bugs and inconsistencies are resolved. If the applied function returns a Series, then pandas will return a DataFrame; otherwise a Series will be returned, this includes the case where a list-like (e.g. `tuple` or `list` is returned) (GH16353, GH17437, GH17970, GH17348, GH17892, GH18573, GH17602, GH18775, GH18901, GH18919).

```
In [80]: df = pd.DataFrame(np.tile(np.arange(3), 6).reshape(6, -1) + 1,
   ....:                   columns=['A', 'B', 'C'])
   ....:

In [81]: df
Out[81]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3

[6 rows x 3 columns]
```

Previous Behavior: if the returned shape happened to match the length of original columns, this would return a `DataFrame`. If the return shape did not match, a `Series` with lists was returned.

```
In [3]: df.apply(lambda x: [1, 2, 3], axis=1)
Out[3]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3

In [4]: df.apply(lambda x: [1, 2], axis=1)
Out[4]:
0    [1, 2]
1    [1, 2]
2    [1, 2]
3    [1, 2]
4    [1, 2]
5    [1, 2]
dtype: object
```

New Behavior: When the applied function returns a list-like, this will now *always* return a `Series`.

```
In [82]: df.apply(lambda x: [1, 2, 3], axis=1)
Out[82]:
0    [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
1    [1, 2, 3]
2    [1, 2, 3]
3    [1, 2, 3]
4    [1, 2, 3]
5    [1, 2, 3]
Length: 6, dtype: object

In [83]: df.apply(lambda x: [1, 2], axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    [1, 2]
1    [1, 2]
2    [1, 2]
3    [1, 2]
4    [1, 2]
5    [1, 2]
Length: 6, dtype: object
```

To have expanded columns, you can use `result_type='expand'`

```
In [84]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='expand')
Out[84]:
   0  1  2
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3

[6 rows x 3 columns]
```

To broadcast the result across the original columns (the old behaviour for list-likes of the correct length), you can use `result_type='broadcast'`. The shape must match the original columns.

```
In [85]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='broadcast')
Out[85]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3

[6 rows x 3 columns]
```

Returning a `Series` allows one to control the exact return structure and column names:

```
In [86]: df.apply(lambda x: pd.Series([1, 2, 3], index=['D', 'E', 'F']), axis=1)
Out[86]:
   D  E  F
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
```

(continues on next page)

(continued from previous page)

```
4  1  2  3
5  1  2  3

[6 rows x 3 columns]
```

Concatenation will no longer sort

In a future version of pandas *pandas.concat()* will no longer sort the non-concatenation axis when it is not already aligned. The current behavior is the same as the previous (sorting), but now a warning is issued when `sort` is not specified and the non-concatenation axis is not aligned (GH4588).

```
In [87]: df1 = pd.DataFrame({"a": [1, 2], "b": [1, 2]}, columns=['b', 'a'])

In [88]: df2 = pd.DataFrame({"a": [4, 5]})

In [89]: pd.concat([df1, df2])
Out[89]:
   a    b
0  1  1.0
1  2  2.0
0  4  NaN
1  5  NaN

[4 rows x 2 columns]
```

To keep the previous behavior (sorting) and silence the warning, pass `sort=True`

```
In [90]: pd.concat([df1, df2], sort=True)
Out[90]:
   a    b
0  1  1.0
1  2  2.0
0  4  NaN
1  5  NaN

[4 rows x 2 columns]
```

To accept the future behavior (no sorting), pass `sort=False`

Note that this change also applies to *DataFrame.append()*, which has also received a `sort` keyword for controlling this behavior.

Build Changes

- Building pandas for development now requires `cython >= 0.24` (GH18613)
- Building from source now explicitly requires `setuptools` in `setup.py` (GH18113)
- Updated conda recipe to be in compliance with conda-build 3.0+ (GH18002)

Index Division By Zero Fills Correctly

Division operations on `Index` and subclasses will now fill division of positive numbers by zero with `np.inf`, division of negative numbers by zero with `-np.inf` and *0 / 0* with `np.nan`. This matches existing `Series` behavior.

(GH19322, GH19347)

Previous Behavior:

```
In [6]: index = pd.Int64Index([-1, 0, 1])

In [7]: index / 0
Out[7]: Int64Index([0, 0, 0], dtype='int64')

# Previous behavior yielded different results depending on the type of zero in the␣
↪divisor
In [8]: index / 0.0
Out[8]: Float64Index([-inf, nan, inf], dtype='float64')

In [9]: index = pd.UInt64Index([0, 1])

In [10]: index / np.array([0, 0], dtype=np.uint64)
Out[10]: UInt64Index([0, 0], dtype='uint64')

In [11]: pd.RangeIndex(1, 5) / 0
ZeroDivisionError: integer division or modulo by zero
```

Current Behavior:

```
In [91]: index = pd.Int64Index([-1, 0, 1])

# division by zero gives -infinity where negative,
# +infinity where positive, and NaN for 0 / 0
In [92]: index / 0
Out[92]: Float64Index([-inf, nan, inf], dtype='float64')

# The result of division by zero should not depend on
# whether the zero is int or float
In [93]: index / 0.0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[93]: Float64Index([-inf,␣
↪nan, inf], dtype='float64')

In [94]: index = pd.UInt64Index([0, 1])

In [95]: index / np.array([0, 0], dtype=np.uint64)
Out[95]: Float64Index([nan, inf], dtype='float64')

In [96]: pd.RangeIndex(1, 5) / 0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[96]: Float64Index([inf, inf,␣
↪inf, inf], dtype='float64')
```

Extraction of matching patterns from strings

By default, extracting matching patterns from strings with `str.extract()` used to return a `Series` if a single group was being extracted (a `DataFrame` if more than one group was extracted). As of Pandas 0.23.0 `str.extract()` always returns a `DataFrame`, unless `expand` is set to `False`. Finally, `None` was an accepted value for the `expand` parameter (which was equivalent to `False`), but now raises a `ValueError`. (GH11386)

Previous Behavior:

```
In [1]: s = pd.Series(['number 10', '12 eggs'])
```

(continues on next page)

(continued from previous page)

```
In [2]: extracted = s.str.extract(r'.*(\d\d).*')

In [3]: extracted
Out [3]:
0     10
1     12
dtype: object

In [4]: type(extracted)
Out [4]:
pandas.core.series.Series
```

New Behavior:

```
In [97]: s = pd.Series(['number 10', '12 eggs'])

In [98]: extracted = s.str.extract(r'.*(\d\d).*')

In [99]: extracted
Out[99]:
    0
0  10
1  12

[2 rows x 1 columns]

In [100]: type(extracted)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[100]: pandas.core.frame.
↪DataFrame
```

To restore previous behavior, simply set `expand` to `False`:

```
In [101]: s = pd.Series(['number 10', '12 eggs'])

In [102]: extracted = s.str.extract(r'.*(\d\d).*', expand=False)

In [103]: extracted
Out[103]:
0     10
1     12
Length: 2, dtype: object

In [104]: type(extracted)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[104]: pandas.core.series.
↪Series
```

Default value for the `ordered` parameter of `CategoricalDtype`

The default value of the `ordered` parameter for `CategoricalDtype` has changed from `False` to `None` to allow updating of `categories` without impacting `ordered`. Behavior should remain consistent for downstream objects, such as *`Categorical`* (GH18790)

In previous versions, the default value for the `ordered` parameter was `False`. This could potentially lead to the `ordered` parameter unintentionally being changed from `True` to `False` when users attempt to update `categories` if `ordered` is not explicitly specified, as it would silently default to `False`. The new behavior for `ordered=None` is to retain the existing value of `ordered`.

New Behavior:

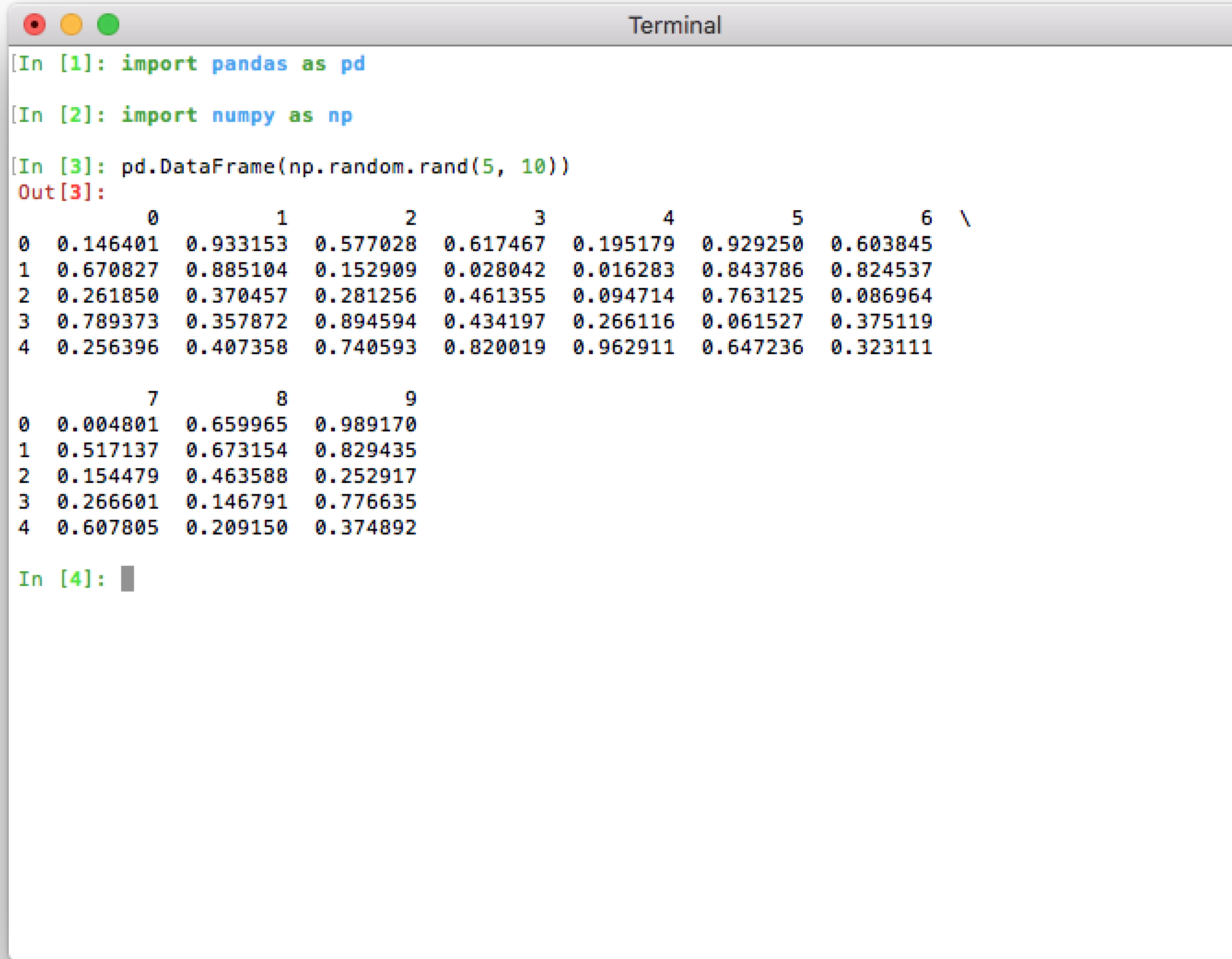
```
In [105]: from pandas.api.types import CategoricalDtype

In [106]: cat = pd.Categorical(list('abcaba'), ordered=True, categories=list('cba'))

In [107]: cat
Out[107]:
[a, b, c, a, b, a]
Categories (3, object): [c < b < a]

In [108]: cdt = CategoricalDtype(categories=list('cbad'))

In [109]: cat.astype(cdt)
Out[109]:
[a, b, c, a, b, a]
Categories (4, object): [c < b < a < d]
```

Notice in the example above that the converted `Categorical` has retained `ordered=True`. Had the default value for `ordered` remained as `False`, the converted `Categorical` would have become unordered, despite `ordered=False` never being explicitly specified. To change the value of `ordered`, explicitly pass it to the new dtype, e.g. `CategoricalDtype(categories=list('cbad'), ordered=False)`.

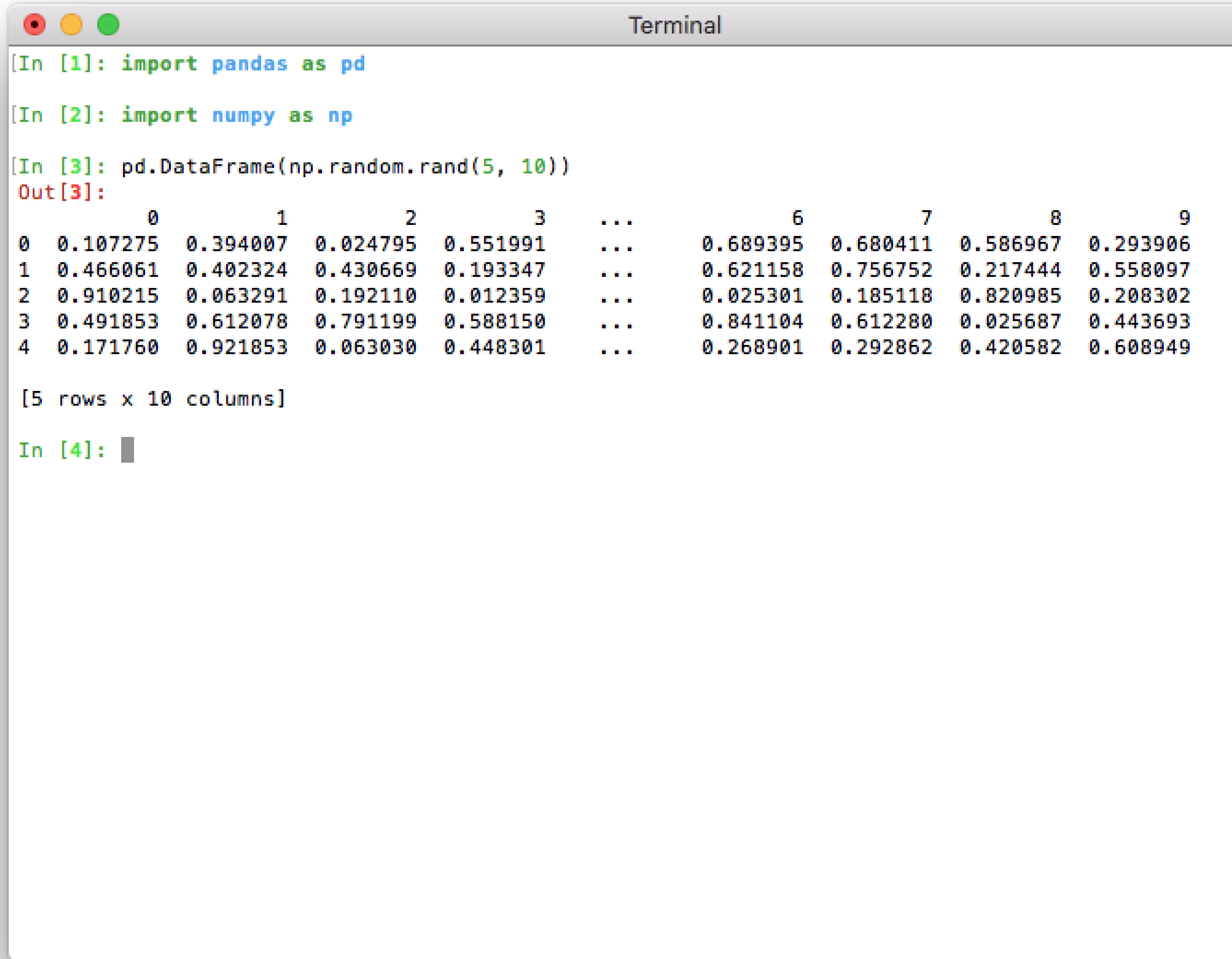
Note that the unintentional conversion of `ordered` discussed above did not arise in previous versions due to separate bugs that prevented `astype` from doing any type of category to category conversion (GH10696, GH18593). These bugs have been fixed in this release, and motivated changing the default value of `ordered`.

Better pretty-printing of DataFrames in a terminal

Previously, the default value for the maximum number of columns was `pd.options.display.max_columns=20`. This meant that relatively wide data frames would not fit within the terminal width, and pandas would introduce line breaks to display these 20 columns. This resulted in an output that was relatively difficult to read:



If Python runs in a terminal, the maximum number of columns is now determined automatically so that the printed data frame fits within the current terminal width (`pd.options.display.max_columns=0`) (GH17023). If Python runs as a Jupyter kernel (such as the Jupyter QtConsole or a Jupyter notebook, as well as in many IDEs), this value cannot be inferred automatically and is thus set to *20* as in previous versions. In a terminal, this results in a much nicer output:



Note that if you don't like the new default, you can always set this option yourself. To revert to the old setting, you can run this line:

```
pd.options.display.max_columns = 20
```

Datetimelike API Changes

- The default `Timedelta` constructor now accepts an `ISO 8601 Duration` string as an argument (GH19040)
- Subtracting `NaT` from a *Series* with `dtype='datetime64[ns]'` returns a `Series` with `dtype='timedelta64[ns]'` instead of `dtype='datetime64[ns]'` (GH18808)

- Addition or subtraction of `NaT` from *TimedeltaIndex* will return `TimedeltaIndex` instead of `DatetimeIndex` (GH19124)
- `DatetimeIndex.shift()` and `TimedeltaIndex.shift()` will now raise `NullFrequencyError` (which subclasses `ValueError`, which was raised in older versions) when the index object frequency is `None` (GH19147)
- Addition and subtraction of `NaN` from a *Series* with `dtype='timedelta64[ns]'` will raise a `TypeError` instead of treating the `NaN` as `NaT` (GH19274)
- `NaT` division with `datetime.timedelta` will now return `NaN` instead of raising (GH17876)
- Operations between a *Series* with dtype `dtype='datetime64[ns]'` and a *PeriodIndex* will correctly raises `TypeError` (GH18850)
- Subtraction of *Series* with timezone-aware `dtype='datetime64[ns]'` with mis-matched timezones will raise `TypeError` instead of `ValueError` (GH18817)
- *Timestamp* will no longer silently ignore unused or invalid `tz` or `tzinfo` keyword arguments (GH17690)
- *Timestamp* will no longer silently ignore invalid `freq` arguments (GH5168)
- `CacheableOffset` and `WeekDay` are no longer available in the `pandas.tseries.offsets` module (GH17830)
- `pandas.tseries.frequencies.get_freq_group()` and `pandas.tseries.frequencies.DAYS` are removed from the public API (GH18034)
- *Series.truncate()* and *DataFrame.truncate()* will raise a `ValueError` if the index is not sorted instead of an unhelpful `KeyError` (GH17935)
- *Series.first* and *DataFrame.first* will now raise a `TypeError` rather than `NotImplementedError` when index is not a *DatetimeIndex* (GH20725).
- *Series.last* and *DataFrame.last* will now raise a `TypeError` rather than `NotImplementedError` when index is not a *DatetimeIndex* (GH20725).
- Restricted `DateOffset` keyword arguments. Previously, `DateOffset` subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176, GH18226).
- *pandas.merge()* provides a more informative error message when trying to merge on timezone-aware and timezone-naive columns (GH15800)
- For *DatetimeIndex* and *TimedeltaIndex* with `freq=None`, addition or subtraction of integer-dtyped array or `Index` will raise `NullFrequencyError` instead of `TypeError` (GH19895)
- *Timestamp* constructor now accepts a *nanosecond* keyword or positional argument (GH18898)
- *DatetimeIndex* will now raise an `AttributeError` when the `tz` attribute is set after instantiation (GH3746)
- *DatetimeIndex* with a `pytz` timezone will now return a consistent `pytz` timezone (GH18595)

Other API Changes

- *Series.astype()* and *Index.astype()* with an incompatible dtype will now raise a `TypeError` rather than a `ValueError` (GH18231)
- `Series` construction with an `object` dtyped tz-aware datetime and `dtype=object` specified, will now return an `object` dtyped `Series`, previously this would infer the datetime dtype (GH18231)

- A *Series* of dtype=category constructed from an empty dict will now have categories of dtype=object rather than dtype=float64, consistently with the case in which an empty list is passed (GH18515)
- All-NaN levels in a MultiIndex are now assigned float rather than object dtype, promoting consistency with Index (GH17929).
- Levels names of a MultiIndex (when not None) are now required to be unique: trying to create a MultiIndex with repeated names will raise a ValueError (GH18872)
- Both construction and renaming of Index/MultiIndex with non-hashable name/names will now raise TypeError (GH20527)
- *Index.map()* can now accept Series and dictionary input objects (GH12756, GH18482, GH18509).
- *DataFrame.unstack()* will now default to filling with np.nan for object columns. (GH12815)
- *IntervalIndex* constructor will raise if the closed parameter conflicts with how the input data is inferred to be closed (GH18421)
- Inserting missing values into indexes will work for all types of indexes and automatically insert the correct type of missing value (NaN, NaT, etc.) regardless of the type passed in (GH18295)
- When created with duplicate labels, MultiIndex now raises a ValueError. (GH17464)
- *Series.fillna()* now raises a TypeError instead of a ValueError when passed a list, tuple or DataFrame as a value (GH18293)
- *pandas.DataFrame.merge()* no longer casts a float column to object when merging on int and float columns (GH16572)
- *pandas.merge()* now raises a ValueError when trying to merge on incompatible data types (GH9780)
- The default NA value for *UInt64Index* has changed from 0 to NaN, which impacts methods that mask with NA, such as UInt64Index.where() (GH18398)
- Refactored setup.py to use find_packages instead of explicitly listing out all subpackages (GH18535)
- Rearranged the order of keyword arguments in *read_excel()* to align with *read_csv()* (GH16672)
- *wide_to_long()* previously kept numeric-like suffixes as object dtype. Now they are cast to numeric if possible (GH17627)
- In *read_excel()*, the comment argument is now exposed as a named parameter (GH18735)
- Rearranged the order of keyword arguments in *read_excel()* to align with *read_csv()* (GH16672)
- The options html.border and mode.use_inf_as_null were deprecated in prior versions, these will now show FutureWarning rather than a DeprecationWarning (GH19003)
- *IntervalIndex* and IntervalDtype no longer support categorical, object, and string subtypes (GH19016)
- IntervalDtype now returns True when compared against 'interval' regardless of subtype, and IntervalDtype.name now returns 'interval' regardless of subtype (GH18980)
- KeyError now raises instead of ValueError in *drop()*, *drop()*, *drop()*, *drop()* when dropping a non-existent element in an axis with duplicates (GH19186)
- *Series.to_csv()* now accepts a compression argument that works in the same way as the compression argument in *DataFrame.to_csv()* (GH18958)
- Set operations (union, difference...) on *IntervalIndex* with incompatible index types will now raise a TypeError rather than a ValueError (GH19329)

- `DateOffset` objects render more simply, e.g. `<DateOffset: days=1>` instead of `<DateOffset: kwds={'days': 1}>` (GH19403)
- `Categorical.fillna` now validates its `value` and `method` keyword arguments. It now raises when both or none are specified, matching the behavior of *Series.fillna()* (GH19682)
- `pd.to_datetime('today')` now returns a datetime, consistent with `pd.Timestamp('today')`; previously `pd.to_datetime('today')` returned a `.normalized()` datetime (GH19935)
- *Series.str.replace()* now takes an optional *regex* keyword which, when set to `False`, uses literal string replacement rather than regex replacement (GH16808)
- *DatetimeIndex.strftime()* and *PeriodIndex.strftime()* now return an `Index` instead of a numpy array to be consistent with similar accessors (GH20127)
- Constructing a Series from a list of length 1 no longer broadcasts this list when a longer index is specified (GH19714, GH20391).
- *DataFrame.to_dict()* with `orient='index'` no longer casts int columns to float for a DataFrame with only int and float columns (GH18580)
- A user-defined-function that is passed to *Series.rolling().aggregate()*, *DataFrame.rolling().aggregate()*, or its expanding cousins, will now *always* be passed a `Series`, rather than a `np.array`; `.apply()` only has the `raw` keyword, see *here*. This is consistent with the signatures of `.aggregate()` across pandas (GH20584)
- Rolling and Expanding types raise `NotImplementedError` upon iteration (GH11704).

Deprecations

- `Series.from_array` and `SparseSeries.from_array` are deprecated. Use the normal constructor `Series(..)` and `SparseSeries(..)` instead (GH18213).
- `DataFrame.as_matrix` is deprecated. Use `DataFrame.values` instead (GH18458).
- `Series.asobject`, `DatetimeIndex.asobject`, `PeriodIndex.asobject` and `TimeDeltaIndex.asobject` have been deprecated. Use `.astype(object)` instead (GH18572)
- Grouping by a tuple of keys now emits a `FutureWarning` and is deprecated. In the future, a tuple passed to `'by'` will always refer to a single key that is the actual tuple, instead of treating the tuple as multiple keys. To retain the previous behavior, use a list instead of a tuple (GH18314)
- `Series.valid` is deprecated. Use *Series.dropna()* instead (GH18800).
- *read_excel()* has deprecated the `skip_footer` parameter. Use `skipfooter` instead (GH18836)
- *ExcelFile.parse()* has deprecated `sheetname` in favor of `sheet_name` for consistency with *read_excel()* (GH20920).
- The `is_copy` attribute is deprecated and will be removed in a future version (GH18801).
- `IntervalIndex.from_intervals` is deprecated in favor of the *IntervalIndex* constructor (GH19263)
- `DataFrame.from_items` is deprecated. Use *DataFrame.from_dict()* instead, or `DataFrame.from_dict(OrderedDict())` if you wish to preserve the key order (GH17320, GH17312)
- Indexing a *MultiIndex* or a `FloatIndex` with a list containing some missing keys will now show a `FutureWarning`, which is consistent with other types of indexes (GH17758).
- The `broadcast` parameter of `.apply()` is deprecated in favor of `result_type='broadcast'` (GH18577)

- The `reduce` parameter of `.apply()` is deprecated in favor of `result_type='reduce'` (GH18577)
- The `order` parameter of *factorize()* is deprecated and will be removed in a future release (GH19727)
- *Timestamp.weekday_name*, `DatetimeIndex.weekday_name`, and `Series.dt.weekday_name` are deprecated in favor of *Timestamp.day_name()*, *DatetimeIndex.day_name()*, and *Series.dt.day_name()* (GH12806)
- `pandas.tseries.plotting.tsplot` is deprecated. Use *Series.plot()* instead (GH18627)
- `Index.summary()` is deprecated and will be removed in a future version (GH18217)
- `NDFrame.get_ftype_counts()` is deprecated and will be removed in a future version (GH18243)
- The `convert_datetime64` parameter in *DataFrame.to_records()* has been deprecated and will be removed in a future version. The NumPy bug motivating this parameter has been resolved. The default value for this parameter has also changed from `True` to `None` (GH18160).
- *Series.rolling().apply()*, *DataFrame.rolling().apply()*, *Series.expanding().apply()*, and *DataFrame.expanding().apply()* have deprecated passing an `np.array` by default. One will need to pass the new `raw` parameter to be explicit about what is passed (GH20584)
- The `data`, `base`, `strides`, `flags` and `itemsize` properties of the `Series` and `Index` classes have been deprecated and will be removed in a future version (GH20419).
- `DatetimeIndex.offset` is deprecated. Use `DatetimeIndex.freq` instead (GH20716)
- Floor division between an integer ndarray and a *Timedelta* is deprecated. Divide by *Timedelta.value* instead (GH19761)
- Setting `PeriodIndex.freq` (which was not guaranteed to work correctly) is deprecated. Use *PeriodIndex.asfreq()* instead (GH20678)
- `Index.get_duplicates()` is deprecated and will be removed in a future version (GH20239)
- The previous default behavior of negative indices in `Categorical.take` is deprecated. In a future version it will change from meaning missing values to meaning positional indices from the right. The future behavior is consistent with *Series.take()* (GH20664).
- Passing multiple axes to the `axis` parameter in *DataFrame.dropna()* has been deprecated and will be removed in a future version (GH20987)

Removal of prior version deprecations/changes

- Warnings against the obsolete usage `Categorical(codes, categories)`, which were emitted for instance when the first two arguments to `Categorical()` had different dtypes, and recommended the use of `Categorical.from_codes`, have now been removed (GH8074)
- The `levels` and `labels` attributes of a `MultiIndex` can no longer be set directly (GH4039).
- `pd.tseries.util.pivot_annual` has been removed (deprecated since v0.19). Use `pivot_table` instead (GH18370)
- `pd.tseries.util.isleapyear` has been removed (deprecated since v0.19). Use `.is_leap_year` property in Datetime-likes instead (GH18370)
- `pd.ordered_merge` has been removed (deprecated since v0.19). Use `pd.merge_ordered` instead (GH18459)
- The `SparseList` class has been removed (GH14007)
- The `pandas.io.wb` and `pandas.io.data` stub modules have been removed (GH13735)
- `Categorical.from_array` has been removed (GH13854)

- The `freq` and `how` parameters have been removed from the `rolling`/`expanding`/`ewm` methods of DataFrame and Series (deprecated since v0.18). Instead, resample before calling the methods. (GH18601 & GH18668)
- `DatetimeIndex.to_datetime`, `Timestamp.to_datetime`, `PeriodIndex.to_datetime`, and `Index.to_datetime` have been removed (GH8254, GH14096, GH14113)
- *read_csv()* has dropped the `skip_footer` parameter (GH13386)
- *read_csv()* has dropped the `as_recarray` parameter (GH13373)
- *read_csv()* has dropped the `buffer_lines` parameter (GH13360)
- *read_csv()* has dropped the `compact_ints` and `use_unsigned` parameters (GH13323)
- The `Timestamp` class has dropped the `offset` attribute in favor of `freq` (GH13593)
- The `Series`, `Categorical`, and `Index` classes have dropped the `reshape` method (GH13012)
- `pandas.tseries.frequencies.get_standard_freq` has been removed in favor of `pandas.tseries.frequencies.to_offset(freq).rule_code` (GH13874)
- The `freqstr` keyword has been removed from `pandas.tseries.frequencies.to_offset` in favor of `freq` (GH13874)
- The `Panel4D` and `PanelND` classes have been removed (GH13776)
- The `Panel` class has dropped the `to_long` and `toLong` methods (GH19077)
- The options `display.line_with` and `display.height` are removed in favor of `display.width` and `display.max_rows` respectively (GH4391, GH19107)
- The `labels` attribute of the `Categorical` class has been removed in favor of *Categorical.codes* (GH7768)
- The `flavor` parameter have been removed from func:*to_sql* method (GH13611)
- The modules `pandas.tools.hashing` and `pandas.util.hashing` have been removed (GH16223)
- The top-level functions `pd.rolling_*`, `pd.expanding_*` and `pd.ewm*` have been removed (Deprecated since v0.18). Instead, use the DataFrame/Series methods *rolling*, *expanding* and *ewm* (GH18723)
- Imports from `pandas.core.common` for functions such as `is_datetime64_dtype` are now removed. These are located in `pandas.api.types`. (GH13634, GH19769)
- The `infer_dst` keyword in *Series.tz_localize()*, *DatetimeIndex.tz_localize()* and *DatetimeIndex* have been removed. `infer_dst=True` is equivalent to `ambiguous='infer'`, and `infer_dst=False` to `ambiguous='raise'` (GH7963).
- When `.resample()` was changed from an eager to a lazy operation, like `.groupby()` in v0.18.0, we put in place compatibility (with a `FutureWarning`), so operations would continue to work. This is now fully removed, so a `Resampler` will no longer forward compat operations (GH20554)
- Remove long deprecated `axis=None` parameter from `.replace()` (GH20271)

Performance Improvements

- Indexers on `Series` or `DataFrame` no longer create a reference cycle (GH17956)
- Added a keyword argument, `cache`, to *to_datetime()* that improved the performance of converting duplicate datetime arguments (GH11665)
- `DateOffset` arithmetic performance is improved (GH18218)

- Converting a `Series` of `Timedelta` objects to days, seconds, etc... sped up through vectorization of underlying methods (GH18092)
- Improved performance of `.map()` with a `Series/dict` input (GH15081)
- The overridden `Timedelta` properties of days, seconds and microseconds have been removed, leveraging their built-in Python versions instead (GH18242)
- `Series` construction will reduce the number of copies made of the input data in certain cases (GH17449)
- Improved performance of *Series.dt.date()* and *DatetimeIndex.date()* (GH18058)
- Improved performance of *Series.dt.time()* and *DatetimeIndex.time()* (GH18461)
- Improved performance of `IntervalIndex.symmetric_difference()` (GH18475)
- Improved performance of `DatetimeIndex` and `Series` arithmetic operations with Business-Month and Business-Quarter frequencies (GH18489)
- *Series()* / *DataFrame()* tab completion limits to 100 values, for better performance. (GH18587)
- Improved performance of *DataFrame.median()* with `axis=1` when bottleneck is not installed (GH16468)
- Improved performance of *MultiIndex.get_loc()* for large indexes, at the cost of a reduction in performance for small ones (GH18519)
- Improved performance of *MultiIndex.remove_unused_levels()* when there are no unused levels, at the cost of a reduction in performance when there are (GH19289)
- Improved performance of *Index.get_loc()* for non-unique indexes (GH19478)
- Improved performance of pairwise `.rolling()` and `.expanding()` with `.cov()` and `.corr()` operations (GH17917)
- Improved performance of *pandas.core.groupby.GroupBy.rank()* (GH15779)
- Improved performance of variable `.rolling()` on `.min()` and `.max()` (GH19521)
- Improved performance of *pandas.core.groupby.GroupBy.ffill()* and *pandas.core.groupby.GroupBy.bfill()* (GH11296)
- Improved performance of *pandas.core.groupby.GroupBy.any()* and *pandas.core.groupby.GroupBy.all()* (GH15435)
- Improved performance of *pandas.core.groupby.GroupBy.pct_change()* (GH19165)
- Improved performance of *Series.isin()* in the case of categorical dtypes (GH20003)
- Improved performance of `getattr(Series, attr)` when the Series has certain index types. This manifested in slow printing of large Series with a `DatetimeIndex` (GH19764)
- Fixed a performance regression for `GroupBy.nth()` and `GroupBy.last()` with some object columns (GH19283)
- Improved performance of `pandas.core.arrays.Categorical.from_codes()` (GH18501)

Documentation Changes

Thanks to all of the contributors who participated in the Pandas Documentation Sprint, which took place on March 10th. We had about 500 participants from over 30 locations across the world. You should notice that many of the *API docstrings* have greatly improved.

There were too many simultaneous contributions to include a release note for each improvement, but this GitHub search should give you an idea of how many docstrings were improved.

Special thanks to Marc Garcia for organizing the sprint. For more information, read the NumFOCUS blogpost recapping the sprint.

- Changed spelling of "numpy" to "NumPy", and "python" to "Python". (GH19017)
- Consistency when introducing code samples, using either colon or period. Rewrote some sentences for greater clarity, added more dynamic references to functions, methods and classes. (GH18941, GH18948, GH18973, GH19017)
- Added a reference to `DataFrame.assign()` in the concatenate section of the merging documentation (GH18665)

Bug Fixes

Categorical

> **Warning:** A class of bugs were introduced in pandas 0.21 with `CategoricalDtype` that affects the correctness of operations like `merge`, `concat`, and indexing when comparing multiple unordered `Categorical` arrays that have the same categories, but in a different order. We highly recommend upgrading or manually aligning your categories before doing these operations.

- Bug in `Categorical.equals` returning the wrong result when comparing two unordered `Categorical` arrays with the same categories, but in a different order (GH16603)
- Bug in `pandas.api.types.union_categoricals()` returning the wrong result when for unordered categoricals with the categories in a different order. This affected `pandas.concat()` with Categorical data (GH19096).
- Bug in `pandas.merge()` returning the wrong result when joining on an unordered `Categorical` that had the same categories but in a different order (GH19551)
- Bug in `CategoricalIndex.get_indexer()` returning the wrong result when `target` was an unordered `Categorical` that had the same categories as `self` but in a different order (GH19551)
- Bug in `Index.astype()` with a categorical dtype where the resultant index is not converted to a `CategoricalIndex` for all types of index (GH18630)
- Bug in `Series.astype()` and `Categorical.astype()` where an existing categorical data does not get updated (GH10696, GH18593)
- Bug in `Series.str.split()` with `expand=True` incorrectly raising an IndexError on empty strings (GH20002).
- Bug in `Index` constructor with `dtype=CategoricalDtype(...)` where `categories` and `ordered` are not maintained (GH19032)
- Bug in `Series` constructor with scalar and `dtype=CategoricalDtype(...)` where `categories` and `ordered` are not maintained (GH19565)
- Bug in `Categorical.__iter__` not converting to Python types (GH19909)
- Bug in `pandas.factorize()` returning the unique codes for the `uniques`. This now returns a `Categorical` with the same dtype as the input (GH19721)
- Bug in `pandas.factorize()` including an item for missing values in the `uniques` return value (GH19721)
- Bug in `Series.take()` with categorical data interpreting `-1` in *indices* as missing value markers, rather than the last element of the Series (GH20664)

Datetimelike

- Bug in `Series.__sub__()` subtracting a non-nanosecond `np.datetime64` object from a `Series` gave incorrect results (GH7996)
- Bug in *DatetimeIndex*, *TimedeltaIndex* addition and subtraction of zero-dimensional integer arrays gave incorrect results (GH19012)
- Bug in *DatetimeIndex* and *TimedeltaIndex* where adding or subtracting an array-like of `DateOffset` objects either raised (`np.array`, `pd.Index`) or broadcast incorrectly (`pd.Series`) (GH18849)
- Bug in `Series.__add__()` adding Series with dtype `timedelta64[ns]` to a timezone-aware `DatetimeIndex` incorrectly dropped timezone information (GH13905)
- Adding a `Period` object to a `datetime` or `Timestamp` object will now correctly raise a `TypeError` (GH17983)
- Bug in *Timestamp* where comparison with an array of `Timestamp` objects would result in a `RecursionError` (GH15183)
- Bug in *Series* floor-division where operating on a scalar `timedelta` raises an exception (GH18846)
- Bug in *DatetimeIndex* where the repr was not showing high-precision time values at the end of a day (e.g., 23:59:59.999999999) (GH19030)
- Bug in `.astype()` to non-ns timedelta units would hold the incorrect dtype (GH19176, GH19223, GH12425)
- Bug in subtracting *Series* from `NaT` incorrectly returning `NaT` (GH19158)
- Bug in *Series.truncate()* which raises `TypeError` with a monotonic `PeriodIndex` (GH17717)
- Bug in *pct_change()* using `periods` and `freq` returned different length outputs (GH7292)
- Bug in comparison of *DatetimeIndex* against `None` or `datetime.date` objects raising `TypeError` for `==` and `!=` comparisons instead of all-`False` and all-`True`, respectively (GH19301)
- Bug in *Timestamp* and *to_datetime()* where a string representing a barely out-of-bounds timestamp would be incorrectly rounded down instead of raising `OutOfBoundsDatetime` (GH19382)
- Bug in *Timestamp.floor()* *DatetimeIndex.floor()* where time stamps far in the future and past were not rounded correctly (GH19206)
- Bug in *to_datetime()* where passing an out-of-bounds datetime with `errors='coerce'` and `utc=True` would raise `OutOfBoundsDatetime` instead of parsing to `NaT` (GH19612)
- Bug in *DatetimeIndex* and *TimedeltaIndex* addition and subtraction where name of the returned object was not always set consistently. (GH19744)
- Bug in *DatetimeIndex* and *TimedeltaIndex* addition and subtraction where operations with numpy arrays raised `TypeError` (GH19847)
- Bug in *DatetimeIndex* and *TimedeltaIndex* where setting the `freq` attribute was not fully supported (GH20678)

Timedelta

- Bug in `Timedelta.__mul__()` where multiplying by `NaT` returned `NaT` instead of raising a `TypeError` (GH19819)
- Bug in *Series* with `dtype='timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` had results cast to `dtype='int64'` (GH17250)

- Bug in *Series* with `dtype='timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` could return a `Series` with an incorrect name (GH19043)
- Bug in `Timedelta.__floordiv__()` and `Timedelta.__rfloordiv__()` dividing by many incompatible numpy objects was incorrectly allowed (GH18846)
- Bug where dividing a scalar timedelta-like object with *TimedeltaIndex* performed the reciprocal operation (GH19125)
- Bug in *TimedeltaIndex* where division by a `Series` would return a `TimedeltaIndex` instead of a `Series` (GH19042)
- Bug in `Timedelta.__add__()`, `Timedelta.__sub__()` where adding or subtracting a `np.timedelta64` object would return another `np.timedelta64` instead of a `Timedelta` (GH19738)
- Bug in `Timedelta.__floordiv__()`, `Timedelta.__rfloordiv__()` where operating with a `Tick` object would raise a `TypeError` instead of returning a numeric value (GH19738)
- Bug in *Period.asfreq()* where periods near `datetime(1, 1, 1)` could be converted incorrectly (GH19643, GH19834)
- Bug in *Timedelta.total_seconds()* causing precision errors, for example `Timedelta('30S').total_seconds()==30.000000000000004` (GH19458)
- Bug in `Timedelta.__rmod__()` where operating with a `numpy.timedelta64` returned a `timedelta64` object instead of a `Timedelta` (GH19820)
- Multiplication of *TimedeltaIndex* by `TimedeltaIndex` will now raise `TypeError` instead of raising `ValueError` in cases of length mis-match (GH19333)
- Bug in indexing a *TimedeltaIndex* with a `np.timedelta64` object which was raising a `TypeError` (GH20393)

Timezones

- Bug in creating a `Series` from an array that contains both tz-naive and tz-aware values will result in a `Series` whose dtype is tz-aware instead of object (GH16406)
- Bug in comparison of timezone-aware *DatetimeIndex* against `NaT` incorrectly raising `TypeError` (GH19276)
- Bug in `DatetimeIndex.astype()` when converting between timezone aware dtypes, and converting from timezone aware to naive (GH18951)
- Bug in comparing *DatetimeIndex*, which failed to raise `TypeError` when attempting to compare timezone-aware and timezone-naive datetimelike objects (GH18162)
- Bug in localization of a naive, datetime string in a `Series` constructor with a `datetime64[ns, tz]` dtype (GH174151)
- *Timestamp.replace()* will now handle Daylight Savings transitions gracefully (GH18319)
- Bug in tz-aware *DatetimeIndex* where addition/subtraction with a *TimedeltaIndex* or array with `dtype='timedelta64[ns]'` was incorrect (GH17558)
- Bug in `DatetimeIndex.insert()` where inserting `NaT` into a timezone-aware index incorrectly raised (GH16357)
- Bug in *DataFrame* constructor, where tz-aware Datetimeindex and a given column name will result in an empty `DataFrame` (GH19157)

- Bug in *Timestamp.tz_localize()* where localizing a timestamp near the minimum or maximum valid values could overflow and return a timestamp with an incorrect nanosecond value (GH12677)
- Bug when iterating over *DatetimeIndex* that was localized with fixed timezone offset that rounded nanosecond precision to microseconds (GH19603)
- Bug in *DataFrame.diff()* that raised an IndexError with tz-aware values (GH18578)
- Bug in *melt()* that converted tz-aware dtypes to tz-naive (GH15785)
- Bug in Dataframe.count() that raised an ValueError, if Dataframe.dropna() was called for a single column with timezone-aware values. (GH13407)

Offsets

- Bug in WeekOfMonth and Week where addition and subtraction did not roll correctly (GH18510, GH18672, GH18864)
- Bug in WeekOfMonth and LastWeekOfMonth where default keyword arguments for constructor raised ValueError (GH19142)
- Bug in FY5253Quarter, LastWeekOfMonth where rollback and rollforward behavior was inconsistent with addition and subtraction behavior (GH18854)
- Bug in FY5253 where datetime addition and subtraction incremented incorrectly for dates on the year-end but not normalized to midnight (GH18854)
- Bug in FY5253 where date offsets could incorrectly raise an AssertionError in arithmetic operations (GH14774)

Numeric

- Bug in *Series* constructor with an int or float list where specifying dtype=str, dtype='str' or dtype='U' failed to convert the data elements to strings (GH16605)
- Bug in *Index* multiplication and division methods where operating with a Series would return an Index object instead of a Series object (GH19042)
- Bug in the *DataFrame* constructor in which data containing very large positive or very large negative numbers was causing OverflowError (GH18584)
- Bug in *Index* constructor with dtype='uint64' where int-like floats were not coerced to *UInt64Index* (GH18400)
- Bug in *DataFrame* flex arithmetic (e.g. df.add(other, fill_value=foo)) with a fill_value other than None failed to raise NotImplementedError in corner cases where either the frame or other has length zero (GH19522)
- Multiplication and division of numeric-dtyped *Index* objects with timedelta-like scalars returns TimedeltaIndex instead of raising TypeError (GH19333)
- Bug where NaN was returned instead of 0 by *Series.pct_change()* and *DataFrame.pct_change()* when fill_method is not None (GH19873)

Strings

- Bug in *Series.str.get()* with a dictionary in the values and the index not in the keys, raising *KeyError* (GH20671)

Indexing

- Bug in *Index* construction from list of mixed type tuples (GH18505)
- Bug in *Index.drop()* when passing a list of both tuples and non-tuples (GH18304)
- Bug in *DataFrame.drop()*, *Panel.drop()*, *Series.drop()*, *Index.drop()* where no `KeyError` is raised when dropping a non-existent element from an axis that contains duplicates (GH19186)
- Bug in indexing a datetimelike `Index` that raised `ValueError` instead of `IndexError` (GH18386).
- *Index.to_series()* now accepts `index` and `name` kwargs (GH18699)
- *DatetimeIndex.to_series()* now accepts `index` and `name` kwargs (GH18699)
- Bug in indexing non-scalar value from `Series` having non-unique `Index` will return value flattened (GH17610)
- Bug in indexing with iterator containing only missing keys, which raised no error (GH20748)
- Fixed inconsistency in `.ix` between list and scalar keys when the index has integer dtype and does not include the desired keys (GH20753)
- Bug in `__setitem__` when indexing a *DataFrame* with a 2-d boolean ndarray (GH18582)
- Bug in `str.extractall` when there were no matches empty *Index* was returned instead of appropriate *MultiIndex* (GH19034)
- Bug in *IntervalIndex* where empty and purely NA data was constructed inconsistently depending on the construction method (GH18421)
- Bug in `IntervalIndex.symmetric_difference()` where the symmetric difference with a non-`IntervalIndex` did not raise (GH18475)
- Bug in *IntervalIndex* where set operations that returned an empty `IntervalIndex` had the wrong dtype (GH19101)
- Bug in *DataFrame.drop_duplicates()* where no `KeyError` is raised when passing in columns that don't exist on the `DataFrame` (GH19726)
- Bug in `Index` subclasses constructors that ignore unexpected keyword arguments (GH19348)
- Bug in *Index.difference()* when taking difference of an `Index` with itself (GH20040)
- Bug in *DataFrame.first_valid_index()* and *DataFrame.last_valid_index()* in presence of entire rows of NaNs in the middle of values (GH20499).
- Bug in *IntervalIndex* where some indexing operations were not supported for overlapping or non-monotonic `uint64` data (GH20636)
- Bug in `Series.is_unique` where extraneous output in stderr is shown if Series contains objects with `__ne__` defined (GH20661)
- Bug in `.loc` assignment with a single-element list-like incorrectly assigns as a list (GH19474)
- Bug in partial string indexing on a `Series/DataFrame` with a monotonic decreasing `DatetimeIndex` (GH19362)
- Bug in performing in-place operations on a `DataFrame` with a duplicate `Index` (GH17105)
- Bug in *IntervalIndex.get_loc()* and *IntervalIndex.get_indexer()* when used with an *IntervalIndex* containing a single interval (GH17284, GH20921)
- Bug in `.loc` with a `uint64` indexer (GH20722)

MultiIndex

- Bug in `MultiIndex.__contains__()` where non-tuple keys would return `True` even if they had been dropped (GH19027)
- Bug in `MultiIndex.set_labels()` which would cause casting (and potentially clipping) of the new labels if the `level` argument is not 0 or a list like [0, 1, …] (GH19057)
- Bug in `MultiIndex.get_level_values()` which would return an invalid index on level of ints with missing values (GH17924)
- Bug in `MultiIndex.unique()` when called on empty `MultiIndex` (GH20568)
- Bug in `MultiIndex.unique()` which would not preserve level names (GH20570)
- Bug in `MultiIndex.remove_unused_levels()` which would fill nan values (GH18417)
- Bug in `MultiIndex.from_tuples()` which would fail to take zipped tuples in python3 (GH18434)
- Bug in `MultiIndex.get_loc()` which would fail to automatically cast values between float and int (GH18818, GH15994)
- Bug in `MultiIndex.get_loc()` which would cast boolean to integer labels (GH19086)
- Bug in `MultiIndex.get_loc()` which would fail to locate keys containing `NaN` (GH18485)
- Bug in `MultiIndex.get_loc()` in large `MultiIndex`, would fail when levels had different dtypes (GH18520)
- Bug in indexing where nested indexers having only numpy arrays are handled incorrectly (GH19686)

I/O

- `read_html()` now rewinds seekable IO objects after parse failure, before attempting to parse with a new parser. If a parser errors and the object is non-seekable, an informative error is raised suggesting the use of a different parser (GH17975)
- `DataFrame.to_html()` now has an option to add an id to the leading *<table>* tag (GH8496)
- Bug in `read_msgpack()` with a non existent file is passed in Python 2 (GH15296)
- Bug in `read_csv()` where a `MultiIndex` with duplicate columns was not being mangled appropriately (GH18062)
- Bug in `read_csv()` where missing values were not being handled properly when `keep_default_na=False` with dictionary `na_values` (GH19227)
- Bug in `read_csv()` causing heap corruption on 32-bit, big-endian architectures (GH20785)
- Bug in `read_sas()` where a file with 0 variables gave an `AttributeError` incorrectly. Now it gives an `EmptyDataError` (GH18184)
- Bug in `DataFrame.to_latex()` where pairs of braces meant to serve as invisible placeholders were escaped (GH18667)
- Bug in `DataFrame.to_latex()` where a `NaN` in a `MultiIndex` would cause an `IndexError` or incorrect output (GH14249)
- Bug in `DataFrame.to_latex()` where a non-string index-level name would result in an `AttributeError` (GH19981)
- Bug in `DataFrame.to_latex()` where the combination of an index name and the *index_names=False* option would result in incorrect output (GH18326)

- Bug in *DataFrame.to_latex()* where a MultiIndex with an empty string as its name would result in incorrect output (GH18669)
- Bug in *DataFrame.to_latex()* where missing space characters caused wrong escaping and produced non-valid latex in some cases (GH20859)
- Bug in *read_json()* where large numeric values were causing an OverflowError (GH18842)
- Bug in *DataFrame.to_parquet()* where an exception was raised if the write destination is S3 (GH19134)
- *Interval* now supported in *DataFrame.to_excel()* for all Excel file types (GH19242)
- *Timedelta* now supported in *DataFrame.to_excel()* for all Excel file types (GH19242, GH9155, GH19900)
- Bug in *pandas.io.stata.StataReader.value_labels()* raising an AttributeError when called on very old files. Now returns an empty dict (GH19417)
- Bug in *read_pickle()* when unpickling objects with *TimedeltaIndex* or *Float64Index* created with pandas prior to version 0.20 (GH19939)
- Bug in *pandas.io.json.json_normalize()* where sub-records are not properly normalized if any sub-records values are NoneType (GH20030)
- Bug in usecols parameter in *read_csv()* where error is not raised correctly when passing a string. (GH20529)
- Bug in *HDFStore.keys()* when reading a file with a soft link causes exception (GH20523)
- Bug in HDFStore.select_column() where a key which is not a valid store raised an AttributeError instead of a KeyError (GH17912)

Plotting

- Better error message when attempting to plot but matplotlib is not installed (GH19810).
- *DataFrame.plot()* now raises a ValueError when the x or y argument is improperly formed (GH18671)
- Bug in *DataFrame.plot()* when x and y arguments given as positions caused incorrect referenced columns for line, bar and area plots (GH20056)
- Bug in formatting tick labels with datetime.time() and fractional seconds (GH18478).
- *Series.plot.kde()* has exposed the args ind and bw_method in the docstring (GH18461). The argument ind may now also be an integer (number of sample points).
- *DataFrame.plot()* now supports multiple columns to the y argument (GH19699)

Groupby/Resample/Rolling

- Bug when grouping by a single column and aggregating with a class like list or tuple (GH18079)
- Fixed regression in *DataFrame.groupby()* which would not emit an error when called with a tuple key not in the index (GH18798)
- Bug in *DataFrame.resample()* which silently ignored unsupported (or mistyped) options for label, closed and convention (GH19303)
- Bug in *DataFrame.groupby()* where tuples were interpreted as lists of keys rather than as keys (GH17979, GH18249)

- Bug in *DataFrame.groupby()* where aggregation by `first`/`last`/`min`/`max` was causing timestamps to lose precision (GH19526)
- Bug in *DataFrame.transform()* where particular aggregation functions were being incorrectly cast to match the dtype(s) of the grouped data (GH19200)
- Bug in *DataFrame.groupby()* passing the *on=* kwarg, and subsequently using `.apply()` (GH17813)
- Bug in *DataFrame.resample().aggregate* not raising a `KeyError` when aggregating a non-existent column (GH16766, GH19566)
- Bug in `DataFrameGroupBy.cumsum()` and `DataFrameGroupBy.cumprod()` when `skipna` was passed (GH19806)
- Bug in *DataFrame.resample()* that dropped timezone information (GH13238)
- Bug in *DataFrame.groupby()* where transformations using `np.all` and `np.any` were raising a `ValueError` (GH20653)
- Bug in *DataFrame.resample()* where `ffill`, `bfill`, `pad`, `backfill`, `fillna`, `interpolate`, and `asfreq` were ignoring `loffset`. (GH20744)
- Bug in *DataFrame.groupby()* when applying a function that has mixed data types and the user supplied function can fail on the grouping column (GH20949)
- Bug in *DataFrameGroupBy.rolling().apply()* where operations performed against the associated `DataFrameGroupBy` object could impact the inclusion of the grouped item(s) in the result (GH14013)

Sparse

- Bug in which creating a `SparseDataFrame` from a dense `Series` or an unsupported type raised an uncontrolled exception (GH19374)
- Bug in `SparseDataFrame.to_csv` causing exception (GH19384)
- Bug in `SparseSeries.memory_usage` which caused segfault by accessing non sparse elements (GH19368)
- Bug in constructing a *SparseArray*: if `data` is a scalar and `index` is defined it will coerce to `float64` regardless of scalar's dtype. (GH19163)

Reshaping

- Bug in *DataFrame.merge()* where referencing a `CategoricalIndex` by name, where the `by` kwarg would `KeyError` (GH20777)
- Bug in *DataFrame.stack()* which fails trying to sort mixed type levels under Python 3 (GH18310)
- Bug in *DataFrame.unstack()* which casts int to float if `columns` is a `MultiIndex` with unused levels (GH17845)
- Bug in *DataFrame.unstack()* which raises an error if `index` is a `MultiIndex` with unused labels on the unstacked level (GH18562)
- Fixed construction of a *Series* from a `dict` containing `NaN` as key (GH18480)
- Fixed construction of a *DataFrame* from a `dict` containing `NaN` as key (GH18455)
- Disabled construction of a *Series* where len(index) > len(data) = 1, which previously would broadcast the data item, and now raises a `ValueError` (GH18819)

- Suppressed error in the construction of a *DataFrame* from a `dict` containing scalar values when the corresponding keys are not included in the passed index (GH18600)
- Fixed (changed from `object` to `float64`) dtype of *DataFrame* initialized with axes, no data, and `dtype=int` (GH19646)
- Bug in *Series.rank()* where `Series` containing `NaT` modifies the `Series` inplace (GH18521)
- Bug in *cut()* which fails when using readonly arrays (GH18773)
- Bug in *DataFrame.pivot_table()* which fails when the `aggfunc` arg is of type string. The behavior is now consistent with other methods like `agg` and `apply` (GH18713)
- Bug in *DataFrame.merge()* in which merging using `Index` objects as vectors raised an Exception (GH19038)
- Bug in *DataFrame.stack()*, *DataFrame.unstack()*, *Series.unstack()* which were not returning subclasses (GH15563)
- Bug in timezone comparisons, manifesting as a conversion of the index to UTC in `.concat()` (GH18523)
- Bug in *concat()* when concatenating sparse and dense series it returns only a `SparseDataFrame`. Should be a `DataFrame`. (GH18914, GH18686, and GH16874)
- Improved error message for *DataFrame.merge()* when there is no common merge key (GH19427)
- Bug in *DataFrame.join()* which does an `outer` instead of a `left` join when being called with multiple DataFrames and some have non-unique indices (GH19624)
- *Series.rename()* now accepts `axis` as a kwarg (GH18589)
- Bug in *rename()* where an Index of same-length tuples was converted to a MultiIndex (GH19497)
- Comparisons between *Series* and *Index* would return a `Series` with an incorrect name, ignoring the `Index`'s name attribute (GH19582)
- Bug in *qcut()* where datetime and timedelta data with `NaT` present raised a `ValueError` (GH19768)
- Bug in *DataFrame.iterrows()*, which would infers strings not compliant to ISO8601 to datetimes (GH19671)
- Bug in *Series* constructor with `Categorical` where a `ValueError` is not raised when an index of different length is given (GH19342)
- Bug in *DataFrame.astype()* where column metadata is lost when converting to categorical or a dictionary of dtypes (GH19920)
- Bug in *cut()* and *qcut()* where timezone information was dropped (GH19872)
- Bug in *Series* constructor with a `dtype=str`, previously raised in some cases (GH19853)
- Bug in *get_dummies()*, and `select_dtypes()`, where duplicate column names caused incorrect behavior (GH20848)
- Bug in *isna()*, which cannot handle ambiguous typed lists (GH20675)
- Bug in *concat()* which raises an error when concatenating TZ-aware dataframes and all-NaT dataframes (GH12396)
- Bug in *concat()* which raises an error when concatenating empty TZ-aware series (GH18447)

Other

- Improved error message when attempting to use a Python keyword as an identifier in a `numexpr` backed query (GH18221)
- Bug in accessing a *pandas.get_option()*, which raised `KeyError` rather than `OptionError` when looking up a non-existent option key in some cases (GH19789)
- Bug in *testing.assert_series_equal()* and *testing.assert_frame_equal()* for Series or DataFrames with differing unicode data (GH20503)

Contributors

A total of 328 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Critchley
- AbdealiJK +
- Adam Hooper +
- Albert Villanova del Moral
- Alejandro Giacometti +
- Alejandro Hohmann +
- Alex Rychyk
- Alexander Buchkovsky
- Alexander Lenail +
- Alexander Michael Schade
- Aly Sivji +
- Andreas Költringer +
- Andrew
- Andrew Bui +
- András Novoszáth +
- Andy Craze +
- Andy R. Terrel
- Anh Le +
- Anil Kumar Pallekonda +
- Antoine Pitrou +
- Antonio Linde +
- Antonio Molina +
- Antonio Quinonez +
- Armin Varshokar +
- Artem Bogachev +
- Avi Sen +

- Azeez Oluwafemi +
- Ben Auffarth +
- Bernhard Thiel +
- Bhavesh Poddar +
- BielStela +
- Blair +
- Bob Haffner
- Brett Naul +
- Brock Mendel
- Bryce Guinta +
- Carlos Eduardo Moreira dos Santos +
- Carlos García Márquez +
- Carol Willing
- Cheuk Ting Ho +
- Chitrank Dixit +
- Chris
- Chris Burr +
- Chris Catalfo +
- Chris Mazzullo
- Christian Chwala +
- Cihan Ceyhan +
- Clemens Brunner
- Colin +
- Cornelius Riemenschneider
- Crystal Gong +
- DaanVanHauwermeiren
- Dan Dixey +
- Daniel Frank +
- Daniel Garrido +
- Daniel Sakuma +
- DataOmbudsman +
- Dave Hirschfeld
- Dave Lewis +
- David Adrián Cañones Castellano +
- David Arcos +
- David C Hall +

- David Fischer
- David Hoese +
- David Lutz +
- David Polo +
- David Stansby
- Dennis Kamau +
- Dillon Niederhut
- Dimitri +
- Dr. Irv
- Dror Atariah
- Eric Chea +
- Eric Kisslinger
- Eric O. LEBIGOT (EOL) +
- FAN-GOD +
- Fabian Retkowski +
- Fer Sar +
- Gabriel de Maeztu +
- Gianpaolo Macario +
- Giftlin Rajaiah
- Gilberto Olimpio +
- Gina +
- Gjelt +
- Graham Inggs +
- Grant Roch
- Grant Smith +
- Grzegorz Konefał +
- Guilherme Beltramini
- HagaiHargil +
- Hamish Pitkeathly +
- Hammad Mashkoor +
- Hannah Ferchland +
- Hans
- Haochen Wu +
- Hissashi Rocha +
- Iain Barr +
- Ibrahim Sharaf ElDen +

- Ignasi Fosch +
- Igor Conrado Alves de Lima +
- Igor Shelvinskyi +
- Imanflow +
- Ingolf Becker
- Israel Saeta Pérez
- Iva Koevska +
- Jakub Nowacki +
- Jan F-F +
- Jan Koch +
- Jan Werkmann
- Janelle Zoutkamp +
- Jason Bandlow +
- Jaume Bonet +
- Jay Alammar +
- Jeff Reback
- JennaVergeynst
- Jimmy Woo +
- Jing Qiang Goh +
- Joachim Wagner +
- Joan Martin Miralles +
- Joel Nothman
- Joeun Park +
- John Cant +
- Johnny Metz +
- Jon Mease
- Jonas Schulze +
- Jongwony +
- Jordi Contestí +
- Joris Van den Bossche
- José F. R. Fonseca +
- Jovixe +
- Julio Martinez +
- Jörg Döpfert
- KOBAYASHI Ittoku +
- Kate Surta +

- Kenneth +
- Kevin Kuhl
- Kevin Sheppard
- Krzysztof Chomski
- Ksenia +
- Ksenia Bobrova +
- Kunal Gosar +
- Kurtis Kerstein +
- Kyle Barron +
- Laksh Arora +
- Laurens Geffert +
- Leif Walsh
- Liam Marshall +
- Liam3851 +
- Licht Takeuchi
- Liudmila +
- Ludovico Russo +
- Mabel Villalba +
- Manan Pal Singh +
- Manraj Singh
- Marc +
- Marc Garcia
- Marco Hemken +
- Maria del Mar Bibiloni +
- Mario Corchero +
- Mark Woodbridge +
- Martin Journois +
- Mason Gallo +
- Matias Heikkilä +
- Matt Braymer-Hayes
- Matt Kirk +
- Matt Maybeno +
- Matthew Kirk +
- Matthew Rocklin +
- Matthew Roeschke
- Matthias Bussonnier +

- Max Mikhaylov +
- Maxim Veksler +
- Maximilian Roos
- Maximiliano Greco +
- Michael Penkov
- Michael Röttger +
- Michael Selik +
- Michael Waskom
- Mie~~~
- Mike Kutzma +
- Ming Li +
- Mitar +
- Mitch Negus +
- Montana Low +
- Moritz Münst +
- Mortada Mehyar
- Myles Braithwaite +
- Nate Yoder
- Nicholas Ursa +
- Nick Chmura
- Nikos Karagiannakis +
- Nipun Sadvilkar +
- Nis Martensen +
- Noah +
- Noémi Éltető +
- Olivier Bilodeau +
- Ondrej Kokes +
- Onno Eberhard +
- Paul Ganssle +
- Paul Mannino +
- Paul Reidy
- Paulo Roberto de Oliveira Castro +
- Pepe Flores +
- Peter Hoffmann
- Phil Ngo +
- Pietro Battiston

- Pranav Suri +
- Priyanka Ojha +
- Pulkit Maloo +
- README Bot +
- Ray Bell +
- Riccardo Magliocchetti +
- Ridhwan Luthra +
- Robert Meyer
- Robin
- Robin Kiplang'at +
- Rohan Pandit +
- Rok Mihevc +
- Rouz Azari
- Ryszard T. Kaleta +
- Sam Cohan
- Sam Foo
- Samir Musali +
- Samuel Sinayoko +
- Sangwoong Yoon
- SarahJessica +
- Sharad Vijalapuram +
- Shubham Chaudhary +
- SiYoungOh +
- Sietse Brouwer
- Simone Basso +
- Stefania Delprete +
- Stefano Cianciulli +
- Stephen Childs +
- StephenVoland +
- Stijn Van Hoey +
- Sven
- Talitha Pumar +
- Tarbo Fukazawa +
- Ted Petrou +
- Thomas A Caswell
- Tim Hoffmann +

- Tim Swast
- Tom Augspurger
- Tommy +
- Tulio Casagrande +
- Tushar Gupta +
- Tushar Mittal +
- Upkar Lidder +
- Victor Villas +
- Vince W +
- Vinícius Figueiredo +
- Vipin Kumar +
- WBare
- Wenhuan +
- Wes Turner
- William Ayd
- Wilson Lin +
- Xbar
- Yaroslav Halchenko
- Yee Mey
- Yeongseon Choe +
- Yian +
- Yimeng Zhang
- ZhuBaohe +
- Zihao Zhao +
- adatasetaday +
- akielbowicz +
- akosel +
- alinde1 +
- amuta +
- bolkedebruin
- cbertinato
- cgohlke
- charlie0389 +
- chris-b1
- csfarkas +
- dajcs +

- deflatSOCO +
- derestle-htwg
- discort
- dmanikowski-reef +
- donK23 +
- elrubio +
- fivemok +
- fjdiod
- fjetter +
- froessler +
- gabrielclow
- gfyoung
- ghasemnaddaf
- h-vetinari +
- himanshu awasthi +
- ignamv +
- jayfoad +
- jazzmuesli +
- jbrockmendel
- jen w +
- jjames34 +
- joaoavf +
- joders +
- jschendel
- juan huguet +
- l736x +
- luzpaz +
- mdeboc +
- miguelmorin +
- miker985
- miquelcamprodon +
- orereta +
- ottiP +
- peterpanmj +
- rafarui +
- raph-m +

- readyready15728 +
- rmihael +
- samghelms +
- scriptomation +
- sfoo +
- stefansimik +
- stonebig
- tmnhat2001 +
- tomneep +
- topper-123
- tv3141 +
- verakai +
- xpvpc +
- zhanghui +

8.3 Version 0.22

8.3.1 v0.22.0 (December 29, 2017)

This is a major release from 0.21.1 and includes a single, API-breaking change. We recommend that all users upgrade to this version after carefully reading the release note (singular!).

Backwards incompatible API changes

Pandas 0.22.0 changes the handling of empty and all-*NA* sums and products. The summary is that

- The sum of an empty or all-*NA* `Series` is now `0`
- The product of an empty or all-*NA* `Series` is now `1`
- We've added a `min_count` parameter to `.sum()` and `.prod()` controlling the minimum number of valid values for the result to be valid. If fewer than `min_count` non-*NA* values are present, the result is *NA*. The default is `0`. To return `NaN`, the 0.21 behavior, use `min_count=1`.

Some background: In pandas 0.21, we fixed a long-standing inconsistency in the return value of all-*NA* series depending on whether or not bottleneck was installed. See *Sum/Prod of all-NaN or empty Series/DataFrames is now consistently NaN*. At the same time, we changed the sum and prod of an empty `Series` to also be `NaN`.

Based on feedback, we've partially reverted those changes.

Arithmetic Operations

The default sum for empty or all-*NA* `Series` is now `0`.

pandas 0.21.x

```
In [1]: pd.Series([]).sum()
Out[1]: nan

In [2]: pd.Series([np.nan]).sum()
Out[2]: nan
```

pandas 0.22.0

```
In [1]: pd.Series([]).sum()
Out[1]: 0.0

In [2]: pd.Series([np.nan]).sum()
\\\\\\\\\\\\Out[2]: 0.0
```

The default behavior is the same as pandas 0.20.3 with bottleneck installed. It also matches the behavior of NumPy's `np.nansum` on empty and all-*NA* arrays.

To have the sum of an empty series return `NaN` (the default behavior of pandas 0.20.3 without bottleneck, or pandas 0.21.x), use the `min_count` keyword.

```
In [3]: pd.Series([]).sum(min_count=1)
Out[3]: nan
```

Thanks to the `skipna` parameter, the `.sum` on an all-*NA* series is conceptually the same as the `.sum` of an empty one with `skipna=True` (the default).

```
In [4]: pd.Series([np.nan]).sum(min_count=1)  # skipna=True by default
Out[4]: nan
```

The `min_count` parameter refers to the minimum number of *non-null* values required for a non-NA sum or product.

`Series.prod()` has been updated to behave the same as *`Series.sum()`*, returning `1` instead.

```
In [5]: pd.Series([]).prod()
Out[5]: 1.0

In [6]: pd.Series([np.nan]).prod()
\\\\\\\\\\\\Out[6]: 1.0

In [7]: pd.Series([]).prod(min_count=1)
\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: nan
```

These changes affect *`DataFrame.sum()`* and *`DataFrame.prod()`* as well. Finally, a few less obvious places in pandas are affected by this change.

Grouping by a Categorical

Grouping by a `Categorical` and summing now returns `0` instead of `NaN` for categories with no observations. The product now returns `1` instead of `NaN`.

pandas 0.21.x

```
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])

In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
```

(continues on next page)

(continued from previous page)

```
a    3.0
b    NaN
dtype: float64
```

pandas 0.22

```
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])

In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
a    3
b    0
Length: 2, dtype: int64
```

To restore the 0.21 behavior of returning `NaN` for unobserved groups, use `min_count>=1`.

```
In [10]: pd.Series([1, 2]).groupby(grouper).sum(min_count=1)
Out[10]:
a    3.0
b    NaN
Length: 2, dtype: float64
```

Resample

The sum and product of all-*NA* bins has changed from `NaN` to `0` for sum and `1` for product.

pandas 0.21.x

```
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
   ....:               index=pd.date_range('2017', periods=4))
   ....: s
Out[11]:
2017-01-01    1.0
2017-01-02    1.0
2017-01-03    NaN
2017-01-04    NaN
Freq: D, dtype: float64

In [12]: s.resample('2d').sum()
Out[12]:
2017-01-01    2.0
2017-01-03    NaN
Freq: 2D, dtype: float64
```

pandas 0.22.0

```
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
   ....:               index=pd.date_range('2017', periods=4))
   ....:

In [12]: s.resample('2d').sum()
Out[12]:
2017-01-01    2.0
2017-01-03    0.0
Freq: 2D, Length: 2, dtype: float64
```

To restore the 0.21 behavior of returning NaN, use min_count>=1.

```
In [13]: s.resample('2d').sum(min_count=1)
Out[13]:
2017-01-01    2.0
2017-01-03    NaN
Freq: 2D, Length: 2, dtype: float64
```

In particular, upsampling and taking the sum or product is affected, as upsampling introduces missing values even if the original series was entirely valid.

pandas 0.21.x

```
In [14]: idx = pd.DatetimeIndex(['2017-01-01', '2017-01-02'])

In [15]: pd.Series([1, 2], index=idx).resample('12H').sum()
Out[15]:
2017-01-01 00:00:00    1.0
2017-01-01 12:00:00    NaN
2017-01-02 00:00:00    2.0
Freq: 12H, dtype: float64
```

pandas 0.22.0

```
In [14]: idx = pd.DatetimeIndex(['2017-01-01', '2017-01-02'])

In [15]: pd.Series([1, 2], index=idx).resample("12H").sum()
Out[15]:
2017-01-01 00:00:00    1
2017-01-01 12:00:00    0
2017-01-02 00:00:00    2
Freq: 12H, Length: 3, dtype: int64
```

Once again, the min_count keyword is available to restore the 0.21 behavior.

```
In [16]: pd.Series([1, 2], index=idx).resample("12H").sum(min_count=1)
Out[16]:
2017-01-01 00:00:00    1.0
2017-01-01 12:00:00    NaN
2017-01-02 00:00:00    2.0
Freq: 12H, Length: 3, dtype: float64
```

Rolling and Expanding

Rolling and expanding already have a min_periods keyword that behaves similar to min_count. The only case that changes is when doing a rolling or expanding sum with min_periods=0. Previously this returned NaN, when fewer than min_periods non-*NA* values were in the window. Now it returns 0.

pandas 0.21.1

```
In [17]: s = pd.Series([np.nan, np.nan])

In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0    NaN
1    NaN
dtype: float64
```

pandas 0.22.0

```
In [17]: s = pd.Series([np.nan, np.nan])

In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0    0.0
1    0.0
Length: 2, dtype: float64
```

The default behavior of `min_periods=None`, implying that `min_periods` equals the window size, is unchanged.

Compatibility

If you maintain a library that should work across pandas versions, it may be easiest to exclude pandas 0.21 from your requirements. Otherwise, all your `sum()` calls would need to check if the `Series` is empty before summing.

With setuptools, in your `setup.py` use:

```
install_requires=['pandas!=0.21.*', ...]
```

With conda, use

```
requirements:
  run:
    - pandas !=0.21.0,!=0.21.1
```

Note that the inconsistency in the return value for all-*NA* series is still there for pandas 0.20.3 and earlier. Avoiding pandas 0.21 will only help with the empty case.

Contributors

A total of 1 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Tom Augspurger

8.4 Version 0.21

8.4.1 v0.21.1 (December 12, 2017)

This is a minor bug-fix release in the 0.21.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Temporarily restore matplotlib datetime plotting functionality. This should resolve issues for users who implicitly relied on pandas to plot datetimes with matplotlib. See *here*.
- Improvements to the Parquet IO functions introduced in 0.21.0. See *here*.

What's new in v0.21.1

Restore Matplotlib datetime Converter Registration

Pandas implements some matplotlib converters for nicely formatting the axis labels on plots with `datetime` or `Period` values. Prior to pandas 0.21.0, these were implicitly registered with matplotlib, as a side effect of `import pandas`.

In pandas 0.21.0, we required users to explicitly register the converter. This caused problems for some users who relied on those converters being present for regular `matplotlib.pyplot` plotting methods, so we're temporarily reverting that change; pandas 0.21.1 again registers the converters on import, just like before 0.21.0.

We've added a new option to control the converters: `pd.options.plotting.matplotlib.register_converters`. By default, they are registered. Toggling this to `False` removes pandas' formatters and restore any converters we overwrote when registering them (GH18301).

We're working with the matplotlib developers to make this easier. We're trying to balance user convenience (automatically registering the converters) with import performance and best practices (importing pandas shouldn't have the side effect of overwriting any custom converters you've already set). In the future we hope to have most of the datetime formatting functionality in matplotlib, with just the pandas-specific converters in pandas. We'll then gracefully deprecate the automatic registration of converters in favor of users explicitly registering them when they want them.

New features

Improvements to the Parquet IO functionality

- *`DataFrame.to_parquet()`* will now write non-default indexes when the underlying engine supports it. The indexes will be preserved when reading back in with *`read_parquet()`* (GH18581).

- *read_parquet()* now allows to specify the columns to read from a parquet file (GH18154)
- *read_parquet()* now allows to specify kwargs which are passed to the respective engine (GH18216)

Other Enhancements

- *Timestamp.timestamp()* is now available in Python 2.7. (GH17329)
- *Grouper* and `TimeGrouper` now have a friendly repr output (GH18203).

Deprecations

- `pandas.tseries.register` has been renamed to *pandas.plotting.register_matplotlib_converters()* (GH18301)

Performance Improvements

- Improved performance of plotting large series/dataframes (GH18236).

Bug Fixes

Conversion

- Bug in *TimedeltaIndex* subtraction could incorrectly overflow when `NaT` is present (GH17791)
- Bug in *DatetimeIndex* subtracting datetimelike from DatetimeIndex could fail to overflow (GH18020)
- Bug in `IntervalIndex.copy()` when copying and `IntervalIndex` with non-default `closed` (GH18339)
- Bug in *DataFrame.to_dict()* where columns of datetime that are tz-aware were not converted to required arrays when used with `orient='records'`, raising `TypeError` (GH18372)
- Bug in `DateTimeIndex` and *date_range()* where mismatching tz-aware `start` and `end` timezones would not raise an err if `end.tzinfo` is None (GH18431)
- Bug in *Series.fillna()* which raised when passed a long integer on Python 2 (GH18159).

Indexing

- Bug in a boolean comparison of a `datetime.datetime` and a `datetime64[ns]` dtype Series (GH17965)
- Bug where a `MultiIndex` with more than a million records was not raising `AttributeError` when trying to access a missing attribute (GH18165)
- Bug in *IntervalIndex* constructor when a list of intervals is passed with non-default `closed` (GH18334)
- Bug in `Index.putmask` when an invalid mask passed (GH18368)
- Bug in masked assignment of a `timedelta64[ns]` dtype `Series`, incorrectly coerced to float (GH18493)

I/O

- Bug in class:~*pandas.io.stata.StataReader* not converting date/time columns with display formatting addressed (GH17990). Previously columns with display formatting were normally left as ordinal numbers and not converted to datetime objects.
- Bug in *read_csv()* when reading a compressed UTF-16 encoded file (GH18071)
- Bug in *read_csv()* for handling null values in index columns when specifying `na_filter=False` (GH5239)
- Bug in *read_csv()* when reading numeric category fields with high cardinality (GH18186)
- Bug in *DataFrame.to_csv()* when the table had `MultiIndex` columns, and a list of strings was passed in for `header` (GH5539)
- Bug in parsing integer datetime-like columns with specified format in `read_sql` (GH17855).
- Bug in *DataFrame.to_msgpack()* when serializing data of the `numpy.bool_` datatype (GH18390)
- Bug in *read_json()* not decoding when reading line delimited JSON from S3 (GH17200)
- Bug in *pandas.io.json.json_normalize()* to avoid modification of `meta` (GH18610)
- Bug in `to_latex()` where repeated MultiIndex values were not printed even though a higher level index differed from the previous row (GH14484)
- Bug when reading NaN-only categorical columns in `HDFStore` (GH18413)
- Bug in *DataFrame.to_latex()* with `longtable=True` where a latex multicolumn always spanned over three columns (GH17959)

Plotting

- Bug in `DataFrame.plot()` and `Series.plot()` with *DatetimeIndex* where a figure generated by them is not pickleable in Python 3 (GH18439)

Groupby/Resample/Rolling

- Bug in `DataFrame.resample(...).apply(...)` when there is a callable that returns different columns (GH15169)
- Bug in `DataFrame.resample(...)` when there is a time change (DST) and resampling frequency is 12h or higher (GH15549)
- Bug in `pd.DataFrameGroupBy.count()` when counting over a datetimelike column (GH13393)
- Bug in `rolling.var` where calculation is inaccurate with a zero-valued array (GH18430)

Reshaping

- Error message in `pd.merge_asof()` for key datatype mismatch now includes datatype of left and right key (GH18068)
- Bug in `pd.concat` when empty and non-empty DataFrames or Series are concatenated (GH18178 GH18187)
- Bug in `DataFrame.filter(...)` when `unicode` is passed as a condition in Python 2 (GH13101)
- Bug when merging empty DataFrames when `np.seterr(divide='raise')` is set (GH17776)

Numeric

- Bug in `pd.Series.rolling.skew()` and `rolling.kurt()` with all equal values has floating issue (GH18044)

Categorical

- Bug in `DataFrame.astype()` where casting to 'category' on an empty `DataFrame` causes a segmentation fault (GH18004)
- Error messages in the testing module have been improved when items have different `CategoricalDtype` (GH18069)
- `CategoricalIndex` can now correctly take a `pd.api.types.CategoricalDtype` as its dtype (GH18116)
- Bug in `Categorical.unique()` returning read-only `codes` array when all categories were `NaN` (GH18051)
- Bug in `DataFrame.groupby(axis=1)` with a `CategoricalIndex` (GH18432)

String

- `Series.str.split()` will now propagate `NaN` values across all expanded columns instead of `None` (GH18450)

Contributors

A total of 46 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Critchley +
- Alex Rychyk
- Alexander Buchkovsky +
- Alexander Michael Schade +
- Chris Mazzullo
- Cornelius Riemenschneider +
- Dave Hirschfeld +
- David Fischer +
- David Stansby +
- Dror Atariah +
- Eric Kisslinger +
- Hans +
- Ingolf Becker +
- Jan Werkmann +
- Jeff Reback

- Joris Van den Bossche
- Jörg Döpfert +
- Kevin Kuhl +
- Krzysztof Chomski +
- Leif Walsh
- Licht Takeuchi
- Manraj Singh +
- Matt Braymer-Hayes +
- Michael Waskom +
- Mie~~~ +
- Peter Hoffmann +
- Robert Meyer +
- Sam Cohan +
- Sietse Brouwer +
- Sven +
- Tim Swast
- Tom Augspurger
- Wes Turner
- William Ayd +
- Yee Mey +
- bolkedebruin +
- cgohlke
- derestle-htwg +
- fjdiod +
- gabrielclow +
- gfyoung
- ghasemnaddaf +
- jbrockmendel
- jschendel
- miker985 +
- topper-123

8.4.2 v0.21.0 (October 27, 2017)

This is a major release from 0.20.3 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Integration with Apache Parquet, including a new top-level `read_parquet()` function and `DataFrame.to_parquet()` method, see *here*.
- New user-facing `pandas.api.types.CategoricalDtype` for specifying categoricals independent of the data, see *here*.
- The behavior of `sum` and `prod` on all-NaN Series/DataFrames is now consistent and no longer depends on whether bottleneck is installed, and `sum` and `prod` on empty Series now return NaN instead of 0, see *here*.
- Compatibility fixes for pypy, see *here*.
- Additions to the `drop`, `reindex` and `rename` API to make them more consistent, see *here*.
- Addition of the new methods `DataFrame.infer_objects` (see *here*) and `GroupBy.pipe` (see *here*).
- Indexing with a list of labels, where one or more of the labels is missing, is deprecated and will raise a KeyError in a future version, see *here*.

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

What's new in v0.21.0

New features

Integration with Apache Parquet file format

Integration with Apache Parquet, including a new top-level `read_parquet()` and `DataFrame.to_parquet()` method, see *here* (GH15838, GH17438).

Apache Parquet provides a cross-language, binary file format for reading and writing data frames efficiently. Parquet is designed to faithfully serialize and de-serialize `DataFrame` s, supporting all of the pandas dtypes, including extension dtypes such as datetime with timezones.

This functionality depends on either the pyarrow or fastparquet library. For more details, see see *the IO docs on Parquet*.

`infer_objects` type conversion

The `DataFrame.infer_objects()` and `Series.infer_objects()` methods have been added to perform dtype inference on object columns, replacing some of the functionality of the deprecated `convert_objects` method. See the documentation *here* for more details. (GH11221)

This method only performs soft conversions on object columns, converting Python objects to native types, but not any coercive conversions. For example:

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3],
   ...:                    'B': np.array([1, 2, 3], dtype='object'),
   ...:                    'C': ['1', '2', '3']})
   ...:

In [2]: df.dtypes
Out[2]:
A     int64
B    object
C    object
Length: 3, dtype: object

In [3]: df.infer_objects().dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]:
A     int64
B     int64
C    object
Length: 3, dtype: object
```

Note that column 'C' was not converted - only scalar numeric types will be converted to a new type. Other types of conversion should be accomplished using the *to_numeric()* function (or *to_datetime()*, *to_timedelta()*).

```
In [4]: df = df.infer_objects()

In [5]: df['C'] = pd.to_numeric(df['C'], errors='coerce')

In [6]: df.dtypes
Out[6]:
A    int64
B    int64
C    int64
Length: 3, dtype: object
```

Improved warnings when attempting to create columns

New users are often puzzled by the relationship between column operations and attribute access on DataFrame instances (GH7175). One specific instance of this confusion is attempting to create a new column by setting an attribute on the DataFrame:

```
In [1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In [2]: df.two = [4, 5, 6]
```

This does not raise any obvious exceptions, but also does not create a new column:

```
In [3]: df
Out[3]:
    one
0   1.0
1   2.0
2   3.0
```

Setting a list-like data structure into a new attribute now raises a UserWarning about the potential for unexpected behavior. See *Attribute Access*.

drop now also accepts index/columns keywords

The *drop()* method has gained index/columns keywords as an alternative to specifying the axis. This is similar to the behavior of reindex (GH12392).

For example:

```
In [7]: df = pd.DataFrame(np.arange(8).reshape(2, 4),
   ...:                   columns=['A', 'B', 'C', 'D'])
   ...:

In [8]: df
Out[8]:
   A  B  C  D
0  0  1  2  3
1  4  5  6  7

[2 rows x 4 columns]

In [9]: df.drop(['B', 'C'], axis=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
   A  D
0  0  3
1  4  7

[2 rows x 2 columns]

# the following is now equivalent
In [10]: df.drop(columns=['B', 'C'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   A  D
0  0  3
1  4  7

[2 rows x 2 columns]
```

rename, reindex now also accept axis keyword

The *DataFrame.rename()* and *DataFrame.reindex()* methods have gained the axis keyword to specify the axis to target with the operation (GH12392).

Here's rename:

```
In [11]: df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

In [12]: df.rename(str.lower, axis='columns')
Out[12]:
   a  b
0  1  4
1  2  5
2  3  6

[3 rows x 2 columns]

In [13]: df.rename(id, axis='index')
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]:
                 A  B
94127814126240  1  4
94127814126272  2  5
94127814126304  3  6

[3 rows x 2 columns]
```

And `reindex`:

```
In [14]: df.reindex(['A', 'B', 'C'], axis='columns')
Out[14]:
   A  B   C
0  1  4 NaN
1  2  5 NaN
2  3  6 NaN

[3 rows x 3 columns]

In [15]: df.reindex([0, 1, 3], axis='index')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]:
↪
     A    B
0  1.0  4.0
1  2.0  5.0
3  NaN  NaN

[3 rows x 2 columns]
```

The “index, columns” style continues to work as before.

```
In [16]: df.rename(index=id, columns=str.lower)
Out[16]:
                 a  b
94127814126240  1  4
94127814126272  2  5
94127814126304  3  6

[3 rows x 2 columns]

In [17]: df.reindex(index=[0, 1, 3], columns=['A', 'B', 'C'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A    B   C
0  1.0  4.0 NaN
1  2.0  5.0 NaN
3  NaN  NaN NaN

[3 rows x 3 columns]
```

We *highly* encourage using named arguments to avoid confusion when using either style.

`CategoricalDtype` for specifying categoricals

`pandas.api.types.CategoricalDtype` has been added to the public API and expanded to include the `categories` and `ordered` attributes. A `CategoricalDtype` can be used to specify the set of categories and orderedness of an array, independent of the data. This can be useful for example, when converting string data to a `Categorical` (GH14711, GH15078, GH16015, GH17643):

```
In [18]: from pandas.api.types import CategoricalDtype

In [19]: s = pd.Series(['a', 'b', 'c', 'a'])  # strings

In [20]: dtype = CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)

In [21]: s.astype(dtype)
Out[21]:
0    a
1    b
2    c
3    a
Length: 4, dtype: category
Categories (4, object): [a < b < c < d]
```

One place that deserves special mention is in *read_csv()*. Previously, with `dtype={'col': 'category'}`, the returned values and categories would always be strings.

```
In [22]: data = 'A,B\na,1\nb,2\nc,3'

In [23]: pd.read_csv(StringIO(data), dtype={'B': 'category'}).B.cat.categories
Out[23]: Index(['1', '2', '3'], dtype='object')
```

Notice the "object" dtype.

With a `CategoricalDtype` of all numerics, datetimes, or timedeltas, we can automatically convert to the correct type

```
In [24]: dtype = {'B': CategoricalDtype([1, 2, 3])}

In [25]: pd.read_csv(StringIO(data), dtype=dtype).B.cat.categories
Out[25]: Int64Index([1, 2, 3], dtype='int64')
```

The values have been correctly interpreted as integers.

The `.dtype` property of a `Categorical`, `CategoricalIndex` or a `Series` with categorical type will now return an instance of `CategoricalDtype`. While the repr has changed, `str(CategoricalDtype())` is still the string `'category'`. We'll take this moment to remind users that the *preferred* way to detect categorical data is to use *pandas.api.types.is_categorical_dtype()*, and not `str(dtype) == 'category'`.

See the *CategoricalDtype docs* for more.

`GroupBy` objects now have a `pipe` method

`GroupBy` objects now have a `pipe` method, similar to the one on `DataFrame` and `Series`, that allow for functions that take a `GroupBy` to be composed in a clean, readable syntax. (GH17871)

For a concrete example on combining `.groupby` and `.pipe` , imagine having a DataFrame with columns for stores, products, revenue and sold quantity. We'd like to do a groupwise calculation of *prices* (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable.

First we set the data:

```
In [26]: import numpy as np

In [27]: n = 1000

In [28]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
   ....:                    'Product': np.random.choice(['Product_1',
   ....:                                                 'Product_2',
   ....:                                                 'Product_3'
   ....:                                                 ], n),
   ....:                    'Revenue': (np.random.random(n) * 50 + 10).round(2),
   ....:                    'Quantity': np.random.randint(1, 10, size=n)})
   ....:

In [29]: df.head(2)
Out[29]:
     Store    Product  Revenue  Quantity
0  Store_2  Product_2    32.09         7
1  Store_1  Product_3    14.20         1

[2 rows x 4 columns]
```

Now, to find prices per store/product, we can simply do:

```
In [30]: (df.groupby(['Store', 'Product'])
   ....:    .pipe(lambda grp: grp.Revenue.sum() / grp.Quantity.sum())
   ....:    .unstack().round(2))
   ....:
Out[30]:
Product  Product_1  Product_2  Product_3
Store
Store_1       6.73       6.72       7.14
Store_2       7.59       6.98       7.23

[2 rows x 3 columns]
```

See the *documentation* for more.

`Categorical.rename_categories` accepts a dict-like

rename_categories() now accepts a dict-like argument for `new_categories`. The previous categories are looked up in the dictionary's keys and replaced if found. The behavior of missing and extra keys is the same as in *DataFrame.rename()*.

```
In [31]: c = pd.Categorical(['a', 'a', 'b'])

In [32]: c.rename_categories({"a": "eh", "b": "bee"})
Out[32]:
[eh, eh, bee]
Categories (2, object): [eh, bee]
```

Warning: To assist with upgrading pandas, `rename_categories` treats `Series` as list-like. Typically, Series are considered to be dict-like (e.g. in `.rename`, `.map`). In a future version of pandas `rename_categories` will change to treat them as dict-like. Follow the warning message's recommendations for writing future-proof code.

```
In [33]: c.rename_categories(pd.Series([0, 1], index=['a', 'c']))
FutureWarning: Treating Series 'new_categories' as a list-like and using the values.
In a future version, 'rename_categories' will treat Series like a dictionary.
For dict-like, use 'new_categories.to_dict()'
For list-like, use 'new_categories.values'.
Out[33]:
[0, 0, 1]
Categories (2, int64): [0, 1]
```

Other Enhancements

New functions or methods

- *nearest()* is added to support nearest-neighbor upsampling (GH17496).
- *Index* has added support for a `to_frame` method (GH15230).

New keywords

- Added a `skipna` parameter to *infer_dtype()* to support type inference in the presence of missing values (GH17059).
- *Series.to_dict()* and *DataFrame.to_dict()* now support an `into` keyword which allows you to specify the `collections.Mapping` subclass that you would like returned. The default is `dict`, which is backwards compatible. (GH16122)
- *Series.set_axis()* and *DataFrame.set_axis()* now support the `inplace` parameter. (GH14636)
- *Series.to_pickle()* and *DataFrame.to_pickle()* have gained a `protocol` parameter (GH16252). By default, this parameter is set to HIGHEST_PROTOCOL
- *read_feather()* has gained the `nthreads` parameter for multi-threaded operations (GH16359)
- *DataFrame.clip()* and *Series.clip()* have gained an `inplace` argument. (GH15388)
- *crosstab()* has gained a `margins_name` parameter to define the name of the row / column that will contain the totals when `margins=True`. (GH15972)
- *read_json()* now accepts a `chunksize` parameter that can be used when `lines=True`. If `chunksize` is passed, read_json now returns an iterator which reads in `chunksize` lines with each iteration. (GH17048)
- *read_json()* and *to_json()* now accept a `compression` argument which allows them to transparently handle compressed files. (GH17798)

Various enhancements

- Improved the import time of pandas by about 2.25x. (GH16764)
- Support for PEP 519 – Adding a file system path protocol on most readers (e.g. *read_csv()*) and writers (e.g. *DataFrame.to_csv()*) (GH13823).
- Added a `__fspath__` method to `pd.HDFStore`, `pd.ExcelFile`, and `pd.ExcelWriter` to work properly with the file system path protocol (GH13823).

- The `validate` argument for `merge()` now checks whether a merge is one-to-one, one-to-many, many-to-one, or many-to-many. If a merge is found to not be an example of specified merge type, an exception of type `MergeError` will be raised. For more, see *here* (GH16270)
- Added support for PEP 518 (`pyproject.toml`) to the build system (GH16745)
- `RangeIndex.append()` now returns a `RangeIndex` object when possible (GH16212)
- `Series.rename_axis()` and `DataFrame.rename_axis()` with `inplace=True` now return `None` while renaming the axis inplace. (GH15704)
- `api.types.infer_dtype()` now infers decimals. (GH15690)
- `DataFrame.select_dtypes()` now accepts scalar values for include/exclude as well as list-like. (GH16855)
- `date_range()` now accepts 'YS' in addition to 'AS' as an alias for start of year. (GH9313)
- `date_range()` now accepts 'Y' in addition to 'A' as an alias for end of year. (GH9313)
- `DataFrame.add_prefix()` and `DataFrame.add_suffix()` now accept strings containing the '%' character. (GH17151)
- Read/write methods that infer compression (`read_csv()`, `read_table()`, `read_pickle()`, and `to_pickle()`) can now infer from path-like objects, such as `pathlib.Path`. (GH17206)
- `read_sas()` now recognizes much more of the most frequently used date (datetime) formats in SAS7BDAT files. (GH15871)
- `DataFrame.items()` and `Series.items()` are now present in both Python 2 and 3 and is lazy in all cases. (GH13918, GH17213)
- `pandas.io.formats.style.Styler.where()` has been implemented as a convenience for `pandas.io.formats.style.Styler.applymap()`. (GH17474)
- `MultiIndex.is_monotonic_decreasing()` has been implemented. Previously returned `False` in all cases. (GH16554)
- `read_excel()` raises `ImportError` with a better message if `xlrd` is not installed. (GH17613)
- `DataFrame.assign()` will preserve the original order of `**kwargs` for Python 3.6+ users instead of sorting the column names. (GH14207)
- `Series.reindex()`, `DataFrame.reindex()`, `Index.get_indexer()` now support list-like argument for `tolerance`. (GH17367)

Backwards incompatible API changes

Dependencies have increased minimum versions

We have updated our minimum supported versions of dependencies (GH15206, GH15543, GH15214). If installed, we now require:

Package	Minimum Version	Required
Numpy	1.9.0	X
Matplotlib	1.4.3	
Scipy	0.14.0	
Bottleneck	1.0.0	

Additionally, support has been dropped for Python 3.4 (GH15251).

Sum/Prod of all-NaN or empty Series/DataFrames is now consistently NaN

Note: The changes described here have been partially reverted. See the *v0.22.0 Whatsnew* for more.

The behavior of `sum` and `prod` on all-NaN Series/DataFrames no longer depends on whether bottleneck is installed, and return value of `sum` and `prod` on an empty Series has changed (GH9422, GH15507).

Calling `sum` or `prod` on an empty or all-`NaN` `Series`, or columns of a `DataFrame`, will result in `NaN`. See the *docs*.

```
In [33]: s = pd.Series([np.nan])
```

Previously WITHOUT `bottleneck` installed:

```
In [2]: s.sum()
Out[2]: np.nan
```

Previously WITH `bottleneck`:

```
In [2]: s.sum()
Out[2]: 0.0
```

New Behavior, without regard to the bottleneck installation:

```
In [34]: s.sum()
Out[34]: 0.0
```

Note that this also changes the sum of an empty `Series`. Previously this always returned 0 regardless of a `bottlenck` installation:

```
In [1]: pd.Series([]).sum()
Out[1]: 0
```

but for consistency with the all-NaN case, this was changed to return NaN as well:

```
In [35]: pd.Series([]).sum()
Out[35]: 0.0
```

Indexing with a list with missing labels is Deprecated

Previously, selecting with a list of labels, where one or more labels were missing would always succeed, returning `NaN` for missing labels. This will now show a `FutureWarning`. In the future this will raise a `KeyError` (GH15747). This warning will trigger on a `DataFrame` or a `Series` for using `.loc[]` or `[[]]` when passing a list-of-labels with at least 1 missing label. See the *deprecation docs*.

```
In [36]: s = pd.Series([1, 2, 3])

In [37]: s
Out[37]: 
0    1
1    2
2    3
Length: 3, dtype: int64
```

Previous Behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Current Behavior

```
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-
↪listlike

Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`

```
In [38]: s.reindex([1, 2, 3])
Out[38]:
1    2.0
2    3.0
3    NaN
Length: 3, dtype: float64
```

Selection with all keys found is unchanged.

```
In [39]: s.loc[[1, 2]]
Out[39]:
1    2
2    3
Length: 2, dtype: int64
```

NA naming Changes

In order to promote more consistency among the pandas API, we have added additional top-level functions *isna()* and *notna()* that are aliases for *isnull()* and *notnull()*. The naming scheme is now more consistent with methods like `.dropna()` and `.fillna()`. Furthermore in all cases where `.isnull()` and `.notnull()` methods are defined, these have additional methods named `.isna()` and `.notna()`, these are included for classes `Categorical`, `Index`, `Series`, and `DataFrame`. (GH15001).

The configuration option `pd.options.mode.use_inf_as_null` is deprecated, and `pd.options.mode.use_inf_as_na` is added as a replacement.

Iteration of Series/Index will now return Python scalars

Previously, when using certain iteration methods for a Series with dtype int or float, you would receive a numpy scalar, e.g. a np.int64, rather than a Python int. Issue (GH10904) corrected this for Series.tolist() and list(Series). This change makes all iteration methods consistent, in particular, for __iter__() and .map(); note that this only affects int/float dtypes. (GH13236, GH13258, GH14216).

```
In [40]: s = pd.Series([1, 2, 3])

In [41]: s
Out[41]:
0    1
1    2
2    3
Length: 3, dtype: int64
```

Previously:

```
In [2]: type(list(s)[0])
Out[2]: numpy.int64
```

New Behaviour:

```
In [42]: type(list(s)[0])
Out[42]: int
```

Furthermore this will now correctly box the results of iteration for *DataFrame.to_dict()* as well.

```
In [43]: d = {'a': [1], 'b': ['b']}

In [44]: df = pd.DataFrame(d)
```

Previously:

```
In [8]: type(df.to_dict()['a'][0])
Out[8]: numpy.int64
```

New Behaviour:

```
In [45]: type(df.to_dict()['a'][0])
Out[45]: int
```

Indexing with a Boolean Index

Previously when passing a boolean Index to .loc, if the index of the Series/DataFrame had boolean labels, you would get a label based selection, potentially duplicating result labels, rather than a boolean indexing selection (where True selects elements), this was inconsistent how a boolean numpy array indexed. The new behavior is to act like a boolean numpy array indexer. (GH17738)

Previous Behavior:

```
In [46]: s = pd.Series([1, 2, 3], index=[False, True, False])

In [47]: s
Out[47]:
```

(continues on next page)

(continued from previous page)

```
False    1
True     2
False    3
Length: 3, dtype: int64
```

```
In [59]: s.loc[pd.Index([True, False, True])]
Out[59]:
True     2
False    1
False    3
True     2
dtype: int64
```

Current Behavior

```
In [48]: s.loc[pd.Index([True, False, True])]
Out[48]:
False    1
False    3
Length: 2, dtype: int64
```

Furthermore, previously if you had an index that was non-numeric (e.g. strings), then a boolean Index would raise a `KeyError`. This will now be treated as a boolean indexer.

Previously Behavior:

```
In [49]: s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [50]: s
Out[50]:
a    1
b    2
c    3
Length: 3, dtype: int64
```

```
In [39]: s.loc[pd.Index([True, False, True])]
KeyError: "None of [Index([True, False, True], dtype='object')] are in the [index]"
```

Current Behavior

```
In [51]: s.loc[pd.Index([True, False, True])]
Out[51]:
a    1
c    3
Length: 2, dtype: int64
```

`PeriodIndex` resampling

In previous versions of pandas, resampling a `Series`/`DataFrame` indexed by a `PeriodIndex` returned a `DatetimeIndex` in some cases (GH12884). Resampling to a multiplied frequency now returns a `PeriodIndex` (GH15944). As a minor enhancement, resampling a `PeriodIndex` can now handle `NaT` values (GH13224)

Previous Behavior:

```
In [1]: pi = pd.period_range('2017-01', periods=12, freq='M')

In [2]: s = pd.Series(np.arange(12), index=pi)

In [3]: resampled = s.resample('2Q').mean()

In [4]: resampled
Out[4]:
2017-03-31     1.0
2017-09-30     5.5
2018-03-31    10.0
Freq: 2Q-DEC, dtype: float64

In [5]: resampled.index
Out[5]: DatetimeIndex(['2017-03-31', '2017-09-30', '2018-03-31'], dtype=
↪'datetime64[ns]', freq='2Q-DEC')
```

New Behavior:

```
In [52]: pi = pd.period_range('2017-01', periods=12, freq='M')

In [53]: s = pd.Series(np.arange(12), index=pi)

In [54]: resampled = s.resample('2Q').mean()

In [55]: resampled
Out[55]:
2017Q1    2.5
2017Q3    8.5
Freq: 2Q-DEC, Length: 2, dtype: float64

In [56]: resampled.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:␣
↪PeriodIndex(['2017Q1', '2017Q3'], dtype='period[2Q-DEC]', freq='2Q-DEC')
```

Upsampling and calling `.ohlc()` previously returned a `Series`, basically identical to calling `.asfreq()`. OHLC upsampling now returns a DataFrame with columns `open`, `high`, `low` and `close` (GH13083). This is consistent with downsampling and `DatetimeIndex` behavior.

Previous Behavior:

```
In [1]: pi = pd.period_range(start='2000-01-01', freq='D', periods=10)

In [2]: s = pd.Series(np.arange(10), index=pi)

In [3]: s.resample('H').ohlc()
Out[3]:
2000-01-01 00:00    0.0
                ...
2000-01-10 23:00    NaN
Freq: H, Length: 240, dtype: float64

In [4]: s.resample('M').ohlc()
Out[4]:
         open  high  low  close
2000-01     0     9    0      9
```

New Behavior:

```
In [57]: pi = pd.period_range(start='2000-01-01', freq='D', periods=10)

In [58]: s = pd.Series(np.arange(10), index=pi)

In [59]: s.resample('H').ohlc()
Out[59]:
                  open  high  low  close
2000-01-01 00:00   0.0   0.0  0.0    0.0
2000-01-01 01:00   NaN   NaN  NaN    NaN
2000-01-01 02:00   NaN   NaN  NaN    NaN
2000-01-01 03:00   NaN   NaN  NaN    NaN
2000-01-01 04:00   NaN   NaN  NaN    NaN
2000-01-01 05:00   NaN   NaN  NaN    NaN
2000-01-01 06:00   NaN   NaN  NaN    NaN
...                ...   ...  ...    ...
2000-01-10 17:00   NaN   NaN  NaN    NaN
2000-01-10 18:00   NaN   NaN  NaN    NaN
2000-01-10 19:00   NaN   NaN  NaN    NaN
2000-01-10 20:00   NaN   NaN  NaN    NaN
2000-01-10 21:00   NaN   NaN  NaN    NaN
2000-01-10 22:00   NaN   NaN  NaN    NaN
2000-01-10 23:00   NaN   NaN  NaN    NaN

[240 rows x 4 columns]

In [60]: s.resample('M').ohlc()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
         open  high  low  close
2000-01     0     9    0      9

[1 rows x 4 columns]
```

Improved error handling during item assignment in pd.eval

`eval()` will now raise a `ValueError` when item assignment malfunctions, or inplace operations are specified, but there is no item assignment in the expression (GH16732)

```
In [61]: arr = np.array([1, 2, 3])
```

Previously, if you attempted the following expression, you would get a not very helpful error message:

```
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...
IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`)
and integer or boolean arrays are valid indices
```

This is a very long way of saying numpy arrays don't support string-item indexing. With this change, the error message is now this:

```
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...
ValueError: Cannot assign expression output to target
```

It also used to be possible to evaluate expressions inplace, even if there was no item assignment:

```
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
Out[4]: 3
```

However, this input does not make much sense because the output is not being assigned to the target. Now, a `ValueError` will be raised when such an input is passed in:

```
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
...
ValueError: Cannot operate inplace if there is no assignment
```

Dtype Conversions

Previously assignments, `.where()` and `.fillna()` with a `bool` assignment, would coerce to same the type (e.g. int / float), or raise for datetimelikes. These will now preserve the bools with `object` dtypes. (GH16821).

```
In [62]: s = pd.Series([1, 2, 3])
```

```
In [5]: s[1] = True

In [6]: s
Out[6]:
0    1
1    1
2    3
dtype: int64
```

New Behavior

```
In [63]: s[1] = True

In [64]: s
Out[64]:
0       1
1    True
2       3
Length: 3, dtype: object
```

Previously, as assignment to a datetimelike with a non-datetimelike would coerce the non-datetime-like item being assigned (GH14145).

```
In [65]: s = pd.Series([pd.Timestamp('2011-01-01'), pd.Timestamp('2012-01-01')])
```

```
In [1]: s[1] = 1

In [2]: s
Out[2]:
0   2011-01-01 00:00:00.000000000
1   1970-01-01 00:00:00.000000001
dtype: datetime64[ns]
```

These now coerce to `object` dtype.

```
In [66]: s[1] = 1
```

(continues on next page)

(continued from previous page)

```
In [67]: s
Out[67]:
0    2011-01-01 00:00:00
1                      1
Length: 2, dtype: object
```

- Inconsistent behavior in `.where()` with datetimelikes which would raise rather than coerce to `object` (GH16402)
- Bug in assignment against `int64` data with `np.ndarray` with `float64` dtype may keep `int64` dtype (GH14001)

MultiIndex Constructor with a Single Level

The `MultiIndex` constructors no longer squeezes a MultiIndex with all length-one levels down to a regular `Index`. This affects all the `MultiIndex` constructors. (GH17178)

Previous behavior:

```
In [2]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[2]: Index(['a', 'b'], dtype='object')
```

Length 1 levels are no longer special-cased. They behave exactly as if you had length 2+ levels, so a *`MultiIndex`* is always returned from all of the `MultiIndex` constructors:

```
In [68]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[68]:
MultiIndex(levels=[['a', 'b']],
           codes=[[0, 1]])
```

UTC Localization with Series

Previously, *`to_datetime()`* did not localize datetime `Series` data when `utc=True` was passed. Now, *`to_datetime()`* will correctly localize `Series` with a `datetime64[ns, UTC]` dtype to be consistent with how list-like and `Index` data are handled. (GH6415).

Previous Behavior

```
In [69]: s = pd.Series(['20130101 00:00:00'] * 3)
```

```
In [12]: pd.to_datetime(s, utc=True)
Out[12]:
0   2013-01-01
1   2013-01-01
2   2013-01-01
dtype: datetime64[ns]
```

New Behavior

```
In [70]: pd.to_datetime(s, utc=True)
Out[70]:
0   2013-01-01 00:00:00+00:00
1   2013-01-01 00:00:00+00:00
```

(continues on next page)

(continued from previous page)

```
2   2013-01-01 00:00:00+00:00
Length: 3, dtype: datetime64[ns, UTC]
```

Additionally, DataFrames with datetime columns that were parsed by *read_sql_table()* and *read_sql_query()* will also be localized to UTC only if the original SQL columns were timezone aware datetime columns.

Consistency of Range Functions

In previous versions, there were some inconsistencies between the various range functions: *date_range()*, *bdate_range()*, *period_range()*, *timedelta_range()*, and *interval_range()*. (GH17471).

One of the inconsistent behaviors occurred when the `start`, `end` and `period` parameters were all specified, potentially leading to ambiguous ranges. When all three parameters were passed, `interval_range` ignored the `period` parameter, `period_range` ignored the `end` parameter, and the other range functions raised. To promote consistency among the range functions, and avoid potentially ambiguous ranges, `interval_range` and `period_range` will now raise when all three parameters are passed.

Previous Behavior:

```
 In [2]: pd.interval_range(start=0, end=4, periods=6)
 Out[2]:
 IntervalIndex([(0, 1], (1, 2], (2, 3]]
               closed='right',
               dtype='interval[int64]')

In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
Out[3]: PeriodIndex(['2017Q1', '2017Q2', '2017Q3', '2017Q4', '2018Q1', '2018Q2'],␣
↪dtype='period[Q-DEC]', freq='Q-DEC')
```

New Behavior:

```
In [2]: pd.interval_range(start=0, end=4, periods=6)
---------------------------------------------------------------------------
ValueError: Of the three parameters: start, end, and periods, exactly two must be␣
↪specified

In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
---------------------------------------------------------------------------
ValueError: Of the three parameters: start, end, and periods, exactly two must be␣
↪specified
```

Additionally, the endpoint parameter `end` was not included in the intervals produced by `interval_range`. However, all other range functions include `end` in their output. To promote consistency among the range functions, `interval_range` will now include `end` as the right endpoint of the final interval, except if `freq` is specified in a way which skips `end`.

Previous Behavior:

```
In [4]: pd.interval_range(start=0, end=4)
Out[4]:
IntervalIndex([(0, 1], (1, 2], (2, 3]]
              closed='right',
              dtype='interval[int64]')
```

New Behavior:

```
In [71]: pd.interval_range(start=0, end=4)
Out[71]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4]],
              closed='right',
              dtype='interval[int64]')
```

No Automatic Matplotlib Converters

Pandas no longer registers our `date`, `time`, `datetime`, `datetime64`, and `Period` converters with matplotlib when pandas is imported. Matplotlib plot methods (`plt.plot`, `ax.plot`, ...), will not nicely format the x-axis for `DatetimeIndex` or `PeriodIndex` values. You must explicitly register these methods:

Pandas built-in `Series.plot` and `DataFrame.plot` *will* register these converters on first-use (GH17710).

Note: This change has been temporarily reverted in pandas 0.21.1, for more details see *here*.

Other API Changes

- The Categorical constructor no longer accepts a scalar for the `categories` keyword. (GH16022)
- Accessing a non-existent attribute on a closed `HDFStore` will now raise an `AttributeError` rather than a `ClosedFileError` (GH16301)
- `read_csv()` now issues a `UserWarning` if the `names` parameter contains duplicates (GH17095)
- `read_csv()` now treats `'null'` and `'n/a'` strings as missing values by default (GH16471, GH16078)
- `pandas.HDFStore`'s string representation is now faster and less detailed. For the previous behavior, use `pandas.HDFStore.info()`. (GH16503).
- Compression defaults in HDF stores now follow pytables standards. Default is no compression and if `complib` is missing and `complevel` > 0 `zlib` is used (GH15943)
- `Index.get_indexer_non_unique()` now returns a ndarray indexer rather than an `Index`; this is consistent with `Index.get_indexer()` (GH16819)
- Removed the `@slow` decorator from `pandas.util.testing`, which caused issues for some downstream packages' test suites. Use `@pytest.mark.slow` instead, which achieves the same thing (GH16850)
- Moved definition of `MergeError` to the `pandas.errors` module.
- The signature of `Series.set_axis()` and `DataFrame.set_axis()` has been changed from `set_axis(axis, labels)` to `set_axis(labels, axis=0)`, for consistency with the rest of the API. The old signature is deprecated and will show a `FutureWarning` (GH14636)
- `Series.argmin()` and `Series.argmax()` will now raise a `TypeError` when used with `object` dtypes, instead of a `ValueError` (GH13595)
- `Period` is now immutable, and will now raise an `AttributeError` when a user tries to assign a new value to the `ordinal` or `freq` attributes (GH17116).
- `to_datetime()` when passed a tz-aware `origin=` kwarg will now raise a more informative `ValueError` rather than a `TypeError` (GH16842)
- `to_datetime()` now raises a `ValueError` when format includes `%W` or `%U` without also including day of the week and calendar year (GH16774)

- Renamed non-functional `index` to `index_col` in `read_stata()` to improve API consistency (GH16342)
- Bug in `DataFrame.drop()` caused boolean labels `False` and `True` to be treated as labels 0 and 1 respectively when dropping indices from a numeric index. This will now raise a ValueError (GH16877)
- Restricted DateOffset keyword arguments. Previously, `DateOffset` subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176).

Deprecations

- `DataFrame.from_csv()` and `Series.from_csv()` have been deprecated in favor of `read_csv()` (GH4191)
- `read_excel()` has deprecated `sheetname` in favor of `sheet_name` for consistency with `.to_excel()` (GH10559).
- `read_excel()` has deprecated `parse_cols` in favor of `usecols` for consistency with `read_csv()` (GH4988)
- `read_csv()` has deprecated the `tupleize_cols` argument. Column tuples will always be converted to a `MultiIndex` (GH17060)
- `DataFrame.to_csv()` has deprecated the `tupleize_cols` argument. MultiIndex columns will be always written as rows in the CSV file (GH17060)
- The `convert` parameter has been deprecated in the `.take()` method, as it was not being respected (GH16948)
- `pd.options.html.border` has been deprecated in favor of `pd.options.display.html.border` (GH15793).
- `SeriesGroupBy.nth()` has deprecated `True` in favor of `'all'` for its kwarg `dropna` (GH11038).
- `DataFrame.as_blocks()` is deprecated, as this is exposing the internal implementation (GH17302)
- `pd.TimeGrouper` is deprecated in favor of `pandas.Grouper` (GH16747)
- `cdate_range` has been deprecated in favor of `bdate_range()`, which has gained `weekmask` and `holidays` parameters for building custom frequency date ranges. See the *documentation* for more details (GH17596)
- passing `categories` or `ordered` kwargs to `Series.astype()` is deprecated, in favor of passing a *CategoricalDtype* (GH17636)
- `.get_value` and `.set_value` on `Series`, `DataFrame`, `Panel`, `SparseSeries`, and `SparseDataFrame` are deprecated in favor of using `.iat[]` or `.at[]` accessors (GH15269)
- Passing a non-existent column in `.to_excel(..., columns=)` is deprecated and will raise a `KeyError` in the future (GH17295)
- `raise_on_error` parameter to `Series.where()`, `Series.mask()`, `DataFrame.where()`, `DataFrame.mask()` is deprecated, in favor of `errors=` (GH14968)
- Using `DataFrame.rename_axis()` and `Series.rename_axis()` to alter index or column *labels* is now deprecated in favor of using `.rename`. `rename_axis` may still be used to alter the name of the index or columns (GH17833).
- `reindex_axis()` has been deprecated in favor of `reindex()`. See *here* for more (GH17833).

Series.select and DataFrame.select

The *Series.select()* and *DataFrame.select()* methods are deprecated in favor of using `df.loc[labels.map(crit)]` (GH12401)

```
In [72]: df = pd.DataFrame({'A': [1, 2, 3]}, index=['foo', 'bar', 'baz'])
```

```
In [3]: df.select(lambda x: x in ['bar', 'baz'])
FutureWarning: select is deprecated and will be removed in a future release. You can␣
↪use .loc[crit] as a replacement
Out[3]:
     A
bar  2
baz  3
```

```
In [73]: df.loc[df.index.map(lambda x: x in ['bar', 'baz'])]
Out[73]:
     A
bar  2
baz  3

[2 rows x 1 columns]
```

Series.argmax and Series.argmin

The behavior of *Series.argmax()* and *Series.argmin()* have been deprecated in favor of *Series.idxmax()* and *Series.idxmin()*, respectively (GH16830).

For compatibility with NumPy arrays, `pd.Series` implements `argmax` and `argmin`. Since pandas 0.13.0, `argmax` has been an alias for *pandas.Series.idxmax()*, and `argmin` has been an alias for *pandas.Series.idxmin()*. They return the *label* of the maximum or minimum, rather than the *position*.

We've deprecated the current behavior of `Series.argmax` and `Series.argmin`. Using either of these will emit a `FutureWarning`. Use *Series.idxmax()* if you want the label of the maximum. Use `Series.values.argmax()` if you want the position of the maximum. Likewise for the minimum. In a future release `Series.argmax` and `Series.argmin` will return the position of the maximum or minimum.

Removal of prior version deprecations/changes

- *read_excel()* has dropped the `has_index_names` parameter (GH10967)
- The `pd.options.display.height` configuration has been dropped (GH3663)
- The `pd.options.display.line_width` configuration has been dropped (GH2881)
- The `pd.options.display.mpl_style` configuration has been dropped (GH12190)
- `Index` has dropped the `.sym_diff()` method in favor of `.symmetric_difference()` (GH12591)
- `Categorical` has dropped the `.order()` and `.sort()` methods in favor of `.sort_values()` (GH12882)
- *eval()* and *DataFrame.eval()* have changed the default of `inplace` from `None` to `False` (GH11149)
- The function `get_offset_name` has been dropped in favor of the `.freqstr` attribute for an offset (GH11834)

- pandas no longer tests for compatibility with hdf5-files created with pandas < 0.11 (GH17404).

Performance Improvements

- Improved performance of instantiating `SparseDataFrame` (GH16773)
- *`Series.dt`* no longer performs frequency inference, yielding a large speedup when accessing the attribute (GH17210)
- Improved performance of *`set_categories()`* by not materializing the values (GH17508)
- *`Timestamp.microsecond`* no longer re-computes on attribute access (GH17331)
- Improved performance of the *`CategoricalIndex`* for data that is already categorical dtype (GH17513)
- Improved performance of `RangeIndex.min()` and `RangeIndex.max()` by using `RangeIndex` properties to perform the computations (GH17607)

Documentation Changes

- Several `NaT` method docstrings (e.g. `NaT.ctime()`) were incorrect (GH17327)
- The documentation has had references to versions < v0.17 removed and cleaned up (GH17442, GH17442, GH17404 & GH17504)

Bug Fixes

Conversion

- Bug in assignment against datetime-like data with `int` may incorrectly convert to datetime-like (GH14145)
- Bug in assignment against `int64` data with `np.ndarray` with `float64` dtype may keep `int64` dtype (GH14001)
- Fixed the return type of `IntervalIndex.is_non_overlapping_monotonic` to be a Python `bool` for consistency with similar attributes/methods. Previously returned a `numpy.bool_`. (GH17237)
- Bug in `IntervalIndex.is_non_overlapping_monotonic` when intervals are closed on both sides and overlap at a point (GH16560)
- Bug in *`Series.fillna()`* returns frame when `inplace=True` and `value` is dict (GH16156)
- Bug in *`Timestamp.weekday_name`* returning a UTC-based weekday name when localized to a timezone (GH17354)
- Bug in `Timestamp.replace` when replacing `tzinfo` around DST changes (GH15683)
- Bug in `Timedelta` construction and arithmetic that would not propagate the `Overflow` exception (GH17367)
- Bug in *`astype()`* converting to object dtype when passed extension type classes (`DatetimeTZDtype`, `CategoricalDtype`) rather than instances. Now a `TypeError` is raised when a class is passed (GH17780).
- Bug in *`to_numeric()`* in which elements were not always being coerced to numeric when `errors='coerce'` (GH17007, GH17125)
- Bug in `DataFrame` and `Series` constructors where `range` objects are converted to `int32` dtype on Windows instead of `int64` (GH16804)

Indexing

- When called with a null slice (e.g. `df.iloc[:]`), the `.iloc` and `.loc` indexers return a shallow copy of the original object. Previously they returned the original object. (GH13873).
- When called on an unsorted `MultiIndex`, the `loc` indexer now will raise `UnsortedIndexError` only if proper slicing is used on non-sorted levels (GH16734).
- Fixes regression in 0.20.3 when indexing with a string on a `TimedeltaIndex` (GH16896).
- Fixed `TimedeltaIndex.get_loc()` handling of `np.timedelta64` inputs (GH16909).
- Fix `MultiIndex.sort_index()` ordering when `ascending` argument is a list, but not all levels are specified, or are in a different order (GH16934).
- Fixes bug where indexing with `np.inf` caused an `OverflowError` to be raised (GH16957)
- Bug in reindexing on an empty `CategoricalIndex` (GH16770)
- Fixes `DataFrame.loc` for setting with alignment and tz-aware `DatetimeIndex` (GH16889)
- Avoids `IndexError` when passing an Index or Series to `.iloc` with older numpy (GH17193)
- Allow unicode empty strings as placeholders in multilevel columns in Python 2 (GH17099)
- Bug in `.iloc` when used with inplace addition or assignment and an int indexer on a `MultiIndex` causing the wrong indexes to be read from and written to (GH17148)
- Bug in `.isin()` in which checking membership in empty `Series` objects raised an error (GH16991)
- Bug in `CategoricalIndex` reindexing in which specified indices containing duplicates were not being respected (GH17323)
- Bug in intersection of `RangeIndex` with negative step (GH17296)
- Bug in `IntervalIndex` where performing a scalar lookup fails for included right endpoints of non-overlapping monotonic decreasing indexes (GH16417, GH17271)
- Bug in *`DataFrame.first_valid_index()`* and *`DataFrame.last_valid_index()`* when no valid entry (GH17400)
- Bug in *`Series.rename()`* when called with a callable, incorrectly alters the name of the `Series`, rather than the name of the `Index`. (GH17407)
- Bug in `String.str_get()` raises `IndexError` instead of inserting NaNs when using a negative index. (GH17704)

I/O

- Bug in *`read_hdf()`* when reading a timezone aware index from `fixed` format HDFStore (GH17618)
- Bug in *`read_csv()`* in which columns were not being thoroughly de-duplicated (GH17060)
- Bug in *`read_csv()`* in which specified column names were not being thoroughly de-duplicated (GH17095)
- Bug in *`read_csv()`* in which non integer values for the header argument generated an unhelpful / unrelated error message (GH16338)
- Bug in *`read_csv()`* in which memory management issues in exception handling, under certain conditions, would cause the interpreter to segfault (GH14696, GH16798).
- Bug in *`read_csv()`* when called with `low_memory=False` in which a CSV with at least one column > 2GB in size would incorrectly raise a `MemoryError` (GH16798).

- Bug in *read_csv()* when called with a single-element list header would return a DataFrame of all NaN values (GH7757)
- Bug in *DataFrame.to_csv()* defaulting to 'ascii' encoding in Python 3, instead of 'utf-8' (GH17097)
- Bug in *read_stata()* where value labels could not be read when using an iterator (GH16923)
- Bug in *read_stata()* where the index was not set (GH16342)
- Bug in *read_html()* where import check fails when run in multiple threads (GH16928)
- Bug in *read_csv()* where automatic delimiter detection caused a TypeError to be thrown when a bad line was encountered rather than the correct error message (GH13374)
- Bug in *DataFrame.to_html()* with notebook=True where DataFrames with named indices or non-MultiIndex indices had undesired horizontal or vertical alignment for column or row labels, respectively (GH16792)
- Bug in *DataFrame.to_html()* in which there was no validation of the justify parameter (GH17527)
- Bug in *HDFStore.select()* when reading a contiguous mixed-data table featuring VLArray (GH17021)
- Bug in to_json() where several conditions (including objects with unprintable symbols, objects with deep recursion, overlong labels) caused segfaults instead of raising the appropriate exception (GH14256)

Plotting

- Bug in plotting methods using secondary_y and fontsize not setting secondary axis font size (GH12565)
- Bug when plotting timedelta and datetime dtypes on y-axis (GH16953)
- Line plots no longer assume monotonic x data when calculating xlims, they show the entire lines now even for unsorted x data. (GH11310, GH11471)
- With matplotlib 2.0.0 and above, calculation of x limits for line plots is left to matplotlib, so that its new default settings are applied. (GH15495)
- Bug in Series.plot.bar or DataFrame.plot.bar with y not respecting user-passed color (GH16822)
- Bug causing plotting.parallel_coordinates to reset the random seed when using random colors (GH17525)

Groupby/Resample/Rolling

- Bug in DataFrame.resample(...).size() where an empty DataFrame did not return a Series (GH14962)
- Bug in *infer_freq()* causing indices with 2-day gaps during the working week to be wrongly inferred as business daily (GH16624)
- Bug in .rolling(...).quantile() which incorrectly used different defaults than *Series.quantile()* and *DataFrame.quantile()* (GH9413, GH16211)
- Bug in groupby.transform() that would coerce boolean dtypes back to float (GH16875)
- Bug in Series.resample(...).apply() where an empty Series modified the source index and did not return the name of a Series (GH14313)
- Bug in .rolling(...).apply(...) with a DataFrame with a DatetimeIndex, a window of a timedelta-convertible and min_periods >= 1 (GH15305)

- Bug in `DataFrame.groupby` where index and column keys were not recognized correctly when the number of keys equaled the number of elements on the groupby axis (GH16859)
- Bug in `groupby.nunique()` with `TimeGrouper` which cannot handle `NaT` correctly (GH17575)
- Bug in `DataFrame.groupby` where a single level selection from a `MultiIndex` unexpectedly sorts (GH17537)
- Bug in `DataFrame.groupby` where spurious warning is raised when `Grouper` object is used to override ambiguous column name (GH17383)
- Bug in `TimeGrouper` differs when passes as a list and as a scalar (GH17530)

Sparse

- Bug in `SparseSeries` raises `AttributeError` when a dictionary is passed in as data (GH16905)
- Bug in `SparseDataFrame.fillna()` not filling all NaNs when frame was instantiated from SciPy sparse matrix (GH16112)
- Bug in `SparseSeries.unstack()` and `SparseDataFrame.stack()` (GH16614, GH15045)
- Bug in `make_sparse()` treating two numeric/boolean data, which have same bits, as same when array `dtype` is `object` (GH17574)
- *`SparseArray.all()`* and *`SparseArray.any()`* are now implemented to handle `SparseArray`, these were used but not implemented (GH17570)

Reshaping

- Joining/Merging with a non unique `PeriodIndex` raised a `TypeError` (GH16871)
- Bug in *`crosstab()`* where non-aligned series of integers were casted to float (GH17005)
- Bug in merging with categorical dtypes with datetimelikes incorrectly raised a `TypeError` (GH16900)
- Bug when using `isin()` on a large object series and large comparison array (GH16012)
- Fixes regression from 0.20, *`Series.aggregate()`* and *`DataFrame.aggregate()`* allow dictionaries as return values again (GH16741)
- Fixes dtype of result with integer dtype input, from *`pivot_table()`* when called with `margins=True` (GH17013)
- Bug in *`crosstab()`* where passing two `Series` with the same name raised a `KeyError` (GH13279)
- *`Series.argmin()`*, *`Series.argmax()`*, and their counterparts on `DataFrame` and groupby objects work correctly with floating point data that contains infinite values (GH13595).
- Bug in *`unique()`* where checking a tuple of strings raised a `TypeError` (GH17108)
- Bug in *`concat()`* where order of result index was unpredictable if it contained non-comparable elements (GH17344)
- Fixes regression when sorting by multiple columns on a `datetime64` dtype `Series` with `NaT` values (GH16836)
- Bug in *`pivot_table()`* where the result's columns did not preserve the categorical dtype of `columns` when `dropna` was `False` (GH17842)
- Bug in `DataFrame.drop_duplicates` where dropping with non-unique column names raised a `ValueError` (GH17836)

- Bug in `unstack()` which, when called on a list of levels, would discard the `fillna` argument (GH13971)
- Bug in the alignment of `range` objects and other list-likes with `DataFrame` leading to operations being performed row-wise instead of column-wise (GH17901)

Numeric

- Bug in `.clip()` with `axis=1` and a list-like for `threshold` is passed; previously this raised `ValueError` (GH15390)
- *`Series.clip()`* and *`DataFrame.clip()`* now treat NA values for upper and lower arguments as `None` instead of raising `ValueError` (GH17276).

Categorical

- Bug in *`Series.isin()`* when called with a categorical (GH16639)
- Bug in the categorical constructor with empty values and categories causing the `.categories` to be an empty `Float64Index` rather than an empty `Index` with object dtype (GH17248)
- Bug in categorical operations with *Series.cat* not preserving the original Series' name (GH17509)
- Bug in *`DataFrame.merge()`* failing for categorical columns with boolean/int data types (GH17187)
- Bug in constructing a `Categorical`/`CategoricalDtype` when the specified `categories` are of categorical type (GH17884).

PyPy

- Compatibility with PyPy in *`read_csv()`* with `usecols=[<unsorted ints>]` and *`read_json()`* (GH17351)
- Split tests into cases for CPython and PyPy where needed, which highlights the fragility of index matching with `float('nan')`, `np.nan` and `NAT` (GH17351)
- Fix *`DataFrame.memory_usage()`* to support PyPy. Objects on PyPy do not have a fixed size, so an approximation is used instead (GH17228)

Other

- Bug where some inplace operators were not being wrapped and produced a copy when invoked (GH12962)
- Bug in *`eval()`* where the `inplace` parameter was being incorrectly handled (GH16732)

Contributors

A total of 206 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- 3553x +
- Aaron Barber
- Adam Gleave +
- Adam Smith +

- AdamShamlian +
- Adrian Liaw +
- Alan Velasco +
- Alan Yee +
- Alex B +
- Alex Lubbock +
- Alex Marchenko +
- Alex Rychyk +
- Amol K +
- Andreas Winkler
- Andrew +
- Andrew
- André Jonasson +
- Becky Sweger
- Berkay +
- Bob Haffner +
- Bran Yang
- Brian Tu +
- Brock Mendel +
- Carol Willing +
- Carter Green +
- Chankey Pathak +
- Chris
- Chris Billington
- Chris Filo Gorgolewski +
- Chris Kerr
- Chris M +
- Chris Mazzullo +
- Christian Prinoth
- Christian Stade-Schuldt
- Christoph Moehl +
- DSM
- Daniel Chen +
- Daniel Grady
- Daniel Himmelstein
- Dave Willmer

- David Cook
- David Gwynne
- David Read +
- Dillon Niederhut +
- Douglas Rudd
- Eric Stein +
- Eric Wieser +
- Erik Fredriksen
- Florian Wilhelm +
- Floris Kint +
- Forbidden Donut
- Gabe F +
- Giftlin +
- Giftlin Rajaiah +
- Giulio Pepe +
- Guilherme Beltramini
- Guillem Borrell +
- Hanmin Qin +
- Hendrik Makait +
- Hugues Valois
- Hussain Tamboli +
- Iva Miholic +
- Jan Novotný +
- Jan Rudolph
- Jean Helie +
- Jean-Baptiste Schiratti +
- Jean-Mathieu Deschenes
- Jeff Knupp +
- Jeff Reback
- Jeff Tratner
- JennaVergeynst
- JimStearns206
- Joel Nothman
- John W. O'Brien
- Jon Crall +
- Jon Mease

- Jonathan J. Helmus +
- Joris Van den Bossche
- JosephWagner
- Juarez Bochi
- Julian Kuhlmann +
- Karel De Brabandere
- Kassandra Keeton +
- Keiron Pizzey +
- Keith Webber
- Kernc
- Kevin Sheppard
- Kirk Hansen +
- Licht Takeuchi +
- Lucas Kushner +
- Mahdi Ben Jelloul +
- Makarov Andrey +
- Malgorzata Turzanska +
- Marc Garcia +
- Margaret Sy +
- MarsGuy +
- Matt Bark +
- Matthew Roeschke
- Matti Picus
- Mehmet Ali "Mali" Akmanalp
- Michael Gasvoda +
- Michael Penkov +
- Milo +
- Morgan Stuart +
- Morgan243 +
- Nathan Ford +
- Nick Eubank
- Nick Garvey +
- Oleg Shteynbuk +
- P-Tillmann +
- Pankaj Pandey
- Patrick Luo

- Patrick O'Melveny
- Paul Reidy +
- Paula +
- Peter Quackenbush
- Peter Yanovich +
- Phillip Cloud
- Pierre Haessig
- Pietro Battiston
- Pradyumna Reddy Chinthala
- Prasanjit Prakash
- RobinFiveWords
- Ryan Hendrickson
- Sam Foo
- Sangwoong Yoon +
- Simon Gibbons +
- SimonBaron
- Steven Cutting +
- Sudeep +
- Sylvia +
- T N +
- Telt
- Thomas A Caswell
- Tim Swast +
- Tom Augspurger
- Tong SHEN
- Tuan +
- Utkarsh Upadhyay +
- Vincent La +
- Vivek +
- WANG Aiyong
- WBare
- Wes McKinney
- XF +
- Yi Liu +
- Yosuke Nakabayashi +
- aaron315 +

- abarber4gh +
- aernlund +
- agustín méndez +
- andymaheshw +
- ante328 +
- aviolov +
- bpraggastis
- cbertinato +
- cclauss +
- chernrick
- chris-b1
- dkamm +
- dwkenefick
- economy
- faic +
- fding253 +
- gfyoung
- guygoldberg +
- hhuuggoo +
- huashuai +
- ian
- iulia +
- jaredsnyder
- jbrockmendel +
- jdeschenes
- jebob +
- jschendel +
- keitakurita
- kernc +
- kiwirob +
- kjford
- linebp
- lloydkirk
- louispotok +
- majiang +
- manikbhandari +

- matthiashuschle +
- mattip
- maxwasserman +
- mjlove12 +
- nmartensen +
- pandas-docs-bot +
- parchd-1 +
- philipphanemann +
- rdk1024 +
- reidy-p +
- ri938
- ruiann +
- rvernica +
- s-weigand +
- scotthavard92 +
- skwbc +
- step4me +
- tobycheese +
- topper-123 +
- tsdlovell
- ysau +
- zzgao +

8.5 Version 0.20

8.5.1 v0.20.3 (July 7, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

What's new in v0.20.3

- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Reshaping*

Bug Fixes

- Fixed a bug in failing to compute rolling computations of a column-MultiIndexed `DataFrame` (GH16789, GH16825)
- Fixed a pytest marker failing downstream packages' tests suites (GH16680)

Conversion

- Bug in pickle compat prior to the v0.20.x series, when `UTC` is a timezone in a Series/DataFrame/Index (GH16608)
- Bug in `Series` construction when passing a `Series` with `dtype='category'` (GH16524).
- Bug in *`DataFrame.astype()`* when passing a `Series` as the `dtype` kwarg. (GH16717).

Indexing

- Bug in `Float64Index` causing an empty array instead of `None` to be returned from `.get(np.nan)` on a Series whose index did not contain any `NaN` s (GH8569)
- Bug in `MultiIndex.isin` causing an error when passing an empty iterable (GH16777)
- Fixed a bug in a slicing DataFrame/Series that have a `TimedeltaIndex` (GH16637)

I/O

- Bug in *`read_csv()`* in which files weren't opened as binary files by the C engine on Windows, causing EOF characters mid-field, which would fail (GH16039, GH16559, GH16675)
- Bug in *`read_hdf()`* in which reading a `Series` saved to an HDF file in 'fixed' format fails when an explicit `mode='r'` argument is supplied (GH16583)
- Bug in *`DataFrame.to_latex()`* where `bold_rows` was wrongly specified to be `True` by default, whereas in reality row labels remained non-bold whatever parameter provided. (GH16707)
- Fixed an issue with *`DataFrame.style()`* where generated element ids were not unique (GH16780)
- Fixed loading a `DataFrame` with a `PeriodIndex`, from a `format='fixed'` HDFStore, in Python 3, that was written in Python 2 (GH16781)

Plotting

- Fixed regression that prevented RGB and RGBA tuples from being used as color arguments (GH16233)
- Fixed an issue with *`DataFrame.plot.scatter()`* that incorrectly raised a `KeyError` when categorical data is used for plotting (GH16199)

Reshaping

- `PeriodIndex` / `TimedeltaIndex.join` was missing the `sort=` kwarg (GH16541)
- Bug in joining on a `MultiIndex` with a `category` dtype for a level (GH16627).
- Bug in *merge()* when merging/joining with multiple categorical columns (GH16767)

Categorical

- Bug in `DataFrame.sort_values` not respecting the `kind` parameter with categorical data (GH16793)

Contributors

A total of 20 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Bran Yang
- Chris
- Chris Kerr +
- DSM
- David Gwynne
- Douglas Rudd
- Forbidden Donut +
- Jeff Reback
- Joris Van den Bossche
- Karel De Brabandere +
- Peter Quackenbush +
- Pradyumna Reddy Chinthala +
- Telt +
- Tom Augspurger
- chris-b1
- gfyoung
- ian +
- jdeschenes +
- kjford +
- ri938 +

8.5.2 v0.20.2 (June 4, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

What's new in v0.20.2

- *Enhancements*
- *Performance Improvements*
- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Groupby/Resample/Rolling*
  - *Sparse*
  - *Reshaping*
  - *Numeric*
  - *Categorical*
  - *Other*
- *Contributors*

Enhancements

- Unblocked access to additional compression types supported in pytables: 'blosc:blosclz, 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd' (GH14478)
- `Series` provides a `to_latex` method (GH16180)
- A new groupby method *ngroup()*, parallel to the existing *cumcount()*, has been added to return the group order (GH11642); see *here*.

Performance Improvements

- Performance regression fix when indexing with a list-like (GH16285)
- Performance regression fix for MultiIndexes (GH16319, GH16346)
- Improved performance of `.clip()` with scalar arguments (GH15400)
- Improved performance of groupby with categorical groupers (GH16413)
- Improved performance of `MultiIndex.remove_unused_levels()` (GH16556)

Bug Fixes

- Silenced a warning on some Windows environments about "tput: terminal attributes: No such device or address" when detecting the terminal size. This fix only applies to python 3 (GH16496)
- Bug in using `pathlib.Path` or `py.path.local` objects with io functions (GH16291)
- Bug in `Index.symmetric_difference()` on two equal MultiIndex's, results in a `TypeError` (GH13490)

- Bug in DataFrame.update() with overwrite=False and NaN values (GH15593)
- Passing an invalid engine to *read_csv()* now raises an informative ValueError rather than UnboundLocalError. (GH16511)
- Bug in *unique()* on an array of tuples (GH16519)
- Bug in *cut()* when labels are set, resulting in incorrect label ordering (GH16459)
- Fixed a compatibility issue with IPython 6.0's tab completion showing deprecation warnings on Categoricals (GH16409)

Conversion

- Bug in *to_numeric()* in which empty data inputs were causing a segfault of the interpreter (GH16302)
- Silence numpy warnings when broadcasting DataFrame to Series with comparison ops (GH16378, GH16306)

Indexing

- Bug in DataFrame.reset_index(level=) with single level index (GH16263)
- Bug in partial string indexing with a monotonic, but not strictly-monotonic, index incorrectly reversing the slice bounds (GH16515)
- Bug in MultiIndex.remove_unused_levels() that would not return a MultiIndex equal to the original. (GH16556)

I/O

- Bug in *read_csv()* when comment is passed in a space delimited text file (GH16472)
- Bug in *read_csv()* not raising an exception with nonexistent columns in usecols when it had the correct length (GH14671)
- Bug that would force importing of the clipboard routines unnecessarily, potentially causing an import error on startup (GH16288)
- Bug that raised IndexError when HTML-rendering an empty DataFrame (GH15953)
- Bug in *read_csv()* in which tarfile object inputs were raising an error in Python 2.x for the C engine (GH16530)
- Bug where DataFrame.to_html() ignored the index_names parameter (GH16493)
- Bug where pd.read_hdf() returns numpy strings for index names (GH13492)
- Bug in HDFStore.select_as_multiple() where start/stop arguments were not respected (GH16209)

Plotting

- Bug in DataFrame.plot with a single column and a list-like color (GH3486)
- Bug in plot where NaT in DatetimeIndex results in Timestamp.min (GH12405)
- Bug in DataFrame.boxplot where figsize keyword was not respected for non-grouped boxplots (GH11959)

Groupby/Resample/Rolling

- Bug in creating a time-based rolling window on an empty `DataFrame` (GH15819)
- Bug in `rolling.cov()` with offset window (GH16058)
- Bug in `.resample()` and `.groupby()` when aggregating on integers (GH16361)

Sparse

- Bug in construction of `SparseDataFrame` from `scipy.sparse.dok_matrix` (GH16179)

Reshaping

- Bug in `DataFrame.stack` with unsorted levels in `MultiIndex` columns (GH16323)
- Bug in `pd.wide_to_long()` where no error was raised when `i` was not a unique identifier (GH16382)
- Bug in `Series.isin(..)` with a list of tuples (GH16394)
- Bug in construction of a `DataFrame` with mixed dtypes including an all-NaT column. (GH16395)
- Bug in `DataFrame.agg()` and `Series.agg()` with aggregating on non-callable attributes (GH16405)

Numeric

- Bug in `.interpolate()`, where `limit_direction` was not respected when `limit=None` (default) was passed (GH16282)

Categorical

- Fixed comparison operations considering the order of the categories when both categoricals are unordered (GH16014)

Other

- Bug in `DataFrame.drop()` with an empty-list with non-unique indices (GH16270)

Contributors

A total of 34 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Barber +
- Andrew +
- Becky Sweger +
- Christian Prinoth +
- Christian Stade-Schuldt +
- DSM

- Erik Fredriksen +
- Hugues Valois +
- Jeff Reback
- Jeff Tratner
- JimStearns206 +
- John W. O'Brien
- Joris Van den Bossche
- JosephWagner +
- Keith Webber +
- Mehmet Ali "Mali" Akmanalp +
- Pankaj Pandey
- Patrick Luo +
- Patrick O'Melveny +
- Pietro Battiston
- RobinFiveWords +
- Ryan Hendrickson +
- SimonBaron +
- Tom Augspurger
- WBare +
- bpraggastis +
- chernrick +
- chris-b1
- economy +
- gfyoung
- jaredsnyder +
- keitakurita +
- linebp
- lloydkirk +

8.5.3 v0.20.1 (May 5, 2017)

This is a major release from 0.19.2 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- New `.agg()` API for Series/DataFrame similar to the groupby-rolling-resample API's, see *here*
- Integration with the `feather-format`, including a new top-level `pd.read_feather()` and `DataFrame.to_feather()` method, see *here*.

- The `.ix` indexer has been deprecated, see *here*
- `Panel` has been deprecated, see *here*
- Addition of an `IntervalIndex` and `Interval` scalar type, see *here*
- Improved user API when grouping by index levels in `.groupby()`, see *here*
- Improved support for `UInt64` dtypes, see *here*
- A new orient for JSON serialization, `orient='table'`, that uses the Table Schema spec and that gives the possibility for a more interactive repr in the Jupyter Notebook, see *here*
- Experimental support for exporting styled DataFrames (`DataFrame.style`) to Excel, see *here*
- Window binary corr/cov operations now return a MultiIndexed `DataFrame` rather than a `Panel`, as `Panel` is now deprecated, see *here*
- Support for S3 handling now uses `s3fs`, see *here*
- Google BigQuery support now uses the `pandas-gbq` library, see *here*

Warning: Pandas has changed the internal structure and layout of the code base. This can affect imports that are not from the top-level `pandas.*` namespace, please see the changes *here*.

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

Note: This is a combined release for 0.20.0 and and 0.20.1. Version 0.20.1 contains one additional change for backwards-compatibility with downstream projects using pandas' `utils` routines. (GH16250)

What's new in v0.20.0

New features

agg API for DataFrame/Series

Series & DataFrame have been enhanced to support the aggregation API. This is a familiar API from groupby, window operations, and resampling. This allows aggregation operations in a concise way by using `agg()` and `transform()`. The full documentation is *here* (GH1623).

Here is a sample

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
   ...:                   index=pd.date_range('1/1/2000', periods=10))
   ...:

In [2]: df.iloc[3:7] = np.nan

In [3]: df
Out[3]:
                   A         B         C
2000-01-01  0.469112 -0.282863 -1.509059
2000-01-02 -1.135632  1.212112 -0.173215
2000-01-03  0.119209 -1.044236 -0.861849
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.113648 -1.478427  0.524988
2000-01-09  0.404705  0.577046 -1.715002
2000-01-10 -1.039268 -0.370647 -1.157892

[10 rows x 3 columns]
```

One can operate using string function names, callables, lists, or dictionaries of these.

Using a single function is equivalent to `.apply`.

```
In [4]: df.agg('sum')
Out[4]:
A   -1.068226
B   -1.387015
C   -4.892029
Length: 3, dtype: float64
```

Multiple aggregations with a list of functions.

```
In [5]: df.agg(['sum', 'min'])
Out[5]:
            A         B         C
sum -1.068226 -1.387015 -4.892029
min -1.135632 -1.478427 -1.715002

[2 rows x 3 columns]
```

Using a dict provides the ability to apply specific aggregations per column. You will get a matrix-like output of all of the aggregators. The output has one column per unique function. Those functions applied to a particular column will be NaN:

```
In [6]: df.agg({'A': ['sum', 'min'], 'B': ['min', 'max']})
Out[6]:
            A         B
max       NaN  1.212112
min -1.135632 -1.478427
sum -1.068226       NaN

[3 rows x 2 columns]
```

The API also supports a .transform() function for broadcasting results.

```
In [7]: df.transform(['abs', lambda x: x - x.min()])
Out[7]:
                   A                   B                   C          
                 abs  <lambda>       abs  <lambda>       abs  <lambda>
2000-01-01  0.469112  1.604745  0.282863  1.195563  1.509059  0.205944
2000-01-02  1.135632  0.000000  1.212112  2.690539  0.173215  1.541787
2000-01-03  0.119209  1.254841  1.044236  0.434191  0.861849  0.853153
2000-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  0.113648  1.249281  1.478427  0.000000  0.524988  2.239990
2000-01-09  0.404705  1.540338  0.577046  2.055473  1.715002  0.000000
2000-01-10  1.039268  0.096364  0.370647  1.107780  1.157892  0.557110

[10 rows x 6 columns]
```

When presented with mixed dtypes that cannot be aggregated, .agg() will only take the valid aggregations. This is similar to how groupby .agg() works. (GH15015)

```
In [8]: df = pd.DataFrame({'A': [1, 2, 3],
   ...:                    'B': [1., 2., 3.],
   ...:                    'C': ['foo', 'bar', 'baz'],
   ...:                    'D': pd.date_range('20130101', periods=3)})
   ...:

In [9]: df.dtypes
Out[9]:
A             int64
B           float64
C            object
D    datetime64[ns]
Length: 4, dtype: object
```

```
In [10]: df.agg(['min', 'sum'])
Out[10]:
     A    B          C          D
min  1  1.0        bar 2013-01-01
sum  6  6.0  foobarbaz        NaT

[2 rows x 4 columns]
```

dtype keyword for data IO

The 'python' engine for *read_csv()*, as well as the *read_fwf()* function for parsing fixed-width text files and *read_excel()* for parsing Excel files, now accept the dtype keyword argument for specifying the types of specific columns (GH14295). See the *io docs* for more information.

```
In [11]: data = "a  b\n1  2\n3  4"

In [12]: pd.read_fwf(StringIO(data)).dtypes
Out[12]:
a    int64
b    int64
Length: 2, dtype: object

In [13]: pd.read_fwf(StringIO(data), dtype={'a': 'float64', 'b': 'object'}).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]:
a    float64
b     object
Length: 2, dtype: object
```

.to_datetime() has gained an origin parameter

to_datetime() has gained a new parameter, origin, to define a reference date from where to compute the resulting timestamps when parsing numerical values with a specific unit specified. (GH11276, GH11745)

For example, with 1960-01-01 as the starting date:

```
In [14]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[14]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

The default is set at origin='unix', which defaults to 1970-01-01 00:00:00, which is commonly called 'unix epoch' or POSIX time. This was the previous default, so this is a backward compatible change.

```
In [15]: pd.to_datetime([1, 2, 3], unit='D')
Out[15]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype=
↪'datetime64[ns]', freq=None)
```

Groupby Enhancements

Strings passed to DataFrame.groupby() as the by parameter may now reference either column names or index level names. Previously, only column names could be referenced. This allows to easily group by a column and index level at the same time. (GH5677)

```
In [16]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
   ....:           ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
   ....:

In [17]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])

In [18]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
   ....:                    'B': np.arange(8)},
   ....:                   index=index)
   ....:

In [19]: df
Out[19]:
              A  B
first second
bar   one     1  0
      two     1  1
baz   one     1  2
      two     1  3
foo   one     2  4
      two     2  5
qux   one     3  6
      two     3  7

[8 rows x 2 columns]

In [20]: df.groupby(['second', 'A']).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          B
second A
one    1  2
       2  4
       3  6
two    1  4
       2  5
       3  7

[6 rows x 1 columns]
```

Better support for compressed URLs in `read_csv`

The compression code was refactored (GH12688). As a result, reading dataframes from URLs in *read_csv()* or *read_table()* now supports additional compression methods: `xz`, `bz2`, and `zip` (GH14570). Previously, only `gzip` compression was supported. By default, compression of URLs and paths are now inferred using their file extensions. Additionally, support for bz2 compression in the python 2 C-engine improved (GH14874).

```
In [21]: url = ('https://github.com/{repo}/raw/{branch}/{path}'
   ....:        .format(repo='pandas-dev/pandas',
   ....:                branch='master',
   ....:                path='pandas/tests/io/parser/data/salaries.csv.bz2'))
   ....:

# default, infer compression
In [22]: df = pd.read_csv(url, sep='\t', compression='infer')
```

(continues on next page)

(continued from previous page)

```
# explicitly specify compression
In [23]: df = pd.read_csv(url, sep='\t', compression='bz2')

In [24]: df.head(2)
Out[24]:
       S  X  E  M
0  13876  1  1  1
1  11608  1  3  0

[2 rows x 4 columns]
```

Pickle file I/O now supports compression

read_pickle(), *DataFrame.to_pickle()* and *Series.to_pickle()* can now read from and write to compressed pickle files. Compression methods can be an explicit parameter or be inferred from the file extension. See *the docs here*.

```
In [25]: df = pd.DataFrame({'A': np.random.randn(1000),
   ....:                    'B': 'foo',
   ....:                    'C': pd.date_range('20130101', periods=1000, freq='s')})
   ....:
```

Using an explicit compression type

```
In [26]: df.to_pickle("data.pkl.compress", compression="gzip")

In [27]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [28]: rt.head()
Out[28]:
          A    B                   C
0 -1.344312  foo 2013-01-01 00:00:00
1  0.844885  foo 2013-01-01 00:00:01
2  1.075770  foo 2013-01-01 00:00:02
3 -0.109050  foo 2013-01-01 00:00:03
4  1.643563  foo 2013-01-01 00:00:04

[5 rows x 3 columns]
```

The default is to infer the compression type from the extension (`compression='infer'`):

```
In [29]: df.to_pickle("data.pkl.gz")

In [30]: rt = pd.read_pickle("data.pkl.gz")

In [31]: rt.head()
Out[31]:
          A    B                   C
0 -1.344312  foo 2013-01-01 00:00:00
1  0.844885  foo 2013-01-01 00:00:01
2  1.075770  foo 2013-01-01 00:00:02
3 -0.109050  foo 2013-01-01 00:00:03
4  1.643563  foo 2013-01-01 00:00:04
```

(continues on next page)

(continued from previous page)

```
[5 rows x 3 columns]

In [32]: df["A"].to_pickle("s1.pkl.bz2")

In [33]: rt = pd.read_pickle("s1.pkl.bz2")

In [34]: rt.head()
Out[34]:
0   -1.344312
1    0.844885
2    1.075770
3   -0.109050
4    1.643563
Name: A, Length: 5, dtype: float64
```

UInt64 Support Improved

Pandas has significantly improved support for operations involving unsigned, or purely non-negative, integers. Previously, handling these integers would result in improper rounding or data-type casting, leading to incorrect results. Notably, a new numerical index, `UInt64Index`, has been created (GH14937)

```
In [35]: idx = pd.UInt64Index([1, 2, 3])

In [36]: df = pd.DataFrame({'A': ['a', 'b', 'c']}, index=idx)

In [37]: df.index
Out[37]: UInt64Index([1, 2, 3], dtype='uint64')
```

- Bug in converting object elements of array-like objects to unsigned 64-bit integers (GH4471, GH14982)
- Bug in `Series.unique()` in which unsigned 64-bit integers were causing overflow (GH14721)
- Bug in `DataFrame` construction in which unsigned 64-bit integer elements were being converted to objects (GH14881)
- Bug in `pd.read_csv()` in which unsigned 64-bit integer elements were being improperly converted to the wrong data types (GH14983)
- Bug in `pd.unique()` in which unsigned 64-bit integers were causing overflow (GH14915)
- Bug in `pd.value_counts()` in which unsigned 64-bit integers were being erroneously truncated in the output (GH14934)

GroupBy on Categoricals

In previous versions, `.groupby(..., sort=False)` would fail with a `ValueError` when grouping on a categorical series with some categories not appearing in the data. (GH13179)

```
In [38]: chromosomes = np.r_[np.arange(1, 23).astype(str), ['X', 'Y']]

In [39]: df = pd.DataFrame({
   ....:     'A': np.random.randint(100),
   ....:     'B': np.random.randint(100),
   ....:     'C': np.random.randint(100),
   ....:     'chromosomes': pd.Categorical(np.random.choice(chromosomes, 100),
```

(continues on next page)

(continued from previous page)

```
   ....:                                                    categories=chromosomes,
   ....:                                                    ordered=True)})
   ....:

In [40]: df
Out[40]:
     A   B   C chromosomes
0   87  22  81           4
1   87  22  81          13
2   87  22  81          22
3   87  22  81           2
4   87  22  81           6
5   87  22  81          18
6   87  22  81           5
..  ..  ..  ..         ...
93  87  22  81           9
94  87  22  81           8
95  87  22  81           8
96  87  22  81          11
97  87  22  81           X
98  87  22  81           1
99  87  22  81          19

[100 rows x 4 columns]
```

Previous Behavior:

```
In [3]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
---------------------------------------------------------------------------
ValueError: items in new_categories are not the same as in old categories
```

New Behavior:

```
In [41]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
Out[41]:
               A    B    C
chromosomes
2            348   88  324
3            348   88  324
4            348   88  324
5            261   66  243
6            174   44  162
7            609  154  567
8            348   88  324
...          ...  ...  ...
19           174   44  162
20           261   66  243
22           348   88  324
X            348   88  324
Y            435  110  405
1              0    0    0
21             0    0    0

[24 rows x 3 columns]
```

Table Schema Output

The new orient 'table' for *DataFrame.to_json()* will generate a Table Schema compatible string representation of the data.

```
In [42]: df = pd.DataFrame(
   ....:     {'A': [1, 2, 3],
   ....:      'B': ['a', 'b', 'c'],
   ....:      'C': pd.date_range('2016-01-01', freq='d', periods=3)},
   ....:     index=pd.Index(range(3), name='idx'))
   ....:

In [43]: df
Out[43]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

[3 rows x 3 columns]

In [44]: df.to_json(orient='table')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'{"schema": {"fields":[{"name":"idx","type":"integer"},{"name":"A","type":"integer"}
↪,{"name":"B","type":"string"},{"name":"C","type":"datetime"}],"primaryKey":["idx"],
↪"pandas_version":"0.20.0"}, "data": [{"idx":0,"A":1,"B":"a","C":"2016-01-
↪01T00:00:00.000Z"},{"idx":1,"A":2,"B":"b","C":"2016-01-02T00:00:00.000Z"},{"idx":2,
↪"A":3,"B":"c","C":"2016-01-03T00:00:00.000Z"}]}'
```

See *IO: Table Schema for more information*.

Additionally, the repr for DataFrame and Series can now publish this JSON Table schema representation of the Series or DataFrame if you are using IPython (or another frontend like nteract using the Jupyter messaging protocol). This gives frontends like the Jupyter notebook and nteract more flexibility in how they display pandas objects, since they have more information about the data. You must enable this by setting the display.html.table_schema option to True.

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.

```
In [45]: from scipy.sparse import csr_matrix

In [46]: arr = np.random.random(size=(1000, 5))

In [47]: arr[arr < .9] = 0

In [48]: sp_arr = csr_matrix(arr)

In [49]: sp_arr
Out[49]:
```

(continues on next page)

(continued from previous page)

```
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 501 stored elements in Compressed Sparse Row format>

In [50]: sdf = pd.SparseDataFrame(sp_arr)

In [51]: sdf
Out[51]:
       0         1         2         3         4
0    NaN       NaN  0.977426       NaN       NaN
1    NaN       NaN       NaN       NaN  0.969340
2    NaN       NaN       NaN       NaN       NaN
3    NaN       NaN       NaN       NaN       NaN
4    NaN       NaN       NaN       NaN       NaN
5    NaN       NaN       NaN       NaN       NaN
6    NaN       NaN       NaN       NaN       NaN
..   ..       ...       ...       ...       ...
993 NaN       NaN       NaN       NaN       NaN
994 NaN  0.915759  0.997955  0.922673       NaN
995 NaN       NaN       NaN       NaN  0.917524
996 NaN       NaN       NaN       NaN       NaN
997 NaN       NaN       NaN       NaN  0.968178
998 NaN       NaN       NaN       NaN  0.901563
999 NaN       NaN       NaN       NaN       NaN

[1000 rows x 5 columns]
```

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

```
In [52]: sdf.to_coo()
Out[52]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 501 stored elements in COOrdinate format>
```

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 [53]: np.random.seed(24)

In [54]: df = pd.DataFrame({'A': np.linspace(1, 10, 10)})

In [55]: df = pd.concat([df, pd.DataFrame(np.random.RandomState(24).randn(10, 4),
   ....:                                   columns=list('BCDE'))],
   ....:                axis=1)
   ....:

In [56]: df.iloc[0, 2] = np.nan

In [57]: df
Out[57]:
       A         B         C         D         E
0    1.0  1.329212       NaN -0.316280 -0.990810
```

(continues on next page)

(continued from previous page)

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

In [59]: 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)
```

(continues on next page)

(continued from previous page)

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

In [61]: c
Out[61]:
[(-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 [62]: c.categories
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [63]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[63]:
[(-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 [64]: df = pd.DataFrame({'A': range(4),
   ....:                    'B': pd.cut([0, 3, 1, 1], bins=c.categories)
   ....:                    }).set_index('B')
   ....:

In [65]: df
Out[65]:
               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 [66]: df.loc[pd.Interval(1.5, 3.0)]
Out[66]:
A    1
Name: (1.5, 3.0], Length: 1, dtype: int64
```

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

```
In [67]: df.loc[0]
Out[67]:
                  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 MutliIndex 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 [68]: idx = pd.Index([1, 2])

In [69]: idx
Out[69]: Int64Index([1, 2], dtype='int64')

In [70]: mi = pd.MultiIndex.from_tuples([(1, 2), (2, 4)])

In [71]: mi
Out[71]:
MultiIndex(levels=[[1, 2], [2, 4]],
           codes=[[0, 1], [0, 1]])
```

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 [72]: idx.map(lambda x: x * 2)
Out[72]: Int64Index([2, 4], dtype='int64')

In [73]: idx.map(lambda x: (x, x * 2))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[73]:
MultiIndex(levels=[[1, 2], [2, 4]],
           codes=[[0, 1], [0, 1]])

In [74]: mi.map(lambda x: x)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[[1, 2], [2, 4]],
           codes=[[0, 1], [0, 1]])

In [75]: mi.map(lambda x: x[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Int64Index([1, 2], dtype='int64')
```

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

```
In [76]: s = pd.Series(pd.date_range('2011-01-02T00:00', '2011-01-02T02:00', freq='H')
   ....:                  .tz_localize('Asia/Tokyo'))
   ....:

In [77]: s
```

(continues on next page)

(continued from previous page)

```
Out[77]:
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]:
0    0
1    1
2    2
dtype: int32
```

New Behavior:

```
In [78]: s.map(lambda x: x.hour)
Out[78]:
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 [79]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [80]: idx.hour
Out[80]: 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'),
   ...:                      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 [81]: pd.Series([pd.Timestamp(r'20160101', tz=r'US/Eastern'),
   ....:            pd.Timestamp(r'20160101', tz=r'US/Eastern')]).unique()
   ....:
Out[81]:
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

In [82]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:            pd.Timestamp('20160101', tz='US/Eastern')]))
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]

# Index, returns a DatetimeIndex
In [83]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:            pd.Timestamp('20160101', tz='US/Eastern')]).unique()
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]',
↪ freq=None)

In [84]: pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
   ....:                      pd.Timestamp('20160101', tz='US/Eastern')]))
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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 [85]: pd.Series(list('baabc'), dtype='category').unique()
Out[85]:
[b, a, c]
Categories (3, object): [b, a, c]

In [86]: pd.unique(pd.Series(list('baabc'), dtype='category'))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[86]:
[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 [87]: 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 [88]: df1 = pd.DataFrame(np.array([1.0], dtype=np.float32, ndmin=2))

In [89]: df1.dtypes
Out[89]:
0    float32
Length: 1, dtype: object

In [90]: df2 = pd.DataFrame(np.array([np.nan], dtype=np.float32, ndmin=2))

In [91]: df2.dtypes
Out[91]:
0    float32
Length: 1, dtype: object
```

Previous Behavior:

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

New Behavior:

```
In [92]: pd.concat([df1, df2]).dtypes
Out[92]:
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')
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 [93]: df = pd.DataFrame(np.arange(6), columns=['value'],
   ....:                   index=pd.MultiIndex.from_product([list('BA'), range(3)]))
   ....:

In [94]: df
Out[94]:
     value
B 0      0
  1      1
  2      2
A 0      3
  1      4
  2      5

[6 rows x 1 columns]
```

```
In [95]: df.index.is_lexsorted()
Out[95]: False

In [96]: df.index.is_monotonic
\\\\\\\\\\\\\\\Out[96]: False
```

Sorting works as expected

```
In [97]: df.sort_index()
Out[97]:
     value
A 0      3
  1      4
  2      5
B 0      0
  1      1
  2      2

[6 rows x 1 columns]
```

```
In [98]: df.sort_index().index.is_lexsorted()
Out[98]: True

In [99]: df.sort_index().index.is_monotonic
\\\\\\\\\\\\\\Out[99]: True
```

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

```
In [100]: df = pd.DataFrame({'value': [1, 2, 3, 4]},
   .....:                   index=pd.MultiIndex([['a', 'b'], ['bb', 'aa']],
   .....:                                       [[0, 0, 1, 1], [0, 1, 0, 1]]))
   .....:

In [101]: df
Out[101]:
      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 [102]: df.sort_index()
Out[102]:
      value
a aa      2
```

(continues on next page)

(continued from previous page)

```
  bb      1
b aa      4
  bb      3

[4 rows x 1 columns]

In [103]: df.sort_index().index.is_lexsorted()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[103]:
↪True

In [104]: df.sort_index().index.is_monotonic
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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 [105]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})

In [106]: df.groupby('A').describe()
Out[106]:
      B
  count mean       std  min   25%  50%   75%  max
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 [107]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     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 [108]: np.random.seed(1234)

In [109]: 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 [110]: df.tail()
Out[110]:
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'>
```

(continues on next page)

(continued from previous page)

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

In [112]: res.tail()
Out[112]:
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 [113]: df.rolling(12).corr().loc['2016-04-07']
Out[113]:
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 [114]: df = pd.DataFrame({'unparsed_date': ['2014-01-01', '2014-01-01']})

In [115]: df.to_hdf('store.h5', 'key', format='table', data_columns=True)

In [116]: df.dtypes
Out[116]:
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 [117]: left = pd.Index([2, 1, 0])

In [118]: left
Out[118]: Int64Index([2, 1, 0], dtype='int64')

In [119]: right = pd.Index([1, 2, 3])

In [120]: right
Out[120]: Int64Index([1, 2, 3], dtype='int64')
```

Previous Behavior:

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

New Behavior:

```
In [121]: left.intersection(right)
Out[121]: Int64Index([1, 2], dtype='int64')
```

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

```
In [122]: left = pd.DataFrame({'a': [20, 10, 0]}, index=[2, 1, 0])

In [123]: left
Out[123]:
    a
2  20
1  10
0   0

[3 rows x 1 columns]

In [124]: right = pd.DataFrame({'b': [100, 200, 300]}, index=[1, 2, 3])

In [125]: right
Out[125]:
     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 [126]: left.join(right, how='inner')
Out[126]:
    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 [127]: df = pd.DataFrame({'col1': [3, 4, 5],
   .....:                    'col2': ['C', 'D', 'E'],
   .....:                    'col3': [1, 3, 9]})
   .....:

In [128]: df
Out[128]:
   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 [129]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[129]:
           col1
col3 col2
1    C        3
3    D        4
9    E        5
```

(continues on next page)

(continued from previous page)

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

In [131]: df
Out[131]:
   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 [132]: df.loc[df.index[[0, 2]], 'A']
Out[132]:
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 [133]: df.iloc[[0, 2], df.columns.get_loc('A')]
Out[133]:
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. For more details see *Deprecate Panel* documentation. (GH13563).

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

In [139]: df
Out[139]:
   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 [140]: df.groupby('A').agg({'B': 'sum', 'C': 'min'})
Out[140]:
   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
is deprecated and will be removed in a future version

Out[6]:
   foo
A
```

(continues on next page)

(continued from previous page)

```
1     3
2     2
```

You can accomplish the same operation, more idiomatically by:

```
In [141]: df.groupby('A').B.agg(['count']).rename(columns={'count': 'foo'})
Out[141]:
   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 [142]: (df.groupby('A')
   .....:    .agg({'B': 'sum', 'C': 'min'})
   .....:    .rename(columns={'B': 'foo', 'C': 'bar'})
   .....:  )
   .....:
Out[142]:
   foo  bar
A
1    3    0
2    7    3

[2 rows x 2 columns]
```

Deprecate .plotting

The `pandas.tools.plotting` module has been deprecated, in favor of the top level `pandas.plotting` module. All the public plotting functions are now available from `pandas.plotting` (GH12548).

Furthermore, the top-level `pandas.scatter_matrix` and `pandas.plot_params` are deprecated. Users can import these from `pandas.plotting` as well.

Previous script:

```
pd.tools.plotting.scatter_matrix(df)
pd.scatter_matrix(df)
```

Should be changed to:

```
pd.plotting.scatter_matrix(df)
```

Other Deprecations

- `SparseArray.to_dense()` has deprecated the `fill` parameter, as that parameter was not being respected (GH14647)
- `SparseSeries.to_dense()` has deprecated the `sparse_only` parameter (GH14647)
- `Series.repeat()` has deprecated the `reps` parameter in favor of `repeats` (GH12662)
- The `Series` constructor and `.astype` method have deprecated accepting timestamp dtypes without a frequency (e.g. `np.datetime64`) for the `dtype` parameter (GH15524)
- `Index.repeat()` and `MultiIndex.repeat()` have deprecated the `n` parameter in favor of `repeats` (GH12662)
- `Categorical.searchsorted()` and `Series.searchsorted()` have deprecated the `v` parameter in favor of `value` (GH12662)
- `TimedeltaIndex.searchsorted()`, `DatetimeIndex.searchsorted()`, and `PeriodIndex.searchsorted()` have deprecated the `key` parameter in favor of `value` (GH12662)
- `DataFrame.astype()` has deprecated the `raise_on_error` parameter in favor of `errors` (GH14878)
- `Series.sortlevel` and `DataFrame.sortlevel` have been deprecated in favor of `Series.sort_index` and `DataFrame.sort_index` (GH15099)
- importing `concat` from `pandas.tools.merge` has been deprecated in favor of imports from the `pandas` namespace. This should only affect explicit imports (GH15358)
- `Series/DataFrame/Panel.consolidate()` been deprecated as a public method. (GH15483)
- The `as_indexer` keyword of `Series.str.match()` has been deprecated (ignored keyword) (GH15257).
- The following top-level pandas functions have been deprecated and will be removed in a future version (GH13790, GH15940)
  - `pd.pnow()`, replaced by `Period.now()`
  - `pd.Term`, is removed, as it is not applicable to user code. Instead use in-line string expressions in the where clause when searching in HDFStore
  - `pd.Expr`, is removed, as it is not applicable to user code.
  - `pd.match()`, is removed.
  - `pd.groupby()`, replaced by using the `.groupby()` method directly on a `Series/DataFrame`
  - `pd.get_store()`, replaced by a direct call to `pd.HDFStore(...)`
- `is_any_int_dtype`, `is_floating_dtype`, and `is_sequence` are deprecated from `pandas.api.types` (GH16042)

Removal of prior version deprecations/changes

- The `pandas.rpy` module is removed. Similar functionality can be accessed through the rpy2 project. See the R interfacing docs for more details.

- The pandas.io.ga module with a google-analytics interface is removed (GH11308). Similar functionality can be found in the Google2Pandas package.
- pd.to_datetime and pd.to_timedelta have dropped the coerce parameter in favor of errors (GH13602)
- pandas.stats.fama_macbeth, pandas.stats.ols, pandas.stats.plm and pandas.stats.var, as well as the top-level pandas.fama_macbeth and pandas.ols routines are removed. Similar functionality can be found in the statsmodels package. (GH11898)
- The TimeSeries and SparseTimeSeries classes, aliases of Series and SparseSeries, are removed (GH10890, GH15098).
- Series.is_time_series is dropped in favor of Series.index.is_all_dates (GH15098)
- The deprecated irow, icol, iget and iget_value methods are removed in favor of iloc and iat as explained *here* (GH10711).
- The deprecated DataFrame.iterkv() has been removed in favor of DataFrame.iteritems() (GH10711)
- The Categorical constructor has dropped the name parameter (GH10632)
- Categorical has dropped support for NaN categories (GH10748)
- The take_last parameter has been dropped from duplicated(), drop_duplicates(), nlargest(), and nsmallest() methods (GH10236, GH10792, GH10920)
- Series, Index, and DataFrame have dropped the sort and order methods (GH10726)
- Where clauses in pytables are only accepted as strings and expressions types and not other data-types (GH12027)
- DataFrame has dropped the combineAdd and combineMult methods in favor of add and mul respectively (GH10735)

Performance Improvements

- Improved performance of pd.wide_to_long() (GH14779)
- Improved performance of pd.factorize() by releasing the GIL with object dtype when inferred as strings (GH14859, GH16057)
- Improved performance of timeseries plotting with an irregular DatetimeIndex (or with compat_x=True) (GH15073).
- Improved performance of groupby().cummin() and groupby().cummax() (GH15048, GH15109, GH15561, GH15635)
- Improved performance and reduced memory when indexing with a MultiIndex (GH15245)
- When reading buffer object in read_sas() method without specified format, filepath string is inferred rather than buffer object. (GH14947)
- Improved performance of .rank() for categorical data (GH15498)
- Improved performance when using .unstack() (GH15503)
- Improved performance of merge/join on category columns (GH10409)
- Improved performance of drop_duplicates() on bool columns (GH12963)
- Improve performance of pd.core.groupby.GroupBy.apply when the applied function used the .name attribute of the group DataFrame (GH15062).

- Improved performance of `iloc` indexing with a list or array (GH15504).
- Improved performance of `Series.sort_index()` with a monotonic index (GH15694)
- Improved performance in `pd.read_csv()` on some platforms with buffered reads (GH16039)

Bug Fixes

Conversion

- Bug in `Timestamp.replace` now raises `TypeError` when incorrect argument names are given; previously this raised `ValueError` (GH15240)
- Bug in `Timestamp.replace` with compat for passing long integers (GH15030)
- Bug in `Timestamp` returning UTC based time/date attributes when a timezone was provided (GH13303, GH6538)
- Bug in `Timestamp` incorrectly localizing timezones during construction (GH11481, GH15777)
- Bug in `TimedeltaIndex` addition where overflow was being allowed without error (GH14816)
- Bug in `TimedeltaIndex` raising a `ValueError` when boolean indexing with `loc` (GH14946)
- Bug in catching an overflow in `Timestamp` + `Timedelta/Offset` operations (GH15126)
- Bug in `DatetimeIndex.round()` and `Timestamp.round()` floating point accuracy when rounding by milliseconds or less (GH14440, GH15578)
- Bug in `astype()` where `inf` values were incorrectly converted to integers. Now raises error now with `astype()` for Series and DataFrames (GH14265)
- Bug in `DataFrame(..).apply(to_numeric)` when values are of type decimal.Decimal. (GH14827)
- Bug in `describe()` when passing a numpy array which does not contain the median to the `percentiles` keyword argument (GH14908)
- Cleaned up `PeriodIndex` constructor, including raising on floats more consistently (GH13277)
- Bug in using `__deepcopy__` on empty NDFrame objects (GH15370)
- Bug in `.replace()` may result in incorrect dtypes. (GH12747, GH15765)
- Bug in `Series.replace` and `DataFrame.replace` which failed on empty replacement dicts (GH15289)
- Bug in `Series.replace` which replaced a numeric by string (GH15743)
- Bug in `Index` construction with `NaN` elements and integer dtype specified (GH15187)
- Bug in `Series` construction with a datetimetz (GH14928)
- Bug in `Series.dt.round()` inconsistent behaviour on `NaT` 's with different arguments (GH14940)
- Bug in `Series` constructor when both `copy=True` and `dtype` arguments are provided (GH15125)
- Incorrect dtyped `Series` was returned by comparison methods (e.g., `lt`, `gt`, `...`) against a constant for an empty `DataFrame` (GH15077)
- Bug in `Series.ffill()` with mixed dtypes containing tz-aware datetimes. (GH14956)
- Bug in `DataFrame.fillna()` where the argument `downcast` was ignored when fillna value was of type `dict` (GH15277)
- Bug in `.asfreq()`, where frequency was not set for empty `Series` (GH14320)
- Bug in `DataFrame` construction with nulls and datetimes in a list-like (GH15869)

- Bug in `DataFrame.fillna()` with tz-aware datetimes (GH15855)
- Bug in `is_string_dtype`, `is_timedelta64_ns_dtype`, and `is_string_like_dtype` in which an error was raised when `None` was passed in (GH15941)
- Bug in the return type of `pd.unique` on a `Categorical`, which was returning an ndarray and not a `Categorical` (GH15903)
- Bug in `Index.to_series()` where the index was not copied (and so mutating later would change the original), (GH15949)
- Bug in indexing with partial string indexing with a len-1 DataFrame (GH16071)
- Bug in `Series` construction where passing invalid dtype didn't raise an error. (GH15520)

Indexing

- Bug in `Index` power operations with reversed operands (GH14973)
- Bug in `DataFrame.sort_values()` when sorting by multiple columns where one column is of type `int64` and contains `NaT` (GH14922)
- Bug in `DataFrame.reindex()` in which `method` was ignored when passing `columns` (GH14992)
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a `Series` indexer (GH14730, GH15424)
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a numpy array (GH15434)
- Bug in `Series.asof` which raised if the series contained all `np.nan` (GH15713)
- Bug in `.at` when selecting from a tz-aware column (GH15822)
- Bug in `Series.where()` and `DataFrame.where()` where array-like conditionals were being rejected (GH15414)
- Bug in `Series.where()` where TZ-aware data was converted to float representation (GH15701)
- Bug in `.loc` that would not return the correct dtype for scalar access for a DataFrame (GH11617)
- Bug in output formatting of a `MultiIndex` when names are integers (GH12223, GH15262)
- Bug in `Categorical.searchsorted()` where alphabetical instead of the provided categorical order was used (GH14522)
- Bug in `Series.iloc` where a `Categorical` object for list-like indexes input was returned, where a `Series` was expected. (GH14580)
- Bug in `DataFrame.isin` comparing datetimelike to empty frame (GH15473)
- Bug in `.reset_index()` when an all `NaN` level of a `MultiIndex` would fail (GH6322)
- Bug in `.reset_index()` when raising error for index name already present in `MultiIndex` columns (GH16120)
- Bug in creating a `MultiIndex` with tuples and not passing a list of names; this will now raise `ValueError` (GH15110)
- Bug in the HTML display with with a `MultiIndex` and truncation (GH14882)
- Bug in the display of `.info()` where a qualifier (+) would always be displayed with a `MultiIndex` that contains only non-strings (GH15245)
- Bug in `pd.concat()` where the names of `MultiIndex` of resulting `DataFrame` are not handled correctly when `None` is presented in the names of `MultiIndex` of input `DataFrame` (GH15787)

- Bug in DataFrame.sort_index() and Series.sort_index() where na_position doesn't work with a MultiIndex (GH14784, GH16604)
- Bug in in pd.concat() when combining objects with a CategoricalIndex (GH16111)
- Bug in indexing with a scalar and a CategoricalIndex (GH16123)

I/O

- Bug in pd.to_numeric() in which float and unsigned integer elements were being improperly casted (GH14941, GH15005)
- Bug in pd.read_fwf() where the skiprows parameter was not being respected during column width inference (GH11256)
- Bug in pd.read_csv() in which the dialect parameter was not being verified before processing (GH14898)
- Bug in pd.read_csv() in which missing data was being improperly handled with usecols (GH6710)
- Bug in pd.read_csv() in which a file containing a row with many columns followed by rows with fewer columns would cause a crash (GH14125)
- Bug in pd.read_csv() for the C engine where usecols were being indexed incorrectly with parse_dates (GH14792)
- Bug in pd.read_csv() with parse_dates when multi-line headers are specified (GH15376)
- Bug in pd.read_csv() with float_precision='round_trip' which caused a segfault when a text entry is parsed (GH15140)
- Bug in pd.read_csv() when an index was specified and no values were specified as null values (GH15835)
- Bug in pd.read_csv() in which certain invalid file objects caused the Python interpreter to crash (GH15337)
- Bug in pd.read_csv() in which invalid values for nrows and chunksize were allowed (GH15767)
- Bug in pd.read_csv() for the Python engine in which unhelpful error messages were being raised when parsing errors occurred (GH15910)
- Bug in pd.read_csv() in which the skipfooter parameter was not being properly validated (GH15925)
- Bug in pd.to_csv() in which there was numeric overflow when a timestamp index was being written (GH15982)
- Bug in pd.util.hashing.hash_pandas_object() in which hashing of categoricals depended on the ordering of categories, instead of just their values. (GH15143)
- Bug in .to_json() where lines=True and contents (keys or values) contain escaped characters (GH15096)
- Bug in .to_json() causing single byte ascii characters to be expanded to four byte unicode (GH15344)
- Bug in .to_json() for the C engine where rollover was not correctly handled for case where frac is odd and diff is exactly 0.5 (GH15716, GH15864)
- Bug in pd.read_json() for Python 2 where lines=True and contents contain non-ascii unicode characters (GH15132)
- Bug in pd.read_msgpack() in which Series categoricals were being improperly processed (GH14901)
- Bug in pd.read_msgpack() which did not allow loading of a dataframe with an index of type CategoricalIndex (GH15487)

- Bug in `pd.read_msgpack()` when deserializing a `CategoricalIndex` (GH15487)
- Bug in `DataFrame.to_records()` with converting a `DatetimeIndex` with a timezone (GH13937)
- Bug in `DataFrame.to_records()` which failed with unicode characters in column names (GH11879)
- Bug in `.to_sql()` when writing a DataFrame with numeric index names (GH15404).
- Bug in `DataFrame.to_html()` with `index=False` and `max_rows` raising in `IndexError` (GH14998)
- Bug in `pd.read_hdf()` passing a `Timestamp` to the `where` parameter with a non date column (GH15492)
- Bug in `DataFrame.to_stata()` and `StataWriter` which produces incorrectly formatted files to be produced for some locales (GH13856)
- Bug in `StataReader` and `StataWriter` which allows invalid encodings (GH15723)
- Bug in the `Series` repr not showing the length when the output was truncated (GH15962).

Plotting

- Bug in `DataFrame.hist` where `plt.tight_layout` caused an `AttributeError` (use `matplotlib >= 2.0.1`) (GH9351)
- Bug in `DataFrame.boxplot` where `fontsize` was not applied to the tick labels on both axes (GH15108)
- Bug in the date and time converters pandas registers with matplotlib not handling multiple dimensions (GH16026)
- Bug in `pd.scatter_matrix()` could accept either `color` or `c`, but not both (GH14855)

Groupby/Resample/Rolling

- Bug in `.groupby(..).resample()` when passed the `on=` kwarg. (GH15021)
- Properly set `__name__` and `__qualname__` for `Groupby.*` functions (GH14620)
- Bug in `GroupBy.get_group()` failing with a categorical grouper (GH15155)
- Bug in `.groupby(...).rolling(...)` when `on` is specified and using a `DatetimeIndex` (GH15130, GH13966)
- Bug in groupby operations with `timedelta64` when passing `numeric_only=False` (GH5724)
- Bug in `groupby.apply()` coercing `object` dtypes to numeric types, when not all values were numeric (GH14423, GH15421, GH15670)
- Bug in `resample`, where a non-string `loffset` argument would not be applied when resampling a timeseries (GH13218)
- Bug in `DataFrame.groupby().describe()` when grouping on `Index` containing tuples (GH14848)
- Bug in `groupby().nunique()` with a datetimelike-grouper where bins counts were incorrect (GH13453)
- Bug in `groupby.transform()` that would coerce the resultant dtypes back to the original (GH10972, GH11444)
- Bug in `groupby.agg()` incorrectly localizing timezone on `datetime` (GH15426, GH10668, GH13046)
- Bug in `.rolling/expanding()` functions where `count()` was not counting `np.Inf`, nor handling `object` dtypes (GH12541)

- Bug in `.rolling()` where `pd.Timedelta` or `datetime.timedelta` was not accepted as a `window` argument (GH15440)
- Bug in `Rolling.quantile` function that caused a segmentation fault when called with a quantile value outside of the range [0, 1] (GH15463)
- Bug in `DataFrame.resample().median()` if duplicate column names are present (GH14233)

Sparse

- Bug in `SparseSeries.reindex` on single level with list of length 1 (GH15447)
- Bug in repr-formatting a `SparseDataFrame` after a value was set on (a copy of) one of its series (GH15488)
- Bug in `SparseDataFrame` construction with lists not coercing to dtype (GH15682)
- Bug in sparse array indexing in which indices were not being validated (GH15863)

Reshaping

- Bug in `pd.merge_asof()` where `left_index` or `right_index` caused a failure when multiple `by` was specified (GH15676)
- Bug in `pd.merge_asof()` where `left_index/right_index` together caused a failure when `tolerance` was specified (GH15135)
- Bug in `DataFrame.pivot_table()` where `dropna=True` would not drop all-NaN columns when the columns was a `category` dtype (GH15193)
- Bug in `pd.melt()` where passing a tuple value for `value_vars` caused a `TypeError` (GH15348)
- Bug in `pd.pivot_table()` where no error was raised when values argument was not in the columns (GH14938)
- Bug in `pd.concat()` in which concatenating with an empty dataframe with `join='inner'` was being improperly handled (GH15328)
- Bug with `sort=True` in `DataFrame.join` and `pd.merge` when joining on indexes (GH15582)
- Bug in `DataFrame.nsmallest` and `DataFrame.nlargest` where identical values resulted in duplicated rows (GH15297)
- Bug in *`pandas.pivot_table()`* incorrectly raising `UnicodeError` when passing unicode input for `margins` keyword (GH13292)

Numeric

- Bug in `.rank()` which incorrectly ranks ordered categories (GH15420)
- Bug in `.corr()` and `.cov()` where the column and index were the same object (GH14617)
- Bug in `.mode()` where `mode` was not returned if was only a single value (GH15714)
- Bug in `pd.cut()` with a single bin on an all 0s array (GH15428)
- Bug in `pd.qcut()` with a single quantile and an array with identical values (GH15431)
- Bug in `pandas.tools.utils.cartesian_product()` with large input can cause overflow on windows (GH15265)

- Bug in `.eval()` which caused multi-line evals to fail with local variables not on the first line (GH15342)

Other

- Compat with SciPy 0.19.0 for testing on `.interpolate()` (GH15662)
- Compat for 32-bit platforms for `.qcut/cut`; bins will now be `int64` dtype (GH14866)
- Bug in interactions with `Qt` when a `QtApplication` already exists (GH14372)
- Avoid use of `np.finfo()` during `import pandas` removed to mitigate deadlock on Python GIL misuse (GH14641)

Contributors

A total of 204 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam J. Stewart +
- Adrian +
- Ajay Saxena
- Akash Tandon +
- Albert Villanova del Moral +
- Aleksey Bilogur +
- Alexis Mignon +
- Amol Kahat +
- Andreas Winkler +
- Andrew Kittredge +
- Anthonios Partheniou
- Arco Bast +
- Ashish Singal +
- Baurzhan Muftakhidinov +
- Ben Kandel
- Ben Thayer +
- Ben Welsh +
- Bill Chambers +
- Brandon M. Burroughs
- Brian +
- Brian McFee +
- Carlos Souza +
- Chris
- Chris Ham

- Chris Warth
- Christoph Gohlke
- Christoph Paulik +
- Christopher C. Aycock
- Clemens Brunner +
- D.S. McNeil +
- DaanVanHauwermeiren +
- Daniel Himmelstein
- Dave Willmer
- David Cook +
- David Gwynne +
- David Hoffman +
- David Krych
- Diego Fernandez +
- Dimitris Spathis +
- Dmitry L +
- Dody Suria Wijaya +
- Dominik Stanczak +
- Dr-Irv
- Dr. Irv +
- Elliott Sales de Andrade +
- Ennemoser Christoph +
- Francesc Alted +
- Fumito Hamamura +
- Giacomo Ferroni
- Graham R. Jeffries +
- Greg Williams +
- Guilherme Beltramini +
- Guilherme Samora +
- Hao Wu +
- Harshit Patni +
- Ilya V. Schurov +
- Iván Vallés Pérez
- Jackie Leng +
- Jaehoon Hwang +
- James Draper +

- James Goppert +
- James McBride +
- James Santucci +
- Jan Schulz
- Jeff Carey
- Jeff Reback
- JennaVergeynst +
- Jim +
- Jim Crist
- Joe Jevnik
- Joel Nothman +
- John +
- John Tucker +
- John W. O'Brien
- John Zwinck
- Jon M. Mease
- Jon Mease
- Jonathan Whitmore +
- Jonathan de Bruin +
- Joost Kranendonk +
- Joris Van den Bossche
- Joshua Bradt +
- Julian Santander
- Julien Marrec +
- Jun Kim +
- Justin Solinsky +
- Kacawi +
- Kamal Kamalaldin +
- Kerby Shedden
- Kernc
- Keshav Ramaswamy
- Kevin Sheppard
- Kyle Kelley
- Larry Ren
- Leon Yin +
- Line Pedersen +

- Lorenzo Cestaro +
- Luca Scarabello
- Lukasz +
- Mahmoud Lababidi
- Mark Mandel +
- Matt Roeschke
- Matthew Brett
- Matthew Roeschke +
- Matti Picus
- Maximilian Roos
- Michael Charlton +
- Michael Felt
- Michael Lamparski +
- Michiel Stock +
- Mikolaj Chwalisz +
- Min RK
- Miroslav Šedivý +
- Mykola Golubyev
- Nate Yoder
- Nathalie Rud +
- Nicholas Ver Halen
- Nick Chmura +
- Nolan Nichols +
- Pankaj Pandey +
- Pawel Kordek
- Pete Huang +
- Peter +
- Peter Csizsek +
- Petio Petrov +
- Phil Ruffwind +
- Pietro Battiston
- Piotr Chromiec
- Prasanjit Prakash +
- Rob Forgione +
- Robert Bradshaw
- Robin +

- Rodolfo Fernandez
- Roger Thomas
- Rouz Azari +
- Sahil Dua
- Sam Foo +
- Sami Salonen +
- Sarah Bird +
- Sarma Tangirala +
- Scott Sanderson
- Sebastian Bank
- Sebastian Gsänger +
- Shawn Heide
- Shyam Saladi +
- Sinhrks
- Stephen Rauch +
- Sébastien de Menten +
- Tara Adiseshan
- Thiago Serafim
- Thoralf Gutierrez +
- Thrasibule +
- Tobias Gustafsson +
- Tom Augspurger
- Tong SHEN +
- Tong Shen +
- TrigonaMinima +
- Uwe +
- Wes Turner
- Wiktor Tomczak +
- WillAyd
- Yaroslav Halchenko
- Yimeng Zhang +
- abaldenko +
- adrian-stepien +
- alexandercbooth +
- atbd +
- bastewart +

- bmagnusson +
- carlosdanielcsantos +
- chaimdemulder +
- chris-b1
- dickreuter +
- discort +
- dr-leo +
- dubourg
- dwkenefick +
- funnycrab +
- gfyoung
- goldenbull +
- hesham.shabana@hotmail.com
- jojomdt +
- linebp +
- manu +
- manuels +
- mattip +
- maxalbert +
- mcocdawc +
- nuffe +
- paul-mannino
- pbreach +
- sakkemo +
- scls19fr
- sinhrks
- stijnvanhoey +
- the-nose-knows +
- themrmax +
- tomrod +
- tzinckgraf
- wandersoncferreira
- watercrossing +
- wcwagner
- xgdgsc +
- yui-knk

8.6 Version 0.19

8.6.1 v0.19.2 (December 24, 2016)

This is a minor bug-fix release in the 0.19.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Compatibility with Python 3.6
- Added a Pandas Cheat Sheet. (GH13202).

Enhancements

The `pd.merge_asof()`, added in 0.19.0, gained some improvements:

- `pd.merge_asof()` gained `left_index`/`right_index` and `left_by`/`right_by` arguments (GH14253)
- `pd.merge_asof()` can take multiple columns in `by` parameter and has specialized dtypes for better performance (GH13936)

Performance Improvements

- Performance regression with `PeriodIndex` (GH14822)
- Performance regression in indexing with getitem (GH14930)
- Improved performance of `.replace()` (GH12745)
- Improved performance `Series` creation with a datetime index and dictionary data (GH14894)

Bug Fixes

- Compat with python 3.6 for pickling of some offsets (GH14685)
- Compat with python 3.6 for some indexing exception types (GH14684, GH14689)
- Compat with python 3.6 for deprecation warnings in the test suite (GH14681)
- Compat with python 3.6 for Timestamp pickles (GH14689)
- Compat with `dateutil==2.6.0`; segfault reported in the testing suite (GH14621)
- Allow `nanoseconds` in `Timestamp.replace` as a kwarg (GH14621)

- Bug in `pd.read_csv` in which aliasing was being done for `na_values` when passed in as a dictionary (GH14203)
- Bug in `pd.read_csv` in which column indices for a dict-like `na_values` were not being respected (GH14203)
- Bug in `pd.read_csv` where reading files fails, if the number of headers is equal to the number of lines in the file (GH14515)
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when multi-char delimiters were not being respected with quotes (GH14582)
- Fix bugs (GH14734, GH13654) in `pd.read_sas` and `pandas.io.sas.sas7bdat.SAS7BDATReader` that caused problems when reading a SAS file incrementally.
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when `skipfooter` was not being respected by Python's CSV library (GH13879)
- Bug in `.fillna()` in which timezone aware datetime64 values were incorrectly rounded (GH14872)
- Bug in `.groupby(..., sort=True)` of a non-lexsorted MultiIndex when grouping with multiple levels (GH14776)
- Bug in `pd.cut` with negative values and a single bin (GH14652)
- Bug in `pd.to_numeric` where a 0 was not unsigned on a `downcast='unsigned'` argument (GH14401)
- Bug in plotting regular and irregular timeseries using shared axes (`sharex=True` or `ax.twinx()`) (GH13341, GH14322).
- Bug in not propagating exceptions in parsing invalid datetimes, noted in python 3.6 (GH14561)
- Bug in resampling a `DatetimeIndex` in local TZ, covering a DST change, which would raise `AmbiguousTimeError` (GH14682)
- Bug in indexing that transformed `RecursionError` into `KeyError` or `IndexingError` (GH14554)
- Bug in `HDFStore` when writing a `MultiIndex` when using `data_columns=True` (GH14435)
- Bug in `HDFStore.append()` when writing a `Series` and passing a `min_itemsize` argument containing a value for the `index` (GH11412)
- Bug when writing to a `HDFStore` in `table` format with a `min_itemsize` value for the `index` and without asking to append (GH10381)
- Bug in `Series.groupby.nunique()` raising an `IndexError` for an empty `Series` (GH12553)
- Bug in `DataFrame.nlargest` and `DataFrame.nsmallest` when the index had duplicate values (GH13412)
- Bug in clipboard functions on linux with python2 with unicode and separators (GH13747)
- Bug in clipboard functions on Windows 10 and python 3 (GH14362, GH12807)
- Bug in `.to_clipboard()` and Excel compat (GH12529)
- Bug in `DataFrame.combine_first()` for integer columns (GH14687).
- Bug in `pd.read_csv()` in which the `dtype` parameter was not being respected for empty data (GH14712)
- Bug in `pd.read_csv()` in which the `nrows` parameter was not being respected for large input when using the C engine for parsing (GH7626)
- Bug in `pd.merge_asof()` could not handle timezone-aware DatetimeIndex when a tolerance was specified (GH14844)
- Explicit check in `to_stata` and `StataWriter` for out-of-range values when writing doubles (GH14618)

- Bug in `.plot(kind='kde')` which did not drop missing values to generate the KDE Plot, instead generating an empty plot. (GH14821)
- Bug in `unstack()` if called with a list of column(s) as an argument, regardless of the dtypes of all columns, they get coerced to `object` (GH11847)

Contributors

A total of 33 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Ajay Saxena +
- Ben Kandel
- Chris
- Chris Ham +
- Christopher C. Aycock
- Daniel Himmelstein +
- Dave Willmer +
- Dr-Irv
- Jeff Carey +
- Jeff Reback
- Joe Jevnik
- Joris Van den Bossche
- Julian Santander +
- Kerby Shedden
- Keshav Ramaswamy
- Kevin Sheppard
- Luca Scarabello +
- Matt Roeschke +
- Matti Picus +
- Maximilian Roos
- Mykola Golubyev +
- Nate Yoder +
- Nicholas Ver Halen +
- Pawel Kordek
- Pietro Battiston
- Rodolfo Fernandez +
- Tara Adiseshan +
- Tom Augspurger
- Yaroslav Halchenko

- gfyoung
- hesham.shabana@hotmail.com +
- sinhrks
- wandersoncferreira +

8.6.2 v0.19.1 (November 3, 2016)

This is a minor bug-fix release from 0.19.0 and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

What's new in v0.19.1

- *Performance Improvements*
- *Bug Fixes*
- *Contributors*

Performance Improvements

- Fixed performance regression in factorization of `Period` data (GH14338)
- Fixed performance regression in `Series.asof(where)` when `where` is a scalar (GH14461)
- Improved performance in `DataFrame.asof(where)` when `where` is a scalar (GH14461)
- Improved performance in `.to_json()` when `lines=True` (GH14408)
- Improved performance in certain types of *loc* indexing with a MultiIndex (GH14551).

Bug Fixes

- Source installs from PyPI will now again work without `cython` installed, as in previous versions (GH14204)
- Compat with Cython 0.25 for building (GH14496)
- Fixed regression where user-provided file handles were closed in `read_csv` (c engine) (GH14418).
- Fixed regression in `DataFrame.quantile` when missing values where present in some columns (GH14357).
- Fixed regression in `Index.difference` where the `freq` of a `DatetimeIndex` was incorrectly set (GH14323)
- Added back `pandas.core.common.array_equivalent` with a deprecation warning (GH14555).
- Bug in `pd.read_csv` for the C engine in which quotation marks were improperly parsed in skipped rows (GH14459)
- Bug in `pd.read_csv` for Python 2.x in which Unicode quote characters were no longer being respected (GH14477)
- Fixed regression in `Index.append` when categorical indices were appended (GH14545).
- Fixed regression in `pd.DataFrame` where constructor fails when given dict with `None` value (GH14381)
- Fixed regression in `DatetimeIndex._maybe_cast_slice_bound` when index is empty (GH14354).

- Bug in localizing an ambiguous timezone when a boolean is passed (GH14402)
- Bug in `TimedeltaIndex` addition with a Datetime-like object where addition overflow in the negative direction was not being caught (GH14068, GH14453)
- Bug in string indexing against data with `object Index` may raise `AttributeError` (GH14424)
- Correctly raise `ValueError` on empty input to `pd.eval()` and `df.query()` (GH13139)
- Bug in `RangeIndex.intersection` when result is a empty set (GH14364).
- Bug in groupby-transform broadcasting that could cause incorrect dtype coercion (GH14457)
- Bug in `Series.__setitem__` which allowed mutating read-only arrays (GH14359).
- Bug in `DataFrame.insert` where multiple calls with duplicate columns can fail (GH14291)
- `pd.merge()` will raise `ValueError` with non-boolean parameters in passed boolean type arguments (GH14434)
- Bug in `Timestamp` where dates very near the minimum (1677-09) could underflow on creation (GH14415)
- Bug in `pd.concat` where names of the `keys` were not propagated to the resulting `MultiIndex` (GH14252)
- Bug in `pd.concat` where `axis` cannot take string parameters `'rows'` or `'columns'` (GH14369)
- Bug in `pd.concat` with dataframes heterogeneous in length and tuple `keys` (GH14438)
- Bug in `MultiIndex.set_levels` where illegal level values were still set after raising an error (GH13754)
- Bug in `DataFrame.to_json` where `lines=True` and a value contained a `}` character (GH14391)
- Bug in `df.groupby` causing an `AttributeError` when grouping a single index frame by a column and the index level (GH14327)
- Bug in `df.groupby` where `TypeError` raised when `pd.Grouper(key=...)` is passed in a list (GH14334)
- Bug in `pd.pivot_table` may raise `TypeError` or `ValueError` when `index` or `columns` is not scalar and `values` is not specified (GH14380)

Contributors

A total of 30 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Chainz +
- Anthonios Partheniou
- Arash Rouhani +
- Ben Kandel
- Brandon M. Burroughs +
- Chris
- Chris Warth
- David Krych +
- Iván Vallés Pérez +
- Jeff Reback
- Joe Jevnik

- Jon M. Mease +
- Jon Mease +
- Joris Van den Bossche
- Josh Owen +
- Keshav Ramaswamy +
- Larry Ren +
- Michael Felt +
- Piotr Chromiec +
- Robert Bradshaw +
- Sinhrks
- Thiago Serafim +
- Tom Bird
- bkandel +
- chris-b1
- dubourg +
- gfyoung
- mattrijk +
- paul-mannino +
- sinhrks

8.6.3 v0.19.0 (October 2, 2016)

This is a major release from 0.18.1 and includes number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- *merge_asof()* for asof-style time-series joining, see *here*
- `.rolling()` is now time-series aware, see *here*
- *read_csv()* now supports parsing `Categorical` data, see *here*
- A function `union_categorical()` has been added for combining categoricals, see *here*
- `PeriodIndex` now has its own `period` dtype, and changed to be more consistent with other `Index` classes. See *here*
- Sparse data structures gained enhanced support of `int` and `bool` dtypes, see *here*
- Comparison operations with `Series` no longer ignores the index, see *here* for an overview of the API changes.
- Introduction of a pandas development API for utility functions, see *here*.
- Deprecation of `Panel4D` and `PanelND`. We recommend to represent these types of n-dimensional data with the xarray package.
- Removal of the previously deprecated modules `pandas.io.data`, `pandas.io.wb`, `pandas.tools.rplot`.

Warning: pandas >= 0.19.0 will no longer silence numpy ufunc warnings upon import, see *here*.

What's new in v0.19.0

- *New features*
  - `merge_asof` *for asof-style time-series joining*
  - `.rolling()` *is now time-series aware*
  - `read_csv` *has improved support for duplicate column names*
  - `read_csv` *supports parsing* `Categorical` *directly*
  - *Categorical Concatenation*
  - *Semi-Month Offsets*
  - *New Index methods*
  - *Google BigQuery Enhancements*
  - *Fine-grained numpy errstate*
  - `get_dummies` *now returns integer dtypes*
  - *Downcast values to smallest possible dtype in* `to_numeric`
  - *pandas development API*
  - *Other enhancements*
- *API changes*
  - `Series.tolist()` *will now return Python types*
  - `Series` *operators for different indexes*
    * *Arithmetic operators*
    * *Comparison operators*
    * *Logical operators*
    * *Flexible comparison methods*
  - `Series` *type promotion on assignment*
  - `.to_datetime()` *changes*
  - *Merging changes*
  - `.describe()` *changes*
  - `Period` *changes*
    * `PeriodIndex` *now has* `period` *dtype*
    * `Period('NaT')` *now returns* `pd.NaT`
    * `PeriodIndex.values` *now returns array of* `Period` *object*
  - *Index* `+` / `-` *no longer used for set operations*
  - `Index.difference` *and* `.symmetric_difference` *changes*
  - `Index.unique` *consistently returns* `Index`

New features

merge_asof for asof-style time-series joining

A long-time requested feature has been added through the *merge_asof()* function, to support asof style joining of time-series (GH1870, GH13695, GH13709, GH13902). Full documentation is *here*.

The *merge_asof()* performs an asof merge, which is similar to a left-join except that we match on nearest key rather than equal keys.

```
In [1]: left = pd.DataFrame({'a': [1, 5, 10],
   ...:                      'left_val': ['a', 'b', 'c']})
   ...:

In [2]: right = pd.DataFrame({'a': [1, 2, 3, 6, 7],
   ...:                       'right_val': [1, 2, 3, 6, 7]})
   ...:

In [3]: left
Out[3]:
    a left_val
0   1        a
1   5        b
2  10        c

[3 rows x 2 columns]

In [4]: right
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:
↪
   a  right_val
0  1          1
1  2          2
2  3          3
```

(continues on next page)

(continued from previous page)

```
3  6            6
4  7            7

[5 rows x 2 columns]
```

We typically want to match exactly when possible, and use the most recent value otherwise.

```
In [5]: pd.merge_asof(left, right, on='a')
Out[5]:
    a left_val  right_val
0   1        a          1
1   5        b          3
2  10        c          7

[3 rows x 3 columns]
```

We can also match rows ONLY with prior data, and not an exact match.

```
In [6]: pd.merge_asof(left, right, on='a', allow_exact_matches=False)
Out[6]:
    a left_val  right_val
0   1        a        NaN
1   5        b        3.0
2  10        c        7.0

[3 rows x 3 columns]
```

In a typical time-series example, we have `trades` and `quotes` and we want to `asof-join` them. This also illustrates using the `by` parameter to group data before merging.

```
In [7]: trades = pd.DataFrame({
   ...:     'time': pd.to_datetime(['20160525 13:30:00.023',
   ...:                             '20160525 13:30:00.038',
   ...:                             '20160525 13:30:00.048',
   ...:                             '20160525 13:30:00.048',
   ...:                             '20160525 13:30:00.048']),
   ...:     'ticker': ['MSFT', 'MSFT',
   ...:                'GOOG', 'GOOG', 'AAPL'],
   ...:     'price': [51.95, 51.95,
   ...:               720.77, 720.92, 98.00],
   ...:     'quantity': [75, 155,
   ...:                  100, 100, 100]},
   ...:     columns=['time', 'ticker', 'price', 'quantity'])
   ...:

In [8]: quotes = pd.DataFrame({
   ...:     'time': pd.to_datetime(['20160525 13:30:00.023',
   ...:                             '20160525 13:30:00.023',
   ...:                             '20160525 13:30:00.030',
   ...:                             '20160525 13:30:00.041',
   ...:                             '20160525 13:30:00.048',
   ...:                             '20160525 13:30:00.049',
   ...:                             '20160525 13:30:00.072',
   ...:                             '20160525 13:30:00.075']),
   ...:     'ticker': ['GOOG', 'MSFT', 'MSFT', 'MSFT',
   ...:                'GOOG', 'AAPL', 'GOOG', 'MSFT'],
```

(continues on next page)

(continued from previous page)

```
   ...:      'bid': [720.50, 51.95, 51.97, 51.99,
   ...:              720.50, 97.99, 720.50, 52.01],
   ...:      'ask': [720.93, 51.96, 51.98, 52.00,
   ...:              720.93, 98.01, 720.88, 52.03]},
   ...:      columns=['time', 'ticker', 'bid', 'ask'])
   ...:
```

```
In [9]: trades
Out[9]:
                     time ticker   price  quantity
0 2016-05-25 13:30:00.023   MSFT   51.95        75
1 2016-05-25 13:30:00.038   MSFT   51.95       155
2 2016-05-25 13:30:00.048   GOOG  720.77       100
3 2016-05-25 13:30:00.048   GOOG  720.92       100
4 2016-05-25 13:30:00.048   AAPL   98.00       100

[5 rows x 4 columns]

In [10]: quotes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                     time ticker     bid     ask
0 2016-05-25 13:30:00.023   GOOG  720.50  720.93
1 2016-05-25 13:30:00.023   MSFT   51.95   51.96
2 2016-05-25 13:30:00.030   MSFT   51.97   51.98
3 2016-05-25 13:30:00.041   MSFT   51.99   52.00
4 2016-05-25 13:30:00.048   GOOG  720.50  720.93
5 2016-05-25 13:30:00.049   AAPL   97.99   98.01
6 2016-05-25 13:30:00.072   GOOG  720.50  720.88
7 2016-05-25 13:30:00.075   MSFT   52.01   52.03

[8 rows x 4 columns]
```

An asof merge joins on the `on`, typically a datetimelike field, which is ordered, and in this case we are using a grouper in the `by` field. This is like a left-outer join, except that forward filling happens automatically taking the most recent non-NaN value.

```
In [11]: pd.merge_asof(trades, quotes,
   ....:               on='time',
   ....:               by='ticker')
   ....:
Out[11]:
                     time ticker   price  quantity     bid     ask
0 2016-05-25 13:30:00.023   MSFT   51.95        75   51.95   51.96
1 2016-05-25 13:30:00.038   MSFT   51.95       155   51.97   51.98
2 2016-05-25 13:30:00.048   GOOG  720.77       100  720.50  720.93
3 2016-05-25 13:30:00.048   GOOG  720.92       100  720.50  720.93
4 2016-05-25 13:30:00.048   AAPL   98.00       100     NaN     NaN

[5 rows x 6 columns]
```

This returns a merged DataFrame with the entries in the same order as the original left passed DataFrame (`trades` in this case), with the fields of the `quotes` merged.

.rolling() is now time-series aware

.rolling() objects are now time-series aware and can accept a time-series offset (or convertible) for the window argument (GH13327, GH12995). See the full documentation *here*.

```
In [12]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.date_range('20130101 09:00:00',
   ....:                                        periods=5, freq='s'))
   ....:

In [13]: dft
Out[13]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0

[5 rows x 1 columns]
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [14]: dft.rolling(2).sum()
Out[14]:
                       B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  NaN

[5 rows x 1 columns]

In [15]: dft.rolling(2, min_periods=1).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0

[5 rows x 1 columns]
```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [16]: dft.rolling('2s').sum()
Out[16]:
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0

[5 rows x 1 columns]
```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```
In [17]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
   ....:                    index=pd.Index([pd.Timestamp('20130101 09:00:00'),
   ....:                                    pd.Timestamp('20130101 09:00:02'),
   ....:                                    pd.Timestamp('20130101 09:00:03'),
   ....:                                    pd.Timestamp('20130101 09:00:05'),
   ....:                                    pd.Timestamp('20130101 09:00:06')],
   ....:                                   name='foo'))
   ....:

In [18]: dft
Out[18]:
                       B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

[5 rows x 1 columns]

In [19]: dft.rolling(2).sum()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                       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
```

(continues on next page)

(continued from previous page)

```
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()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                  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)
Out[26]:
   a  b  a.1
0  0  1    2
1  3  4    5

[2 rows x 3 columns]
```

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
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
col1    object
col2    object
col3     int64
Length: 3, dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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
2    3
Name: col3, Length: 3, dtype: category
```

(continues on next page)

(continued from previous page)

```
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)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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(levels=[[1, 2, 4], [1, 2]],
           codes=[[0, 1, -1, 2], [0, 1, -1, -1]])

In [57]: midx.dropna()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪
MultiIndex(levels=[[1, 2, 4], [1, 2]],
           codes=[[0, 1], [0, 1]])

In [58]: midx.dropna(how='all')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[[1, 2, 4], [1, 2]],
           codes=[[0, 1, 2], [0, 1, -1]])
```

`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_datetimetz',
 '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',
 'is_period_dtype',
 'is_re',
 'is_re_compilable',
 'is_scalar',
 'is_signed_integer_dtype',
 'is_sparse',
 'is_string_dtype',
 '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()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            a
date
2015-01-31  6
2015-02-28  4

[2 rows x 1 columns]

In [74]: df.resample('M', level='d').sum()
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            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

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

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

(continues on next page)

(continued from previous page)

```
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], dtype=bool)
```

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

(continues on next page)

(continued from previous page)

```
[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")

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

In [102]: s
Out[102]:
a    2016-01-01 00:00:00
b                      3
```

(continues on next page)

(continued from previous page)

```
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

[1 rows x 2 columns]

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

(continues on next page)

(continued from previous page)

```
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]

In [112]: pd.merge(df1, df2, how='outer', on='key').dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[112]:␣
↪
```

(continues on next page)

(continued from previous page)

```
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
99.9%     3.996000
99.95%    3.998000
99.99%    3.999600
```

(continues on next page)

(continued from previous page)

```
max        4.000000
Length: 12, dtype: float64

In [116]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
               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
```

(continues on next page)

(continued from previous page)

```
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
↪period[D]

In [122]: type(pi.dtype)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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'],
```

(continues on next page)

(continued from previous page)

```
   .....:                    tz='Asia/Tokyo').unique()
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\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(levels=[['b', 'a', 'c'], ['bar', 'foo']],
           codes=[[1, 0], [1, 0]])
```

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)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪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')
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
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A       int64
C    category
B     float64
Length: 3, dtype: object

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

In [150]: df_set_idx.reset_index().dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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
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)
```

As of v0.19.0, sparse data keeps the input dtype, and uses more appropriate `fill_value` defaults (`0` for `int64` dtype, `False` for `bool` dtype).

```
In [153]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[153]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)

In [154]: pd.SparseArray([True, False, False, False])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[154]:␣
↪
[True, False, False, False]
Fill: False
IntIndex
Indices: array([0], dtype=int32)
```

See the *docs* for more details.

Operators now preserve dtypes

- Sparse data structure now can preserve `dtype` after arithmetic ops (GH13848)

```
In [155]: s = pd.SparseSeries([0, 2, 0, 1], fill_value=0, dtype=np.int64)

In [156]: s.dtype
Out[156]: Sparse[int64, 0]

In [157]: s + 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[157]:
0    1
1    3
2    1
3    2
Length: 4, dtype: Sparse[int64, 1]
BlockIndex
Block locations: array([1, 3], dtype=int32)
Block lengths: array([1, 1], dtype=int32)
```

- Sparse data structure now support `astype` to convert internal `dtype` (GH13900)

```
In [158]: s = pd.SparseSeries([1., 0., 2., 0.], fill_value=0)

In [159]: s
Out[159]:
0    1.0
1    0.0
2    2.0
3    0.0
Length: 4, dtype: Sparse[float64, 0]
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)

In [160]: s.astype(np.int64)
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    0
2    2
3    0
Length: 4, dtype: Sparse[int64, 0]
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)
```

`astype` fails if data contains values which cannot be converted to specified `dtype`. Note that the limitation is applied to `fill_value` which default is `np.nan`.

```
In [7]: pd.SparseSeries([1., np.nan, 2., np.nan], fill_value=np.nan).astype(np.int64)
Out[7]:
ValueError: unable to coerce current fill_value nan to int64 dtype
```

Other sparse fixes

- Subclassed `SparseDataFrame` and `SparseSeries` now preserve class types when slicing or transposing. (GH13787)
- `SparseArray` with `bool` dtype now supports logical (bool) operators (GH14000)
- Bug in `SparseSeries` with `MultiIndex []` indexing may raise `IndexError` (GH13144)
- Bug in `SparseSeries` with `MultiIndex []` indexing result may have normal `Index` (GH13144)
- Bug in `SparseDataFrame` in which `axis=None` did not default to `axis=0` (GH13048)
- Bug in `SparseSeries` and `SparseDataFrame` creation with `object` dtype may raise `TypeError` (GH11633)
- Bug in `SparseDataFrame` doesn't respect passed `SparseArray` or `SparseSeries` 's dtype and `fill_value` (GH13866)
- Bug in `SparseArray` and `SparseSeries` don't apply ufunc to `fill_value` (GH13853)
- Bug in `SparseSeries.abs` incorrectly keeps negative `fill_value` (GH13853)
- Bug in single row slicing on multi-type `SparseDataFrame` s, types were previously forced to float (GH13917)
- Bug in `SparseSeries` slicing changes integer dtype to float (GH8292)
- Bug in `SparseDataFarme` comparison ops may raise `TypeError` (GH13001)
- Bug in `SparseDataFarme.isnull` raises `ValueError` (GH8276)
- Bug in `SparseSeries` representation with `bool` dtype may raise `IndexError` (GH13110)
- Bug in `SparseSeries` and `SparseDataFrame` of `bool` or `int64` dtype may display its values like `float64` dtype (GH13110)
- Bug in sparse indexing using `SparseArray` with `bool` dtype may return incorrect result (GH13985)
- Bug in `SparseArray` created from `SparseSeries` may lose `dtype` (GH13999)
- Bug in `SparseSeries` comparison with dense returns normal `Series` rather than `SparseSeries` (GH13999)

Indexer dtype changes

Note: This change only affects 64 bit python running on Windows, and only affects relatively advanced indexing operations

Methods such as `Index.get_indexer` that return an indexer array, coerce that array to a "platform int", so that it can be directly used in 3rd party library operations like `numpy.take`. Previously, a platform int was defined as `np.int_` which corresponds to a C integer, but the correct type, and what is being used now, is `np.intp`, which corresponds to the C integer size that can hold a pointer (GH3033, GH13972).

These types are the same on many platform, but for 64 bit python on Windows, `np.int_` is 32 bits, and `np.intp` is 64 bits. Changing this behavior improves performance for many operations on that platform.

Previous behavior:

```
In [1]: i = pd.Index(['a', 'b', 'c'])

In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int32')
```

New behavior:

```
In [1]: i = pd.Index(['a', 'b', 'c'])

In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int64')
```

Other API Changes

- `Timestamp.to_pydatetime` will issue a `UserWarning` when `warn=True`, and the instance has a non-zero number of nanoseconds, previously this would print a message to stdout (GH14101).
- `Series.unique()` with datetime and timezone now returns return array of `Timestamp` with timezone (GH13565).
- `Panel.to_sparse()` will raise a `NotImplementedError` exception when called (GH13778).
- `Index.reshape()` will raise a `NotImplementedError` exception when called (GH12882).
- `.filter()` enforces mutual exclusion of the keyword arguments (GH12399).
- `eval`'s upcasting rules for `float32` types have been updated to be more consistent with NumPy's rules. New behavior will not upcast to `float64` if you multiply a pandas `float32` object by a scalar float64 (GH12388).
- An `UnsupportedFunctionCall` error is now raised if NumPy ufuncs like `np.mean` are called on groupby or resample objects (GH12811).
- `__setitem__` will no longer apply a callable rhs as a function instead of storing it. Call `where` directly to get the previous behavior (GH13299).
- Calls to `.sample()` will respect the random seed set via `numpy.random.seed(n)` (GH13161)
- `Styler.apply` is now more strict about the outputs your function must return. For `axis=0` or `axis=1`, the output shape must be identical. For `axis=None`, the output must be a DataFrame with identical columns and index labels (GH13222).
- `Float64Index.astype(int)` will now raise `ValueError` if `Float64Index` contains `NaN` values (GH13149)

- `TimedeltaIndex.astype(int)` and `DatetimeIndex.astype(int)` will now return `Int64Index` instead of `np.array` (GH13209)
- Passing `Period` with multiple frequencies to normal `Index` now returns `Index` with `object` dtype (GH13664)
- `PeriodIndex.fillna` with `Period` has different freq now coerces to `object` dtype (GH13664)
- Faceted boxplots from `DataFrame.boxplot(by=col)` now return a `Series` when `return_type` is not None. Previously these returned an `OrderedDict`. Note that when `return_type=None`, the default, these still return a 2-D NumPy array (GH12216, GH7096).
- `pd.read_hdf` will now raise a `ValueError` instead of `KeyError`, if a mode other than `r`, `r+` and `a` is supplied. (GH13623)
- `pd.read_csv()`, `pd.read_table()`, and `pd.read_hdf()` raise the builtin `FileNotFoundError` exception for Python 3.x when called on a nonexistent file; this is back-ported as `IOError` in Python 2.x (GH14086)
- More informative exceptions are passed through the csv parser. The exception type would now be the original exception type instead of `CParserError` (GH13652).
- `pd.read_csv()` in the C engine will now issue a `ParserWarning` or raise a `ValueError` when `sep` encoded is more than one character long (GH14065)
- `DataFrame.values` will now return `float64` with a `DataFrame` of mixed `int64` and `uint64` dtypes, conforming to `np.find_common_type` (GH10364, GH13917)
- `.groupby.groups` will now return a dictionary of `Index` objects, rather than a dictionary of `np.ndarray` or `lists` (GH14293)

Deprecations

- `Series.reshape` and `Categorical.reshape` have been deprecated and will be removed in a subsequent release (GH12882, GH12882)
- `PeriodIndex.to_datetime` has been deprecated in favor of `PeriodIndex.to_timestamp` (GH8254)
- `Timestamp.to_datetime` has been deprecated in favor of `Timestamp.to_pydatetime` (GH8254)
- `Index.to_datetime` and `DatetimeIndex.to_datetime` have been deprecated in favor of `pd.to_datetime` (GH8254)
- `pandas.core.datetools` module has been deprecated and will be removed in a subsequent release (GH14094)
- `SparseList` has been deprecated and will be removed in a future version (GH13784)
- `DataFrame.to_html()` and `DataFrame.to_latex()` have dropped the `colSpace` parameter in favor of `col_space` (GH13857)
- `DataFrame.to_sql()` has deprecated the `flavor` parameter, as it is superfluous when SQLAlchemy is not installed (GH13611)
- Deprecated `read_csv` keywords:
  - `compact_ints` and `use_unsigned` have been deprecated and will be removed in a future version (GH13320)
  - `buffer_lines` has been deprecated and will be removed in a future version (GH13360)
  - `as_recarray` has been deprecated and will be removed in a future version (GH13373)

  - `skip_footer` has been deprecated in favor of `skipfooter` and will be removed in a future version (GH13349)

- top-level `pd.ordered_merge()` has been renamed to `pd.merge_ordered()` and the original name will be removed in a future version (GH13358)
- `Timestamp.offset` property (and named arg in the constructor), has been deprecated in favor of `freq` (GH12160)
- `pd.tseries.util.pivot_annual` is deprecated. Use `pivot_table` as alternative, an example is *here* (GH736)
- `pd.tseries.util.isleapyear` has been deprecated and will be removed in a subsequent release. Datetime-likes now have a `.is_leap_year` property (GH13727)
- `Panel4D` and `PanelND` constructors are deprecated and will be removed in a future version. The recommended way to represent these types of n-dimensional data are with the xarray package. Pandas provides a `to_xarray()` method to automate this conversion (GH13564).
- `pandas.tseries.frequencies.get_standard_freq` is deprecated. Use `pandas.tseries.frequencies.to_offset(freq).rule_code` instead (GH13874)
- `pandas.tseries.frequencies.to_offset`'s `freqstr` keyword is deprecated in favor of `freq` (GH13874)
- `Categorical.from_array` has been deprecated and will be removed in a future version (GH13854)

Removal of prior version deprecations/changes

- The `SparsePanel` class has been removed (GH13778)
- The `pd.sandbox` module has been removed in favor of the external library `pandas-qt` (GH13670)
- The `pandas.io.data` and `pandas.io.wb` modules are removed in favor of the pandas-datareader package (GH13724).
- The `pandas.tools.rplot` module has been removed in favor of the seaborn package (GH13855)
- `DataFrame.to_csv()` has dropped the `engine` parameter, as was deprecated in 0.17.1 (GH11274, GH13419)
- `DataFrame.to_dict()` has dropped the `outtype` parameter in favor of `orient` (GH13627, GH8486)
- `pd.Categorical` has dropped setting of the `ordered` attribute directly in favor of the `set_ordered` method (GH13671)
- `pd.Categorical` has dropped the `levels` attribute in favor of `categories` (GH8376)
- `DataFrame.to_sql()` has dropped the `mysql` option for the `flavor` parameter (GH13611)
- `Panel.shift()` has dropped the `lags` parameter in favor of `periods` (GH14041)
- `pd.Index` has dropped the `diff` method in favor of `difference` (GH13669)
- `pd.DataFrame` has dropped the `to_wide` method in favor of `to_panel` (GH14039)
- `Series.to_csv` has dropped the `nanRep` parameter in favor of `na_rep` (GH13804)
- `Series.xs`, `DataFrame.xs`, `Panel.xs`, `Panel.major_xs`, and `Panel.minor_xs` have dropped the `copy` parameter (GH13781)
- `str.split` has dropped the `return_type` parameter in favor of `expand` (GH13701)

- Removal of the legacy time rules (offset aliases), deprecated since 0.17.0 (this has been alias since 0.8.0) (GH13590, GH13868). Now legacy time rules raises `ValueError`. For the list of currently supported offsets, see *here*.
- The default value for the `return_type` parameter for `DataFrame.plot.box` and `DataFrame.boxplot` changed from `None` to `"axes"`. These methods will now return a matplotlib axes by default instead of a dictionary of artists. See *here* (GH6581).
- The `tquery` and `uquery` functions in the `pandas.io.sql` module are removed (GH5950).

Performance Improvements

- Improved performance of sparse `IntIndex.intersect` (GH13082)
- Improved performance of sparse arithmetic with `BlockIndex` when the number of blocks are large, though recommended to use `IntIndex` in such cases (GH13082)
- Improved performance of `DataFrame.quantile()` as it now operates per-block (GH11623)
- Improved performance of float64 hash table operations, fixing some very slow indexing and groupby operations in python 3 (GH13166, GH13334)
- Improved performance of `DataFrameGroupBy.transform` (GH12737)
- Improved performance of `Index` and `Series` `.duplicated` (GH10235)
- Improved performance of `Index.difference` (GH12044)
- Improved performance of `RangeIndex.is_monotonic_increasing` and `is_monotonic_decreasing` (GH13749)
- Improved performance of datetime string parsing in `DatetimeIndex` (GH13692)
- Improved performance of hashing `Period` (GH12817)
- Improved performance of `factorize` of datetime with timezone (GH13750)
- Improved performance of by lazily creating indexing hashtables on larger Indexes (GH14266)
- Improved performance of `groupby.groups` (GH14293)
- Unnecessary materializing of a MultiIndex when introspecting for memory usage (GH14308)

Bug Fixes

- Bug in `groupby().shift()`, which could cause a segfault or corruption in rare circumstances when grouping by columns with missing values (GH13813)
- Bug in `groupby().cumsum()` calculating `cumprod` when `axis=1`. (GH13994)
- Bug in `pd.to_timedelta()` in which the `errors` parameter was not being respected (GH13613)
- Bug in `io.json.json_normalize()`, where non-ascii keys raised an exception (GH13213)
- Bug when passing a not-default-indexed `Series` as `xerr` or `yerr` in `.plot()` (GH11858)
- Bug in area plot draws legend incorrectly if subplot is enabled or legend is moved after plot (matplotlib 1.5.0 is required to draw area plot legend properly) (GH9161, GH13544)
- Bug in `DataFrame` assignment with an object-dtyped `Index` where the resultant column is mutable to the original object. (GH13522)
- Bug in matplotlib `AutoDataFormatter`; this restores the second scaled formatting and re-adds micro-second scaled formatting (GH13131)

- Bug in selection from a `HDFStore` with a fixed format and `start` and/or `stop` specified will now return the selected range (GH8287)
- Bug in `Categorical.from_codes()` where an unhelpful error was raised when an invalid `ordered` parameter was passed in (GH14058)
- Bug in `Series` construction from a tuple of integers on windows not returning default dtype (int64) (GH13646)
- Bug in `TimedeltaIndex` addition with a Datetime-like object where addition overflow was not being caught (GH14068)
- Bug in `.groupby(..).resample(..)` when the same object is called multiple times (GH13174)
- Bug in `.to_records()` when index name is a unicode string (GH13172)
- Bug in calling `.memory_usage()` on object which doesn't implement (GH12924)
- Regression in `Series.quantile` with nans (also shows up in `.median()` and `.describe()`); furthermore now names the `Series` with the quantile (GH13098, GH13146)
- Bug in `SeriesGroupBy.transform` with datetime values and missing groups (GH13191)
- Bug where empty `Series` were incorrectly coerced in datetime-like numeric operations (GH13844)
- Bug in `Categorical` constructor when passed a `Categorical` containing datetimes with timezones (GH14190)
- Bug in `Series.str.extractall()` with `str` index raises `ValueError` (GH13156)
- Bug in `Series.str.extractall()` with single group and quantifier (GH13382)
- Bug in `DatetimeIndex` and `Period` subtraction raises `ValueError` or `AttributeError` rather than `TypeError` (GH13078)
- Bug in `Index` and `Series` created with `NaN` and `NaT` mixed data may not have `datetime64` dtype (GH13324)
- Bug in `Index` and `Series` may ignore `np.datetime64('nat')` and `np.timdelta64('nat')` to infer dtype (GH13324)
- Bug in `PeriodIndex` and `Period` subtraction raises `AttributeError` (GH13071)
- Bug in `PeriodIndex` construction returning a `float64` index in some circumstances (GH13067)
- Bug in `.resample(..)` with a `PeriodIndex` not changing its `freq` appropriately when empty (GH13067)
- Bug in `.resample(..)` with a `PeriodIndex` not retaining its type or name with an empty `DataFrame` appropriately when empty (GH13212)
- Bug in `groupby(..).apply(..)` when the passed function returns scalar values per group (GH13468).
- Bug in `groupby(..).resample(..)` where passing some keywords would raise an exception (GH13235)
- Bug in `.tz_convert` on a tz-aware `DateTimeIndex` that relied on index being sorted for correct results (GH13306)
- Bug in `.tz_localize` with `dateutil.tz.tzlocal` may return incorrect result (GH13583)
- Bug in `DatetimeTZDtype` dtype with `dateutil.tz.tzlocal` cannot be regarded as valid dtype (GH13583)
- Bug in `pd.read_hdf()` where attempting to load an HDF file with a single dataset, that had one or more categorical columns, failed unless the key argument was set to the name of the dataset. (GH13231)
- Bug in `.rolling()` that allowed a negative integer window in construction of the `Rolling()` object, but would later fail on aggregation (GH13383)

- Bug in `Series` indexing with tuple-valued data and a numeric index (GH13509)
- Bug in printing `pd.DataFrame` where unusual elements with the `object` dtype were causing segfaults (GH13717)
- Bug in ranking `Series` which could result in segfaults (GH13445)
- Bug in various index types, which did not propagate the name of passed index (GH12309)
- Bug in `DatetimeIndex`, which did not honour the `copy=True` (GH13205)
- Bug in `DatetimeIndex.is_normalized` returns incorrectly for normalized date_range in case of local timezones (GH13459)
- Bug in `pd.concat` and `.append` may coerces `datetime64` and `timedelta` to `object` dtype containing python built-in `datetime` or `timedelta` rather than `Timestamp` or `Timedelta` (GH13626)
- Bug in `PeriodIndex.append` may raises `AttributeError` when the result is `object` dtype (GH13221)
- Bug in `CategoricalIndex.append` may accept normal `list` (GH13626)
- Bug in `pd.concat` and `.append` with the same timezone get reset to UTC (GH7795)
- Bug in `Series` and `DataFrame` `.append` raises `AmbiguousTimeError` if data contains datetime near DST boundary (GH13626)
- Bug in `DataFrame.to_csv()` in which float values were being quoted even though quotations were specified for non-numeric values only (GH12922, GH13259)
- Bug in `DataFrame.describe()` raising `ValueError` with only boolean columns (GH13898)
- Bug in `MultiIndex` slicing where extra elements were returned when level is non-unique (GH12896)
- Bug in `.str.replace` does not raise `TypeError` for invalid replacement (GH13438)
- Bug in `MultiIndex.from_arrays` which didn't check for input array lengths matching (GH13599)
- Bug in `cartesian_product` and `MultiIndex.from_product` which may raise with empty input arrays (GH12258)
- Bug in `pd.read_csv()` which may cause a segfault or corruption when iterating in large chunks over a stream/file under rare circumstances (GH13703)
- Bug in `pd.read_csv()` which caused errors to be raised when a dictionary containing scalars is passed in for `na_values` (GH12224)
- Bug in `pd.read_csv()` which caused BOM files to be incorrectly parsed by not ignoring the BOM (GH4793)
- Bug in `pd.read_csv()` with `engine='python'` which raised errors when a numpy array was passed in for `usecols` (GH12546)
- Bug in `pd.read_csv()` where the index columns were being incorrectly parsed when parsed as dates with a `thousands` parameter (GH14066)
- Bug in `pd.read_csv()` with `engine='python'` in which `NaN` values weren't being detected after data was converted to numeric values (GH13314)
- Bug in `pd.read_csv()` in which the `nrows` argument was not properly validated for both engines (GH10476)
- Bug in `pd.read_csv()` with `engine='python'` in which infinities of mixed-case forms were not being interpreted properly (GH13274)
- Bug in `pd.read_csv()` with `engine='python'` in which trailing `NaN` values were not being parsed (GH13320)

- Bug in `pd.read_csv()` with `engine='python'` when reading from a `tempfile.TemporaryFile` on Windows with Python 3 (GH13398)
- Bug in `pd.read_csv()` that prevents `usecols` kwarg from accepting single-byte unicode strings (GH13219)
- Bug in `pd.read_csv()` that prevents `usecols` from being an empty set (GH13402)
- Bug in `pd.read_csv()` in the C engine where the NULL character was not being parsed as NULL (GH14012)
- Bug in `pd.read_csv()` with `engine='c'` in which NULL `quotechar` was not accepted even though `quoting` was specified as `None` (GH13411)
- Bug in `pd.read_csv()` with `engine='c'` in which fields were not properly cast to float when quoting was specified as non-numeric (GH13411)
- Bug in `pd.read_csv()` in Python 2.x with non-UTF8 encoded, multi-character separated data (GH3404)
- Bug in `pd.read_csv()`, where aliases for utf-xx (e.g. UTF-xx, UTF_xx, utf_xx) raised UnicodeDecodeError (GH13549)
- Bug in `pd.read_csv`, `pd.read_table`, `pd.read_fwf`, `pd.read_stata` and `pd.read_sas` where files were opened by parsers but not closed if both `chunksize` and `iterator` were `None`. (GH13940)
- Bug in `StataReader`, `StataWriter`, `XportReader` and `SAS7BDATReader` where a file was not properly closed when an error was raised. (GH13940)
- Bug in `pd.pivot_table()` where `margins_name` is ignored when `aggfunc` is a list (GH13354)
- Bug in `pd.Series.str.zfill`, `center`, `ljust`, `rjust`, and `pad` when passing non-integers, did not raise `TypeError` (GH13598)
- Bug in checking for any null objects in a `TimedeltaIndex`, which always returned `True` (GH13603)
- Bug in `Series` arithmetic raises `TypeError` if it contains datetime-like as `object` dtype (GH13043)
- Bug `Series.isnull()` and `Series.notnull()` ignore `Period('NaT')` (GH13737)
- Bug `Series.fillna()` and `Series.dropna()` don't affect to `Period('NaT')` (GH13737
- Bug in `.fillna(value=np.nan)` incorrectly raises `KeyError` on a `category` dtyped `Series` (GH14021)
- Bug in extension dtype creation where the created types were not is/identical (GH13285)
- Bug in `.resample(..)` where incorrect warnings were triggered by IPython introspection (GH13618)
- Bug in `NaT` - `Period` raises `AttributeError` (GH13071)
- Bug in `Series` comparison may output incorrect result if rhs contains `NaT` (GH9005)
- Bug in `Series` and `Index` comparison may output incorrect result if it contains `NaT` with `object` dtype (GH13592)
- Bug in `Period` addition raises `TypeError` if `Period` is on right hand side (GH13069)
- Bug in `Peirod` and `Series` or `Index` comparison raises `TypeError` (GH13200)
- Bug in `pd.set_eng_float_format()` that would prevent NaN and Inf from formatting (GH11981)
- Bug in `.unstack` with `Categorical` dtype resets `.ordered` to `True` (GH13249)
- Clean some compile time warnings in datetime parsing (GH13607)
- Bug in `factorize` raises `AmbiguousTimeError` if data contains datetime near DST boundary (GH13750)

- Bug in .set_index raises AmbiguousTimeError if new index contains DST boundary and multi levels (GH12920)
- Bug in .shift raises AmbiguousTimeError if data contains datetime near DST boundary (GH13926)
- Bug in pd.read_hdf() returns incorrect result when a DataFrame with a categorical column and a query which doesn't match any values (GH13792)
- Bug in .iloc when indexing with a non lexsorted MultiIndex (GH13797)
- Bug in .loc when indexing with date strings in a reverse sorted DatetimeIndex (GH14316)
- Bug in Series comparison operators when dealing with zero dim NumPy arrays (GH13006)
- Bug in .combine_first may return incorrect dtype (GH7630, GH10567)
- Bug in groupby where apply returns different result depending on whether first result is None or not (GH12824)
- Bug in groupby(..).nth() where the group key is included inconsistently if called after .head()/.tail() (GH12839)
- Bug in .to_html, .to_latex and .to_string silently ignore custom datetime formatter passed through the formatters key word (GH10690)
- Bug in DataFrame.iterrows(), not yielding a Series subclasse if defined (GH13977)
- Bug in pd.to_numeric when errors='coerce' and input contains non-hashable objects (GH13324)
- Bug in invalid Timedelta arithmetic and comparison may raise ValueError rather than TypeError (GH13624)
- Bug in invalid datetime parsing in to_datetime and DatetimeIndex may raise TypeError rather than ValueError (GH11169, GH11287)
- Bug in Index created with tz-aware Timestamp and mismatched tz option incorrectly coerces timezone (GH13692)
- Bug in DatetimeIndex with nanosecond frequency does not include timestamp specified with end (GH13672)
- Bug in `Series when setting a slice with a np.timedelta64 (GH14155)
- Bug in Index raises OutOfBoundsDatetime if datetime exceeds datetime64[ns] bounds, rather than coercing to object dtype (GH13663)
- Bug in Index may ignore specified datetime64 or timedelta64 passed as dtype (GH13981)
- Bug in RangeIndex can be created without no arguments rather than raises TypeError (GH13793)
- Bug in .value_counts() raises OutOfBoundsDatetime if data exceeds datetime64[ns] bounds (GH13663)
- Bug in DatetimeIndex may raise OutOfBoundsDatetime if input np.datetime64 has other unit than ns (GH9114)
- Bug in Series creation with np.datetime64 which has other unit than ns as object dtype results in incorrect values (GH13876)
- Bug in resample with timedelta data where data was casted to float (GH13119).
- Bug in pd.isnull() pd.notnull() raise TypeError if input datetime-like has other unit than ns (GH13389)
- Bug in pd.merge() may raise TypeError if input datetime-like has other unit than ns (GH13389)
- Bug in HDFStore/read_hdf() discarded DatetimeIndex.name if tz was set (GH13884)

- Bug in `Categorical.remove_unused_categories()` changes `.codes` dtype to platform int (GH13261)
- Bug in `groupby` with `as_index=False` returns all NaN's when grouping on multiple columns including a categorical one (GH13204)
- Bug in `df.groupby(...)[...]` where getitem with `Int64Index` raised an error (GH13731)
- Bug in the CSS classes assigned to `DataFrame.style` for index names. Previously they were assigned `"col_heading level<n> col<c>"` where `n` was the number of levels + 1. Now they are assigned `"index_name level<n>"`, where `n` is the correct level for that MultiIndex.
- Bug where `pd.read_gbq()` could throw `ImportError: No module named discovery` as a result of a naming conflict with another python package called apiclient (GH13454)
- Bug in `Index.union` returns an incorrect result with a named empty index (GH13432)
- Bugs in `Index.difference` and `DataFrame.join` raise in Python3 when using mixed-integer indexes (GH13432, GH12814)
- Bug in subtract tz-aware `datetime.datetime` from tz-aware `datetime64` series (GH14088)
- Bug in `.to_excel()` when DataFrame contains a MultiIndex which contains a label with a NaN value (GH13511)
- Bug in invalid frequency offset string like "D1", "-2-3H" may not raise `ValueError` (GH13930)
- Bug in `concat` and `groupby` for hierarchical frames with `RangeIndex` levels (GH13542).
- Bug in `Series.str.contains()` for Series containing only `NaN` values of `object` dtype (GH14171)
- Bug in `agg()` function on groupby dataframe changes dtype of `datetime64[ns]` column to `float64` (GH12821)
- Bug in using NumPy ufunc with `PeriodIndex` to add or subtract integer raise `IncompatibleFrequency`. Note that using standard operator like `+` or `-` is recommended, because standard operators use more efficient path (GH13980)
- Bug in operations on `NaT` returning `float` instead of `datetime64[ns]` (GH12941)
- Bug in `Series` flexible arithmetic methods (like `.add()`) raises `ValueError` when `axis=None` (GH13894)
- Bug in `DataFrame.to_csv()` with `MultiIndex` columns in which a stray empty line was added (GH6618)
- Bug in `DatetimeIndex`, `TimedeltaIndex` and `PeriodIndex.equals()` may return `True` when input isn't `Index` but contains the same values (GH13107)
- Bug in assignment against datetime with timezone may not work if it contains datetime near DST boundary (GH14146)
- Bug in `pd.eval()` and `HDFStore` query truncating long float literals with python 2 (GH14241)
- Bug in `Index` raises `KeyError` displaying incorrect column when column is not in the df and columns contains duplicate values (GH13822)
- Bug in `Period` and `PeriodIndex` creating wrong dates when frequency has combined offset aliases (GH13874)
- Bug in `.to_string()` when called with an integer `line_width` and `index=False` raises an UnboundLocalError exception because `idx` referenced before assignment.
- Bug in `eval()` where the `resolvers` argument would not accept a list (GH14095)

- Bugs in `stack`, `get_dummies`, `make_axis_dummies` which don't preserve categorical dtypes in (multi)indexes (GH13854)
- `PeriodIndex` can now accept `list` and `array` which contains `pd.NaT` (GH13430)
- Bug in `df.groupby` where `.median()` returns arbitrary values if grouped dataframe contains empty bins (GH13629)
- Bug in `Index.copy()` where `name` parameter was ignored (GH14302)

Contributors

A total of 117 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adrien Emery +
- Alex Alekseyev
- Alex Vig +
- Allen Riddell +
- Amol +
- Amol Agrawal +
- Andy R. Terrel +
- Anthonios Partheniou
- Ben Kandel +
- Bob Baxley +
- Brett Rosen +
- Camilo Cota +
- Chris
- Chris Grinolds
- Chris Warth
- Christian Hudon
- Christopher C. Aycock
- Daniel Siladji +
- Douglas McNeil
- Drewrey Lupton +
- Eduardo Blancas Reyes +
- Elliot Marsden +
- Evan Wright
- Felix Marczinowski +
- Francis T. O'Donovan
- Geraint Duck +
- Giacomo Ferroni +

- Grant Roch +
- Gábor Lipták
- Haleemur Ali +
- Hassan Shamim +
- Iulius Curt +
- Ivan Nazarov +
- Jeff Reback
- Jeffrey Gerard +
- Jenn Olsen +
- Jim Crist
- Joe Jevnik
- John Evans +
- John Freeman
- John Liekezer +
- John W. O'Brien
- John Zwinck +
- Johnny Gill +
- Jordan Erenrich +
- Joris Van den Bossche
- Josh Howes +
- Jozef Brandys +
- Ka Wo Chen
- Kamil Sindi +
- Kerby Shedden
- Kernc +
- Kevin Sheppard
- Matthieu Brucher +
- Maximilian Roos
- Michael Scherer +
- Mike Graham +
- Mortada Mehyar
- Muhammad Haseeb Tariq +
- Nate George +
- Neil Parley +
- Nicolas Bonnotte
- OXPHOS

- Pan Deng / Zora +
- Paul +
- Paul Mestemaker +
- Pauli Virtanen
- Pawel Kordek +
- Pietro Battiston
- Piotr Jucha +
- Ravi Kumar Nimmi +
- Robert Gieseke
- Robert Kern +
- Roger Thomas
- Roy Keyes +
- Russell Smith +
- Sahil Dua +
- Sanjiv Lobo +
- Sašo Stanovnik +
- Shawn Heide +
- Sinhrks
- Stephen Kappel +
- Steve Choi +
- Stewart Henderson +
- Sudarshan Konge +
- Thomas A Caswell
- Tom Augspurger
- Tom Bird +
- Uwe Hoffmann +
- WillAyd +
- Xiang Zhang +
- YG-Riku +
- Yadunandan +
- Yaroslav Halchenko
- Yuichiro Kaneko +
- adneu
- agraboso +
- babakkeyvani +
- c123w +

- chris-b1
- cmazzullo +
- conquistador1492 +
- cr3 +
- dsm054
- gfyoung
- harshul1610 +
- iamsimha +
- jackieleng +
- mpuels +
- pijucha +
- priyankjain +
- sinhrks
- wcwagner +
- yui-knk +
- zhangjinjie +
- znmean +
- Yan Facai +

8.7 Version 0.18

8.7.1 v0.18.1 (May 3, 2016)

This is a minor bug-fix release from 0.18.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- `.groupby(...)` has been enhanced to provide convenient syntax when working with `.rolling(..)`, `.expanding(..)` and `.resample(..)` per group, see *here*
- `pd.to_datetime()` has gained the ability to assemble dates from a `DataFrame`, see *here*
- Method chaining improvements, see *here*.
- Custom business hour offset, see *here*.
- Many bug fixes in the handling of `sparse`, see *here*
- Expanded the *Tutorials section* with a feature on modern pandas, courtesy of @TomAugsburger. (GH13045).

What's new in v0.18.1

- *New features*
  - *Custom Business Hour*

New features

Custom Business Hour

The `CustomBusinessHour` is a mixture of `BusinessHour` and `CustomBusinessDay` which allows you to specify arbitrary holidays. For details, see *Custom Business Hour* (GH11514)

```
In [1]: from pandas.tseries.offsets import CustomBusinessHour

In [2]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [3]: bhour_us = CustomBusinessHour(calendar=USFederalHolidayCalendar())
```

Friday before MLK Day

```
In [4]: import datetime

In [5]: dt = datetime.datetime(2014, 1, 17, 15)

In [6]: dt + bhour_us
Out[6]: Timestamp('2014-01-17 16:00:00')
```

Tuesday after MLK Day (Monday is skipped because it’s a holiday)

```
In [7]: dt + bhour_us * 2
Out[7]: Timestamp('2014-01-20 09:00:00')
```

.groupby(..) syntax with window and resample operations

.groupby(...) has been enhanced to provide convenient syntax when working with .rolling(..), .expanding(..) and .resample(..) per group, see (GH12486, GH12738).

You can now use .rolling(..) and .expanding(..) as methods on groupbys. These return another deferred object (similar to what .rolling() and .expanding() do on ungrouped pandas objects). You can then operate on these RollingGroupby objects in a similar manner.

Previously you would have to do this to get a rolling window mean per-group:

```
In [8]: df = pd.DataFrame({'A': [1] * 20 + [2] * 12 + [3] * 8,
   ...:                    'B': np.arange(40)})
   ...:

In [9]: df
Out[9]:
    A   B
0   1   0
1   1   1
2   1   2
3   1   3
4   1   4
5   1   5
6   1   6
.. ..  ..
33  3  33
34  3  34
35  3  35
36  3  36
37  3  37
38  3  38
39  3  39

[40 rows x 2 columns]
```

```
In [10]: df.groupby('A').apply(lambda x: x.rolling(4).B.mean())
Out[10]:
A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
         ...
3  33     NaN
   34     NaN
   35    33.5
   36    34.5
   37    35.5
   38    36.5
```

(continues on next page)

(continued from previous page)

```
   39    37.5
Name: B, Length: 40, dtype: float64
```

Now you can do:

```
In [11]: df.groupby('A').rolling(4).B.mean()
Out[11]:
A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
         ...
3  33     NaN
   34     NaN
   35    33.5
   36    34.5
   37    35.5
   38    36.5
   39    37.5
Name: B, Length: 40, dtype: float64
```

For `.resample(..)` type of operations, previously you would have to:

```
In [12]: df = pd.DataFrame({'date': pd.date_range(start='2016-01-01',
   ....:                                         periods=4,
   ....:                                         freq='W'),
   ....:                    'group': [1, 1, 2, 2],
   ....:                    'val': [5, 6, 7, 8]}).set_index('date')
   ....:

In [13]: df
Out[13]:
            group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

[4 rows x 2 columns]
```

```
In [14]: df.groupby('group').apply(lambda x: x.resample('1D').ffill())
Out[14]:
                  group  val
group date
1     2016-01-03      1    5
      2016-01-04      1    5
      2016-01-05      1    5
      2016-01-06      1    5
      2016-01-07      1    5
      2016-01-08      1    5
      2016-01-09      1    5
```

(continues on next page)

(continued from previous page)

```
...                    ...  ...
2      2016-01-18        2    7
       2016-01-19        2    7
       2016-01-20        2    7
       2016-01-21        2    7
       2016-01-22        2    7
       2016-01-23        2    7
       2016-01-24        2    8

[16 rows x 2 columns]
```

Now you can do:

```
In [15]: df.groupby('group').resample('1D').ffill()
Out[15]:
                  group  val
group date
1      2016-01-03        1    5
       2016-01-04        1    5
       2016-01-05        1    5
       2016-01-06        1    5
       2016-01-07        1    5
       2016-01-08        1    5
       2016-01-09        1    5
...                    ...  ...
2      2016-01-18        2    7
       2016-01-19        2    7
       2016-01-20        2    7
       2016-01-21        2    7
       2016-01-22        2    7
       2016-01-23        2    7
       2016-01-24        2    8

[16 rows x 2 columns]
```

Method chaining improvements

The following methods / indexers now accept a `callable`. It is intended to make these more useful in method chains, see the *documentation*. (GH11485, GH12533)

- `.where()` and `.mask()`
- `.loc[]`, `iloc[]` and `.ix[]`
- `[]` indexing

`.where()` and `.mask()`

These can accept a callable for the condition and `other` arguments.

```
In [16]: df = pd.DataFrame({'A': [1, 2, 3],
   ....:                    'B': [4, 5, 6],
   ....:                    'C': [7, 8, 9]})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [17]: df.where(lambda x: x > 4, lambda x: x + 10)
Out[17]:
    A   B  C
0  11  14  7
1  12   5  8
2  13   6  9

[3 rows x 3 columns]
```

.loc[], .iloc[], .ix[]

These can accept a callable, and a tuple of callable as a slicer. The callable can return a valid boolean indexer or anything which is valid for these indexer's input.

```
# callable returns bool indexer
In [18]: df.loc[lambda x: x.A >= 2, lambda x: x.sum() > 10]
Out[18]:
   B  C
1  5  8
2  6  9

[2 rows x 2 columns]

# callable returns list of labels
In [19]: df.loc[lambda x: [1, 2], lambda x: ['A', 'B']]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:
   A  B
1  2  5
2  3  6

[2 rows x 2 columns]
```

[] indexing

Finally, you can use a callable in `[]` indexing of Series, DataFrame and Panel. The callable must return a valid input for `[]` indexing depending on its class and index type.

```
In [20]: df[lambda x: 'A']
Out[20]:
0    1
1    2
2    3
Name: A, Length: 3, dtype: int64
```

Using these methods / indexers, you can chain data selection operations without using temporary variable.

```
In [21]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [22]: (bb.groupby(['year', 'team'])
   ....:    .sum()
   ....:    .loc[lambda df: df.r > 100])
   ....:
```

(continues on next page)

(continued from previous page)

```
Out[22]:
           stint    g    ab    r    h  X2b  X3b  hr    rbi    sb   cs   bb     so  ␣
↪ibb   hbp    sh    sf  gidp
year team                                                                          ␣
↪
2007 CIN       6  379   745  101  203   35    2  36  125.0  10.0  1.0  105  127.0  14.
↪0   1.0   1.0  15.0  18.0
     DET       5  301  1062  162  283   54    4  37  144.0  24.0  7.0   97  176.0   3.
↪0  10.0   4.0   8.0  28.0
     HOU       4  311   926  109  218   47    6  14   77.0  10.0  4.0   60  212.0   3.
↪0   9.0  16.0   6.0  17.0
     LAN      11  413  1021  153  293   61    3  36  154.0   7.0  5.0  114  141.0   8.
↪0   9.0   3.0   8.0  29.0
     NYN      13  622  1854  240  509  101    3  61  243.0  22.0  4.0  174  310.0  24.
↪0  23.0  18.0  15.0  48.0
     SFN       5  482  1305  198  337   67    6  40  171.0  26.0  7.0  235  188.0  51.
↪0   8.0  16.0   6.0  41.0
     TEX       2  198   729  115  200   40    4  28  115.0  21.0  4.0   73  140.0   4.
↪0   5.0   2.0   8.0  16.0
     TOR       4  459  1408  187  378   96    2  58  223.0   4.0  2.0  190  265.0  16.
↪0  12.0   4.0  16.0  38.0

[8 rows x 18 columns]
```

Partial string indexing on `DateTimeIndex` when part of a `MultiIndex`

Partial string indexing now matches on `DateTimeIndex` when part of a `MultiIndex` (GH10331)

```
In [23]: dft2 = pd.DataFrame(
   ....:     np.random.randn(20, 1),
   ....:     columns=['A'],
   ....:     index=pd.MultiIndex.from_product([pd.date_range('20130101',
   ....:                                                      periods=10,
   ....:                                                      freq='12H'),
   ....:                                       ['a', 'b']]))
   ....:

In [24]: dft2
Out[24]:
                              A
2013-01-01 00:00:00 a  0.469112
                    b -0.282863
2013-01-01 12:00:00 a -1.509059
                    b -1.135632
2013-01-02 00:00:00 a  1.212112
                    b -0.173215
2013-01-02 12:00:00 a  0.119209
...                         ...
2013-01-04 00:00:00 b -0.706771
2013-01-04 12:00:00 a -1.039575
                    b  0.271860
2013-01-05 00:00:00 a -0.424972
                    b  0.567020
2013-01-05 12:00:00 a  0.276232
                    b -1.087401
```

(continues on next page)

(continued from previous page)

```
[20 rows x 1 columns]

In [25]: dft2.loc['2013-01-05']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                                A
2013-01-05 00:00:00 a -0.424972
                    b  0.567020
2013-01-05 12:00:00 a  0.276232
                    b -1.087401

[4 rows x 1 columns]
```

On other levels

```
In [26]: idx = pd.IndexSlice

In [27]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [28]: dft2
Out[28]:
                                A
a 2013-01-01 00:00:00  0.469112
  2013-01-01 12:00:00 -1.509059
  2013-01-02 00:00:00  1.212112
  2013-01-02 12:00:00  0.119209
  2013-01-03 00:00:00 -0.861849
  2013-01-03 12:00:00 -0.494929
  2013-01-04 00:00:00  0.721555
...                           ...
b 2013-01-02 12:00:00 -1.044236
  2013-01-03 00:00:00 -2.104569
  2013-01-03 12:00:00  1.071804
  2013-01-04 00:00:00 -0.706771
  2013-01-04 12:00:00  0.271860
  2013-01-05 00:00:00  0.567020
  2013-01-05 12:00:00 -1.087401

[20 rows x 1 columns]

In [29]: dft2.loc[idx[:, '2013-01-05'], :]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                                A
a 2013-01-05 00:00:00 -0.424972
  2013-01-05 12:00:00  0.276232
b 2013-01-05 00:00:00  0.567020
  2013-01-05 12:00:00 -1.087401

[4 rows x 1 columns]
```

Assembling Datetimes

`pd.to_datetime()` has gained the ability to assemble datetimes from a passed in `DataFrame` or a dict. (GH8158).

```
In [30]: df = pd.DataFrame({'year': [2015, 2016],
   ....:                    'month': [2, 3],
   ....:                    'day': [4, 5],
   ....:                    'hour': [2, 3]})
   ....:

In [31]: df
Out[31]:
   year  month  day  hour
0  2015      2    4     2
1  2016      3    5     3

[2 rows x 4 columns]
```

Assembling using the passed frame.

```
In [32]: pd.to_datetime(df)
Out[32]:
0   2015-02-04 02:00:00
1   2016-03-05 03:00:00
Length: 2, dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```
In [33]: pd.to_datetime(df[['year', 'month', 'day']])
Out[33]:
0   2015-02-04
1   2016-03-05
Length: 2, dtype: datetime64[ns]
```

Other Enhancements

- `pd.read_csv()` now supports `delim_whitespace=True` for the Python engine (GH12958)
- `pd.read_csv()` now supports opening ZIP files that contains a single CSV, via extension inference or explicit `compression='zip'` (GH12175)
- `pd.read_csv()` now supports opening files using xz compression, via extension inference or explicit `compression='xz'` is specified; `xz` compressions is also supported by `DataFrame.to_csv` in the same way (GH11852)
- `pd.read_msgpack()` now always gives writeable ndarrays even when compression is used (GH12359).
- `pd.read_msgpack()` now supports serializing and de-serializing categoricals with msgpack (GH12573)
- `.to_json()` now supports `NDFrames` that contain categorical and sparse data (GH10778)
- `interpolate()` now supports `method='akima'` (GH7588).
- `pd.read_excel()` now accepts path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path, in line with other `read_*` functions (GH12655)
- Added `.weekday_name` property as a component to `DatetimeIndex` and the `.dt` accessor. (GH11128)

- `Index.take` now handles `allow_fill` and `fill_value` consistently (GH12631)

```
In [34]: idx = pd.Index([1., 2., 3., 4.], dtype='float')

# default, allow_fill=True, fill_value=None
In [35]: idx.take([2, -1])
Out[35]: Float64Index([3.0, 4.0], dtype='float64')

In [36]: idx.take([2, -1], fill_value=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[36]: Float64Index([3.0,␣
↪nan], dtype='float64')
```

- `Index` now supports `.str.get_dummies()` which returns `MultiIndex`, see *Creating Indicator Variables* (GH10008, GH10103)

```
In [37]: idx = pd.Index(['a|b', 'a|c', 'b|c'])

In [38]: idx.str.get_dummies('|')
Out[38]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
           codes=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
           names=['a', 'b', 'c'])
```

- `pd.crosstab()` has gained a `normalize` argument for normalizing frequency tables (GH12569). Examples in the updated docs *here*.
- `.resample(..).interpolate()` is now supported (GH12925)
- `.isin()` now accepts passed `sets` (GH12988)

Sparse changes

These changes conform sparse handling to return the correct types and work to make a smoother experience with indexing.

`SparseArray.take` now returns a scalar for scalar input, `SparseArray` for others. Furthermore, it handles a negative indexer with the same rule as `Index` (GH10560, GH12796)

```
s = pd.SparseArray([np.nan, np.nan, 1, 2, 3, np.nan, 4, 5, np.nan, 6])
s.take(0)
s.take([1, 2, 3])
```

- Bug in `SparseSeries[]` indexing with `Ellipsis` raises `KeyError` (GH9467)
- Bug in `SparseArray[]` indexing with tuples are not handled properly (GH12966)
- Bug in `SparseSeries.loc[]` with list-like input raises `TypeError` (GH10560)
- Bug in `SparseSeries.iloc[]` with scalar input may raise `IndexError` (GH10560)
- Bug in `SparseSeries.loc[]`, `.iloc[]` with `slice` returns `SparseArray`, rather than `SparseSeries` (GH10560)
- Bug in `SparseDataFrame.loc[]`, `.iloc[]` may results in dense `Series`, rather than `SparseSeries` (GH12787)
- Bug in `SparseArray` addition ignores `fill_value` of right hand side (GH12910)
- Bug in `SparseArray` mod raises `AttributeError` (GH12910)
- Bug in `SparseArray` pow calculates `1 ** np.nan` as `np.nan` which must be 1 (GH12910)

- Bug in `SparseArray` comparison output may incorrect result or raise `ValueError` (GH12971)
- Bug in `SparseSeries.__repr__` raises `TypeError` when it is longer than `max_rows` (GH10560)
- Bug in `SparseSeries.shape` ignores `fill_value` (GH10452)
- Bug in `SparseSeries` and `SparseArray` may have different `dtype` from its dense values (GH12908)
- Bug in `SparseSeries.reindex` incorrectly handle `fill_value` (GH12797)
- Bug in `SparseArray.to_frame()` results in `DataFrame`, rather than `SparseDataFrame` (GH9850)
- Bug in `SparseSeries.value_counts()` does not count `fill_value` (GH6749)
- Bug in `SparseArray.to_dense()` does not preserve `dtype` (GH10648)
- Bug in `SparseArray.to_dense()` incorrectly handle `fill_value` (GH12797)
- Bug in `pd.concat()` of `SparseSeries` results in dense (GH10536)
- Bug in `pd.concat()` of `SparseDataFrame` incorrectly handle `fill_value` (GH9765)
- Bug in `pd.concat()` of `SparseDataFrame` may raise `AttributeError` (GH12174)
- Bug in `SparseArray.shift()` may raise `NameError` or `TypeError` (GH12908)

API changes

`.groupby(..).nth()` changes

The index in `.groupby(..).nth()` output is now more consistent when the `as_index` argument is passed (GH11039):

```
In [39]: df = pd.DataFrame({'A': ['a', 'b', 'a'],
   ....:                    'B': [1, 2, 3]})
   ....:

In [40]: df
Out[40]:
   A  B
0  a  1
1  b  2
2  a  3

[3 rows x 2 columns]
```

Previous Behavior:

```
In [3]: df.groupby('A', as_index=True)['B'].nth(0)
Out[3]:
0    1
1    2
Name: B, dtype: int64

In [4]: df.groupby('A', as_index=False)['B'].nth(0)
Out[4]:
0    1
1    2
Name: B, dtype: int64
```

New Behavior:

```
In [41]: df.groupby('A', as_index=True)['B'].nth(0)
Out[41]:
A
a    1
b    2
Name: B, Length: 2, dtype: int64

In [42]: df.groupby('A', as_index=False)['B'].nth(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[42]:
0    1
1    2
Name: B, Length: 2, dtype: int64
```

Furthermore, previously, a `.groupby` would always sort, regardless if `sort=False` was passed with `.nth()`.

```
In [43]: np.random.seed(1234)

In [44]: df = pd.DataFrame(np.random.randn(100, 2), columns=['a', 'b'])

In [45]: df['c'] = np.random.randint(0, 4, 100)
```

Previous Behavior:

```
In [4]: df.groupby('c', sort=True).nth(1)
Out[4]:
          a         b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524

In [5]: df.groupby('c', sort=False).nth(1)
Out[5]:
          a         b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524
```

New Behavior:

```
In [46]: df.groupby('c', sort=True).nth(1)
Out[46]:
          a         b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524

[4 rows x 2 columns]

In [47]: df.groupby('c', sort=False).nth(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
          a         b
c
2 -0.720589  0.887163
3  0.859588 -0.636524
0 -0.334077  0.002118
1  0.036142 -2.074978

[4 rows x 2 columns]
```

numpy function compatibility

Compatibility between pandas array-like methods (e.g. `sum` and `take`) and their `numpy` counterparts has been greatly increased by augmenting the signatures of the `pandas` methods so as to accept arguments that can be passed in from `numpy`, even if they are not necessarily used in the `pandas` implementation (GH12644, GH12638, GH12687)

- `.searchsorted()` for `Index` and `TimedeltaIndex` now accept a `sorter` argument to maintain compatibility with numpy's `searchsorted` function (GH12238)
- Bug in numpy compatibility of `np.round()` on a `Series` (GH12600)

An example of this signature augmentation is illustrated below:

```
In [48]: sp = pd.SparseDataFrame([1, 2, 3])

In [49]: sp
Out[49]:
   0
0  1
1  2
2  3

[3 rows x 1 columns]
```

Previous behaviour:

```
In [2]: np.cumsum(sp, axis=0)
...
TypeError: cumsum() takes at most 2 arguments (4 given)
```

New behaviour:

```
In [50]: np.cumsum(sp, axis=0)
Out[50]:
   0
0  1
1  3
2  6

[3 rows x 1 columns]
```

Using `.apply` on groupby resampling

Using `apply` on resampling groupby operations (using a `pd.TimeGrouper`) now has the same output types as similar `apply` calls on other groupby operations. (GH11742).

```
In [51]: df = pd.DataFrame({'date': pd.to_datetime(['10/10/2000', '11/10/2000']),
   ....:                    'value': [10, 13]})
   ....:

In [52]: df
Out[52]:
        date  value
0 2000-10-10     10
1 2000-11-10     13

[2 rows x 2 columns]
```

Previous behavior:

```
In [1]: df.groupby(pd.TimeGrouper(key='date',
   ...:                           freq='M')).apply(lambda x: x.value.sum())
Out[1]:
...
TypeError: cannot concatenate a non-NDFrame object

# Output is a Series
In [2]: df.groupby(pd.TimeGrouper(key='date',
   ...:                           freq='M')).apply(lambda x: x[['value']].sum())
Out[2]:
date
2000-10-31  value    10
2000-11-30  value    13
dtype: int64
```

New Behavior:

```
# Output is a Series
In [55]: df.groupby(pd.TimeGrouper(key='date',
    ...:                           freq='M')).apply(lambda x: x.value.sum())
Out[55]:
date
2000-10-31    10
2000-11-30    13
Freq: M, dtype: int64

# Output is a DataFrame
In [56]: df.groupby(pd.TimeGrouper(key='date',
    ...:                           freq='M')).apply(lambda x: x[['value']].sum())
Out[56]:
            value
date
2000-10-31     10
2000-11-30     13
```

Changes in `read_csv` exceptions

In order to standardize the `read_csv` API for both the `c` and `python` engines, both will now raise an `EmptyDataError`, a subclass of `ValueError`, in response to empty columns or header (GH12493, GH12506)

Previous behaviour:

```
In [1]: import io

In [2]: df = pd.read_csv(io.StringIO(''), engine='c')
...
ValueError: No columns to parse from file

In [3]: df = pd.read_csv(io.StringIO(''), engine='python')
...
StopIteration
```

New behaviour:

```
In [1]: df = pd.read_csv(io.StringIO(''), engine='c')
...
pandas.io.common.EmptyDataError: No columns to parse from file

In [2]: df = pd.read_csv(io.StringIO(''), engine='python')
...
pandas.io.common.EmptyDataError: No columns to parse from file
```

In addition to this error change, several others have been made as well:

- `CParserError` now sub-classes `ValueError` instead of just a `Exception` (GH12551)
- A `CParserError` is now raised instead of a generic `Exception` in `read_csv` when the `c` engine cannot parse a column (GH12506)
- A `ValueError` is now raised instead of a generic `Exception` in `read_csv` when the `c` engine encounters a `NaN` value in an integer column (GH12506)
- A `ValueError` is now raised instead of a generic `Exception` in `read_csv` when `true_values` is specified, and the `c` engine encounters an element in a column containing unencodable bytes (GH12506)
- `pandas.parser.OverflowError` exception has been removed and has been replaced with Python's built-in `OverflowError` exception (GH12506)
- `pd.read_csv()` no longer allows a combination of strings and integers for the `usecols` parameter (GH12678)

`to_datetime` error changes

Bugs in `pd.to_datetime()` when passing a `unit` with convertible entries and `errors='coerce'` or non-convertible with `errors='ignore'`. Furthermore, an `OutOfBoundsDateime` exception will be raised when an out-of-range value is encountered for that unit when `errors='raise'`. (GH11758, GH13052, GH13059)

Previous behaviour:

```
In [27]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[27]: NaT

In [28]: pd.to_datetime(11111111, unit='D', errors='ignore')
OverflowError: Python int too large to convert to C long

In [29]: pd.to_datetime(11111111, unit='D', errors='raise')
OverflowError: Python int too large to convert to C long
```

New behaviour:

```
In [2]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[2]: Timestamp('2014-12-31 16:31:00')

In [3]: pd.to_datetime(11111111, unit='D', errors='ignore')
Out[3]: 11111111

In [4]: pd.to_datetime(11111111, unit='D', errors='raise')
OutOfBoundsDatetime: cannot convert input with unit 'D'
```

Other API changes

- `.swaplevel()` for `Series`, `DataFrame`, `Panel`, and `MultiIndex` now features defaults for its first two parameters `i` and `j` that swap the two innermost levels of the index. (GH12934)
- `.searchsorted()` for `Index` and `TimedeltaIndex` now accept a `sorter` argument to maintain compatibility with numpy's `searchsorted` function (GH12238)
- `Period` and `PeriodIndex` now raises `IncompatibleFrequency` error which inherits `ValueError` rather than raw `ValueError` (GH12615)
- `Series.apply` for category dtype now applies the passed function to each of the `.categories` (and not the `.codes`), and returns a `category` dtype if possible (GH12473)
- `read_csv` will now raise a `TypeError` if `parse_dates` is neither a boolean, list, or dictionary (matches the doc-string) (GH5636)
- The default for `.query()/.eval()` is now `engine=None`, which will use `numexpr` if it's installed; otherwise it will fallback to the `python` engine. This mimics the pre-0.18.1 behavior if `numexpr` is installed (and which, previously, if numexpr was not installed, `.query()/.eval()` would raise). (GH12749)
- `pd.show_versions()` now includes `pandas_datareader` version (GH12740)
- Provide a proper `__name__` and `__qualname__` attributes for generic functions (GH12021)
- `pd.concat(ignore_index=True)` now uses `RangeIndex` as default (GH12695)
- `pd.merge()` and `DataFrame.join()` will show a `UserWarning` when merging/joining a single- with a multi-leveled dataframe (GH9455, GH12219)
- Compat with `scipy` > 0.17 for deprecated `piecewise_polynomial` interpolation method; support for the replacement `from_derivatives` method (GH12887)

Deprecations

- The method name `Index.sym_diff()` is deprecated and can be replaced by `Index.symmetric_difference()` (GH12591)
- The method name `Categorical.sort()` is deprecated in favor of `Categorical.sort_values()` (GH12882)

Performance Improvements

- Improved speed of SAS reader (GH12656, GH12961)
- Performance improvements in `.groupby(..).cumcount()` (GH11039)
- Improved memory usage in `pd.read_csv()` when using `skiprows=an_integer` (GH13005)

- Improved performance of `DataFrame.to_sql` when checking case sensitivity for tables. Now only checks if table has been created correctly when table name is not lower case. (GH12876)
- Improved performance of `Period` construction and time series plotting (GH12903, GH11831).
- Improved performance of `.str.encode()` and `.str.decode()` methods (GH13008)
- Improved performance of `to_numeric` if input is numeric dtype (GH12777)
- Improved performance of sparse arithmetic with `IntIndex` (GH13036)

Bug Fixes

- `usecols` parameter in `pd.read_csv` is now respected even when the lines of a CSV file are not even (GH12203)
- Bug in `groupby.transform(..)` when `axis=1` is specified with a non-monotonic ordered index (GH12713)
- Bug in `Period` and `PeriodIndex` creation raises `KeyError` if `freq="Minute"` is specified. Note that "Minute" freq is deprecated in v0.17.0, and recommended to use `freq="T"` instead (GH11854)
- Bug in `.resample(...).count()` with a `PeriodIndex` always raising a `TypeError` (GH12774)
- Bug in `.resample(...)` with a `PeriodIndex` casting to a `DatetimeIndex` when empty (GH12868)
- Bug in `.resample(...)` with a `PeriodIndex` when resampling to an existing frequency (GH12770)
- Bug in printing data which contains `Period` with different `freq` raises `ValueError` (GH12615)
- Bug in `Series` construction with `Categorical` and `dtype='category'` is specified (GH12574)
- Bugs in concatenation with a coercible dtype was too aggressive, resulting in different dtypes in output formatting when an object was longer than `display.max_rows` (GH12411, GH12045, GH11594, GH10571, GH12211)
- Bug in `float_format` option with option not being validated as a callable. (GH12706)
- Bug in `GroupBy.filter` when `dropna=False` and no groups fulfilled the criteria (GH12768)
- Bug in `__name__` of `.cum*` functions (GH12021)
- Bug in `.astype()` of a `Float64Inde/Int64Index` to an `Int64Index` (GH12881)
- Bug in round tripping an integer based index in `.to_json()/.read_json()` when `orient='index'` (the default) (GH12866)
- Bug in plotting `Categorical` dtypes cause error when attempting stacked bar plot (GH13019)
- Compat with >= `numpy` 1.11 for `NaT` comparisons (GH12969)
- Bug in `.drop()` with a non-unique `MultiIndex`. (GH12701)
- Bug in `.concat` of datetime tz-aware and naive DataFrames (GH12467)
- Bug in correctly raising a `ValueError` in `.resample(..).fillna(..)` when passing a non-string (GH12952)
- Bug fixes in various encoding and header processing issues in `pd.read_sas()` (GH12659, GH12654, GH12647, GH12809)
- Bug in `pd.crosstab()` where would silently ignore `aggfunc` if `values=None` (GH12569).
- Potential segfault in `DataFrame.to_json` when serialising `datetime.time` (GH11473).
- Potential segfault in `DataFrame.to_json` when attempting to serialise 0d array (GH11299).

- Segfault in `to_json` when attempting to serialise a `DataFrame` or `Series` with non-ndarray values; now supports serialization of `category`, `sparse`, and `datetime64[ns, tz]` dtypes (GH10778).
- Bug in `DataFrame.to_json` with unsupported dtype not passed to default handler (GH12554).
- Bug in `.align` not returning the sub-class (GH12983)
- Bug in aligning a `Series` with a `DataFrame` (GH13037)
- Bug in `ABCPanel` in which `Panel4D` was not being considered as a valid instance of this generic type (GH12810)
- Bug in consistency of `.name` on `.groupby(..).apply(..)` cases (GH12363)
- Bug in `Timestamp.__repr__` that caused `pprint` to fail in nested structures (GH12622)
- Bug in `Timedelta.min` and `Timedelta.max`, the properties now report the true minimum/maximum `timedeltas` as recognized by pandas. See the *documentation*. (GH12727)
- Bug in `.quantile()` with interpolation may coerce to `float` unexpectedly (GH12772)
- Bug in `.quantile()` with empty `Series` may return scalar rather than empty `Series` (GH12772)
- Bug in `.loc` with out-of-bounds in a large indexer would raise `IndexError` rather than `KeyError` (GH12527)
- Bug in resampling when using a `TimedeltaIndex` and `.asfreq()`, would previously not include the final fencepost (GH12926)
- Bug in equality testing with a `Categorical` in a `DataFrame` (GH12564)
- Bug in `GroupBy.first()`, `.last()` returns incorrect row when `TimeGrouper` is used (GH7453)
- Bug in `pd.read_csv()` with the `c` engine when specifying `skiprows` with newlines in quoted items (GH10911, GH12775)
- Bug in `DataFrame` timezone lost when assigning tz-aware datetime `Series` with alignment (GH12981)
- Bug in `.value_counts()` when `normalize=True` and `dropna=True` where nulls still contributed to the normalized count (GH12558)
- Bug in `Series.value_counts()` loses name if its dtype is `category` (GH12835)
- Bug in `Series.value_counts()` loses timezone info (GH12835)
- Bug in `Series.value_counts(normalize=True)` with `Categorical` raises `UnboundLocalError` (GH12835)
- Bug in `Panel.fillna()` ignoring `inplace=True` (GH12633)
- Bug in `pd.read_csv()` when specifying `names`, `usecols`, and `parse_dates` simultaneously with the `c` engine (GH9755)
- Bug in `pd.read_csv()` when specifying `delim_whitespace=True` and `lineterminator` simultaneously with the `c` engine (GH12912)
- Bug in `Series.rename`, `DataFrame.rename` and `DataFrame.rename_axis` not treating `Series` as mappings to relabel (GH12623).
- Clean in `.rolling.min` and `.rolling.max` to enhance dtype handling (GH12373)
- Bug in `groupby` where complex types are coerced to float (GH12902)
- Bug in `Series.map` raises `TypeError` if its dtype is `category` or tz-aware `datetime` (GH12473)
- Bugs on 32bit platforms for some test comparisons (GH12972)
- Bug in index coercion when falling back from `RangeIndex` construction (GH12893)

- Better error message in window functions when invalid argument (e.g. a float window) is passed (GH12669)
- Bug in slicing subclassed DataFrame defined to return subclassed Series may return normal Series (GH11559)
- Bug in .str accessor methods may raise ValueError if input has name and the result is DataFrame or MultiIndex (GH12617)
- Bug in DataFrame.last_valid_index() and DataFrame.first_valid_index() on empty frames (GH12800)
- Bug in CategoricalIndex.get_loc returns different result from regular Index (GH12531)
- Bug in PeriodIndex.resample where name not propagated (GH12769)
- Bug in date_range closed keyword and timezones (GH12684).
- Bug in pd.concat raises AttributeError when input data contains tz-aware datetime and timedelta (GH12620)
- Bug in pd.concat did not handle empty Series properly (GH11082)
- Bug in .plot.bar alignment when width is specified with int (GH12979)
- Bug in fill_value is ignored if the argument to a binary operator is a constant (GH12723)
- Bug in pd.read_html() when using bs4 flavor and parsing table with a header and only one column (GH9178)
- Bug in .pivot_table when margins=True and dropna=True where nulls still contributed to margin count (GH12577)
- Bug in .pivot_table when dropna=False where table index/column names disappear (GH12133)
- Bug in pd.crosstab() when margins=True and dropna=False which raised (GH12642)
- Bug in Series.name when name attribute can be a hashable type (GH12610)
- Bug in .describe() resets categorical columns information (GH11558)
- Bug where loffset argument was not applied when calling resample().count() on a timeseries (GH12725)
- pd.read_excel() now accepts column names associated with keyword argument names (GH12870)
- Bug in pd.to_numeric() with Index returns np.ndarray, rather than Index (GH12777)
- Bug in pd.to_numeric() with datetime-like may raise TypeError (GH12777)
- Bug in pd.to_numeric() with scalar raises ValueError (GH12777)

Contributors

A total of 60 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Andrew Fiore-Gartland +
- Bastiaan +
- Benoît Vinot +
- Brandon Rhodes +
- DaCoEx +
- Drew Fustin +

- Ernesto Freitas +
- Filip Ter +
- Gregory Livschitz +
- Gábor Lipták
- Hassan Kibirige +
- Iblis Lin
- Israel Saeta Pérez +
- Jason Wolosonovich +
- Jeff Reback
- Joe Jevnik
- Joris Van den Bossche
- Joshua Storck +
- Ka Wo Chen
- Kerby Shedden
- Kieran O'Mahony
- Leif Walsh +
- Mahmoud Lababidi +
- Maoyuan Liu +
- Mark Roth +
- Matt Wittmann
- MaxU +
- Maximilian Roos
- Michael Droettboom +
- Nick Eubank
- Nicolas Bonnotte
- OXPHOS +
- Pauli Virtanen +
- Peter Waller +
- Pietro Battiston
- Prabhjot Singh +
- Robin Wilson
- Roger Thomas +
- Sebastian Bank
- Stephen Hoover
- Tim Hopper +
- Tom Augspurger

- WANG Aiyong
- Wes Turner
- Winand +
- Xbar +
- Yan Facai +
- adneu +
- ajenkins-cargometrics +
- behzad nouri
- chinskiy +
- gfyoung
- jeps-journal +
- jonaslb +
- kotrfa +
- nileracecrew +
- onesandzeroes
- rs2 +
- sinhrks
- tsdlovell +

8.7.2 v0.18.0 (March 13, 2016)

This is a major release from 0.17.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

> **Warning:** pandas >= 0.18.0 no longer supports compatibility with Python version 2.6 and 3.3 (GH7718, GH11273)

> **Warning:** `numexpr` version 2.4.4 will now show a warning and not be used as a computation back-end for pandas because of some buggy behavior. This does not affect other versions (>= 2.1 and >= 2.4.6). (GH12489)

Highlights include:

- Moving and expanding window functions are now methods on Series and DataFrame, similar to `.groupby`, see *here*.
- Adding support for a `RangeIndex` as a specialized form of the `Int64Index` for memory savings, see *here*.
- API breaking change to the `.resample` method to make it more `.groupby` like, see *here*.
- Removal of support for positional indexing with floats, which was deprecated since 0.14.0. This will now raise a `TypeError`, see *here*.
- The `.to_xarray()` function has been added for compatibility with the xarray package, see *here*.

- The `read_sas` function has been enhanced to read `sas7bdat` files, see *here*.
- Addition of the *.str.extractall() method*, and API changes to the *.str.extract() method* and *.str.cat() method*.
- `pd.test()` top-level nose test runner is available (GH4327).

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

What's new in v0.18.0

- *Contributors*

New features

Window functions are now methods

Window functions have been refactored to be methods on `Series/DataFrame` objects, rather than top-level functions, which are now deprecated. This allows these window-type functions, to have a similar API to that of `.groupby`. See the full documentation *here* (GH11603, GH12373)

```
In [1]: np.random.seed(1234)

In [2]: df = pd.DataFrame({'A': range(10), 'B': np.random.randn(10)})

In [3]: df
Out[3]:
   A         B
0  0  0.471435
1  1 -1.190976
2  2  1.432707
3  3 -0.312652
4  4 -0.720589
5  5  0.887163
6  6  0.859588
7  7 -0.636524
8  8  0.015696
9  9 -2.242685

[10 rows x 2 columns]
```

Previous Behavior:

```
In [8]: pd.rolling_mean(df, window=3)
        FutureWarning: pd.rolling_mean is deprecated for DataFrame and will be␣
↪removed in a future version, replace with
                       DataFrame.rolling(window=3,center=False).mean()
Out[8]:
    A         B
0 NaN       NaN
1 NaN       NaN
2   1  0.237722
3   2 -0.023640
4   3  0.133155
5   4 -0.048693
6   5  0.342054
7   6  0.370076
8   7  0.079587
9   8 -0.954504
```

New Behavior:

```
In [4]: r = df.rolling(window=3)
```

These show a descriptive repr

```
In [5]: r
Out[5]: Rolling [window=3,center=False,axis=0]
```

with tab-completion of available methods and properties.

```
In [9]: r.<TAB>  # noqa E225, E999
r.A             r.agg           r.apply         r.count         r.exclusions   r.max           r.
↪median         r.name          r.skew          r.sum
r.B             r.aggregate     r.corr          r.cov           r.kurt         r.mean          r.
↪min            r.quantile      r.std           r.var
```

The methods operate on the `Rolling` object itself

```
In [6]: r.mean()
Out[6]:
     A         B
0  NaN       NaN
1  NaN       NaN
2  1.0  0.237722
3  2.0 -0.023640
4  3.0  0.133155
5  4.0 -0.048693
6  5.0  0.342054
7  6.0  0.370076
8  7.0  0.079587
9  8.0 -0.954504

[10 rows x 2 columns]
```

They provide getitem accessors

```
In [7]: r['A'].mean()
Out[7]:
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
5    4.0
6    5.0
7    6.0
8    7.0
9    8.0
Name: A, Length: 10, dtype: float64
```

And multiple aggregations

```
In [8]: r.agg({'A': ['mean', 'std'],
   ...:        'B': ['mean', 'std']})
   ...:
Out[8]:
     A              B
  mean  std      mean       std
0  NaN  NaN       NaN       NaN
1  NaN  NaN       NaN       NaN
2  1.0  1.0  0.237722  1.327364
3  2.0  1.0 -0.023640  1.335505
```

(continues on next page)

(continued from previous page)

```
4  3.0  1.0  0.133155  1.143778
5  4.0  1.0 -0.048693  0.835747
6  5.0  1.0  0.342054  0.920379
7  6.0  1.0  0.370076  0.871850
8  7.0  1.0  0.079587  0.750099
9  8.0  1.0 -0.954504  1.162285

[10 rows x 4 columns]
```

Changes to rename

`Series.rename` and `NDFrame.rename_axis` can now take a scalar or list-like argument for altering the Series or axis *name*, in addition to their old behaviors of altering labels. (GH9494, GH11965)

```
In [9]: s = pd.Series(np.random.randn(5))

In [10]: s.rename('newname')
Out[10]:
0    1.150036
1    0.991946
2    0.953324
3   -2.021255
4   -0.334077
Name: newname, Length: 5, dtype: float64
```

```
In [11]: df = pd.DataFrame(np.random.randn(5, 2))

In [12]: (df.rename_axis("indexname")
   ....:    .rename_axis("columns_name", axis="columns"))
   ....:
Out[12]:
columns_name         0         1
indexname
0             0.002118  0.405453
1             0.289092  1.321158
2            -1.546906 -0.202646
3            -0.655969  0.193421
4             0.553439  1.318152

[5 rows x 2 columns]
```

The new functionality works well in method chains. Previously these methods only accepted functions or dicts mapping a *label* to a new label. This continues to work as before for function or dict-like values.

Range Index

A `RangeIndex` has been added to the `Int64Index` sub-classes to support a memory saving alternative for common use cases. This has a similar implementation to the python `range` object (`xrange` in python 2), in that it only stores the start, stop, and step values for the index. It will transparently interact with the user API, converting to `Int64Index` if needed.

This will now be the default constructed index for `NDFrame` objects, rather than previous an `Int64Index`. (GH939, GH12070, GH12071, GH12109, GH12888)

Previous Behavior:

```
In [3]: s = pd.Series(range(1000))

In [4]: s.index
Out[4]:
Int64Index([  0,    1,    2,    3,    4,    5,    6,    7,    8,    9,
            ...
            990, 991, 992, 993, 994, 995, 996, 997, 998, 999], dtype='int64',␣
↪length=1000)

In [6]: s.index.nbytes
Out[6]: 8000
```

New Behavior:

```
In [13]: s = pd.Series(range(1000))

In [14]: s.index
Out[14]: RangeIndex(start=0, stop=1000, step=1)

In [15]: s.index.nbytes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]: 80
```

Changes to str.extract

The *.str.extract* method takes a regular expression with capture groups, finds the first match in each subject string, and returns the contents of the capture groups (GH11386).

In v0.18.0, the `expand` argument was added to `extract`.

- `expand=False`: it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0).
- `expand=True`: it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user.

Currently the default is `expand=None` which gives a `FutureWarning` and uses `expand=False`. To avoid this warning, please explicitly specify `expand`.

```
In [1]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=None)
FutureWarning: currently extract(expand=None) means expand=False (return Index/Series/
↪DataFrame)
but in a future version of pandas this will be changed to expand=True (return␣
↪DataFrame)

Out[1]:
0      1
1      2
2    NaN
dtype: object
```

Extracting a regular expression with one group returns a Series if `expand=False`.

```
In [16]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=False)
Out[16]:
0      1
```

(continues on next page)

(continued from previous page)

```
1      2
2    NaN
Length: 3, dtype: object
```

It returns a DataFrame with one column if expand=True.

```
In [17]: pd.Series(['a1', 'b2', 'c3']).str.extract(r'[ab](\d)', expand=True)
Out[17]:
     0
0    1
1    2
2  NaN

[3 rows x 1 columns]
```

Calling on an Index with a regex with exactly one capture group returns an Index if expand=False.

```
In [18]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])

In [19]: s.index
Out[19]: Index(['A11', 'B22', 'C33'], dtype='object')

In [20]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]: Index(['A', 'B', 'C'],␣
↪dtype='object', name='letter')
```

It returns a DataFrame with one column if expand=True.

```
In [21]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[21]:
  letter
0      A
1      B
2      C

[3 rows x 1 columns]
```

Calling on an Index with a regex with more than one capture group raises ValueError if expand=False.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

It returns a DataFrame if expand=True.

```
In [22]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[22]:
  letter   1
0      A  11
1      B  22
2      C  33

[3 rows x 2 columns]
```

In summary, extract(expand=True) always returns a DataFrame with a row for every subject string, and a column for every capture group.

Addition of str.extractall

The *.str.extractall* method was added (GH11386). Unlike `extract`, which returns only the first match.

```
In [23]: s = pd.Series(["a1a2", "b1", "c1"], ["A", "B", "C"])

In [24]: s
Out[24]:
A    a1a2
B      b1
C      c1
Length: 3, dtype: object

In [25]: s.str.extract(r"(?P<letter>[ab])(?P<digit>\d)", expand=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]:
  letter digit
A      a     1
B      b     1
C    NaN   NaN

[3 rows x 2 columns]
```

The `extractall` method returns all matches.

```
In [26]: s.str.extractall(r"(?P<letter>[ab])(?P<digit>\d)")
Out[26]:
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1

[3 rows x 2 columns]
```

Changes to str.cat

The method `.str.cat()` concatenates the members of a `Series`. Before, if `NaN` values were present in the Series, calling `.str.cat()` on it would return `NaN`, unlike the rest of the `Series.str.*` API. This behavior has been amended to ignore `NaN` values by default. (GH11435).

A new, friendlier `ValueError` is added to protect against the mistake of supplying the `sep` as an arg, rather than as a kwarg. (GH11334).

```
In [27]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ')
Out[27]: 'a b c'

In [28]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ', na_rep='?')
\\\\\\\\\\\\\\\\Out[28]: 'a b ? c'
```

```
In [2]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(' ')
ValueError: Did you mean to supply a `sep` keyword?
```

Datetimelike rounding

`DatetimeIndex`, `Timestamp`, `TimedeltaIndex`, `Timedelta` have gained the `.round()`, `.floor()` and `.ceil()` method for datetimelike rounding, flooring and ceiling. (GH4314, GH11963)

Naive datetimes

```
In [29]: dr = pd.date_range('20130101 09:12:56.1234', periods=3)

In [30]: dr
Out[30]:
DatetimeIndex(['2013-01-01 09:12:56.123400', '2013-01-02 09:12:56.123400',
               '2013-01-03 09:12:56.123400'],
              dtype='datetime64[ns]', freq='D')

In [31]: dr.round('s')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2013-01-01 09:12:56', '2013-01-02 09:12:56',
               '2013-01-03 09:12:56'],
              dtype='datetime64[ns]', freq=None)

# Timestamp scalar
In [32]: dr[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-01-01 09:12:56.123400', freq='D')

In [33]: dr[0].round('10s')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2013-01-01 09:13:00')
```

Tz-aware are rounded, floored and ceiled in local times

```
In [34]: dr = dr.tz_localize('US/Eastern')

In [35]: dr
Out[35]:
DatetimeIndex(['2013-01-01 09:12:56.123400-05:00',
               '2013-01-02 09:12:56.123400-05:00',
               '2013-01-03 09:12:56.123400-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')

In [36]: dr.round('s')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2013-01-01 09:12:56-05:00', '2013-01-02 09:12:56-05:00',
               '2013-01-03 09:12:56-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

Timedeltas

```
In [37]: t = pd.timedelta_range('1 days 2 hr 13 min 45 us', periods=3, freq='d')

In [38]: t
Out[38]:
TimedeltaIndex(['1 days 02:13:00.000045', '2 days 02:13:00.000045',
                '3 days 02:13:00.000045'],
               dtype='timedelta64[ns]', freq='D')
```

(continues on next page)

(continued from previous page)

```
In [39]: t.round('10min')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['1 days 02:10:00', '2 days 02:10:00', '3 days 02:10:00'], dtype=
↪'timedelta64[ns]', freq=None)

# Timedelta scalar
In [40]: t[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 02:13:00.000045')

In [41]: t[0].round('2h')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timedelta('1 days 02:00:00')
```

In addition, `.round()`, `.floor()` and `.ceil()` will be available through the `.dt` accessor of `Series`.

```
In [42]: s = pd.Series(dr)

In [43]: s
Out[43]:
0   2013-01-01 09:12:56.123400-05:00
1   2013-01-02 09:12:56.123400-05:00
2   2013-01-03 09:12:56.123400-05:00
Length: 3, dtype: datetime64[ns, US/Eastern]

In [44]: s.dt.round('D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2013-01-01 00:00:00-05:00
1   2013-01-02 00:00:00-05:00
2   2013-01-03 00:00:00-05:00
Length: 3, dtype: datetime64[ns, US/Eastern]
```

Formatting of Integers in FloatIndex

Integers in `FloatIndex`, e.g. 1., are now formatted with a decimal point and a `0` digit, e.g. `1.0` (GH11713) This change not only affects the display to the console, but also the output of IO methods like `.to_csv` or `.to_html`.

Previous Behavior:

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

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

In [4]: s.index
Out[4]: Float64Index([0.0, 1.0, 2.0], dtype='float64')

In [5]: print(s.to_csv(path=None))
0,1
```

(continues on next page)

(continued from previous page)

```
1,2
2,3
```

New Behavior:

```
In [45]: s = pd.Series([1, 2, 3], index=np.arange(3.))

In [46]: s
Out[46]:
0.0    1
1.0    2
2.0    3
Length: 3, dtype: int64

In [47]: s.index
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]: Float64Index([0.
↪0, 1.0, 2.0], dtype='float64')

In [48]: print(s.to_csv(path_or_buf=None, header=False))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪0,1
1.0,2
2.0,3
```

Changes to dtype assignment behaviors

When a DataFrame's slice is updated with a new slice of the same dtype, the dtype of the DataFrame will now remain the same. (GH10503)

Previous Behavior:

```
In [5]: df = pd.DataFrame({'a': [0, 1, 1],
                           'b': pd.Series([100, 200, 300], dtype='uint32')})

In [7]: df.dtypes
Out[7]:
a      int64
b     uint32
dtype: object

In [8]: ix = df['a'] == 1

In [9]: df.loc[ix, 'b'] = df.loc[ix, 'b']

In [11]: df.dtypes
Out[11]:
a    int64
b    int64
dtype: object
```

New Behavior:

```
In [49]: df = pd.DataFrame({'a': [0, 1, 1],
   ....:                    'b': pd.Series([100, 200, 300], dtype='uint32')})
   ....:
```

(continues on next page)

(continued from previous page)

```
In [50]: df.dtypes
Out[50]:
a      int64
b    uint32
Length: 2, dtype: object

In [51]: ix = df['a'] == 1

In [52]: df.loc[ix, 'b'] = df.loc[ix, 'b']

In [53]: df.dtypes
Out[53]:
a      int64
b    uint32
Length: 2, dtype: object
```

When a DataFrame's integer slice is partially updated with a new slice of floats that could potentially be down-casted to integer without losing precision, the dtype of the slice will be set to float instead of integer.

Previous Behavior:

```
In [4]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
                          columns=list('abc'),
                          index=[[4,4,8], [8,10,12]])

In [5]: df
Out[5]:
      a  b  c
4 8   1  2  3
  10  4  5  6
8 12  7  8  9

In [7]: df.ix[4, 'c'] = np.array([0., 1.])

In [8]: df
Out[8]:
      a  b  c
4 8   1  2  0
  10  4  5  1
8 12  7  8  9
```

New Behavior:

```
In [54]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
   ....:                   columns=list('abc'),
   ....:                   index=[[4,4,8], [8,10,12]])
   ....:

In [55]: df
Out[55]:
      a  b  c
4 8   1  2  3
  10  4  5  6
8 12  7  8  9

[3 rows x 3 columns]
```

(continues on next page)

(continued from previous page)

```
In [56]: df.loc[4, 'c'] = np.array([0., 1.])

In [57]: df
Out[57]:
      a  b    c
4 8   1  2  0.0
  10  4  5  1.0
8 12  7  8  9.0

[3 rows x 3 columns]
```

to_xarray

In a future version of pandas, we will be deprecating `Panel` and other > 2 ndim objects. In order to provide for continuity, all `NDFrame` objects have gained the `.to_xarray()` method in order to convert to `xarray` objects, which has a pandas-like interface for > 2 ndim. (GH11972)

See the xarray full-documentation here.

```
In [1]: p = Panel(np.arange(2*3*4).reshape(2,3,4))

In [2]: p.to_xarray()
Out[2]:
<xarray.DataArray (items: 2, major_axis: 3, minor_axis: 4)>
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
Coordinates:
  * items        (items) int64 0 1
  * major_axis   (major_axis) int64 0 1 2
  * minor_axis   (minor_axis) int64 0 1 2 3
```

Latex Representation

`DataFrame` has gained a `._repr_latex_()` method in order to allow for conversion to latex in a ipython/jupyter notebook using nbconvert. (GH11778)

Note that this must be activated by setting the option `pd.display.latex.repr=True` (GH12182)

For example, if you have a jupyter notebook you plan to convert to latex using nbconvert, place the statement `pd.display.latex.repr=True` in the first cell to have the contained DataFrame output also stored as latex.

The options `display.latex.escape` and `display.latex.longtable` have also been added to the configuration and are used automatically by the `to_latex` method. See the *available options docs* for more info.

`pd.read_sas()` changes

`read_sas` has gained the ability to read SAS7BDAT files, including compressed files. The files can be read in entirety, or incrementally. For full details see *here*. (GH4052)

Other enhancements

- Handle truncated floats in SAS xport files (GH11713)
- Added option to hide index in `Series.to_string` (GH11729)
- `read_excel` now supports s3 urls of the format `s3://bucketname/filename` (GH11447)
- add support for `AWS_S3_HOST` env variable when reading from s3 (GH12198)
- A simple version of `Panel.round()` is now implemented (GH11763)
- For Python 3.x, `round(DataFrame)`, `round(Series)`, `round(Panel)` will work (GH11763)
- `sys.getsizeof(obj)` returns the memory usage of a pandas object, including the values it contains (GH11597)
- `Series` gained an `is_unique` attribute (GH11946)
- `DataFrame.quantile` and `Series.quantile` now accept `interpolation` keyword (GH10174).
- Added `DataFrame.style.format` for more flexible formatting of cell values (GH11692)
- `DataFrame.select_dtypes` now allows the `np.float16` type code (GH11990)
- `pivot_table()` now accepts most iterables for the `values` parameter (GH12017)
- Added Google `BigQuery` service account authentication support, which enables authentication on remote servers. (GH11881, GH12572). For further details see here
- `HDFStore` is now iterable: `for k in store` is equivalent to `for k in store.keys()` (GH12221).
- Add missing methods/fields to `.dt` for `Period` (GH8848)
- The entire code base has been `PEP`-ified (GH12096)

Backwards incompatible API changes

- the leading white spaces have been removed from the output of `.to_string(index=False)` method (GH11833)
- the `out` parameter has been removed from the `Series.round()` method. (GH11763)
- `DataFrame.round()` leaves non-numeric columns unchanged in its return, rather than raises. (GH11885)
- `DataFrame.head(0)` and `DataFrame.tail(0)` return empty frames, rather than `self`. (GH11937)
- `Series.head(0)` and `Series.tail(0)` return empty series, rather than `self`. (GH11937)
- `to_msgpack` and `read_msgpack` encoding now defaults to `'utf-8'`. (GH12170)
- the order of keyword arguments to text file parsing functions (`.read_csv()`, `.read_table()`, `.read_fwf()`) changed to group related arguments. (GH11555)
- `NaTType.isoformat` now returns the string `'NaT` to allow the result to be passed to the constructor of `Timestamp`. (GH12300)

NaT and Timedelta operations

NaT and Timedelta have expanded arithmetic operations, which are extended to Series arithmetic where applicable. Operations defined for datetime64[ns] or timedelta64[ns] are now also defined for NaT (GH11564).

NaT now supports arithmetic operations with integers and floats.

```
In [58]: pd.NaT * 1
Out[58]: NaT

In [59]: pd.NaT * 1.5
\\\\\\\\\\\\\Out[59]: NaT

In [60]: pd.NaT / 2
\\\\\\\\\\\\\\\\\\\\\\\\\\Out[60]: NaT

In [61]: pd.NaT * np.nan
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[61]: NaT
```

NaT defines more arithmetic operations with datetime64[ns] and timedelta64[ns].

```
In [62]: pd.NaT / pd.NaT
Out[62]: nan

In [63]: pd.Timedelta('1s') / pd.NaT
\\\\\\\\\\\\\Out[63]: nan
```

NaT may represent either a datetime64[ns] null or a timedelta64[ns] null. Given the ambiguity, it is treated as a timedelta64[ns], which allows more operations to succeed.

```
In [64]: pd.NaT + pd.NaT
Out[64]: NaT

# same as
In [65]: pd.Timedelta('1s') + pd.Timedelta('1s')
\\\\\\\\\\\\\Out[65]: Timedelta('0 days 00:00:02')
```

as opposed to

```
In [3]: pd.Timestamp('19900315') + pd.Timestamp('19900315')
TypeError: unsupported operand type(s) for +: 'Timestamp' and 'Timestamp'
```

However, when wrapped in a Series whose dtype is datetime64[ns] or timedelta64[ns], the dtype information is respected.

```
In [1]: pd.Series([pd.NaT], dtype='<M8[ns]') + pd.Series([pd.NaT], dtype='<M8[ns]')
TypeError: can only operate on a datetimes for subtraction,
           but the operator [__add__] was passed
```

```
In [66]: pd.Series([pd.NaT], dtype='<m8[ns]') + pd.Series([pd.NaT], dtype='<m8[ns]')
Out[66]:
0   NaT
Length: 1, dtype: timedelta64[ns]
```

Timedelta division by floats now works.

```
In [67]: pd.Timedelta('1s') / 2.0
Out[67]: Timedelta('0 days 00:00:00.500000')
```

Subtraction by `Timedelta` in a `Series` by a `Timestamp` works (GH11925)

```
In [68]: ser = pd.Series(pd.timedelta_range('1 day', periods=3))

In [69]: ser
Out[69]:
0   1 days
1   2 days
2   3 days
Length: 3, dtype: timedelta64[ns]

In [70]: pd.Timestamp('2012-01-01') - ser
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[70]:
0   2011-12-31
1   2011-12-30
2   2011-12-29
Length: 3, dtype: datetime64[ns]
```

`NaT.isoformat()` now returns `'NaT'`. This change allows allows `pd.Timestamp` to rehydrate any timestamp like object from its isoformat (GH12300).

Changes to msgpack

Forward incompatible changes in `msgpack` writing format were made over 0.17.0 and 0.18.0; older versions of pandas cannot read files packed by newer versions (GH12129, GH10527)

Bugs in `to_msgpack` and `read_msgpack` introduced in 0.17.0 and fixed in 0.18.0, caused files packed in Python 2 unreadable by Python 3 (GH12142). The following table describes the backward and forward compat of msgpacks.

Warning:

Packed with	Can be unpacked with
pre-0.17 / Python 2	any
pre-0.17 / Python 3	any
0.17 / Python 2	• ==0.17 / Python 2 • >=0.18 / any Python
0.17 / Python 3	>=0.18 / any Python
0.18	>= 0.18

0.18.0 is backward-compatible for reading files packed by older versions, except for files packed with 0.17 in Python 2, in which case only they can only be unpacked in Python 2.

Signature change for .rank

`Series.rank` and `DataFrame.rank` now have the same signature (GH11759)

Previous signature

```
In [3]: pd.Series([0,1]).rank(method='average', na_option='keep',
                              ascending=True, pct=False)
Out[3]:
0    1
1    2
dtype: float64

In [4]: pd.DataFrame([0,1]).rank(axis=0, numeric_only=None,
                                 method='average', na_option='keep',
                                 ascending=True, pct=False)
Out[4]:
   0
0  1
1  2
```

New signature

```
In [71]: pd.Series([0,1]).rank(axis=0, method='average', numeric_only=None,
   ....:                       na_option='keep', ascending=True, pct=False)
   ....:
Out[71]:
0    1.0
1    2.0
Length: 2, dtype: float64

In [72]: pd.DataFrame([0,1]).rank(axis=0, method='average', numeric_only=None,
   ....:                          na_option='keep', ascending=True, pct=False)
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[72]:
     0
0  1.0
1  2.0

[2 rows x 1 columns]
```

Bug in QuarterBegin with n=0

In previous versions, the behavior of the QuarterBegin offset was inconsistent depending on the date when the `n` parameter was 0. (GH11406)

The general semantics of anchored offsets for `n=0` is to not move the date when it is an anchor point (e.g., a quarter start date), and otherwise roll forward to the next anchor point.

```
In [73]: d = pd.Timestamp('2014-02-01')

In [74]: d
Out[74]: Timestamp('2014-02-01 00:00:00')

In [75]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[75]: Timestamp('2014-02-01 00:00:00')

In [76]: d + pd.offsets.QuarterBegin(n=0, startingMonth=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[76]:
→Timestamp('2014-04-01 00:00:00')
```

For the `QuarterBegin` offset in previous versions, the date would be rolled *backwards* if date was in the same month as the quarter start date.

```
In [3]: d = pd.Timestamp('2014-02-15')

In [4]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[4]: Timestamp('2014-02-01 00:00:00')
```

This behavior has been corrected in version 0.18.0, which is consistent with other anchored offsets like `MonthBegin` and `YearBegin`.

```
In [77]: d = pd.Timestamp('2014-02-15')

In [78]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[78]: Timestamp('2014-05-01 00:00:00')
```

Resample API

Like the change in the window functions API *above*, `.resample(...)` is changing to have a more groupby-like API. (GH11732, GH12702, GH12202, GH12332, GH12334, GH12348, GH12448).

```
In [79]: np.random.seed(1234)

In [80]: df = pd.DataFrame(np.random.rand(10,4),
   ....:                   columns=list('ABCD'),
   ....:                   index=pd.date_range('2010-01-01 09:00:00',
   ....:                                       periods=10, freq='s'))
   ....:

In [81]: df
Out[81]:
                            A         B         C         D
2010-01-01 09:00:00  0.191519  0.622109  0.437728  0.785359
2010-01-01 09:00:01  0.779976  0.272593  0.276464  0.801872
2010-01-01 09:00:02  0.958139  0.875933  0.357817  0.500995
2010-01-01 09:00:03  0.683463  0.712702  0.370251  0.561196
2010-01-01 09:00:04  0.503083  0.013768  0.772827  0.882641
2010-01-01 09:00:05  0.364886  0.615396  0.075381  0.368824
2010-01-01 09:00:06  0.933140  0.651378  0.397203  0.788730
2010-01-01 09:00:07  0.316836  0.568099  0.869127  0.436173
2010-01-01 09:00:08  0.802148  0.143767  0.704261  0.704581
2010-01-01 09:00:09  0.218792  0.924868  0.442141  0.909316

[10 rows x 4 columns]
```

Previous API:

You would write a resampling operation that immediately evaluates. If a `how` parameter was not provided, it would default to `how='mean'`.

```
In [6]: df.resample('2s')
Out[6]:
                         A         B         C         D
2010-01-01 09:00:00  0.485748  0.447351  0.357096  0.793615
2010-01-01 09:00:02  0.820801  0.794317  0.364034  0.531096
2010-01-01 09:00:04  0.433985  0.314582  0.424104  0.625733
2010-01-01 09:00:06  0.624988  0.609738  0.633165  0.612452
2010-01-01 09:00:08  0.510470  0.534317  0.573201  0.806949
```

You could also specify a how directly

```
In [7]: df.resample('2s', how='sum')
Out[7]:
                            A         B         C         D
2010-01-01 09:00:00  0.971495  0.894701  0.714192  1.587231
2010-01-01 09:00:02  1.641602  1.588635  0.728068  1.062191
2010-01-01 09:00:04  0.867969  0.629165  0.848208  1.251465
2010-01-01 09:00:06  1.249976  1.219477  1.266330  1.224904
2010-01-01 09:00:08  1.020940  1.068634  1.146402  1.613897
```

New API:

Now, you can write .resample(..) as a 2-stage operation like .groupby(...), which yields a Resampler.

```
In [82]: r = df.resample('2s')

In [83]: r
Out[83]: DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left, label=left,␣
↪convention=start, base=0]
```

Downsampling

You can then use this object to perform operations. These are downsampling operations (going from a higher frequency to a lower one).

```
In [84]: r.mean()
Out[84]:
                            A         B         C         D
2010-01-01 09:00:00  0.485748  0.447351  0.357096  0.793615
2010-01-01 09:00:02  0.820801  0.794317  0.364034  0.531096
2010-01-01 09:00:04  0.433985  0.314582  0.424104  0.625733
2010-01-01 09:00:06  0.624988  0.609738  0.633165  0.612452
2010-01-01 09:00:08  0.510470  0.534317  0.573201  0.806949

[5 rows x 4 columns]
```

```
In [85]: r.sum()
Out[85]:
                            A         B         C         D
2010-01-01 09:00:00  0.971495  0.894701  0.714192  1.587231
2010-01-01 09:00:02  1.641602  1.588635  0.728068  1.062191
2010-01-01 09:00:04  0.867969  0.629165  0.848208  1.251465
2010-01-01 09:00:06  1.249976  1.219477  1.266330  1.224904
2010-01-01 09:00:08  1.020940  1.068634  1.146402  1.613897

[5 rows x 4 columns]
```

Furthermore, resample now supports getitem operations to perform the resample on specific columns.

```
In [86]: r[['A','C']].mean()
Out[86]:
                            A         C
2010-01-01 09:00:00  0.485748  0.357096
2010-01-01 09:00:02  0.820801  0.364034
2010-01-01 09:00:04  0.433985  0.424104
```

(continues on next page)

(continued from previous page)

```
2010-01-01 09:00:06  0.624988  0.633165
2010-01-01 09:00:08  0.510470  0.573201

[5 rows x 2 columns]
```

and `.aggregate` type operations.

```
In [87]: r.agg({'A' : 'mean', 'B' : 'sum'})
Out[87]:
                            A         B
2010-01-01 09:00:00  0.485748  0.894701
2010-01-01 09:00:02  0.820801  1.588635
2010-01-01 09:00:04  0.433985  0.629165
2010-01-01 09:00:06  0.624988  1.219477
2010-01-01 09:00:08  0.510470  1.068634

[5 rows x 2 columns]
```

These accessors can of course, be combined

```
In [88]: r[['A','B']].agg(['mean','sum'])
Out[88]:
                            A                   B
                         mean       sum      mean       sum
2010-01-01 09:00:00  0.485748  0.971495  0.447351  0.894701
2010-01-01 09:00:02  0.820801  1.641602  0.794317  1.588635
2010-01-01 09:00:04  0.433985  0.867969  0.314582  0.629165
2010-01-01 09:00:06  0.624988  1.249976  0.609738  1.219477
2010-01-01 09:00:08  0.510470  1.020940  0.534317  1.068634

[5 rows x 4 columns]
```

Upsampling

Upsampling operations take you from a lower frequency to a higher frequency. These are now performed with the `Resampler` objects with `backfill()`, `ffill()`, `fillna()` and `asfreq()` methods.

```
In [89]: s = pd.Series(np.arange(5, dtype='int64'),
   ....:               index=pd.date_range('2010-01-01', periods=5, freq='Q'))
   ....:

In [90]: s
Out[90]:
2010-03-31    0
2010-06-30    1
2010-09-30    2
2010-12-31    3
2011-03-31    4
Freq: Q-DEC, Length: 5, dtype: int64
```

Previously

```
In [6]: s.resample('M', fill_method='ffill')
Out[6]:
2010-03-31    0
```

(continues on next page)

(continued from previous page)

```
2010-04-30    0
2010-05-31    0
2010-06-30    1
2010-07-31    1
2010-08-31    1
2010-09-30    2
2010-10-31    2
2010-11-30    2
2010-12-31    3
2011-01-31    3
2011-02-28    3
2011-03-31    4
Freq: M, dtype: int64
```

New API

```
In [91]: s.resample('M').ffill()
Out[91]:
2010-03-31    0
2010-04-30    0
2010-05-31    0
2010-06-30    1
2010-07-31    1
2010-08-31    1
2010-09-30    2
2010-10-31    2
2010-11-30    2
2010-12-31    3
2011-01-31    3
2011-02-28    3
2011-03-31    4
Freq: M, Length: 13, dtype: int64
```

Note: In the new API, you can either downsample OR upsample. The prior implementation would allow you to pass an aggregator function (like `mean`) even though you were upsampling, providing a bit of confusion.

Previous API will work but with deprecations

Warning: This new API for resample includes some internal changes for the prior-to-0.18.0 API, to work with a deprecation warning in most cases, as the resample operation returns a deferred object. We can intercept operations and just do what the (pre 0.18.0) API did (with a warning). Here is a typical use case:

```
In [4]: r = df.resample('2s')

In [6]: r*10
pandas/tseries/resample.py:80: FutureWarning: .resample() is now a deferred
↪operation
use .resample(...).mean() instead of .resample(...)

Out[6]:
                          A         B         C         D
2010-01-01 09:00:00  4.857476  4.473507  3.570960  7.936154
2010-01-01 09:00:02  8.208011  7.943173  3.640340  5.310957
2010-01-01 09:00:04  4.339846  3.145823  4.241039  6.257326
2010-01-01 09:00:06  6.249881  6.097384  6.331650  6.124518
2010-01-01 09:00:08  5.104699  5.343172  5.732009  8.069486
```

However, getting and assignment operations directly on a `Resampler` will raise a `ValueError`:

```
In [7]: r.iloc[0] = 5
ValueError: .resample() is now a deferred operation
use .resample(...).mean() instead of .resample(...)
```

There is a situation where the new API can not perform all the operations when using original code. This code is intending to resample every 2s, take the `mean` AND then take the `min` of those results.

```
In [4]: df.resample('2s').min()
Out[4]:
A    0.433985
B    0.314582
C    0.357096
D    0.531096
dtype: float64
```

The new API will:

```
In [92]: df.resample('2s').min()
Out[92]:
                            A         B         C         D
2010-01-01 09:00:00  0.191519  0.272593  0.276464  0.785359
2010-01-01 09:00:02  0.683463  0.712702  0.357817  0.500995
2010-01-01 09:00:04  0.364886  0.013768  0.075381  0.368824
2010-01-01 09:00:06  0.316836  0.568099  0.397203  0.436173
2010-01-01 09:00:08  0.218792  0.143767  0.442141  0.704581

[5 rows x 4 columns]
```

The good news is the return dimensions will differ between the new API and the old API, so this should loudly raise an exception.

To replicate the original operation

```
In [93]: df.resample('2s').mean().min()
Out[93]:
A    0.433985
B    0.314582
C    0.357096
D    0.531096
Length: 4, dtype: float64
```

Changes to eval

In prior versions, new columns assignments in an `eval` expression resulted in an inplace change to the `DataFrame`. (GH9297, GH8664, GH10486)

```
In [94]: df = pd.DataFrame({'a': np.linspace(0, 10, 5), 'b': range(5)})

In [95]: df
Out[95]:
      a  b
0   0.0  0
1   2.5  1
2   5.0  2
3   7.5  3
```

(continues on next page)

(continued from previous page)

```
4  10.0  4

[5 rows x 2 columns]
```

```
In [12]: df.eval('c = a + b')
FutureWarning: eval expressions containing an assignment currentlydefault to␣
↪operating inplace.
This will change in a future version of pandas, use inplace=True to avoid this␣
↪warning.

In [13]: df
Out[13]:
      a  b     c
0   0.0  0   0.0
1   2.5  1   3.5
2   5.0  2   7.0
3   7.5  3  10.5
4  10.0  4  14.0
```

In version 0.18.0, a new `inplace` keyword was added to choose whether the assignment should be done inplace or return a copy.

```
In [96]: df
Out[96]:
      a  b     c
0   0.0  0   0.0
1   2.5  1   3.5
2   5.0  2   7.0
3   7.5  3  10.5
4  10.0  4  14.0

[5 rows x 3 columns]

In [97]: df.eval('d = c - b', inplace=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      a  b     c     d
0   0.0  0   0.0   0.0
1   2.5  1   3.5   2.5
2   5.0  2   7.0   5.0
3   7.5  3  10.5   7.5
4  10.0  4  14.0  10.0

[5 rows x 4 columns]

In [98]: df
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
      a  b     c
0   0.0  0   0.0
1   2.5  1   3.5
2   5.0  2   7.0
3   7.5  3  10.5
4  10.0  4  14.0

[5 rows x 3 columns]
```

(continues on next page)

(continued from previous page)

```
In [99]: df.eval('d = c - b', inplace=True)

In [100]: df
Out[100]:
      a  b     c     d
0   0.0  0   0.0   0.0
1   2.5  1   3.5   2.5
2   5.0  2   7.0   5.0
3   7.5  3  10.5   7.5
4  10.0  4  14.0  10.0

[5 rows x 4 columns]
```

> **Warning:** For backwards compatibility, `inplace` defaults to `True` if not specified. This will change in a future version of pandas. If your code depends on an inplace assignment you should update to explicitly set `inplace=True`

The `inplace` keyword parameter was also added the `query` method.

```
In [101]: df.query('a > 5')
Out[101]:
      a  b     c     d
3   7.5  3  10.5   7.5
4  10.0  4  14.0  10.0

[2 rows x 4 columns]

In [102]: df.query('a > 5', inplace=True)

In [103]: df
Out[103]:
      a  b     c     d
3   7.5  3  10.5   7.5
4  10.0  4  14.0  10.0

[2 rows x 4 columns]
```

> **Warning:** Note that the default value for `inplace` in a `query` is `False`, which is consistent with prior versions.

`eval` has also been updated to allow multi-line expressions for multiple assignments. These expressions will be evaluated one at a time in order. Only assignments are valid for multi-line expressions.

```
In [104]: df
Out[104]:
      a  b     c     d
3   7.5  3  10.5   7.5
4  10.0  4  14.0  10.0

[2 rows x 4 columns]

In [105]: df.eval("""
```

(continues on next page)

(continued from previous page)

```
   .....: e = d + a
   .....: f = e - 22
   .....: g = f / 2.0""", inplace=True)
   .....:

In [106]: df
Out[106]:
      a  b     c     d     e    f    g
3   7.5  3  10.5   7.5  15.0 -7.0 -3.5
4  10.0  4  14.0  10.0  20.0 -2.0 -1.0

[2 rows x 7 columns]
```

Other API Changes

- `DataFrame.between_time` and `Series.between_time` now only parse a fixed set of time strings. Parsing of date strings is no longer supported and raises a `ValueError`. (GH11818)

  ```
  In [107]: s = pd.Series(range(10), pd.date_range('2015-01-01', freq='H',␣
  ↪periods=10))

  In [108]: s.between_time("7:00am", "9:00am")
  Out[108]:
  2015-01-01 07:00:00    7
  2015-01-01 08:00:00    8
  2015-01-01 09:00:00    9
  Freq: H, Length: 3, dtype: int64
  ```

  This will now raise.

  ```
  In [2]: s.between_time('20150101 07:00:00','20150101 09:00:00')
  ValueError: Cannot convert arg ['20150101 07:00:00'] to a time.
  ```

- `.memory_usage()` now includes values in the index, as does memory_usage in `.info()` (GH11597)
- `DataFrame.to_latex()` now supports non-ascii encodings (eg `utf-8`) in Python 2 with the parameter `encoding` (GH7061)
- `pandas.merge()` and `DataFrame.merge()` will show a specific error message when trying to merge with an object that is not of type `DataFrame` or a subclass (GH12081)
- `DataFrame.unstack` and `Series.unstack` now take `fill_value` keyword to allow direct replacement of missing values when an unstack results in missing values in the resulting `DataFrame`. As an added benefit, specifying `fill_value` will preserve the data type of the original stacked data. (GH9746)
- As part of the new API for *window functions* and *resampling*, aggregation functions have been clarified, raising more informative error messages on invalid aggregations. (GH9052). A full set of examples are presented in *groupby*.
- Statistical functions for `NDFrame` objects (like `sum(), mean(), min()`) will now raise if non-numpy-compatible arguments are passed in for `**kwargs` (GH12301)
- `.to_latex` and `.to_html` gain a `decimal` parameter like `.to_csv`; the default is `'.'` (GH12031)
- More helpful error message when constructing a `DataFrame` with empty data but with indices (GH8020)
- `.describe()` will now properly handle bool dtype as a categorical (GH6625)

- More helpful error message with an invalid `.transform` with user defined input (GH10165)
- Exponentially weighted functions now allow specifying alpha directly (GH10789) and raise `ValueError` if parameters violate `0 < alpha <= 1` (GH12492)

Deprecations

- The functions `pd.rolling_*`, `pd.expanding_*`, and `pd.ewm*` are deprecated and replaced by the corresponding method call. Note that the new suggested syntax includes all of the arguments (even if default) (GH11603)

```
In [1]: s = pd.Series(range(3))

In [2]: pd.rolling_mean(s,window=2,min_periods=1)
        FutureWarning: pd.rolling_mean is deprecated for Series and
             will be removed in a future version, replace with
             Series.rolling(min_periods=1,window=2,center=False).mean()
Out[2]:
        0    0.0
        1    0.5
        2    1.5
        dtype: float64

In [3]: pd.rolling_cov(s, s, window=2)
        FutureWarning: pd.rolling_cov is deprecated for Series and
             will be removed in a future version, replace with
             Series.rolling(window=2).cov(other=<Series>)
Out[3]:
        0    NaN
        1    0.5
        2    0.5
        dtype: float64
```

- The `freq` and `how` arguments to the `.rolling`, `.expanding`, and `.ewm` (new) functions are deprecated, and will be removed in a future version. You can simply resample the input prior to creating a window function. (GH11603).

  For example, instead of `s.rolling(window=5,freq='D').max()` to get the max value on a rolling 5 Day window, one could use `s.resample('D').mean().rolling(window=5).max()`, which first resamples the data to daily data, then provides a rolling 5 day window.

- `pd.tseries.frequencies.get_offset_name` function is deprecated. Use offset's `.freqstr` property as alternative (GH11192)
- `pandas.stats.fama_macbeth` routines are deprecated and will be removed in a future version (GH6077)
- `pandas.stats.ols`, `pandas.stats.plm` and `pandas.stats.var` routines are deprecated and will be removed in a future version (GH6077)
- show a `FutureWarning` rather than a `DeprecationWarning` on using long-time deprecated syntax in `HDFStore.select`, where the `where` clause is not a string-like (GH12027)
- The `pandas.options.display.mpl_style` configuration has been deprecated and will be removed in a future version of pandas. This functionality is better handled by matplotlib's style sheets (GH11783).

Removal of deprecated float indexers

In GH4892 indexing with floating point numbers on a non-`Float64Index` was deprecated (in version 0.14.0). In 0.18.0, this deprecation warning is removed and these will now raise a `TypeError`. (GH12165, GH12333)

```
In [109]: s = pd.Series([1, 2, 3], index=[4, 5, 6])

In [110]: s
Out[110]:
4    1
5    2
6    3
Length: 3, dtype: int64

In [111]: s2 = pd.Series([1, 2, 3], index=list('abc'))

In [112]: s2
Out[112]:
a    1
b    2
c    3
Length: 3, dtype: int64
```

Previous Behavior:

```
# this is label indexing
In [2]: s[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not␣
↪floating point
Out[2]: 2

# this is positional indexing
In [3]: s.iloc[1.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not␣
↪floating point
Out[3]: 2

# this is label indexing
In [4]: s.loc[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not␣
↪floating point
Out[4]: 2

# .ix would coerce 1.0 to the positional 1, and index
In [5]: s2.ix[1.0] = 10
FutureWarning: scalar indexers for index type Index should be integers and not␣
↪floating point

In [6]: s2
Out[6]:
a     1
b    10
c     3
dtype: int64
```

New Behavior:

For iloc, getting & setting via a float scalar will always raise.

```
In [3]: s.iloc[2.0]
TypeError: cannot do label indexing on <class 'pandas.indexes.numeric.Int64Index'>␣
↪with these indexers [2.0] of <type 'float'>
```

Other indexers will coerce to a like integer for both getting and setting. The `FutureWarning` has been dropped for `.loc`, `.ix` and `[]`.

```
In [113]: s[5.0]
Out[113]: 2

In [114]: s.loc[5.0]
\\\\\\\\\\\\Out[114]: 2
```

and setting

```
In [115]: s_copy = s.copy()

In [116]: s_copy[5.0] = 10

In [117]: s_copy
Out[117]:
4     1
5    10
6     3
Length: 3, dtype: int64

In [118]: s_copy = s.copy()

In [119]: s_copy.loc[5.0] = 10

In [120]: s_copy
Out[120]:
4     1
5    10
6     3
Length: 3, dtype: int64
```

Positional setting with `.ix` and a float indexer will ADD this value to the index, rather than previously setting the value by position.

```
In [3]: s2.ix[1.0] = 10
In [4]: s2
Out[4]:
a       1
b       2
c       3
1.0    10
dtype: int64
```

Slicing will also coerce integer-like floats to integers for a non-`Float64Index`.

```
In [121]: s.loc[5.0:6]
Out[121]:
5    2
6    3
Length: 2, dtype: int64
```

Note that for floats that are NOT coercible to ints, the label based bounds will be excluded

```
In [122]: s.loc[5.1:6]
Out[122]:
6    3
Length: 1, dtype: int64
```

Float indexing on a `Float64Index` is unchanged.

```
In [123]: s = pd.Series([1, 2, 3], index=np.arange(3.))

In [124]: s[1.0]
Out[124]: 2

In [125]: s[1.0:2.5]
\\\\\\\\\\\\Out[125]:
1.0    2
2.0    3
Length: 2, dtype: int64
```

Removal of prior version deprecations/changes

- Removal of `rolling_corr_pairwise` in favor of `.rolling().corr(pairwise=True)` (GH4950)
- Removal of `expanding_corr_pairwise` in favor of `.expanding().corr(pairwise=True)` (GH4950)
- Removal of `DataMatrix` module. This was not imported into the pandas namespace in any event (GH12111)
- Removal of `cols` keyword in favor of `subset` in `DataFrame.duplicated()` and `DataFrame.drop_duplicates()` (GH6680)
- Removal of the `read_frame` and `frame_query` (both aliases for `pd.read_sql`) and `write_frame` (alias of `to_sql`) functions in the `pd.io.sql` namespace, deprecated since 0.14.0 (GH6292).
- Removal of the `order` keyword from `.factorize()` (GH6930)

Performance Improvements

- Improved performance of `andrews_curves` (GH11534)
- Improved huge `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex`'s ops performance including `NaT` (GH10277)
- Improved performance of `pandas.concat` (GH11958)
- Improved performance of `StataReader` (GH11591)
- Improved performance in construction of `Categoricals` with `Series` of datetimes containing `NaT` (GH12077)
- Improved performance of ISO 8601 date parsing for dates without separators (GH11899), leading zeros (GH11871) and with white space preceding the time zone (GH9714)

Bug Fixes

- Bug in `GroupBy.size` when data-frame is empty. (GH11699)
- Bug in `Period.end_time` when a multiple of time period is requested (GH11738)

- Regression in `.clip` with tz-aware datetimes (GH11838)
- Bug in `date_range` when the boundaries fell on the frequency (GH11804, GH12409)
- Bug in consistency of passing nested dicts to `.groupby(...).agg(...)` (GH9052)
- Accept unicode in `Timedelta` constructor (GH11995)
- Bug in value label reading for `StataReader` when reading incrementally (GH12014)
- Bug in vectorized `DateOffset` when `n` parameter is `0` (GH11370)
- Compat for numpy 1.11 w.r.t. `NaT` comparison changes (GH12049)
- Bug in `read_csv` when reading from a `StringIO` in threads (GH11790)
- Bug in not treating `NaT` as a missing value in datetimelikes when factorizing & with `Categoricals` (GH12077)
- Bug in getitem when the values of a `Series` were tz-aware (GH12089)
- Bug in `Series.str.get_dummies` when one of the variables was 'name' (GH12180)
- Bug in `pd.concat` while concatenating tz-aware NaT series. (GH11693, GH11755, GH12217)
- Bug in `pd.read_stata` with version <= 108 files (GH12232)
- Bug in `Series.resample` using a frequency of `Nano` when the index is a `DatetimeIndex` and contains non-zero nanosecond parts (GH12037)
- Bug in resampling with `.nunique` and a sparse index (GH12352)
- Removed some compiler warnings (GH12471)
- Work around compat issues with `boto` in python 3.5 (GH11915)
- Bug in `NaT` subtraction from `Timestamp` or `DatetimeIndex` with timezones (GH11718)
- Bug in subtraction of `Series` of a single tz-aware `Timestamp` (GH12290)
- Use compat iterators in PY2 to support `.next()` (GH12299)
- Bug in `Timedelta.round` with negative values (GH11690)
- Bug in `.loc` against `CategoricalIndex` may result in normal `Index` (GH11586)
- Bug in `DataFrame.info` when duplicated column names exist (GH11761)
- Bug in `.copy` of datetime tz-aware objects (GH11794)
- Bug in `Series.apply` and `Series.map` where `timedelta64` was not boxed (GH11349)
- Bug in `DataFrame.set_index()` with tz-aware `Series` (GH12358)
- Bug in subclasses of `DataFrame` where `AttributeError` did not propagate (GH11808)
- Bug groupby on tz-aware data where selection not returning `Timestamp` (GH11616)
- Bug in `pd.read_clipboard` and `pd.to_clipboard` functions not supporting Unicode; upgrade included `pyperclip` to v1.5.15 (GH9263)
- Bug in `DataFrame.query` containing an assignment (GH8664)
- Bug in `from_msgpack` where `__contains__()` fails for columns of the unpacked `DataFrame`, if the `DataFrame` has object columns. (GH11880)
- Bug in `.resample` on categorical data with `TimedeltaIndex` (GH12169)
- Bug in timezone info lost when broadcasting scalar datetime to `DataFrame` (GH11682)

- Bug in `Index` creation from `Timestamp` with mixed tz coerces to UTC (GH11488)
- Bug in `to_numeric` where it does not raise if input is more than one dimension (GH11776)
- Bug in parsing timezone offset strings with non-zero minutes (GH11708)
- Bug in `df.plot` using incorrect colors for bar plots under matplotlib 1.5+ (GH11614)
- Bug in the `groupby` `plot` method when using keyword arguments (GH11805).
- Bug in `DataFrame.duplicated` and `drop_duplicates` causing spurious matches when setting `keep=False` (GH11864)
- Bug in `.loc` result with duplicated key may have `Index` with incorrect dtype (GH11497)
- Bug in `pd.rolling_median` where memory allocation failed even with sufficient memory (GH11696)
- Bug in `DataFrame.style` with spurious zeros (GH12134)
- Bug in `DataFrame.style` with integer columns not starting at 0 (GH12125)
- Bug in `.style.bar` may not rendered properly using specific browser (GH11678)
- Bug in rich comparison of `Timedelta` with a `numpy.array` of `Timedelta` that caused an infinite recursion (GH11835)
- Bug in `DataFrame.round` dropping column index name (GH11986)
- Bug in `df.replace` while replacing value in mixed dtype `Dataframe` (GH11698)
- Bug in `Index` prevents copying name of passed `Index`, when a new name is not provided (GH11193)
- Bug in `read_excel` failing to read any non-empty sheets when empty sheets exist and `sheetname=None` (GH11711)
- Bug in `read_excel` failing to raise `NotImplemented` error when keywords `parse_dates` and `date_parser` are provided (GH11544)
- Bug in `read_sql` with `pymysql` connections failing to return chunked data (GH11522)
- Bug in `.to_csv` ignoring formatting parameters `decimal`, `na_rep`, `float_format` for float indexes (GH11553)
- Bug in `Int64Index` and `Float64Index` preventing the use of the modulo operator (GH9244)
- Bug in `MultiIndex.drop` for not lexsorted MultiIndexes (GH12078)
- Bug in `DataFrame` when masking an empty `DataFrame` (GH11859)
- Bug in `.plot` potentially modifying the `colors` input when the number of columns didn't match the number of series provided (GH12039).
- Bug in `Series.plot` failing when index has a `CustomBusinessDay` frequency (GH7222).
- Bug in `.to_sql` for `datetime.time` values with sqlite fallback (GH8341)
- Bug in `read_excel` failing to read data with one column when `squeeze=True` (GH12157)
- Bug in `read_excel` failing to read one empty column (GH12292, GH9002)
- Bug in `.groupby` where a `KeyError` was not raised for a wrong column if there was only one row in the dataframe (GH11741)
- Bug in `.read_csv` with dtype specified on empty data producing an error (GH12048)
- Bug in `.read_csv` where strings like `'2E'` are treated as valid floats (GH12237)
- Bug in building *pandas* with debugging symbols (GH12123)

- Removed `millisecond` property of `DatetimeIndex`. This would always raise a `ValueError` (GH12019).
- Bug in `Series` constructor with read-only data (GH11502)
- Removed `pandas.util.testing.choice()`. Should use `np.random.choice()`, instead. (GH12386)
- Bug in `.loc` setitem indexer preventing the use of a TZ-aware DatetimeIndex (GH12050)
- Bug in `.style` indexes and MultiIndexes not appearing (GH11655)
- Bug in `to_msgpack` and `from_msgpack` which did not correctly serialize or deserialize `NaT` (GH12307).
- Bug in `.skew` and `.kurt` due to roundoff error for highly similar values (GH11974)
- Bug in `Timestamp` constructor where microsecond resolution was lost if HHMMSS were not separated with ':' (GH10041)
- Bug in `buffer_rd_bytes` src->buffer could be freed more than once if reading failed, causing a segfault (GH12098)
- Bug in `crosstab` where arguments with non-overlapping indexes would return a `KeyError` (GH10291)
- Bug in `DataFrame.apply` in which reduction was not being prevented for cases in which `dtype` was not a numpy dtype (GH12244)
- Bug when initializing categorical series with a scalar value. (GH12336)
- Bug when specifying a UTC `DatetimeIndex` by setting `utc=True` in `.to_datetime` (GH11934)
- Bug when increasing the buffer size of CSV reader in `read_csv` (GH12494)
- Bug when setting columns of a `DataFrame` with duplicate column names (GH12344)

Contributors

A total of 101 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- ARF +
- Alex Alekseyev +
- Andrew McPherson +
- Andrew Rosenfeld
- Andy Hayden
- Anthonios Partheniou
- Anton I. Sipos
- Ben +
- Ben North +
- Bran Yang +
- Chris
- Chris Carroux +
- Christopher C. Aycock +
- Christopher Scanlin +

- Cody +
- Da Wang +
- Daniel Grady +
- Dorozhko Anton +
- Dr-Irv +
- Erik M. Bray +
- Evan Wright
- Francis T. O'Donovan +
- Frank Cleary +
- Gianluca Rossi
- Graham Jeffries +
- Guillaume Horel
- Henry Hammond +
- Isaac Schwabacher +
- Jean-Mathieu Deschenes
- Jeff Reback
- Joe Jevnik +
- John Freeman +
- John Fremlin +
- Jonas Hoersch +
- Joris Van den Bossche
- Joris Vankerschaver
- Justin Lecher
- Justin Lin +
- Ka Wo Chen
- Keming Zhang +
- Kerby Shedden
- Kyle +
- Marco Farrugia +
- MasonGallo +
- MattRijk +
- Matthew Lurie +
- Maximilian Roos
- Mayank Asthana +
- Mortada Mehyar
- Moussa Taifi +

- Navreet Gill +
- Nicolas Bonnotte
- Paul Reiners +
- Philip Gura +
- Pietro Battiston
- RahulHP +
- Randy Carnevale
- Rinoc Johnson
- Rishipuri +
- Sangmin Park +
- Scott E Lasley
- Sereger13 +
- Shannon Wang +
- Skipper Seabold
- Thierry Moisan
- Thomas A Caswell
- Toby Dylan Hocking +
- Tom Augspurger
- Travis +
- Trent Hauck
- Tux1
- Varun
- Wes McKinney
- Will Thompson +
- Yoav Ram
- Yoong Kang Lim +
- Yoshiki Vázquez Baeza
- Young Joong Kim +
- Younggun Kim
- Yuval Langer +
- alex argunov +
- behzad nouri
- boombard +
- brian-pantano +
- chromy +
- daniel +

- dgram0 +
- gfyoung +
- hack-c +
- hcontrast +
- jfoo +
- kaustuv deolal +
- llllllllll
- ranarag +
- rockg
- scls19fr
- seales +
- sinhrks
- srib +
- surveymedia.ca +
- tworec +

8.8 Version 0.17

8.8.1 v0.17.1 (November 21, 2015)

Note: We are proud to announce that *pandas* has become a sponsored project of the (NumFOCUS organization). This will help ensure the success of development of *pandas* as a world-class open-source project.

This is a minor bug-fix release from 0.17.0 and includes a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Support for Conditional HTML Formatting, see *here*
- Releasing the GIL on the csv reader & other ops, see *here*
- Fixed regression in `DataFrame.drop_duplicates` from 0.16.2, causing incorrect results on integer values (GH11376)

What's new in v0.17.1

- *New features*
  - *Conditional HTML Formatting*
- *Enhancements*
- *API changes*
  - *Deprecations*

- *Performance Improvements*
- *Bug Fixes*
- *Contributors*

New features

Conditional HTML Formatting

Warning: This is a new feature and is under active development. We'll be adding features an possibly making breaking changes in future releases. Feedback is welcome.

We've added *experimental* support for conditional HTML formatting: the visual styling of a DataFrame based on the data. The styling is accomplished with HTML and CSS. Accesses the styler class with the `pandas.DataFrame.style`, attribute, an instance of `Styler` with your data attached.

Here's a quick example:

```
In [1]: np.random.seed(123)

In [2]: df = pd.DataFrame(np.random.randn(10, 5), columns=list('abcde'))

In [3]: html = df.style.background_gradient(cmap='viridis', low=.5)
```

We can render the HTML to get the following table.

`Styler` interacts nicely with the Jupyter Notebook. See the documentation for more.

Enhancements

- `DatetimeIndex` now supports conversion to strings with `astype(str)` (GH10442)
- Support for `compression` (gzip/bz2) in `pandas.DataFrame.to_csv()` (GH7615)
- `pd.read_*` functions can now also accept `pathlib.Path`, or `py._path.local.LocalPath` objects for the `filepath_or_buffer` argument. (GH11033) - The `DataFrame` and `Series` functions `.to_csv()`, `.to_html()` and `.to_latex()` can now handle paths beginning with tildes (e.g. `~/Documents/`) (GH11438)
- `DataFrame` now uses the fields of a `namedtuple` as columns, if columns are not supplied (GH11181)
- `DataFrame.itertuples()` now returns `namedtuple` objects, when possible. (GH11269, GH11625)
- Added `axvlines_kwds` to parallel coordinates plot (GH10709)
- Option to `.info()` and `.memory_usage()` to provide for deep introspection of memory consumption. Note that this can be expensive to compute and therefore is an optional parameter. (GH11595)

```
In [4]: df = pd.DataFrame({'A': ['foo'] * 1000})  # noqa: F821

In [5]: df['B'] = df['A'].astype('category')

# shows the '+' as we have object dtypes
In [6]: df.info()
```

(continues on next page)

(continued from previous page)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
A    1000 non-null object
B    1000 non-null category
dtypes: category(1), object(1)
memory usage: 9.0+ KB

# we have an accurate memory assessment (but can be expensive to compute this)
In [7]: df.info(memory_usage='deep')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
A    1000 non-null object
B    1000 non-null category
dtypes: category(1), object(1)
memory usage: 75.4 KB
```

- `Index` now has a `fillna` method (GH10089)

```
In [8]: pd.Index([1, np.nan, 3]).fillna(2)
Out[8]: Float64Index([1.0, 2.0, 3.0], dtype='float64')
```

- Series of type `category` now make `.str.<...>` and `.dt.<...>` accessor methods / properties available, if the categories are of that type. (GH10661)

```
In [9]: s = pd.Series(list('aabb')).astype('category')

In [10]: s
Out[10]:
0    a
1    a
2    b
3    b
Length: 4, dtype: category
Categories (2, object): [a, b]

In [11]: s.str.contains("a")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0     True
1     True
2    False
3    False
Length: 4, dtype: bool

In [12]: date = pd.Series(pd.date_range('1/1/2015', periods=5)).astype('category')

In [13]: date
Out[13]:
0   2015-01-01
1   2015-01-02
2   2015-01-03
3   2015-01-04
4   2015-01-05
```

(continues on next page)

(continued from previous page)

```
Length: 5, dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-04,␣
↪2015-01-05]

In [14]: date.dt.day
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    1
1    2
2    3
3    4
4    5
Length: 5, dtype: int64
```

- `pivot_table` now has a `margins_name` argument so you can use something other than the default of 'All' (GH3335)
- Implement export of `datetime64[ns, tz]` dtypes with a fixed HDF5 store (GH11411)
- Pretty printing sets (e.g. in DataFrame cells) now uses set literal syntax (`{x, y}`) instead of Legacy Python syntax (`set([x, y])`) (GH11215)
- Improve the error message in `pandas.io.gbq.to_gbq()` when a streaming insert fails (GH11285) and when the DataFrame does not match the schema of the destination table (GH11359)

API changes

- raise `NotImplementedError` in `Index.shift` for non-supported index types (GH8038)
- `min` and `max` reductions on `datetime64` and `timedelta64` dtyped series now result in `NaT` and not `nan` (GH11245).
- Indexing with a null key will raise a `TypeError`, instead of a `ValueError` (GH11356)
- `Series.ptp` will now ignore missing values by default (GH11163)

Deprecations

- The `pandas.io.ga` module which implements `google-analytics` support is deprecated and will be removed in a future version (GH11308)
- Deprecate the `engine` keyword in `.to_csv()`, which will be removed in a future version (GH11274)

Performance Improvements

- Checking monotonic-ness before sorting on an index (GH11080)
- `Series.dropna` performance improvement when its dtype can't contain `NaN` (GH11159)
- Release the GIL on most datetime field operations (e.g. `DatetimeIndex.year`, `Series.dt.year`), normalization, and conversion to and from `Period`, `DatetimeIndex.to_period` and `PeriodIndex.to_timestamp` (GH11263)
- Release the GIL on some rolling algos: `rolling_median`, `rolling_mean`, `rolling_max`, `rolling_min`, `rolling_var`, `rolling_kurt`, `rolling_skew` (GH11450)
- Release the GIL when reading and parsing text files in `read_csv`, `read_table` (GH11272)

- Improved performance of `rolling_median` (GH11450)
- Improved performance of `to_excel` (GH11352)
- Performance bug in repr of `Categorical` categories, which was rendering the strings before chopping them for display (GH11305)
- Performance improvement in `Categorical.remove_unused_categories`, (GH11643).
- Improved performance of `Series` constructor with no data and `DatetimeIndex` (GH11433)
- Improved performance of `shift`, `cumprod`, and `cumsum` with groupby (GH4095)

Bug Fixes

- `SparseArray.__iter__()` now does not cause `PendingDeprecationWarning` in Python 3.5 (GH11622)
- Regression from 0.16.2 for output formatting of long floats/nan, restored in (GH11302)
- `Series.sort_index()` now correctly handles the `inplace` option (GH11402)
- Incorrectly distributed .c file in the build on `PyPi` when reading a csv of floats and passing `na_values=<a scalar>` would show an exception (GH11374)
- Bug in `.to_latex()` output broken when the index has a name (GH10660)
- Bug in `HDFStore.append` with strings whose encoded length exceeded the max unencoded length (GH11234)
- Bug in merging `datetime64[ns, tz]` dtypes (GH11405)
- Bug in `HDFStore.select` when comparing with a numpy scalar in a where clause (GH11283)
- Bug in using `DataFrame.ix` with a MultiIndex indexer (GH11372)
- Bug in `date_range` with ambiguous endpoints (GH11626)
- Prevent adding new attributes to the accessors `.str`, `.dt` and `.cat`. Retrieving such a value was not possible, so error out on setting it. (GH10673)
- Bug in tz-conversions with an ambiguous time and `.dt` accessors (GH11295)
- Bug in output formatting when using an index of ambiguous times (GH11619)
- Bug in comparisons of Series vs list-likes (GH11339)
- Bug in `DataFrame.replace` with a `datetime64[ns, tz]` and a non-compat to_replace (GH11326, GH11153)
- Bug in `isnull` where `numpy.datetime64('NaT')` in a `numpy.array` was not determined to be null(GH11206)
- Bug in list-like indexing with a mixed-integer Index (GH11320)
- Bug in `pivot_table` with `margins=True` when indexes are of `Categorical` dtype (GH10993)
- Bug in `DataFrame.plot` cannot use hex strings colors (GH10299)
- Regression in `DataFrame.drop_duplicates` from 0.16.2, causing incorrect results on integer values (GH11376)
- Bug in `pd.eval` where unary ops in a list error (GH11235)
- Bug in `squeeze()` with zero length arrays (GH11230, GH8999)
- Bug in `describe()` dropping column names for hierarchical indexes (GH11517)

- Bug in `DataFrame.pct_change()` not propagating `axis` keyword on `.fillna` method (GH11150)
- Bug in `.to_csv()` when a mix of integer and string column names are passed as the `columns` parameter (GH11637)
- Bug in indexing with a `range`, (GH11652)
- Bug in inference of numpy scalars and preserving dtype when setting columns (GH11638)
- Bug in `to_sql` using unicode column names giving UnicodeEncodeError with (GH11431).
- Fix regression in setting of `xticks` in `plot` (GH11529).
- Bug in `holiday.dates` where observance rules could not be applied to holiday and doc enhancement (GH11477, GH11533)
- Fix plotting issues when having plain `Axes` instances instead of `SubplotAxes` (GH11520, GH11556).
- Bug in `DataFrame.to_latex()` produces an extra rule when `header=False` (GH7124)
- Bug in `df.groupby(...).apply(func)` when a func returns a `Series` containing a new datetimelike column (GH11324)
- Bug in `pandas.json` when file to load is big (GH11344)
- Bugs in `to_excel` with duplicate columns (GH11007, GH10982, GH10970)
- Fixed a bug that prevented the construction of an empty series of dtype `datetime64[ns, tz]` (GH11245).
- Bug in `read_excel` with MultiIndex containing integers (GH11317)
- Bug in `to_excel` with openpyxl 2.2+ and merging (GH11408)
- Bug in `DataFrame.to_dict()` produces a `np.datetime64` object instead of `Timestamp` when only datetime is present in data (GH11327)
- Bug in `DataFrame.corr()` raises exception when computes Kendall correlation for DataFrames with boolean and not boolean columns (GH11560)
- Bug in the link-time error caused by C `inline` functions on FreeBSD 10+ (with `clang`) (GH10510)
- Bug in `DataFrame.to_csv` in passing through arguments for formatting `MultiIndexes`, including `date_format` (GH7791)
- Bug in `DataFrame.join()` with `how='right'` producing a `TypeError` (GH11519)
- Bug in `Series.quantile` with empty list results has `Index` with `object` dtype (GH11588)
- Bug in `pd.merge` results in empty `Int64Index` rather than `Index(dtype=object)` when the merge result is empty (GH11588)
- Bug in `Categorical.remove_unused_categories` when having `NaN` values (GH11599)
- Bug in `DataFrame.to_sparse()` loses column names for MultiIndexes (GH11600)
- Bug in `DataFrame.round()` with non-unique column index producing a Fatal Python error (GH11611)
- Bug in `DataFrame.round()` with `decimals` being a non-unique indexed Series producing extra columns (GH11618)

Contributors

A total of 63 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aleksandr Drozd +

- Alex Chase +
- Anthonios Partheniou
- BrenBarn +
- Brian J. McGuirk +
- Chris
- Christian Berendt +
- Christian Perez +
- Cody Piersall +
- Data & Code Expert Experimenting with Code on Data
- DrIrv +
- Evan Wright
- Guillaume Gay
- Hamed Saljooghinejad +
- Iblis Lin +
- Jake VanderPlas
- Jan Schulz
- Jean-Mathieu Deschenes +
- Jeff Reback
- Jimmy Callin +
- Joris Van den Bossche
- K.-Michael Aye
- Ka Wo Chen
- Loïc Séguin-C +
- Luo Yicheng +
- Magnus Jöud +
- Manuel Leonhardt +
- Matthew Gilbert
- Maximilian Roos
- Michael +
- Nicholas Stahl +
- Nicolas Bonnotte +
- Pastafarianist +
- Petra Chong +
- Phil Schaf +
- Philipp A +
- Rob deCarvalho +

- Roman Khomenko +
- Rémy Léone +
- Sebastian Bank +
- Sinhrks
- Stephan Hoyer
- Thierry Moisan
- Tom Augspurger
- Tux1 +
- Varun +
- Wieland Hoffmann +
- Winterflower
- Yoav Ram +
- Younggun Kim
- Zeke +
- ajcr
- azuranski +
- behzad nouri
- cel4
- emilydolson +
- hironow +
- lexual
- llllllllll +
- rockg
- silentquasar +
- sinhrks
- taeold +

8.8.2 v0.17.0 (October 9, 2015)

This is a major release from 0.16.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Warning: pandas >= 0.17.0 will no longer support compatibility with Python version 3.2 (GH9118)

Warning: The `pandas.io.data` package is deprecated and will be replaced by the pandas-datareader package. This will allow the data modules to be independently updated to your pandas installation. The API for `pandas-datareader v0.1.1` is exactly the same as in `pandas v0.17.0` (GH8961, GH10861).

After installing pandas-datareader, you can easily change your imports:

```
from pandas.io import data, wb
```

becomes

```
from pandas_datareader import data, wb
```

Highlights include:

- Release the Global Interpreter Lock (GIL) on some cython operations, see *here*
- Plotting methods are now available as attributes of the `.plot` accessor, see *here*
- The sorting API has been revamped to remove some long-time inconsistencies, see *here*
- Support for a `datetime64[ns]` with timezones as a first-class dtype, see *here*
- The default for `to_datetime` will now be to `raise` when presented with unparseable formats, previously this would return the original input. Also, date parse functions now return consistent results. See *here*
- The default for `dropna` in `HDFStore` has changed to `False`, to store by default all rows even if they are all `NaN`, see *here*
- Datetime accessor (`dt`) now supports `Series.dt.strftime` to generate formatted strings for datetime-likes, and `Series.dt.total_seconds` to generate each duration of the timedelta in seconds. See *here*
- `Period` and `PeriodIndex` can handle multiplied freq like `3D`, which corresponding to 3 days span. See *here*
- Development installed versions of pandas will now have `PEP440` compliant version strings (GH9518)
- Development support for benchmarking with the Air Speed Velocity library (GH8361)
- Support for reading SAS xport files, see *here*
- Documentation comparing SAS to *pandas*, see *here*
- Removal of the automatic TimeSeries broadcasting, deprecated since 0.8.0, see *here*
- Display format with plain text can optionally align with Unicode East Asian Width, see *here*
- Compatibility with Python 3.5 (GH11097)
- Compatibility with matplotlib 1.5.0 (GH11111)

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

What's new in v0.17.0

New features

Datetime with TZ

We are adding an implementation that natively supports datetime with timezones. A `Series` or a `DataFrame` column previously *could* be assigned a datetime with timezones, and would work as an `object` dtype. This had performance issues with a large number rows. See the *docs* for more details. (GH8260, GH10763, GH11034).

The new implementation allows for having a single-timezone across all rows, with operations in a performant manner.

```
In [1]: df = pd.DataFrame({'A': pd.date_range('20130101', periods=3),
   ...:                    'B': pd.date_range('20130101', periods=3, tz='US/Eastern'),
   ...:                    'C': pd.date_range('20130101', periods=3, tz='CET')})
   ...:
```

(continues on next page)

(continued from previous page)

```
In [2]: df
Out[2]:
           A                         B                         C
0 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00+01:00
1 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-02 00:00:00+01:00
2 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-03 00:00:00+01:00

[3 rows x 3 columns]

In [3]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A                 datetime64[ns]
B    datetime64[ns, US/Eastern]
C           datetime64[ns, CET]
Length: 3, dtype: object
```

```
In [4]: df.B
Out[4]:
0   2013-01-01 00:00:00-05:00
1   2013-01-02 00:00:00-05:00
2   2013-01-03 00:00:00-05:00
Name: B, Length: 3, dtype: datetime64[ns, US/Eastern]

In [5]: df.B.dt.tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   2013-01-01
1   2013-01-02
2   2013-01-03
Name: B, Length: 3, dtype: datetime64[ns]
```

This uses a new-dtype representation as well, that is very similar in look-and-feel to its numpy cousin `datetime64[ns]`

```
In [6]: df['B'].dtype
Out[6]: datetime64[ns, US/Eastern]

In [7]: type(df['B'].dtype)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: pandas.core.dtypes.dtypes.DatetimeTZDtype
```

Note: There is a slightly different string repr for the underlying `DatetimeIndex` as a result of the dtype changes, but functionally these are the same.

Previous Behavior:

```
In [1]: pd.date_range('20130101', periods=3, tz='US/Eastern')
Out[1]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
                       '2013-01-03 00:00:00-05:00'],
                      dtype='datetime64[ns]', freq='D', tz='US/Eastern')

In [2]: pd.date_range('20130101', periods=3, tz='US/Eastern').dtype
Out[2]: dtype('<M8[ns]')
```

New Behavior:

```
In [8]: pd.date_range('20130101', periods=3, tz='US/Eastern')
Out[8]:
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
               '2013-01-03 00:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')

In [9]: pd.date_range('20130101', periods=3, tz='US/Eastern').dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪datetime64[ns, US/Eastern]
```

Releasing the GIL

We are releasing the global-interpreter-lock (GIL) on some cython operations. This will allow other threads to run simultaneously during computation, potentially allowing performance improvements from multi-threading. Notably `groupby`, `nsmallest`, `value_counts` and some indexing operations benefit from this. (GH8882)

For example the groupby expression in the following code will have the GIL released during the factorization step, e.g. `df.groupby('key')` as well as the `.sum()` operation.

```
N = 1000000
ngroups = 10
df = DataFrame({'key': np.random.randint(0, ngroups, size=N),
                'data': np.random.randn(N)})
df.groupby('key')['data'].sum()
```

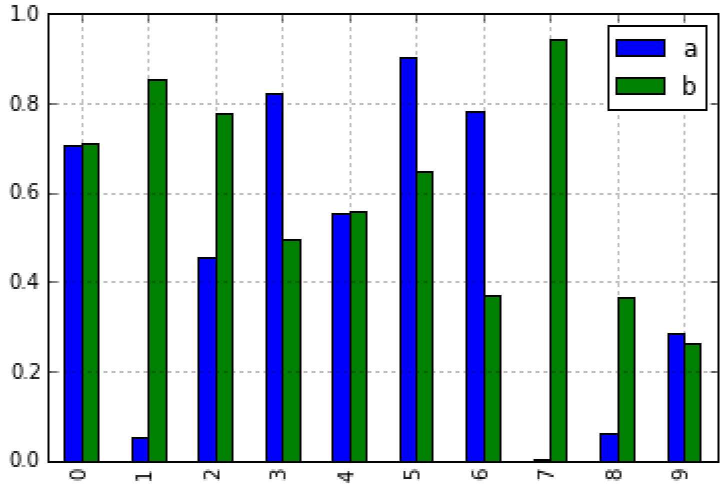
Releasing of the GIL could benefit an application that uses threads for user interactions (e.g. QT), or performing multi-threaded computations. A nice example of a library that can handle these types of computation-in-parallel is the dask library.

Plot submethods

The Series and DataFrame `.plot()` method allows for customizing *plot types* by supplying the `kind` keyword arguments. Unfortunately, many of these kinds of plots use different required and optional keyword arguments, which makes it difficult to discover what any given plot kind uses out of the dozens of possible arguments.

To alleviate this issue, we have added a new, optional plotting interface, which exposes each kind of plot as a method of the `.plot` attribute. Instead of writing `series.plot(kind=<kind>, ...)`, you can now also use `series.plot.<kind>(...)`:

```
In [10]: df = pd.DataFrame(np.random.rand(10, 2), columns=['a', 'b'])

In [11]: df.plot.bar()
```



As a result of this change, these methods are now all discoverable via tab-completion:

```
In [12]: df.plot.<TAB>  # noqa: E225, E999
df.plot.area     df.plot.barh     df.plot.density  df.plot.hist     df.plot.line     ␣
↪df.plot.scatter
df.plot.bar      df.plot.box      df.plot.hexbin   df.plot.kde      df.plot.pie
```

Each method signature only includes relevant arguments. Currently, these are limited to required arguments, but in the future these will include optional arguments, as well. For an overview, see the new *Plotting* API documentation.

Additional methods for `dt` accessor

strftime

We are now supporting a `Series.dt.strftime` method for datetime-likes to generate a formatted string (GH10110). Examples:

```
# DatetimeIndex
In [13]: s = pd.Series(pd.date_range('20130101', periods=4))

In [14]: s
Out[14]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
Length: 4, dtype: datetime64[ns]

In [15]: s.dt.strftime('%Y/%m/%d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
Length: 4, dtype: object
```

```
# PeriodIndex
In [16]: s = pd.Series(pd.period_range('20130101', periods=4))

In [17]: s
Out[17]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
Length: 4, dtype: period[D]

In [18]: s.dt.strftime('%Y/%m/%d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
Length: 4, dtype: object
```

The string format is as the python standard library and details can be found here

total_seconds

pd.Series of type timedelta64 has new method .dt.total_seconds() returning the duration of the timedelta in seconds (GH10817)

```
# TimedeltaIndex
In [19]: s = pd.Series(pd.timedelta_range('1 minutes', periods=4))

In [20]: s
Out[20]:
0   0 days 00:01:00
1   1 days 00:01:00
2   2 days 00:01:00
3   3 days 00:01:00
Length: 4, dtype: timedelta64[ns]

In [21]: s.dt.total_seconds()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0        60.0
1     86460.0
2    172860.0
3    259260.0
Length: 4, dtype: float64
```

Period Frequency Enhancement

Period, PeriodIndex and period_range can now accept multiplied freq. Also, Period.freq and PeriodIndex.freq are now stored as a DateOffset instance like DatetimeIndex, and not as str (GH7811)

A multiplied freq represents a span of corresponding length. The example below creates a period of 3 days. Addition and subtraction will shift the period by its span.

```
In [22]: p = pd.Period('2015-08-01', freq='3D')

In [23]: p
Out[23]: Period('2015-08-01', '3D')

In [24]: p + 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[24]: Period('2015-08-04', '3D')

In [25]: p - 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]:␣
↪Period('2015-07-26', '3D')

In [26]: p.to_timestamp()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2015-08-01 00:00:00')

In [27]: p.to_timestamp(how='E')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2015-08-03 23:59:59.999999999')
```

You can use the multiplied freq in `PeriodIndex` and `period_range`.

```
In [28]: idx = pd.period_range('2015-08-01', periods=4, freq='2D')

In [29]: idx
Out[29]: PeriodIndex(['2015-08-01', '2015-08-03', '2015-08-05', '2015-08-07'], dtype=
↪'period[2D]', freq='2D')

In [30]: idx + 1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪PeriodIndex(['2015-08-03', '2015-08-05', '2015-08-07', '2015-08-09'], dtype=
↪'period[2D]', freq='2D')
```

Support for SAS XPORT files

`read_sas()` provides support for reading *SAS XPORT* format files. (GH4052).

```
df = pd.read_sas('sas_xport.xpt')
```

It is also possible to obtain an iterator and read an XPORT file incrementally.

```
for df in pd.read_sas('sas_xport.xpt', chunksize=10000):
    do_something(df)
```

See the *docs* for more details.

Support for Math Functions in .eval()

`eval()` now supports calling math functions (GH4893)

```
df = pd.DataFrame({'a': np.random.randn(10)})
df.eval("b = sin(a)")
```

The support math functions are *sin*, *cos*, *exp*, *log*, *expm1*, *log1p*, *sqrt*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan*, *arccosh*, *arcsinh*, *arctanh*, *abs* and *arctan2*.

These functions map to the intrinsics for the NumExpr engine. For the Python engine, they are mapped to NumPy calls.

Changes to Excel with MultiIndex

In version 0.16.2 a DataFrame with MultiIndex columns could not be written to Excel via to_excel. That functionality has been added (GH10564), along with updating read_excel so that the data can be read back with, no loss of information, by specifying which columns/rows make up the MultiIndex in the header and index_col parameters (GH4679)

See the *documentation* for more details.

```
In [31]: df = pd.DataFrame([[1, 2, 3, 4], [5, 6, 7, 8]],
   ....:                   columns=pd.MultiIndex.from_product(
   ....:                   [['foo', 'bar'], ['a', 'b']], names=['col1', 'col2']),
   ....:                   index=pd.MultiIndex.from_product([['j'], ['l', 'k']],
   ....:                                                    names=['i1', 'i2']))
   ....:

In [32]: df
Out[32]:
col1  foo    bar
col2    a  b   a  b
i1 i2
j  l    1  2   3  4
   k    5  6   7  8

[2 rows x 4 columns]

In [33]: df.to_excel('test.xlsx')

In [34]: df = pd.read_excel('test.xlsx', header=[0, 1], index_col=[0, 1])

In [35]: df
Out[35]:
col1  foo    bar
col2    a  b   a  b
i1 i2
j  l    1  2   3  4
   k    5  6   7  8

[2 rows x 4 columns]
```

Previously, it was necessary to specify the has_index_names argument in read_excel, if the serialized data had index names. For version 0.17.0 the output format of to_excel has been changed to make this keyword unnecessary - the change is shown below.

Old

	A	B	C	D	E	F
1		A	B	C	D	
2	idx_name					
3	2000-01-07 00:00:00	0.968129	0.906529	0.05343	0.02619	
4	2000-01-10 00:00:00	-0.16632	1.981993	1.833093	0.803685	
5	2000-01-11 00:00:00	0.121057	0.36946	-0.02888	1.683975	
6	2000-01-12 00:00:00	-1.70456	-0.73098	-0.38088	0.020946	
7	2000-01-13 00:00:00	-1.20024	1.907733	0.629318	1.507033	
8	2000-01-14 00:00:00	-0.66344	0.073188	1.583482	0.735205	
9	2000-01-17 00:00:00	0.716635	-2.07952	1.760536	0.970309	

New

	A	B	C	D	E
1	idx_name	A	B	C	D
2	2000-01-07 00:00:00	0.968129	0.906529	0.05343	0.02619
3	2000-01-10 00:00:00	-0.16632	1.981993	1.833093	0.803685
4	2000-01-11 00:00:00	0.121057	0.36946	-0.02888	1.683975
5	2000-01-12 00:00:00	-1.70456	-0.73098	-0.38088	0.020946
6	2000-01-13 00:00:00	-1.20024	1.907733	0.629318	1.507033
7	2000-01-14 00:00:00	-0.66344	0.073188	1.583482	0.735205
8	2000-01-17 00:00:00	0.716635	-2.07952	1.760536	0.970309
9	2000-01-18 00:00:00	0.737628	2.32367	2.706276	0.691843

> **Warning:** Excel files saved in version 0.16.2 or prior that had index names will still able to be read in, but the `has_index_names` argument must specified to `True`.

Google BigQuery Enhancements

- Added ability to automatically create a table/dataset using the `pandas.io.gbq.to_gbq()` function if the destination table/dataset does not exist. (GH8325, GH11121).
- Added ability to replace an existing table and schema when calling the `pandas.io.gbq.to_gbq()` function via the `if_exists` argument. See the docs for more details (GH8325).
- `InvalidColumnOrder` and `InvalidPageToken` in the gbq module will raise `ValueError` instead of `IOError`.
- The `generate_bq_schema()` function is now deprecated and will be removed in a future version (GH11121)
- The gbq module will now support Python 3 (GH11094).

Display Alignment with Unicode East Asian Width

Warning: Enabling this option will affect the performance for printing of `DataFrame` and `Series` (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters its width is corresponding to 2 alphabets. If a `DataFrame` or `Series` contains these characters, the default output cannot be aligned properly. The following options are added to enable precise handling for these characters.

- `display.unicode.east_asian_width`: Whether to use the Unicode East Asian Width to calculate the display text width. (GH2612)
- `display.unicode.ambiguous_as_wide`: Whether to handle Unicode characters belong to Ambiguous as Wide. (GH11102)

```
In [36]: df = pd.DataFrame({u'': ['UK', u''], u'': ['Alice', u'']})

In [37]: df;
```

```
>>> df = pd.DataFrame({u'国籍': ['UK', u'日本'], u'名前': ['Alice', u'しのぶ']})
>>> df
      名前   国籍
0  Alice   UK
1    しのぶ   日本
```

```
In [38]: pd.set_option('display.unicode.east_asian_width', True)

In [39]: df;
```

```
>>> pd.set_option('display.unicode.east_asian_width', True)
>>> df
     名前  国籍
0   Alice    UK
1  しのぶ  日本
```

For further details, see *here*

Other enhancements

- Support for `openpyxl` >= 2.2. The API for style support is now stable (GH10125)
- `merge` now accepts the argument `indicator` which adds a Categorical-type column (by default called `_merge`) to the output object that takes on the values (GH8790)

Observation Origin	`_merge` value
Merge key only in `'left'` frame	`left_only`
Merge key only in `'right'` frame	`right_only`
Merge key in both frames	`both`

```
In [40]: df1 = pd.DataFrame({'col1':[0,1], 'col_left':['a','b']})

In [41]: df2 = pd.DataFrame({'col1':[1,2,2],'col_right':[2,2,2]})
```

(continues on next page)

(continued from previous page)

```
In [42]: pd.merge(df1, df2, on='col1', how='outer', indicator=True)
Out[42]:
   col1 col_left  col_right      _merge
0     0        a        NaN   left_only
1     1        b        2.0        both
2     2      NaN        2.0  right_only
3     2      NaN        2.0  right_only

[4 rows x 4 columns]
```

For more, see the *updated docs*

- `pd.to_numeric` is a new function to coerce strings to numbers (possibly with coercion) (GH11133)
- `pd.merge` will now allow duplicate column names if they are not merged upon (GH10639).
- `pd.pivot` will now allow passing index as `None` (GH3962).
- `pd.concat` will now use existing Series names if provided (GH10698).

```
In [43]: foo = pd.Series([1, 2], name='foo')

In [44]: bar = pd.Series([1, 2])

In [45]: baz = pd.Series([4, 5])
```

Previous Behavior:

```
In [1]: pd.concat([foo, bar, baz], 1)
Out[1]:
      0  1  2
   0  1  1  4
   1  2  2  5
```

New Behavior:

```
In [46]: pd.concat([foo, bar, baz], 1)
Out[46]:
   foo  0  1
0    1  1  4
1    2  2  5

[2 rows x 3 columns]
```

- `DataFrame` has gained the `nlargest` and `nsmallest` methods (GH10393)
- Add a `limit_direction` keyword argument that works with `limit` to enable `interpolate` to fill `NaN` values forward, backward, or both (GH9218, GH10420, GH11115)

```
In [47]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13])

In [48]: ser.interpolate(limit=1, limit_direction='both')
Out[48]:
0     NaN
1     5.0
2     5.0
3     7.0
4     NaN
```

(continues on next page)

(continued from previous page)

```
5    11.0
6    13.0
Length: 7, dtype: float64
```

- Added a `DataFrame.round` method to round the values to a variable number of decimal places (GH10568).

```
In [49]: df = pd.DataFrame(np.random.random([3, 3]),
   ....:                   columns=['A', 'B', 'C'],
   ....:                   index=['first', 'second', 'third'])
   ....: 

In [50]: df
Out[50]: 
               A         B         C
first   0.126970  0.966718  0.260476
second  0.897237  0.376750  0.336222
third   0.451376  0.840255  0.123102

[3 rows x 3 columns]

In [51]: df.round(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
           A     B     C
first   0.13  0.97  0.26
second  0.90  0.38  0.34
third   0.45  0.84  0.12

[3 rows x 3 columns]

In [52]: df.round({'A': 0, 'C': 2})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B     C
first   0.0  0.966718  0.26
second  1.0  0.376750  0.34
third   0.0  0.840255  0.12

[3 rows x 3 columns]
```

- `drop_duplicates` and `duplicated` now accept a `keep` keyword to target first, last, and all duplicates. The `take_last` keyword is deprecated, see *here* (GH6511, GH8505)

```
In [53]: s = pd.Series(['A', 'B', 'C', 'A', 'B', 'D'])

In [54]: s.drop_duplicates()
Out[54]: 
0    A
1    B
2    C
5    D
Length: 4, dtype: object

In [55]: s.drop_duplicates(keep='last')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[55]: 
2    C
```

(continues on next page)

(continued from previous page)

```
3    A
4    B
5    D
Length: 4, dtype: object

In [56]: s.drop_duplicates(keep=False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2    C
5    D
Length: 2, dtype: object
```

- Reindex now has a `tolerance` argument that allows for finer control of *Limits on filling while reindexing* (GH10411):

```
In [57]: df = pd.DataFrame({'x': range(5),
   ....:                    't': pd.date_range('2000-01-01', periods=5)})
   ....:

In [58]: df.reindex([0.1, 1.9, 3.5],
   ....:            method='nearest',
   ....:            tolerance=0.2)
   ....:
Out[58]:
       x          t
0.1  0.0 2000-01-01
1.9  2.0 2000-01-03
3.5  NaN        NaT

[3 rows x 2 columns]
```

When used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will coerced into a `Timedelta` if possible. This allows you to specify tolerance with a string:

```
In [59]: df = df.set_index('t')

In [60]: df.reindex(pd.to_datetime(['1999-12-31']),
   ....:            method='nearest',
   ....:            tolerance='1 day')
   ....:
Out[60]:
            x
1999-12-31  0

[1 rows x 1 columns]
```

`tolerance` is also exposed by the lower level `Index.get_indexer` and `Index.get_loc` methods.

- Added functionality to use the `base` argument when resampling a `TimeDeltaIndex` (GH10530)
- `DatetimeIndex` can be instantiated using strings contains `NaT` (GH7599)
- `to_datetime` can now accept the `yearfirst` keyword (GH7599)
- `pandas.tseries.offsets` larger than the `Day` offset can now be used with a `Series` for addition/subtraction (GH10699). See the *docs* for more details.
- `pd.Timedelta.total_seconds()` now returns Timedelta duration to ns precision (previously microsecond precision) (GH10939)

- `PeriodIndex` now supports arithmetic with `np.ndarray` (GH10638)
- Support pickling of `Period` objects (GH10439)
- `.as_blocks` will now take a `copy` optional argument to return a copy of the data, default is to copy (no change in behavior from prior versions), (GH9607)
- `regex` argument to `DataFrame.filter` now handles numeric column names instead of raising `ValueError` (GH10384).
- Enable reading gzip compressed files via URL, either by explicitly setting the compression parameter or by inferring from the presence of the HTTP Content-Encoding header in the response (GH8685)
- Enable writing Excel files in *memory* using StringIO/BytesIO (GH7074)
- Enable serialization of lists and dicts to strings in `ExcelWriter` (GH8188)
- SQL io functions now accept a SQLAlchemy connectable. (GH7877)
- `pd.read_sql` and `to_sql` can accept database URI as `con` parameter (GH10214)
- `read_sql_table` will now allow reading from views (GH10750).
- Enable writing complex values to `HDFStores` when using the `table` format (GH10447)
- Enable `pd.read_hdf` to be used without specifying a key when the HDF file contains a single dataset (GH10443)
- `pd.read_stata` will now read Stata 118 type files. (GH9882)
- `msgpack` submodule has been updated to 0.4.6 with backward compatibility (GH10581)
- `DataFrame.to_dict` now accepts `orient='index'` keyword argument (GH10844).
- `DataFrame.apply` will return a Series of dicts if the passed function returns a dict and `reduce=True` (GH8735).
- Allow passing *kwargs* to the interpolation methods (GH10378).
- Improved error message when concatenating an empty iterable of `Dataframe` objects (GH9157)
- `pd.read_csv` can now read bz2-compressed files incrementally, and the C parser can read bz2-compressed files from AWS S3 (GH11070, GH11072).
- In `pd.read_csv`, recognize `s3n://` and `s3a://` URLs as designating S3 file storage (GH11070, GH11071).
- Read CSV files from AWS S3 incrementally, instead of first downloading the entire file. (Full file download still required for compressed files in Python 2.) (GH11070, GH11073)
- `pd.read_csv` is now able to infer compression type for files read from AWS S3 storage (GH11070, GH11074).

Backwards incompatible API changes

Changes to sorting API

The sorting API has had some longtime inconsistencies. (GH9816, GH8239).

Here is a summary of the API **PRIOR** to 0.17.0:

- `Series.sort` is **INPLACE** while `DataFrame.sort` returns a new object.
- `Series.order` returns a new object

- It was possible to use `Series/DataFrame.sort_index` to sort by **values** by passing the `by` keyword.
- `Series/DataFrame.sortlevel` worked only on a `MultiIndex` for sorting by index.

To address these issues, we have revamped the API:

- We have introduced a new method, *`DataFrame.sort_values()`*, which is the merger of `DataFrame.sort()`, `Series.sort()`, and `Series.order()`, to handle sorting of **values**.
- The existing methods `Series.sort()`, `Series.order()`, and `DataFrame.sort()` have been deprecated and will be removed in a future version.
- The `by` argument of `DataFrame.sort_index()` has been deprecated and will be removed in a future version.
- The existing method `.sort_index()` will gain the `level` keyword to enable level sorting.

We now have two distinct and non-overlapping methods of sorting. A `*` marks items that will show a `FutureWarning`.

To sort by the **values**:

Previous	Replacement
* `Series.order()`	`Series.sort_values()`
* `Series.sort()`	`Series.sort_values(inplace=True)`
* `DataFrame.sort(columns=...)`	`DataFrame.sort_values(by=...)`

To sort by the **index**:

Previous	Replacement
`Series.sort_index()`	`Series.sort_index()`
`Series.sortlevel(level=...)`	`Series.sort_index(level=...)`
`DataFrame.sort_index()`	`DataFrame.sort_index()`
`DataFrame.sortlevel(level=...)`	`DataFrame.sort_index(level=...)`
* `DataFrame.sort()`	`DataFrame.sort_index()`

We have also deprecated and changed similar methods in two Series-like classes, `Index` and `Categorical`.

Previous	Replacement
* `Index.order()`	`Index.sort_values()`
* `Categorical.order()`	`Categorical.sort_values()`

Changes to to_datetime and to_timedelta

Error handling

The default for `pd.to_datetime` error handling has changed to `errors='raise'`. In prior versions it was `errors='ignore'`. Furthermore, the `coerce` argument has been deprecated in favor of `errors='coerce'`. This means that invalid parsing will raise rather that return the original input as in previous versions. (GH10636)

Previous Behavior:

```
In [2]: pd.to_datetime(['2009-07-31', 'asd'])
Out[2]: array(['2009-07-31', 'asd'], dtype=object)
```

New Behavior:

```
In [3]: pd.to_datetime(['2009-07-31', 'asd'])
ValueError: Unknown string format
```

Of course you can coerce this as well.

```
In [61]: pd.to_datetime(['2009-07-31', 'asd'], errors='coerce')
Out[61]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

To keep the previous behavior, you can use `errors='ignore'`:

```
In [62]: pd.to_datetime(['2009-07-31', 'asd'], errors='ignore')
Out[62]: Index(['2009-07-31', 'asd'], dtype='object')
```

Furthermore, `pd.to_timedelta` has gained a similar API, of `errors='raise'|'ignore'|'coerce'`, and the `coerce` keyword has been deprecated in favor of `errors='coerce'`.

Consistent Parsing

The string parsing of `to_datetime`, `Timestamp` and `DatetimeIndex` has been made consistent. (GH7599)

Prior to v0.17.0, `Timestamp` and `to_datetime` may parse year-only datetime-string incorrectly using today's date, otherwise `DatetimeIndex` uses the beginning of the year. `Timestamp` and `to_datetime` may raise `ValueError` in some types of datetime-string which `DatetimeIndex` can parse, such as a quarterly string.

Previous Behavior:

```
In [1]: pd.Timestamp('2012Q2')
Traceback
   ...
ValueError: Unable to parse 2012Q2

# Results in today's date.
In [2]: pd.Timestamp('2014')
Out [2]: 2014-08-12 00:00:00
```

v0.17.0 can parse them as below. It works on `DatetimeIndex` also.

New Behavior:

```
In [63]: pd.Timestamp('2012Q2')
Out[63]: Timestamp('2012-04-01 00:00:00')

In [64]: pd.Timestamp('2014')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[64]: Timestamp('2014-01-01 00:00:00')

In [65]: pd.DatetimeIndex(['2012Q2', '2014'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[65]:
↪DatetimeIndex(['2012-04-01', '2014-01-01'], dtype='datetime64[ns]', freq=None)
```

Note: If you want to perform calculations based on today's date, use `Timestamp.now()` and `pandas.tseries.offsets`.

```
In [66]: import pandas.tseries.offsets as offsets

In [67]: pd.Timestamp.now()
```

(continues on next page)

(continued from previous page)

```
Out[67]: Timestamp('2019-01-25 16:35:24.831831')

In [68]: pd.Timestamp.now() + offsets.DateOffset(years=1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[68]: Timestamp('2020-01-25␣
↪16:35:24.832741')
```

Changes to Index Comparisons

Operator equal on `Index` should behavior similarly to `Series` (GH9947, GH10637)

Starting in v0.17.0, comparing `Index` objects of different lengths will raise a `ValueError`. This is to be consistent with the behavior of `Series`.

Previous Behavior:

```
In [2]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[2]: array([ True, False, False], dtype=bool)

In [3]: pd.Index([1, 2, 3]) == pd.Index([2])
Out[3]: array([False,  True, False], dtype=bool)

In [4]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
Out[4]: False
```

New Behavior:

```
In [8]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[8]: array([ True, False, False], dtype=bool)

In [9]: pd.Index([1, 2, 3]) == pd.Index([2])
ValueError: Lengths must match to compare

In [10]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
ValueError: Lengths must match to compare
```

Note that this is different from the `numpy` behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([1])
Out[69]: array([ True, False, False], dtype=bool)
```

or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

Changes to Boolean Comparisons vs. None

Boolean comparisons of a `Series` vs `None` will now be equivalent to comparing with `np.nan`, rather than raise `TypeError`. (GH1079).

```
In [71]: s = pd.Series(range(3))
```

(continues on next page)

(continued from previous page)

```
In [72]: s.iloc[1] = None

In [73]: s
Out[73]: 
0    0.0
1    NaN
2    2.0
Length: 3, dtype: float64
```

Previous Behavior:

```
In [5]: s == None
TypeError: Could not compare <type 'NoneType'> type with Series
```

New Behavior:

```
In [74]: s == None
Out[74]: 
0    False
1    False
2    False
Length: 3, dtype: bool
```

Usually you simply want to know which values are null.

```
In [75]: s.isnull()
Out[75]: 
0    False
1     True
2    False
Length: 3, dtype: bool
```

Warning: You generally will want to use `isnull/notnull` for these types of comparisons, as `isnull/notnull` tells you which elements are null. One has to be mindful that `nan's` don't compare equal, but `None's` do. Note that Pandas/numpy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`.

```
In [76]: None == None
Out[76]: True

In [77]: np.nan == np.nan
\\\\\\\\\\\\\Out[77]: False
```

HDFStore dropna behavior

The default behavior for HDFStore write functions with `format='table'` is now to keep rows that are all missing. Previously, the behavior was to drop rows that were all missing save the index. The previous behavior can be replicated using the `dropna=True` option. (GH9382)

Previous Behavior:

```
In [78]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
   ....:                                 'col2': [1, np.nan, np.nan]})
   ....: 
```

(continues on next page)

(continued from previous page)

```
In [79]: df_with_missing
Out[79]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

[3 rows x 2 columns]
```

```
In [27]:
df_with_missing.to_hdf('file.h5',
                       'df_with_missing',
                       format='table',
                       mode='w')

In [28]: pd.read_hdf('file.h5', 'df_with_missing')

Out [28]:
      col1  col2
  0      0     1
  2      2   NaN
```

New Behavior:

```
In [80]: df_with_missing.to_hdf('file.h5',
   ....:                        'df_with_missing',
   ....:                        format='table',
   ....:                        mode='w')
   ....:

In [81]: pd.read_hdf('file.h5', 'df_with_missing')
Out[81]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

[3 rows x 2 columns]
```

See the *docs* for more details.

Changes to `display.precision` option

The `display.precision` option has been clarified to refer to decimal places (GH10451).

Earlier versions of pandas would format floating point numbers to have one less decimal place than the value in `display.precision`.

```
In [1]: pd.set_option('display.precision', 2)

In [2]: pd.DataFrame({'x': [123.456789]})
Out[2]:
       x
0  123.5
```

If interpreting precision as "significant figures" this did work for scientific notation but that same interpretation did not work for values with standard formatting. It was also out of step with how numpy handles formatting.

Going forward the value of `display.precision` will directly control the number of places after the decimal, for regular formatting as well as scientific notation, similar to how numpy's `precision` print option works.

```
In [82]: pd.set_option('display.precision', 2)

In [83]: pd.DataFrame({'x': [123.456789]})
Out[83]:
        x
0  123.46

[1 rows x 1 columns]
```

To preserve output behavior with prior versions the default value of `display.precision` has been reduced to `6` from `7`.

Changes to `Categorical.unique`

`Categorical.unique` now returns new `Categoricals` with `categories` and `codes` that are unique, rather than returning `np.array` (GH10508)

- unordered category: values and categories are sorted by appearance order.
- ordered category: values are sorted by appearance order, categories keep existing order.

```
In [84]: cat = pd.Categorical(['C', 'A', 'B', 'C'],
   ....:                      categories=['A', 'B', 'C'],
   ....:                      ordered=True)
   ....:

In [85]: cat
Out[85]:
[C, A, B, C]
Categories (3, object): [A < B < C]

In [86]: cat.unique()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[86]:
[C, A, B]
Categories (3, object): [A < B < C]

In [87]: cat = pd.Categorical(['C', 'A', 'B', 'C'],
   ....:                      categories=['A', 'B', 'C'])
   ....:

In [88]: cat
Out[88]:
[C, A, B, C]
Categories (3, object): [A, B, C]

In [89]: cat.unique()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[89]:
[C, A, B]
Categories (3, object): [C, A, B]
```

Changes to `bool` passed as `header` in Parsers

In earlier versions of pandas, if a bool was passed the `header` argument of `read_csv`, `read_excel`, or `read_html` it was implicitly converted to an integer, resulting in `header=0` for `False` and `header=1` for `True` (GH6113)

A `bool` input to `header` will now raise a `TypeError`

```
In [29]: df = pd.read_csv('data.csv', header=False)
TypeError: Passing a bool to header is invalid. Use header=None for no header or
header=int or list-like of ints to specify the row(s) making up the column names
```

Other API Changes

- Line and kde plot with `subplots=True` now uses default colors, not all black. Specify `color='k'` to draw all lines in black (GH9894)
- Calling the `.value_counts()` method on a Series with a `categorical` dtype now returns a Series with a `CategoricalIndex` (GH10704)
- The metadata properties of subclasses of pandas objects will now be serialized (GH10553).
- `groupby` using `Categorical` follows the same rule as `Categorical.unique` described above (GH10508)
- When constructing `DataFrame` with an array of `complex64` dtype previously meant the corresponding column was automatically promoted to the `complex128` dtype. Pandas will now preserve the itemsize of the input for complex data (GH10952)
- some numeric reduction operators would return `ValueError`, rather than `TypeError` on object types that includes strings and numbers (GH11131)
- Passing currently unsupported `chunksize` argument to `read_excel` or `ExcelFile.parse` will now raise `NotImplementedError` (GH8011)
- Allow an `ExcelFile` object to be passed into `read_excel` (GH11198)
- `DatetimeIndex.union` does not infer `freq` if `self` and the input have `None` as `freq` (GH11086)
- `NaT`'s methods now either raise `ValueError`, or return `np.nan` or `NaT` (GH9513)

Behavior	Methods
return `np.nan`	`weekday`, `isoweekday`
return `NaT`	`date`, `now`, `replace`, `to_datetime`, `today`
return `np.datetime64('NaT')`	`to_datetime64` (unchanged)
raise `ValueError`	All other public methods (names not beginning with underscores)

Deprecations

- For `Series` the following indexing functions are deprecated (GH10177).

Deprecated Function	Replacement
`.irow(i)`	`.iloc[i]` or `.iat[i]`
`.iget(i)`	`.iloc[i]` or `.iat[i]`
`.iget_value(i)`	`.iloc[i]` or `.iat[i]`

- For `DataFrame` the following indexing functions are deprecated (GH10177).

Deprecated Function	Replacement
`.irow(i)`	`.iloc[i]`
`.iget_value(i, j)`	`.iloc[i, j]` or `.iat[i, j]`
`.icol(j)`	`.iloc[:, j]`

Note: These indexing function have been deprecated in the documentation since 0.11.0.

- `Categorical.name` was deprecated to make `Categorical` more `numpy.ndarray` like. Use `Series(cat, name="whatever")` instead (GH10482).
- Setting missing values (NaN) in a `Categorical`'s `categories` will issue a warning (GH10748). You can still have missing values in the `values`.
- `drop_duplicates` and `duplicated`'s `take_last` keyword was deprecated in favor of `keep`. (GH6511, GH8505)
- `Series.nsmallest` and `nlargest`'s `take_last` keyword was deprecated in favor of `keep`. (GH10792)
- `DataFrame.combineAdd` and `DataFrame.combineMult` are deprecated. They can easily be replaced by using the `add` and `mul` methods: `DataFrame.add(other, fill_value=0)` and `DataFrame.mul(other, fill_value=1.)` (GH10735).
- `TimeSeries` deprecated in favor of `Series` (note that this has been an alias since 0.13.0), (GH10890)
- `SparsePanel` deprecated and will be removed in a future version (GH11157).
- `Series.is_time_series` deprecated in favor of `Series.index.is_all_dates` (GH11135)
- Legacy offsets (like `'A@JAN'`) are deprecated (note that this has been alias since 0.8.0) (GH10878)
- `WidePanel` deprecated in favor of `Panel`, `LongPanel` in favor of `DataFrame` (note these have been aliases since < 0.11.0), (GH10892)
- `DataFrame.convert_objects` has been deprecated in favor of type-specific functions `pd.to_datetime`, `pd.to_timestamp` and `pd.to_numeric` (new in 0.17.0) (GH11133).

Removal of prior version deprecations/changes

- Removal of `na_last` parameters from `Series.order()` and `Series.sort()`, in favor of `na_position`. (GH5231)
- Remove of `percentile_width` from `.describe()`, in favor of `percentiles`. (GH7088)
- Removal of `colSpace` parameter from `DataFrame.to_string()`, in favor of `col_space`, circa 0.8.0 version.
- Removal of automatic time-series broadcasting (GH2304)

```
In [90]: np.random.seed(1234)

In [91]: df = pd.DataFrame(np.random.randn(5, 2),
   ....:                   columns=list('AB'),
   ....:                   index=pd.date_range('2013-01-01', periods=5))
   ....:
```

(continues on next page)

(continued from previous page)

```
In [92]: df
Out[92]:
                   A         B
2013-01-01  0.471435 -1.190976
2013-01-02  1.432707 -0.312652
2013-01-03 -0.720589  0.887163
2013-01-04  0.859588 -0.636524
2013-01-05  0.015696 -2.242685

[5 rows x 2 columns]
```

Previously

```
In [3]: df + df.A
FutureWarning: TimeSeries broadcasting along DataFrame index by default is␣
↪deprecated.
Please use DataFrame.<op> to explicitly broadcast arithmetic operations along the␣
↪index

Out[3]:
                    A         B
2013-01-01  0.942870 -0.719541
2013-01-02  2.865414  1.120055
2013-01-03 -1.441177  0.166574
2013-01-04  1.719177  0.223065
2013-01-05  0.031393 -2.226989
```

Current

```
In [93]: df.add(df.A, axis='index')
Out[93]:
                   A         B
2013-01-01  0.942870 -0.719541
2013-01-02  2.865414  1.120055
2013-01-03 -1.441177  0.166574
2013-01-04  1.719177  0.223065
2013-01-05  0.031393 -2.226989

[5 rows x 2 columns]
```

- Remove `table` keyword in `HDFStore.put/append`, in favor of using `format=` (GH4645)
- Remove `kind` in `read_excel/ExcelFile` as its unused (GH4712)
- Remove `infer_type` keyword from `pd.read_html` as its unused (GH4770, GH7032)
- Remove `offset` and `timeRule` keywords from `Series.tshift/shift`, in favor of `freq` (GH4853, GH4864)
- Remove `pd.load/pd.save` aliases in favor of `pd.to_pickle/pd.read_pickle` (GH3787)

Performance Improvements

- Development support for benchmarking with the Air Speed Velocity library (GH8361)
- Added vbench benchmarks for alternative ExcelWriter engines and reading Excel files (GH7171)
- Performance improvements in `Categorical.value_counts` (GH10804)

- Performance improvements in SeriesGroupBy.nunique and SeriesGroupBy.value_counts and SeriesGroupby.transform (GH10820, GH11077)
- Performance improvements in DataFrame.drop_duplicates with integer dtypes (GH10917)
- Performance improvements in DataFrame.duplicated with wide frames. (GH10161, GH11180)
- 4x improvement in timedelta string parsing (GH6755, GH10426)
- 8x improvement in timedelta64 and datetime64 ops (GH6755)
- Significantly improved performance of indexing MultiIndex with slicers (GH10287)
- 8x improvement in iloc using list-like input (GH10791)
- Improved performance of Series.isin for datetimelike/integer Series (GH10287)
- 20x improvement in concat of Categoricals when categories are identical (GH10587)
- Improved performance of to_datetime when specified format string is ISO8601 (GH10178)
- 2x improvement of Series.value_counts for float dtype (GH10821)
- Enable infer_datetime_format in to_datetime when date components do not have 0 padding (GH11142)
- Regression from 0.16.1 in constructing DataFrame from nested dictionary (GH11084)
- Performance improvements in addition/subtraction operations for DateOffset with Series or DatetimeIndex (GH10744, GH11205)

Bug Fixes

- Bug in incorrect computation of .mean() on timedelta64[ns] because of overflow (GH9442)
- Bug in .isin on older numpies (GH11232)
- Bug in DataFrame.to_html(index=False) renders unnecessary name row (GH10344)
- Bug in DataFrame.to_latex() the column_format argument could not be passed (GH9402)
- Bug in DatetimeIndex when localizing with NaT (GH10477)
- Bug in Series.dt ops in preserving meta-data (GH10477)
- Bug in preserving NaT when passed in an otherwise invalid to_datetime construction (GH10477)
- Bug in DataFrame.apply when function returns categorical series. (GH9573)
- Bug in to_datetime with invalid dates and formats supplied (GH10154)
- Bug in Index.drop_duplicates dropping name(s) (GH10115)
- Bug in Series.quantile dropping name (GH10881)
- Bug in pd.Series when setting a value on an empty Series whose index has a frequency. (GH10193)
- Bug in pd.Series.interpolate with invalid order keyword values. (GH10633)
- Bug in DataFrame.plot raises ValueError when color name is specified by multiple characters (GH10387)
- Bug in Index construction with a mixed list of tuples (GH10697)
- Bug in DataFrame.reset_index when index contains NaT. (GH10388)
- Bug in ExcelReader when worksheet is empty (GH6403)

- Bug in `BinGrouper.group_info` where returned values are not compatible with base class (GH10914)
- Bug in clearing the cache on `DataFrame.pop` and a subsequent inplace op (GH10912)
- Bug in indexing with a mixed-integer `Index` causing an `ImportError` (GH10610)
- Bug in `Series.count` when index has nulls (GH10946)
- Bug in pickling of a non-regular freq `DatetimeIndex` (GH11002)
- Bug causing `DataFrame.where` to not respect the `axis` parameter when the frame has a symmetric shape. (GH9736)
- Bug in `Table.select_column` where name is not preserved (GH10392)
- Bug in `offsets.generate_range` where `start` and `end` have finer precision than `offset` (GH9907)
- Bug in `pd.rolling_*` where `Series.name` would be lost in the output (GH10565)
- Bug in `stack` when index or columns are not unique. (GH10417)
- Bug in setting a `Panel` when an axis has a MultiIndex (GH10360)
- Bug in `USFederalHolidayCalendar` where `USMemorialDay` and `USMartinLutherKingJr` were incorrect (GH10278 and GH9760)
- Bug in `.sample()` where returned object, if set, gives unnecessary `SettingWithCopyWarning` (GH10738)
- Bug in `.sample()` where weights passed as `Series` were not aligned along axis before being treated positionally, potentially causing problems if weight indices were not aligned with sampled object. (GH10738)
- Regression fixed in (GH9311, GH6620, GH9345), where groupby with a datetime-like converting to float with certain aggregators (GH10979)
- Bug in `DataFrame.interpolate` with `axis=1` and `inplace=True` (GH10395)
- Bug in `io.sql.get_schema` when specifying multiple columns as primary key (GH10385).
- Bug in `groupby(sort=False)` with datetime-like `Categorical` raises `ValueError` (GH10505)
- Bug in `groupby(axis=1)` with `filter()` throws `IndexError` (GH11041)
- Bug in `test_categorical` on big-endian builds (GH10425)
- Bug in `Series.shift` and `DataFrame.shift` not supporting categorical data (GH9416)
- Bug in `Series.map` using categorical `Series` raises `AttributeError` (GH10324)
- Bug in `MultiIndex.get_level_values` including `Categorical` raises `AttributeError` (GH10460)
- Bug in `pd.get_dummies` with `sparse=True` not returning `SparseDataFrame` (GH10531)
- Bug in `Index` subtypes (such as `PeriodIndex`) not returning their own type for `.drop` and `.insert` methods (GH10620)
- Bug in `algos.outer_join_indexer` when `right` array is empty (GH10618)
- Bug in `filter` (regression from 0.16.0) and `transform` when grouping on multiple keys, one of which is datetime-like (GH10114)
- Bug in `to_datetime` and `to_timedelta` causing `Index` name to be lost (GH10875)
- Bug in `len(DataFrame.groupby)` causing `IndexError` when there's a column containing only NaNs (GH11016)
- Bug that caused segfault when resampling an empty Series (GH10228)

- Bug in `DatetimeIndex` and `PeriodIndex.value_counts` resets name from its result, but retains in result's `Index`. (GH10150)
- Bug in `pd.eval` using `numexpr` engine coerces 1 element numpy array to scalar (GH10546)
- Bug in `pd.concat` with `axis=0` when column is of dtype `category` (GH10177)
- Bug in `read_msgpack` where input type is not always checked (GH10369, GH10630)
- Bug in `pd.read_csv` with kwargs `index_col=False`, `index_col=['a', 'b']` or `dtype` (GH10413, GH10467, GH10577)
- Bug in `Series.from_csv` with `header` kwarg not setting the `Series.name` or the `Series.index.name` (GH10483)
- Bug in `groupby.var` which caused variance to be inaccurate for small float values (GH10448)
- Bug in `Series.plot(kind='hist')` Y Label not informative (GH10485)
- Bug in `read_csv` when using a converter which generates a `uint8` type (GH9266)
- Bug causes memory leak in time-series line and area plot (GH9003)
- Bug when setting a `Panel` sliced along the major or minor axes when the right-hand side is a `DataFrame` (GH11014)
- Bug that returns `None` and does not raise `NotImplementedError` when operator functions (e.g. `.add`) of `Panel` are not implemented (GH7692)
- Bug in line and kde plot cannot accept multiple colors when `subplots=True` (GH9894)
- Bug in `DataFrame.plot` raises `ValueError` when color name is specified by multiple characters (GH10387)
- Bug in left and right `align` of `Series` with `MultiIndex` may be inverted (GH10665)
- Bug in left and right `join` of with `MultiIndex` may be inverted (GH10741)
- Bug in `read_stata` when reading a file with a different order set in `columns` (GH10757)
- Bug in `Categorical` may not representing properly when category contains `tz` or `Period` (GH10713)
- Bug in `Categorical.__iter__` may not returning correct `datetime` and `Period` (GH10713)
- Bug in indexing with a `PeriodIndex` on an object with a `PeriodIndex` (GH4125)
- Bug in `read_csv` with `engine='c'`: EOF preceded by a comment, blank line, etc. was not handled correctly (GH10728, GH10548)
- Reading "famafrench" data via `DataReader` results in HTTP 404 error because of the website url is changed (GH10591).
- Bug in `read_msgpack` where DataFrame to decode has duplicate column names (GH9618)
- Bug in `io.common.get_filepath_or_buffer` which caused reading of valid S3 files to fail if the bucket also contained keys for which the user does not have read permission (GH10604)
- Bug in vectorised setting of timestamp columns with python `datetime.date` and numpy `datetime64` (GH10408, GH10412)
- Bug in `Index.take` may add unnecessary `freq` attribute (GH10791)
- Bug in `merge` with empty `DataFrame` may raise `IndexError` (GH10824)
- Bug in `to_latex` where unexpected keyword argument for some documented arguments (GH10888)
- Bug in indexing of large `DataFrame` where `IndexError` is uncaught (GH10645 and GH10692)

- Bug in `read_csv` when using the `nrows` or `chunksize` parameters if file contains only a header line (GH9535)
- Bug in serialization of `category` types in HDF5 in presence of alternate encodings. (GH10366)
- Bug in `pd.DataFrame` when constructing an empty DataFrame with a string dtype (GH9428)
- Bug in `pd.DataFrame.diff` when DataFrame is not consolidated (GH10907)
- Bug in `pd.unique` for arrays with the `datetime64` or `timedelta64` dtype that meant an array with object dtype was returned instead the original dtype (GH9431)
- Bug in `Timedelta` raising error when slicing from 0s (GH10583)
- Bug in `DatetimeIndex.take` and `TimedeltaIndex.take` may not raise `IndexError` against invalid index (GH10295)
- Bug in `Series([np.nan]).astype('M8[ms]')`, which now returns `Series([pd.NaT])` (GH10747)
- Bug in `PeriodIndex.order` reset freq (GH10295)
- Bug in `date_range` when `freq` divides `end` as nanos (GH10885)
- Bug in `iloc` allowing memory outside bounds of a Series to be accessed with negative integers (GH10779)
- Bug in `read_msgpack` where encoding is not respected (GH10581)
- Bug preventing access to the first index when using `iloc` with a list containing the appropriate negative integer (GH10547, GH10779)
- Bug in `TimedeltaIndex` formatter causing error while trying to save `DataFrame` with `TimedeltaIndex` using `to_csv` (GH10833)
- Bug in `DataFrame.where` when handling Series slicing (GH10218, GH9558)
- Bug where `pd.read_gbq` throws `ValueError` when Bigquery returns zero rows (GH10273)
- Bug in `to_json` which was causing segmentation fault when serializing 0-rank ndarray (GH9576)
- Bug in plotting functions may raise `IndexError` when plotted on `GridSpec` (GH10819)
- Bug in plot result may show unnecessary minor ticklabels (GH10657)
- Bug in `groupby` incorrect computation for aggregation on `DataFrame` with `NaT` (E.g `first`, `last`, `min`). (GH10590, GH11010)
- Bug when constructing `DataFrame` where passing a dictionary with only scalar values and specifying columns did not raise an error (GH10856)
- Bug in `.var()` causing roundoff errors for highly similar values (GH10242)
- Bug in `DataFrame.plot(subplots=True)` with duplicated columns outputs incorrect result (GH10962)
- Bug in `Index` arithmetic may result in incorrect class (GH10638)
- Bug in `date_range` results in empty if freq is negative annually, quarterly and monthly (GH11018)
- Bug in `DatetimeIndex` cannot infer negative freq (GH11018)
- Remove use of some deprecated numpy comparison operations, mainly in tests. (GH10569)
- Bug in `Index` dtype may not applied properly (GH11017)
- Bug in `io.gbq` when testing for minimum google api client version (GH10652)
- Bug in `DataFrame` construction from nested `dict` with `timedelta` keys (GH11129)
- Bug in `.fillna` against may raise `TypeError` when data contains datetime dtype (GH7095, GH11153)

- Bug in .groupby when number of keys to group by is same as length of index (GH11185)
- Bug in convert_objects where converted values might not be returned if all null and coerce (GH9589)
- Bug in convert_objects where copy keyword was not respected (GH9589)

Contributors

A total of 112 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Alex Rothberg
- Andrea Bedini +
- Andrew Rosenfeld
- Andy Hayden
- Andy Li +
- Anthonios Partheniou +
- Artemy Kolchinsky
- Bernard Willers
- Charlie Clark +
- Chris +
- Chris Whelan
- Christoph Gohlke +
- Christopher Whelan
- Clark Fitzgerald
- Clearfield Christopher +
- Dan Ringwalt +
- Daniel Ni +
- Data & Code Expert Experimenting with Code on Data +
- David Cottrell
- David John Gagne +
- David Kelly +
- ETF +
- Eduardo Schettino +
- Egor +
- Egor Panfilov +
- Evan Wright
- Frank Pinter +
- Gabriel Araujo +
- Garrett-R

- Gianluca Rossi +
- Guillaume Gay
- Guillaume Poulin
- Harsh Nisar +
- Ian Henriksen +
- Ian Hoegen +
- Jaidev Deshpande +
- Jan Rudolph +
- Jan Schulz
- Jason Swails +
- Jeff Reback
- Jonas Buyl +
- Joris Van den Bossche
- Joris Vankerschaver +
- Josh Levy-Kramer +
- Julien Danjou
- Ka Wo Chen
- Karrie Kehoe +
- Kelsey Jordahl
- Kerby Shedden
- Kevin Sheppard
- Lars Buitinck
- Leif Johnson +
- Luis Ortiz +
- Mac +
- Matt Gambogi +
- Matt Savoie +
- Matthew Gilbert +
- Maximilian Roos +
- Michelangelo D'Agostino +
- Mortada Mehyar
- Nick Eubank
- Nipun Batra
- Ondřej Čertík
- Phillip Cloud
- Pratap Vardhan +

- Rafal Skolasinski +
- Richard Lewis +
- Rinoc Johnson +
- Rob Levy
- Robert Gieseke
- Safia Abdalla +
- Samuel Denny +
- Saumitra Shahapure +
- Sebastian Pölsterl +
- Sebastian Rubbert +
- Sheppard, Kevin +
- Sinhrks
- Siu Kwan Lam +
- Skipper Seabold
- Spencer Carrucciu +
- Stephan Hoyer
- Stephen Hoover +
- Stephen Pascoe +
- Terry Santegoeds +
- Thomas Grainger
- Tjerk Santegoeds +
- Tom Augspurger
- Vincent Davis +
- Winterflower +
- Yaroslav Halchenko
- Yuan Tang (Terry) +
- agijsberts
- ajcr +
- behzad nouri
- cel4
- chris-b1 +
- cyrusmaher +
- davidovitch +
- ganego +
- jreback
- juricast +

- larvian +
- maximilianr +
- msund +
- rekcahpassyla
- robertzk +
- scls19fr
- seth-p
- sinhrks
- springcoil +
- terrytangyuan +
- tzinckgraf +

8.9 Version 0.16

8.9.1 v0.16.2 (June 12, 2015)

This is a minor bug-fix release from 0.16.1 and includes a a large number of bug fixes along some new features (`pipe()` method), enhancements, and performance improvements.

We recommend that all users upgrade to this version.

Highlights include:

- A new `pipe` method, see *here*
- Documentation on how to use numba with *pandas*, see *here*

What's new in v0.16.2

- *New features*
  - *Pipe*
  - *Other Enhancements*
- *API Changes*
- *Performance Improvements*
- *Bug Fixes*
- *Contributors*

New features

Pipe

We've introduced a new method `DataFrame.pipe()`. As suggested by the name, `pipe` should be used to pipe data through a chain of function calls. The goal is to avoid confusing nested function calls like

```
# df is a DataFrame
# f, g, and h are functions that take and return DataFrames
f(g(h(df), arg1=1), arg2=2, arg3=3)  # noqa F821
```

The logic flows from inside out, and function names are separated from their keyword arguments. This can be rewritten as

```
(df.pipe(h)                      # noqa F821
   .pipe(g, arg1=1)              # noqa F821
   .pipe(f, arg2=2, arg3=3)      # noqa F821
)
```

Now both the code and the logic flow from top to bottom. Keyword arguments are next to their functions. Overall the code is much more readable.

In the example above, the functions `f`, `g`, and `h` each expected the DataFrame as the first positional argument. When the function you wish to apply takes its data anywhere other than the first argument, pass a tuple of `(function, keyword)` indicating where the DataFrame should flow. For example:

```
In [1]: import statsmodels.formula.api as sm

In [2]: bb = pd.read_csv('data/baseball.csv', index_col='id')

# sm.ols takes (formula, data)
In [3]: (bb.query('h > 0')
   ...:     .assign(ln_h=lambda df: np.log(df.h))
   ...:     .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
   ...:     .fit()
   ...:     .summary()
   ...:  )
   ...:
Out[3]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                     hr   R-squared:                       0.685
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Fri, 25 Jan 2019   Prob (F-statistic):           3.48e-15
Time:                        16:35:23   Log-Likelihood:                -205.92
No. Observations:                  68   AIC:                             421.8
Df Residuals:                      63   BIC:                             432.9
Df Model:                           4
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept   -8484.7720   4664.146     -1.819      0.074   -1.78e+04     835.780
C(lg)[T.NL]    -2.2736      1.325     -1.716      0.091      -4.922       0.375
ln_h           -1.3542      0.875     -1.547      0.127      -3.103       0.395
year            4.2277      2.324      1.819      0.074      -0.417       8.872
g               0.1841      0.029      6.258      0.000       0.125       0.243
===============================================================================
Omnibus:                       10.875   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               17.298
Skew:                           0.537   Prob(JB):                     0.000175
```

(continues on next page)

(continued from previous page)

```
Kurtosis:                       5.225   Cond. No.                     1.49e+07
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

The pipe method is inspired by unix pipes, which stream text through processes. More recently dplyr and magrittr have introduced the popular `(%>%)` pipe operator for R.

See the *documentation* for more. (GH10129)

Other Enhancements

- Added *rsplit* to Index/Series StringMethods (GH10303)
- Removed the hard-coded size limits on the `DataFrame` HTML representation in the IPython notebook, and leave this to IPython itself (only for IPython v3.0 or greater). This eliminates the duplicate scroll bars that appeared in the notebook with large frames (GH10231).

  Note that the notebook has a `toggle output scrolling` feature to limit the display of very large frames (by clicking left of the output). You can also configure the way DataFrames are displayed using the pandas options, see here *here*.
- `axis` parameter of `DataFrame.quantile` now accepts also `index` and `column`. (GH9543)

API Changes

- `Holiday` now raises `NotImplementedError` if both `offset` and `observance` are used in the constructor instead of returning an incorrect result (GH10217).

Performance Improvements

- Improved `Series.resample` performance with `dtype=datetime64[ns]` (GH7754)
- Increase performance of `str.split` when `expand=True` (GH10081)

Bug Fixes

- Bug in `Series.hist` raises an error when a one row `Series` was given (GH10214)
- Bug where `HDFStore.select` modifies the passed columns list (GH7212)
- Bug in `Categorical` repr with `display.width` of `None` in Python 3 (GH10087)
- Bug in `to_json` with certain orients and a `CategoricalIndex` would segfault (GH10317)
- Bug where some of the nan functions do not have consistent return dtypes (GH10251)
- Bug in `DataFrame.quantile` on checking that a valid axis was passed (GH9543)
- Bug in `groupby.apply` aggregation for `Categorical` not preserving categories (GH10138)

- Bug in `to_csv` where `date_format` is ignored if the `datetime` is fractional (GH10209)
- Bug in `DataFrame.to_json` with mixed data types (GH10289)
- Bug in cache updating when consolidating (GH10264)
- Bug in `mean()` where integer dtypes can overflow (GH10172)
- Bug where `Panel.from_dict` does not set dtype when specified (GH10058)
- Bug in `Index.union` raises `AttributeError` when passing array-likes. (GH10149)
- Bug in `Timestamp`'s' `microsecond`, `quarter`, `dayofyear`, `week` and `daysinmonth` properties return `np.int` type, not built-in `int`. (GH10050)
- Bug in `NaT` raises `AttributeError` when accessing to `daysinmonth`, `dayofweek` properties. (GH10096)
- Bug in Index repr when using the `max_seq_items=None` setting (GH10182).
- Bug in getting timezone data with `dateutil` on various platforms (GH9059, GH8639, GH9663, GH10121)
- Bug in displaying datetimes with mixed frequencies; display 'ms' datetimes to the proper precision. (GH10170)
- Bug in `setitem` where type promotion is applied to the entire block (GH10280)
- Bug in `Series` arithmetic methods may incorrectly hold names (GH10068)
- Bug in `GroupBy.get_group` when grouping on multiple keys, one of which is categorical. (GH10132)
- Bug in `DatetimeIndex` and `TimedeltaIndex` names are lost after timedelta arithmetics (GH9926)
- Bug in `DataFrame` construction from nested `dict` with `datetime64` (GH10160)
- Bug in `Series` construction from `dict` with `datetime64` keys (GH9456)
- Bug in `Series.plot(label="LABEL")` not correctly setting the label (GH10119)
- Bug in `plot` not defaulting to matplotlib `axes.grid` setting (GH9792)
- Bug causing strings containing an exponent, but no decimal to be parsed as `int` instead of `float` in `engine='python'` for the `read_csv` parser (GH9565)
- Bug in `Series.align` resets `name` when `fill_value` is specified (GH10067)
- Bug in `read_csv` causing index name not to be set on an empty DataFrame (GH10184)
- Bug in `SparseSeries.abs` resets `name` (GH10241)
- Bug in `TimedeltaIndex` slicing may reset freq (GH10292)
- Bug in `GroupBy.get_group` raises `ValueError` when group key contains `NaT` (GH6992)
- Bug in `SparseSeries` constructor ignores input data name (GH10258)
- Bug in `Categorical.remove_categories` causing a `ValueError` when removing the `NaN` category if underlying dtype is floating-point (GH10156)
- Bug where infer_freq infers time rule (WOM-5XXX) unsupported by to_offset (GH9425)
- Bug in `DataFrame.to_hdf()` where table format would raise a seemingly unrelated error for invalid (non-string) column names. This is now explicitly forbidden. (GH9057)
- Bug to handle masking empty `DataFrame` (GH10126).
- Bug where MySQL interface could not handle numeric table/column names (GH10255)
- Bug in `read_csv` with a `date_parser` that returned a `datetime64` array of other time resolution than `[ns]` (GH10245)

- Bug in `Panel.apply` when the result has ndim=0 (GH10332)
- Bug in `read_hdf` where `auto_close` could not be passed (GH9327).
- Bug in `read_hdf` where open stores could not be used (GH10330).
- Bug in adding empty `DataFrames`, now results in a `DataFrame` that `.equals` an empty `DataFrame` (GH10181).
- Bug in `to_hdf` and `HDFStore` which did not check that complib choices were valid (GH4582, GH8874).

Contributors

A total of 34 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Andrew Rosenfeld
- Artemy Kolchinsky
- Bernard Willers +
- Christer van der Meeren
- Christian Hudon +
- Constantine Glen Evans +
- Daniel Julius Lasiman +
- Evan Wright
- Francesco Brundu +
- Gaëtan de Menten +
- Jake VanderPlas
- James Hiebert +
- Jeff Reback
- Joris Van den Bossche
- Justin Lecher +
- Ka Wo Chen +
- Kevin Sheppard
- Mortada Mehyar
- Morton Fox +
- Robin Wilson +
- Sinhrks
- Stephan Hoyer
- Thomas Grainger
- Tom Ajamian
- Tom Augspurger
- Yoshiki Vázquez Baeza
- Younggun Kim

- austinc +
- behzad nouri
- jreback
- lexual
- rekcahpassyla +
- scls19fr
- sinhrks

8.9.2 v0.16.1 (May 11, 2015)

This is a minor bug-fix release from 0.16.0 and includes a a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Support for a `CategoricalIndex`, a category based index, see *here*
- New section on how-to-contribute to *pandas*, see *here*
- Revised "Merge, join, and concatenate" documentation, including graphical examples to make it easier to understand each operations, see *here*
- New method `sample` for drawing random samples from Series, DataFrames and Panels. See *here*
- The default `Index` printing has changed to a more uniform format, see *here*
- `BusinessHour` datetime-offset is now supported, see *here*
- Further enhancement to the `.str` accessor to make string operations easier, see *here*

What's new in v0.16.1

Warning: In pandas 0.17.0, the sub-package `pandas.io.data` will be removed in favor of a separately installable package (GH8961).

Enhancements

CategoricalIndex

We introduce a `CategoricalIndex`, a new type of index object that is useful for supporting indexing with duplicates. This is a container around a `Categorical` (introduced in v0.15.0) and allows efficient indexing and storage of an index with a large number of duplicated elements. Prior to 0.16.1, setting the index of a `DataFrame/Series` with a `category` dtype would convert this to regular object-based `Index`.

```
In [1]: df = pd.DataFrame({'A': np.arange(6),
   ...:                     'B': pd.Series(list('aabbca'))
   ...:                            .astype('category', categories=list('cab'))
   ...:                     })
   ...:

In [2]: df
Out[2]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [3]: df.dtypes
Out[3]:
A       int64
B    category
dtype: object

In [4]: df.B.cat.categories
Out[4]: Index(['c', 'a', 'b'], dtype='object')
```

setting the index, will create create a `CategoricalIndex`

```
In [5]: df2 = df.set_index('B')

In [6]: df2.index
Out[6]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],␣
↪ordered=False, name='B', dtype='category')
```

indexing with `__getitem__/.iloc/.loc/.ix` works similarly to an Index with duplicates. The indexers MUST be in the category or the operation will raise.

```
In [7]: df2.loc['a']
Out[7]:
   A
B
a  0
a  1
a  5
```

and preserves the `CategoricalIndex`

```
In [8]: df2.loc['a'].index
Out[8]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,␣
↪name='B', dtype='category')
```

(continues on next page)

(continued from previous page)

```

```

sorting will order by the order of the categories

```
In [9]: df2.sort_index()
Out[9]:
   A
B
c  4
a  0
a  1
a  5
b  2
b  3
```

groupby operations on the index will preserve the index nature as well

```
In [10]: df2.groupby(level=0).sum()
Out[10]:
   A
B
c  4
a  6
b  5

In [11]: df2.groupby(level=0).sum().index
Out[11]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False,␣
↪name='B', dtype='category')
```

reindexing operations, will return a resulting index based on the type of the passed indexer, meaning that passing a list will return a plain-old-`Index`; indexing with a `Categorical` will return a `CategoricalIndex`, indexed according to the categories of the PASSED `Categorical` dtype. This allows one to arbitrarily index these even with values NOT in the categories, similarly to how you can reindex ANY pandas index.

```
In [12]: df2.reindex(['a', 'e'])
Out[12]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [13]: df2.reindex(['a', 'e']).index
Out[13]: pd.Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [14]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde')))
Out[14]:
     A
B
a  0.0
a  1.0
a  5.0
e  NaN

In [15]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde'))).index
Out[15]: pd.CategoricalIndex(['a', 'a', 'a', 'e'],
```

(continues on next page)

(continued from previous page)

```
                                categories=['a', 'b', 'c', 'd', 'e'],
                                ordered=False, name='B',
                                dtype='category')
```

See the *documentation* for more. (GH7629, GH10038, GH10039)

Sample

Series, DataFrames, and Panels now have a new method: `sample()`. The method accepts a specific number of rows or columns to return, or a fraction of the total number or rows or columns. It also has options for sampling with or without replacement, for passing in a column for weights for non-uniform sampling, and for setting seed values to facilitate replication. (GH2419)

```
In [1]: example_series = pd.Series([0, 1, 2, 3, 4, 5])

# When no arguments are passed, returns 1
In [2]: example_series.sample()
Out[2]:
3    3
Length: 1, dtype: int64

# One may specify either a number of rows:
In [3]: example_series.sample(n=3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]:
2    2
1    1
0    0
Length: 3, dtype: int64

# Or a fraction of the rows:
In [4]: example_series.sample(frac=0.5)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]
↪
1    1
5    5
3    3
Length: 3, dtype: int64

# weights are accepted.
In [5]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [6]: example_series.sample(n=3, weights=example_weights)
Out[6]:
2    2
4    4
3    3
Length: 3, dtype: int64

# weights will also be normalized if they do not sum to one,
# and missing values will be treated as zeros.
In [7]: example_weights2 = [0.5, 0, 0, 0, None, np.nan]

In [8]: example_series.sample(n=1, weights=example_weights2)
Out[8]:
0    0
```

(continues on next page)

(continued from previous page)

```
Length: 1, dtype: int64
```

When applied to a DataFrame, one may pass the name of a column to specify sampling weights when sampling from rows.

```
In [9]: df = pd.DataFrame({'col1': [9, 8, 7, 6],
   ...:                    'weight_column': [0.5, 0.4, 0.1, 0]})
   ...:

In [10]: df.sample(n=3, weights='weight_column')
Out[10]:
   col1  weight_column
0     9            0.5
1     8            0.4
2     7            0.1

[3 rows x 2 columns]
```

String Methods Enhancements

Continuing from v0.16.0, the following enhancements make string operations easier and more consistent with standard python string operations.

- Added `StringMethods` (`.str` accessor) to `Index` (GH9068)

  The `.str` accessor is now available for both `Series` and `Index`.

  ```
  In [11]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

  In [12]: idx.str.strip()
  Out[12]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')
  ```

  One special case for the *.str* accessor on `Index` is that if a string method returns `bool`, the `.str` accessor will return a `np.array` instead of a boolean `Index` (GH8875). This enables the following expression to work naturally:

  ```
  In [13]: idx = pd.Index(['a1', 'a2', 'b1', 'b2'])

  In [14]: s = pd.Series(range(4), index=idx)

  In [15]: s
  Out[15]:
  a1    0
  a2    1
  b1    2
  b2    3
  Length: 4, dtype: int64

  In [16]: idx.str.startswith('a')
  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16]:␣
  ↪array([ True,  True, False, False], dtype=bool)

  In [17]: s[s.index.str.startswith('a')]
  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  ↪
  ```

(continues on next page)

(continued from previous page)

```
a1    0
a2    1
Length: 2, dtype: int64
```

- The following new methods are accessible via `.str` accessor to apply the function to each values. (GH9766, GH9773, GH10031, GH10045, GH10052)

		Methods		
capitalize()	swapcase()	normalize()	partition()	rpartition()
index()	rindex()	translate()		

- `split` now takes `expand` keyword to specify whether to expand dimensionality. `return_type` is deprecated. (GH9847)

```
In [18]: s = pd.Series(['a,b', 'a,c', 'b,c'])

# return Series
In [19]: s.str.split(',')
Out[19]:
0    [a, b]
1    [a, c]
2    [b, c]
Length: 3, dtype: object

# return DataFrame
In [20]: s.str.split(',', expand=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]:
   0  1
0  a  b
1  a  c
2  b  c

[3 rows x 2 columns]

In [21]: idx = pd.Index(['a,b', 'a,c', 'b,c'])

# return Index
In [22]: idx.str.split(',')
Out[22]: Index([['a', 'b'], ['a', 'c'], ['b', 'c']], dtype='object')

# return MultiIndex
In [23]: idx.str.split(',', expand=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:
MultiIndex(levels=[['a', 'b'], ['b', 'c']],
           codes=[[0, 0, 1], [0, 1, 1]])
```

- Improved `extract` and `get_dummies` methods for `Index.str` (GH9980)

Other Enhancements

- `BusinessHour` offset is now supported, which represents business hours starting from 09:00 - 17:00 on `BusinessDay` by default. See *Here* for details. (GH7905)

```
In [24]: pd.Timestamp('2014-08-01 09:00') + pd.tseries.offsets.BusinessHour()
Out[24]: Timestamp('2014-08-01 10:00:00')

In [25]: pd.Timestamp('2014-08-01 07:00') + pd.tseries.offsets.BusinessHour()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]: Timestamp('2014-08-01 10:00:00
↪')

In [26]: pd.Timestamp('2014-08-01 16:30') + pd.tseries.offsets.BusinessHour()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]:␣
↪Timestamp('2014-08-04 09:30:00')
```

- `DataFrame.diff` now takes an `axis` parameter that determines the direction of differencing (GH9727)
- Allow `clip`, `clip_lower`, and `clip_upper` to accept array-like arguments as thresholds (This is a regression from 0.11.0). These methods now have an `axis` parameter which determines how the Series or DataFrame will be aligned with the threshold(s). (GH6966)
- `DataFrame.mask()` and `Series.mask()` now support same keywords as `where` (GH8801)
- `drop` function can now accept `errors` keyword to suppress `ValueError` raised when any of label does not exist in the target data. (GH6736)

```
In [27]: df = pd.DataFrame(np.random.randn(3, 3), columns=['A', 'B', 'C'])

In [28]: df.drop(['A', 'X'], axis=1, errors='ignore')
Out[28]:
          B         C
0 -0.706771 -1.039575
1 -0.424972  0.567020
2 -1.087401 -0.673690

[3 rows x 2 columns]
```

- Add support for separating years and quarters using dashes, for example 2014-Q1. (GH9688)
- Allow conversion of values with dtype `datetime64` or `timedelta64` to strings using `astype(str)` (GH9757)
- `get_dummies` function now accepts `sparse` keyword. If set to `True`, the return `DataFrame` is sparse, e.g. `SparseDataFrame`. (GH8823)
- `Period` now accepts `datetime64` as value input. (GH9054)
- Allow timedelta string conversion when leading zero is missing from time definition, ie *0:00:00* vs *00:00:00*. (GH9570)
- Allow `Panel.shift` with `axis='items'` (GH9890)
- Trying to write an excel file now raises `NotImplementedError` if the `DataFrame` has a `MultiIndex` instead of writing a broken Excel file. (GH9794)
- Allow `Categorical.add_categories` to accept `Series` or `np.array`. (GH9927)
- Add/delete `str/dt/cat` accessors dynamically from `__dir__`. (GH9910)
- Add `normalize` as a `dt` accessor method. (GH10047)
- `DataFrame` and `Series` now have `_constructor_expanddim` property as overridable constructor for one higher dimensionality data. This should be used only when it is really needed, see *here*
- `pd.lib.infer_dtype` now returns `'bytes'` in Python 3 where appropriate. (GH10032)

API changes

- When passing in an ax to `df.plot( ..., ax=ax)`, the *sharex* kwarg will now default to *False*. The result is that the visibility of xlabels and xticklabels will not anymore be changed. You have to do that by yourself for the right axes in your figure or set `sharex=True` explicitly (but this changes the visible for all axes in the figure, not only the one which is passed in!). If pandas creates the subplots itself (e.g. no passed in *ax* kwarg), then the default is still `sharex=True` and the visibility changes are applied.
- *`assign()`* now inserts new columns in alphabetical order. Previously the order was arbitrary. (GH9777)
- By default, `read_csv` and `read_table` will now try to infer the compression type based on the file extension. Set `compression=None` to restore the previous behavior (no decompression). (GH9770)

Deprecations

- `Series.str.split`'s `return_type` keyword was removed in favor of `expand` (GH9847)

Index Representation

The string representation of `Index` and its sub-classes have now been unified. These will show a single-line display if there are few values; a wrapped multi-line display for a lot of values (but less than `display.max_seq_items`; if lots of items (> `display.max_seq_items`) will show a truncated display (the head and tail of the data). The formatting for `MultiIndex` is unchanged (a multi-line wrapped display). The display width responds to the option `display.max_seq_items`, which is defaulted to 100. (GH6482)

Previous Behavior

```
In [2]: pd.Index(range(4), name='foo')
Out[2]: Int64Index([0, 1, 2, 3], dtype='int64')

In [3]: pd.Index(range(104), name='foo')
Out[3]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,␣
↪19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,␣
↪40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,␣
↪61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,␣
↪82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...], dtype=
↪'int64')

In [4]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')
Out[4]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-01-04 00:00:00-05:00]
Length: 4, Freq: D, Timezone: US/Eastern

In [5]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')
Out[5]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-04-14 00:00:00-04:00]
Length: 104, Freq: D, Timezone: US/Eastern
```

New Behavior

```
In [29]: pd.set_option('display.width', 80)

In [30]: pd.Index(range(4), name='foo')
```

(continues on next page)

(continued from previous page)

```
Out[30]: RangeIndex(start=0, stop=4, step=1, name='foo')

In [31]: pd.Index(range(30), name='foo')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[31]: RangeIndex(start=0,␣
↪stop=30, step=1, name='foo')

In [32]: pd.Index(range(104), name='foo')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪RangeIndex(start=0, stop=104, step=1, name='foo')

In [33]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd'],
   ....:                     ordered=True, name='foobar')
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪CategoricalIndex(['a', 'bb', 'ccc', 'dddd'], categories=['a', 'bb', 'ccc', 'dddd'],␣
↪ordered=True, name='foobar', dtype='category')

In [34]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd'] * 10,
   ....:                     ordered=True, name='foobar')
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
CategoricalIndex(['a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a',
                  'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb',
                  'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc',
                  'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd',
                  'a', 'bb', 'ccc', 'dddd'],
                 categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar',␣
↪dtype='category')

In [35]: pd.CategoricalIndex(['a', 'bb', 'ccc', 'dddd'] * 100,
   ....:                     ordered=True, name='foobar')
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
CategoricalIndex(['a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a',
                  'bb',
                  ...
                  'ccc', 'dddd', 'a', 'bb', 'ccc', 'dddd', 'a', 'bb', 'ccc',
                  'dddd'],
                 categories=['a', 'bb', 'ccc', 'dddd'], ordered=True, name='foobar',␣
↪dtype='category', length=400)

In [36]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
               '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', name='foo', freq='D')

In [37]: pd.date_range('20130101', periods=25, freq='D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06', '2013-01-07', '2013-01-08',
               '2013-01-09', '2013-01-10', '2013-01-11', '2013-01-12',
               '2013-01-13', '2013-01-14', '2013-01-15', '2013-01-16',
```

(continues on next page)

(continued from previous page)

```
                 '2013-01-17', '2013-01-18', '2013-01-19', '2013-01-20',
                 '2013-01-21', '2013-01-22', '2013-01-23', '2013-01-24',
                 '2013-01-25'],
                dtype='datetime64[ns]', freq='D')

In [38]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
                 '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00',
                 '2013-01-05 00:00:00-05:00', '2013-01-06 00:00:00-05:00',
                 '2013-01-07 00:00:00-05:00', '2013-01-08 00:00:00-05:00',
                 '2013-01-09 00:00:00-05:00', '2013-01-10 00:00:00-05:00',
                 ...
                 '2013-04-05 00:00:00-04:00', '2013-04-06 00:00:00-04:00',
                 '2013-04-07 00:00:00-04:00', '2013-04-08 00:00:00-04:00',
                 '2013-04-09 00:00:00-04:00', '2013-04-10 00:00:00-04:00',
                 '2013-04-11 00:00:00-04:00', '2013-04-12 00:00:00-04:00',
                 '2013-04-13 00:00:00-04:00', '2013-04-14 00:00:00-04:00'],
                dtype='datetime64[ns, US/Eastern]', name='foo', length=104, freq='D')
```

Performance Improvements

- Improved csv write performance with mixed dtypes, including datetimes by up to 5x (GH9940)
- Improved csv write performance generally by 2x (GH9940)
- Improved the performance of `pd.lib.max_len_string_array` by 5-7x (GH10024)

Bug Fixes

- Bug where labels did not appear properly in the legend of `DataFrame.plot()`, passing `label=` arguments works, and Series indices are no longer mutated. (GH9542)
- Bug in json serialization causing a segfault when a frame had zero length. (GH9805)
- Bug in `read_csv` where missing trailing delimiters would cause segfault. (GH5664)
- Bug in retaining index name on appending (GH9862)
- Bug in `scatter_matrix` draws unexpected axis ticklabels (GH5662)
- Fixed bug in `StataWriter` resulting in changes to input `DataFrame` upon save (GH9795).
- Bug in `transform` causing length mismatch when null entries were present and a fast aggregator was being used (GH9697)
- Bug in `equals` causing false negatives when block order differed (GH9330)
- Bug in grouping with multiple `pd.Grouper` where one is non-time based (GH10063)
- Bug in `read_sql_table` error when reading postgres table with timezone (GH7139)
- Bug in `DataFrame` slicing may not retain metadata (GH9776)
- Bug where `TimdeltaIndex` were not properly serialized in fixed `HDFStore` (GH9635)
- Bug with `TimedeltaIndex` constructor ignoring `name` when given another `TimedeltaIndex` as data (GH10025).

- Bug in `DataFrameFormatter._get_formatted_index` with not applying `max_colwidth` to the `DataFrame` index (GH7856)
- Bug in `.loc` with a read-only ndarray data source (GH10043)
- Bug in `groupby.apply()` that would raise if a passed user defined function either returned only `None` (for all input). (GH9685)
- Always use temporary files in pytables tests (GH9992)
- Bug in plotting continuously using `secondary_y` may not show legend properly. (GH9610, GH9779)
- Bug in `DataFrame.plot(kind="hist")` results in `TypeError` when `DataFrame` contains non-numeric columns (GH9853)
- Bug where repeated plotting of `DataFrame` with a `DatetimeIndex` may raise `TypeError` (GH9852)
- Bug in `setup.py` that would allow an incompat cython version to build (GH9827)
- Bug in plotting `secondary_y` incorrectly attaches `right_ax` property to secondary axes specifying itself recursively. (GH9861)
- Bug in `Series.quantile` on empty Series of type `Datetime` or `Timedelta` (GH9675)
- Bug in `where` causing incorrect results when upcasting was required (GH9731)
- Bug in `FloatArrayFormatter` where decision boundary for displaying "small" floats in decimal format is off by one order of magnitude for a given display.precision (GH9764)
- Fixed bug where `DataFrame.plot()` raised an error when both `color` and `style` keywords were passed and there was no color symbol in the style strings (GH9671)
- Not showing a `DeprecationWarning` on combining list-likes with an `Index` (GH10083)
- Bug in `read_csv` and `read_table` when using `skip_rows` parameter if blank lines are present. (GH9832)
- Bug in `read_csv()` interprets `index_col=True` as `1` (GH9798)
- Bug in index equality comparisons using == failing on Index/MultiIndex type incompatibility (GH9785)
- Bug in which `SparseDataFrame` could not take *nan* as a column name (GH8822)
- Bug in `to_msgpack` and `read_msgpack` zlib and blosc compression support (GH9783)
- Bug `GroupBy.size` doesn't attach index name properly if grouped by `TimeGrouper` (GH9925)
- Bug causing an exception in slice assignments because `length_of_indexer` returns wrong results (GH9995)
- Bug in csv parser causing lines with initial white space plus one non-space character to be skipped. (GH9710)
- Bug in C csv parser causing spurious NaNs when data started with newline followed by white space. (GH10022)
- Bug causing elements with a null group to spill into the final group when grouping by a `Categorical` (GH9603)
- Bug where .iloc and .loc behavior is not consistent on empty dataframes (GH9964)
- Bug in invalid attribute access on a `TimedeltaIndex` incorrectly raised `ValueError` instead of `AttributeError` (GH9680)
- Bug in unequal comparisons between categorical data and a scalar, which was not in the categories (e.g. `Series(Categorical(list("abc"), ordered=True)) > "d"`. This returned `False` for all elements, but now raises a `TypeError`. Equality comparisons also now return `False` for `==` and `True` for `!=`. (GH9848)
- Bug in DataFrame `__setitem__` when right hand side is a dictionary (GH9874)

- Bug in `where` when dtype is `datetime64/timedelta64`, but dtype of other is not (GH9804)
- Bug in `MultiIndex.sortlevel()` results in unicode level name breaks (GH9856)
- Bug in which `groupby.transform` incorrectly enforced output dtypes to match input dtypes. (GH9807)
- Bug in `DataFrame` constructor when `columns` parameter is set, and `data` is an empty list (GH9939)
- Bug in bar plot with `log=True` raises `TypeError` if all values are less than 1 (GH9905)
- Bug in horizontal bar plot ignores `log=True` (GH9905)
- Bug in PyTables queries that did not return proper results using the index (GH8265, GH9676)
- Bug where dividing a dataframe containing values of type `Decimal` by another `Decimal` would raise. (GH9787)
- Bug where using DataFrames asfreq would remove the name of the index. (GH9885)
- Bug causing extra index point when resample BM/BQ (GH9756)
- Changed caching in `AbstractHolidayCalendar` to be at the instance level rather than at the class level as the latter can result in unexpected behaviour. (GH9552)
- Fixed latex output for MultiIndexed dataframes (GH9778)
- Bug causing an exception when setting an empty range using `DataFrame.loc` (GH9596)
- Bug in hiding ticklabels with subplots and shared axes when adding a new plot to an existing grid of axes (GH9158)
- Bug in `transform` and `filter` when grouping on a categorical variable (GH9921)
- Bug in `transform` when groups are equal in number and dtype to the input index (GH9700)
- Google BigQuery connector now imports dependencies on a per-method basis.(GH9713)
- Updated BigQuery connector to no longer use deprecated `oauth2client.tools.run()` (GH8327)
- Bug in subclassed `DataFrame`. It may not return the correct class, when slicing or subsetting it. (GH9632)
- Bug in `.median()` where non-float null values are not handled correctly (GH10040)
- Bug in Series.fillna() where it raises if a numerically convertible string is given (GH10092)

Contributors

A total of 58 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Alfonso MHC +
- Andy Hayden
- Artemy Kolchinsky
- Chris Gilmer +
- Chris Grinolds +
- Dan Birken
- David BROCHART +
- David Hirschfeld +
- David Stephens

- Dr. Leo +
- Evan Wright +
- Frans van Dunné +
- Hatem Nassrat +
- Henning Sperr +
- Hugo Herter +
- Jan Schulz
- Jeff Blackburne +
- Jeff Reback
- Jim Crist +
- Jonas Abernot +
- Joris Van den Bossche
- Kerby Shedden
- Leo Razoumov +
- Manuel Riel +
- Mortada Mehyar
- Nick Burns +
- Nick Eubank +
- Olivier Grisel
- Phillip Cloud
- Pietro Battiston
- Roy Hyunjin Han
- Sam Zhang +
- Scott Sanderson +
- Sinhrks +
- Stephan Hoyer
- Tiago Antao
- Tom Ajamian +
- Tom Augspurger
- Tomaz Berisa +
- Vikram Shirgur +
- Vladimir Filimonov
- William Hogman +
- Yasin A +
- Younggun Kim +
- behzad nouri

- dsm054
- floydsoft +
- flying-sheep +
- gfr +
- jnmclarty
- jreback
- ksanghai +
- lucas +
- mschmohl +
- ptype +
- rockg
- scls19fr +
- sinhrks

8.9.3 v0.16.0 (March 22, 2015)

This is a major release from 0.15.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `DataFrame.assign` method, see *here*
- `Series.to_coo/from_coo` methods to interact with `scipy.sparse`, see *here*
- Backwards incompatible change to `Timedelta` to conform the `.seconds` attribute with `datetime.timedelta`, see *here*
- Changes to the `.loc` slicing API to conform with the behavior of `.ix` see *here*
- Changes to the default for ordering in the `Categorical` constructor, see *here*
- Enhancement to the `.str` accessor to make string operations easier, see *here*
- The `pandas.tools.rplot`, `pandas.sandbox.qtpandas` and `pandas.rpy` modules are deprecated. We refer users to external packages like seaborn, pandas-qt and rpy2 for similar or equivalent functionality, see *here*

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

What's new in v0.16.0

- *New features*
  - *DataFrame Assign*
  - *Interaction with scipy.sparse*
  - *String Methods Enhancements*
  - *Other enhancements*

New features

DataFrame Assign

Inspired by dplyr's `mutate` verb, DataFrame has a new `assign()` method. The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. The new values are inserted, and the entire DataFrame (with all original and new columns) is returned.

```
In [1]: iris = pd.read_csv('data/iris.data')

In [2]: iris.head()
Out[2]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

[5 rows x 5 columns]

In [3]: iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength']).head()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000

[5 rows x 6 columns]
```

Above was an example of inserting a precomputed value. We can also pass in a function to be evaluated.

```
In [4]: iris.assign(sepal_ratio=lambda x: (x['SepalWidth']
   ...:                                     / x['SepalLength'])).head()
```

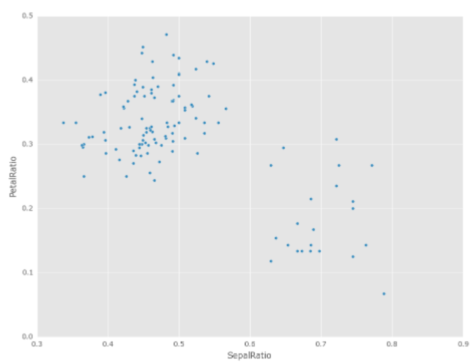
(continues on next page)

(continued from previous page)

```
   ...:
Out[4]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000

[5 rows x 6 columns]
```

The power of `assign` comes when used in chains of operations. For example, we can limit the DataFrame to just those with a Sepal Length greater than 5, calculate the ratio, and plot

```
In [5]: iris = pd.read_csv('data/iris.data')

In [6]: (iris.query('SepalLength > 5')
   ...:      .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ...:              PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
   ...:      .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
   ...:
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7f37dd27e400>
```



See the *documentation* for more. (GH9229)

Interaction with scipy.sparse

Added *SparseSeries.to_coo()* and *SparseSeries.from_coo()* methods (GH8048) for converting to and from `scipy.sparse.coo_matrix` instances (see *here*). For example, given a SparseSeries with MultiIndex we can convert to a *scipy.sparse.coo_matrix* by specifying the row and column labels as index levels:

```
In [7]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [8]: s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
   ...:                                      (1, 2, 'a', 1),
   ...:                                      (1, 1, 'b', 0),
   ...:                                      (1, 1, 'b', 1),
   ...:                                      (2, 1, 'b', 0),
   ...:                                      (2, 1, 'b', 1)],
   ...:                                     names=['A', 'B', 'C', 'D'])
   ...:
```

(continues on next page)

(continued from previous page)

```
In [9]: s
Out[9]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
Length: 6, dtype: float64

# SparseSeries
In [10]: ss = s.to_sparse()

In [11]: ss
Out[11]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
Length: 6, dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 2], dtype=int32)

In [12]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
   ....:                              column_levels=['C', 'D'],
   ....:                              sort_labels=False)
   ....:

In [13]: A
Out[13]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>

In [14]: A.todense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
matrix([[ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  3.],
        [ 0.,  0.,  0.,  0.]])

In [15]: rows
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[(1, 2), (1, 1), (2, 1)]

In [16]: columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

The from_coo method is a convenience method for creating a `SparseSeries` from a `scipy.sparse.coo_matrix`:

```
In [17]: from scipy import sparse

In [18]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
   ....:                       shape=(3, 4))
   ....:

In [19]: A
Out[19]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>

In [20]: A.todense()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])

In [21]: ss = pd.SparseSeries.from_coo(A)

In [22]: ss
Out[22]:
0  2    1.0
   3    2.0
1  0    3.0
Length: 3, dtype: Sparse[float64, nan]
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

String Methods Enhancements

- Following new methods are accessible via `.str` accessor to apply the function to each values. This is intended to make it more consistent with standard methods on strings. (GH9282, GH9352, GH9386, GH9387, GH9439)

		Methods		
`isalnum()`	`isalpha()`	`isdigit()`	`isdigit()`	`isspace()`
`islower()`	`isupper()`	`istitle()`	`isnumeric()`	`isdecimal()`
`find()`	`rfind()`	`ljust()`	`rjust()`	`zfill()`

```
In [23]: s = pd.Series(['abcd', '3456', 'EFGH'])

In [24]: s.str.isalpha()
Out[24]:
0     True
1    False
2     True
Length: 3, dtype: bool

In [25]: s.str.find('ab')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]:
0    0
1   -1
```

(continues on next page)

(continued from previous page)

```
2   -1
Length: 3, dtype: int64
```

- *Series.str.pad()* and *Series.str.center()* now accept `fillchar` option to specify filling character (GH9352)

```
In [26]: s = pd.Series(['12', '300', '25'])

In [27]: s.str.pad(5, fillchar='_')
Out[27]:
0    ___12
1    __300
2    ___25
Length: 3, dtype: object
```

- Added *Series.str.slice_replace()*, which previously raised `NotImplementedError` (GH8888)

```
In [28]: s = pd.Series(['ABCD', 'EFGH', 'IJK'])

In [29]: s.str.slice_replace(1, 3, 'X')
Out[29]:
0    AXD
1    EXH
2     IX
Length: 3, dtype: object

# replaced with empty char
In [30]: s.str.slice_replace(0, 1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]:
0    BCD
1    FGH
2     JK
Length: 3, dtype: object
```

Other enhancements

- Reindex now supports `method='nearest'` for frames or series with a monotonic increasing or decreasing index (GH9258):

```
In [31]: df = pd.DataFrame({'x': range(5)})

In [32]: df.reindex([0.2, 1.8, 3.5], method='nearest')
Out[32]:
     x
0.2  0
1.8  2
3.5  4

[3 rows x 1 columns]
```

  This method is also exposed by the lower level `Index.get_indexer` and `Index.get_loc` methods.

- The `read_excel()` function's *sheetname* argument now accepts a list and `None`, to get multiple or all sheets respectively. If more than one sheet is specified, a dictionary is returned. (GH9450)

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel('path_to_file.xls', sheetname=['Sheet1', 3])
```

- Allow Stata files to be read incrementally with an iterator; support for long strings in Stata files. See the docs *here* (GH9493:).
- Paths beginning with ~ will now be expanded to begin with the user's home directory (GH9066)
- Added time interval selection in `get_data_yahoo` (GH9071)
- Added `Timestamp.to_datetime64()` to complement `Timedelta.to_timedelta64()` (GH9255)
- `tseries.frequencies.to_offset()` now accepts `Timedelta` as input (GH9064)
- Lag parameter was added to the autocorrelation method of `Series`, defaults to lag-1 autocorrelation (GH9192)
- `Timedelta` will now accept `nanoseconds` keyword in constructor (GH9273)
- SQL code now safely escapes table and column names (GH8986)
- Added auto-complete for `Series.str.<tab>`, `Series.dt.<tab>` and `Series.cat.<tab>` (GH9322)
- `Index.get_indexer` now supports `method='pad'` and `method='backfill'` even for any target array, not just monotonic targets. These methods also work for monotonic decreasing as well as monotonic increasing indexes (GH9258).
- `Index.asof` now works on all index types (GH9258).
- A `verbose` argument has been augmented in `io.read_excel()`, defaults to False. Set to True to print sheet names as they are parsed. (GH9450)
- Added `days_in_month` (compatibility alias `daysinmonth`) property to `Timestamp`, `DatetimeIndex`, `Period`, `PeriodIndex`, and `Series.dt` (GH9572)
- Added `decimal` option in `to_csv` to provide formatting for non-'.' decimal separators (GH781)
- Added `normalize` option for `Timestamp` to normalized to midnight (GH8794)
- Added example for `DataFrame` import to R using HDF5 file and `rhdf5` library. See the *documentation* for more (GH9636).

Backwards incompatible API changes

Changes in Timedelta

In v0.15.0 a new scalar type `Timedelta` was introduced, that is a sub-class of `datetime.timedelta`. Mentioned *here* was a notice of an API change w.r.t. the `.seconds` accessor. The intent was to provide a user-friendly set of accessors that give the 'natural' value for that unit, e.g. if you had a `Timedelta('1 day, 10:11:12')`, then `.seconds` would return 12. However, this is at odds with the definition of `datetime.timedelta`, which defines `.seconds` as `10 * 3600 + 11 * 60 + 12 == 36672`.

So in v0.16.0, we are restoring the API to match that of `datetime.timedelta`. Further, the component values are still available through the `.components` accessor. This affects the `.seconds` and `.microseconds` accessors, and removes the `.hours`, `.minutes`, `.milliseconds` accessors. These changes affect `TimedeltaIndex` and the Series `.dt` accessor as well. (GH9185, GH9139)

Previous Behavior

```
In [2]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [3]: t.days
Out[3]: 1

In [4]: t.seconds
Out[4]: 12

In [5]: t.microseconds
Out[5]: 123
```

New Behavior

```
In [33]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [34]: t.days
Out[34]: 1

In [35]: t.seconds
\\\\\\\\\\\Out[35]: 36672

In [36]: t.microseconds
\\\\\\\\\\\\\\\\\\\\\\\\\Out[36]: 100123
```

Using `.components` allows the full component access

```
In [37]: t.components
Out[37]: Components(days=1, hours=10, minutes=11, seconds=12, milliseconds=100,␣
↪microseconds=123, nanoseconds=0)

In [38]: t.components.seconds
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪12
```

Indexing Changes

The behavior of a small sub-set of edge cases for using `.loc` have changed (GH8613). Furthermore we have improved the content of the error messages that are raised:

- Slicing with `.loc` where the start and/or stop bound is not found in the index is now allowed; this previously would raise a `KeyError`. This makes the behavior the same as `.ix` in this case. This change is only for slicing, not when indexing with a single label.

```
In [39]: df = pd.DataFrame(np.random.randn(5, 4),
   ....:                   columns=list('ABCD'),
   ....:                   index=pd.date_range('20130101', periods=5))
   ....:

In [40]: df
Out[40]: 
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

(continues on next page)

(continued from previous page)

```
[5 rows x 4 columns]

In [41]: s = pd.Series(range(5), [-2, -1, 1, 2, 3])

In [42]: s
Out[42]:
-2    0
-1    1
 1    2
 2    3
 3    4
Length: 5, dtype: int64
```

Previous Behavior

```
In [4]: df.loc['2013-01-02':'2013-01-10']
KeyError: 'stop bound [2013-01-10] is not in the [index]'

In [6]: s.loc[-10:3]
KeyError: 'start bound [-10] is not the [index]'
```

New Behavior

```
In [43]: df.loc['2013-01-02':'2013-01-10']
Out[43]:
                   A         B         C         D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401

[4 rows x 4 columns]

In [44]: s.loc[-10:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
-2    0
-1    1
 1    2
 2    3
 3    4
Length: 5, dtype: int64
```

- Allow slicing with float-like values on an integer index for `.ix`. Previously this was only enabled for `.loc`:

  Previous Behavior

```
In [8]: s.ix[-1.0:2]
TypeError: the slice start value [-1.0] is not a proper indexer for this index␣
↪type (Int64Index)
```

  New Behavior

```
In [2]: s.ix[-1.0:2]
Out[2]:
-1    1
```

(continues on next page)

(continued from previous page)

```
 1    2
 2    3
dtype: int64
```

- Provide a useful exception for indexing with an invalid type for that index when using `.loc`. For example trying to use `.loc` on an index of type `DatetimeIndex` or `PeriodIndex` or `TimedeltaIndex`, with an integer (or a float).

  Previous Behavior

  ```
  In [4]: df.loc[2:3]
  KeyError: 'start bound [2] is not the [index]'
  ```

  New Behavior

  ```
  In [4]: df.loc[2:3]
  TypeError: Cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex
  ↪'> with <type 'int'> keys
  ```

Categorical Changes

In prior versions, `Categoricals` that had an unspecified ordering (meaning no `ordered` keyword was passed) were defaulted as `ordered` Categoricals. Going forward, the `ordered` keyword in the `Categorical` constructor will default to `False`. Ordering must now be explicit.

Furthermore, previously you *could* change the `ordered` attribute of a Categorical by just setting the attribute, e.g. `cat.ordered=True`; This is now deprecated and you should use `cat.as_ordered()` or `cat.as_unordered()`. These will by default return a **new** object and not modify the existing object. (GH9347, GH9190)

Previous Behavior

```
In [3]: s = pd.Series([0, 1, 2], dtype='category')

In [4]: s
Out[4]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [5]: s.cat.ordered
Out[5]: True

In [6]: s.cat.ordered = False

In [7]: s
Out[7]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0, 1, 2]
```

New Behavior

```
In [45]: s = pd.Series([0, 1, 2], dtype='category')

In [46]: s
Out[46]:
0    0
1    1
2    2
Length: 3, dtype: category
Categories (3, int64): [0, 1, 2]

In [47]: s.cat.ordered
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]:␣
↪False

In [48]: s = s.cat.as_ordered()

In [49]: s
Out[49]:
0    0
1    1
2    2
Length: 3, dtype: category
Categories (3, int64): [0 < 1 < 2]

In [50]: s.cat.ordered
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[50]
↪True

# you can set in the constructor of the Categorical
In [51]: s = pd.Series(pd.Categorical([0, 1, 2], ordered=True))

In [52]: s
Out[52]:
0    0
1    1
2    2
Length: 3, dtype: category
Categories (3, int64): [0 < 1 < 2]

In [53]: s.cat.ordered
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[53]
↪True
```

For ease of creation of series of categorical data, we have added the ability to pass keywords when calling `.astype()`. These are passed directly to the constructor.

```
In [54]: s = pd.Series(["a", "b", "c", "a"]).astype('category', ordered=True)

In [55]: s
Out[55]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a < b < c]
```

(continues on next page)

(continued from previous page)

```
In [56]: s = (pd.Series(["a", "b", "c", "a"])
   ....:         .astype('category', categories=list('abcdef'), ordered=False))

In [57]: s
Out[57]:
0    a
1    b
2    c
3    a
dtype: category
Categories (6, object): [a, b, c, d, e, f]
```

Other API Changes

- `Index.duplicated` now returns `np.array(dtype=bool)` rather than `Index(dtype=object)` containing `bool` values. (GH8875)
- `DataFrame.to_json` now returns accurate type serialisation for each column for frames of mixed dtype (GH9037)

  Previously data was coerced to a common dtype before serialisation, which for example resulted in integers being serialised to floats:

  ```
  In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
  Out[2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1.0,"1":2.0}}'
  ```

  Now each column is serialised using its correct dtype:

  ```
  In [2]:  pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
  Out[2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1,"1":2}}'
  ```

- `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex.summary` now output the same format. (GH9116)
- `TimedeltaIndex.freqstr` now output the same string format as `DatetimeIndex`. (GH9116)
- Bar and horizontal bar plots no longer add a dashed line along the info axis. The prior style can be achieved with matplotlib's `axhline` or `axvline` methods (GH9088).
- `Series` accessors `.dt`, `.cat` and `.str` now raise `AttributeError` instead of `TypeError` if the series does not contain the appropriate type of data (GH9617). This follows Python's built-in exception hierarchy more closely and ensures that tests like `hasattr(s, 'cat')` are consistent on both Python 2 and 3.
- `Series` now supports bitwise operation for integral types (GH9016). Previously even if the input dtypes were integral, the output dtype was coerced to `bool`.

  Previous Behavior

  ```
  In [2]: pd.Series([0, 1, 2, 3], list('abcd')) | pd.Series([4, 4, 4, 4], list('abcd
  ↪'))
  Out[2]:
  a    True
  b    True
  c    True
  d    True
  dtype: bool
  ```

New Behavior. If the input dtypes are integral, the output dtype is also integral and the output values are the result of the bitwise operation.

```
In [2]: pd.Series([0, 1, 2, 3], list('abcd')) | pd.Series([4, 4, 4, 4], list('abcd
↪'))
Out[2]:
a    4
b    5
c    6
d    7
dtype: int64
```

- During division involving a `Series` or `DataFrame`, `0/0` and `0//0` now give `np.nan` instead of `np.inf`. (GH9144, GH8445)

  Previous Behavior

```
In [2]: p = pd.Series([0, 1])

In [3]: p / 0
Out[3]:
0    inf
1    inf
dtype: float64

In [4]: p // 0
Out[4]:
0    inf
1    inf
dtype: float64
```

  New Behavior

```
In [54]: p = pd.Series([0, 1])

In [55]: p / 0
Out[55]:
0    NaN
1    inf
Length: 2, dtype: float64

In [56]: p // 0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[56]:
0    NaN
1    inf
Length: 2, dtype: float64
```

- `Series.values_counts` and `Series.describe` for categorical data will now put `NaN` entries at the end. (GH9443)
- `Series.describe` for categorical data will now give counts and frequencies of 0, not `NaN`, for unused categories (GH9443)
- Due to a bug fix, looking up a partial string label with `DatetimeIndex.asof` now includes values that match the string, even if they are after the start of the partial string label (GH9258).

  Old behavior:

```
In [4]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
Out[4]: Timestamp('2000-01-31 00:00:00')
```

Fixed behavior:

```
In [57]: pd.to_datetime(['2000-01-31', '2000-02-28']).asof('2000-02')
Out[57]: Timestamp('2000-02-28 00:00:00')
```

To reproduce the old behavior, simply add more precision to the label (e.g., use `2000-02-01` instead of `2000-02`).

Deprecations

- The `rplot` trellis plotting interface is deprecated and will be removed in a future version. We refer to external packages like seaborn for similar but more refined functionality (GH3445). The documentation includes some examples how to convert your existing code using `rplot` to seaborn: *rplot docs*.
- The `pandas.sandbox.qtpandas` interface is deprecated and will be removed in a future version. We refer users to the external package pandas-qt. (GH9615)
- The `pandas.rpy` interface is deprecated and will be removed in a future version. Similar functionality can be accessed through the rpy2 project (GH9602)
- Adding `DatetimeIndex/PeriodIndex` to another `DatetimeIndex/PeriodIndex` is being deprecated as a set-operation. This will be changed to a `TypeError` in a future version. `.union()` should be used for the union set operation. (GH9094)
- Subtracting `DatetimeIndex/PeriodIndex` from another `DatetimeIndex/PeriodIndex` is being deprecated as a set-operation. This will be changed to an actual numeric subtraction yielding a `TimeDeltaIndex` in a future version. `.difference()` should be used for the differencing set operation. (GH9094)

Removal of prior version deprecations/changes

- `DataFrame.pivot_table` and `crosstab`'s `rows` and `cols` keyword arguments were removed in favor of `index` and `columns` (GH6581)
- `DataFrame.to_excel` and `DataFrame.to_csv` `cols` keyword argument was removed in favor of `columns` (GH6581)
- Removed `convert_dummies` in favor of `get_dummies` (GH6581)
- Removed `value_range` in favor of `describe` (GH6581)

Performance Improvements

- Fixed a performance regression for `.loc` indexing with an array or list-like (GH9126:).
- `DataFrame.to_json` 30x performance improvement for mixed dtype frames. (GH9037)
- Performance improvements in `MultiIndex.duplicated` by working with labels instead of values (GH9125)
- Improved the speed of `nunique` by calling `unique` instead of `value_counts` (GH9129, GH7771)
- Performance improvement of up to 10x in `DataFrame.count` and `DataFrame.dropna` by taking advantage of homogeneous/heterogeneous dtypes appropriately (GH9136)

- Performance improvement of up to 20x in `DataFrame.count` when using a `MultiIndex` and the `level` keyword argument (GH9163)
- Performance and memory usage improvements in `merge` when key space exceeds `int64` bounds (GH9151)
- Performance improvements in multi-key `groupby` (GH9429)
- Performance improvements in `MultiIndex.sortlevel` (GH9445)
- Performance and memory usage improvements in `DataFrame.duplicated` (GH9398)
- Cythonized `Period` (GH9440)
- Decreased memory usage on `to_hdf` (GH9648)

Bug Fixes

- Changed `.to_html` to remove leading/trailing spaces in table body (GH4987)
- Fixed issue using `read_csv` on s3 with Python 3 (GH9452)
- Fixed compatibility issue in `DatetimeIndex` affecting architectures where `numpy.int_` defaults to `numpy.int32` (GH8943)
- Bug in Panel indexing with an object-like (GH9140)
- Bug in the returned `Series.dt.components` index was reset to the default index (GH9247)
- Bug in `Categorical.__getitem__/__setitem__` with listlike input getting incorrect results from indexer coercion (GH9469)
- Bug in partial setting with a DatetimeIndex (GH9478)
- Bug in groupby for integer and datetime64 columns when applying an aggregator that caused the value to be changed when the number was sufficiently large (GH9311, GH6620)
- Fixed bug in `to_sql` when mapping a `Timestamp` object column (datetime column with timezone info) to the appropriate sqlalchemy type (GH9085).
- Fixed bug in `to_sql` `dtype` argument not accepting an instantiated SQLAlchemy type (GH9083).
- Bug in `.loc` partial setting with a `np.datetime64` (GH9516)
- Incorrect dtypes inferred on datetimelike looking `Series` & on `.xs` slices (GH9477)
- Items in `Categorical.unique()` (and `s.unique()` if `s` is of dtype `category`) now appear in the order in which they are originally found, not in sorted order (GH9331). This is now consistent with the behavior for other dtypes in pandas.
- Fixed bug on big endian platforms which produced incorrect results in `StataReader` (GH8688).
- Bug in `MultiIndex.has_duplicates` when having many levels causes an indexer overflow (GH9075, GH5873)
- Bug in `pivot` and `unstack` where `nan` values would break index alignment (GH4862, GH7401, GH7403, GH7405, GH7466, GH9497)
- Bug in left `join` on MultiIndex with `sort=True` or null values (GH9210).
- Bug in `MultiIndex` where inserting new keys would fail (GH9250).
- Bug in `groupby` when key space exceeds `int64` bounds (GH9096).
- Bug in `unstack` with `TimedeltaIndex` or `DatetimeIndex` and nulls (GH9491).
- Bug in `rank` where comparing floats with tolerance will cause inconsistent behaviour (GH8365).

- Fixed character encoding bug in read_stata and StataReader when loading data from a URL (GH9231).
- Bug in adding offsets.Nano to other offsets raises TypeError (GH9284)
- Bug in DatetimeIndex iteration, related to (GH8890), fixed in (GH9100)
- Bugs in resample around DST transitions. This required fixing offset classes so they behave correctly on DST transitions. (GH5172, GH8744, GH8653, GH9173, GH9468).
- Bug in binary operator method (eg .mul()) alignment with integer levels (GH9463).
- Bug in boxplot, scatter and hexbin plot may show an unnecessary warning (GH8877)
- Bug in subplot with layout kw may show unnecessary warning (GH9464)
- Bug in using grouper functions that need passed through arguments (e.g. axis), when using wrapped function (e.g. fillna), (GH9221)
- DataFrame now properly supports simultaneous copy and dtype arguments in constructor (GH9099)
- Bug in read_csv when using skiprows on a file with CR line endings with the c engine. (GH9079)
- isnull now detects NaT in PeriodIndex (GH9129)
- Bug in groupby .nth() with a multiple column groupby (GH8979)
- Bug in DataFrame.where and Series.where coerce numerics to string incorrectly (GH9280)
- Bug in DataFrame.where and Series.where raise ValueError when string list-like is passed. (GH9280)
- Accessing Series.str methods on with non-string values now raises TypeError instead of producing incorrect results (GH9184)
- Bug in DatetimeIndex.__contains__ when index has duplicates and is not monotonic increasing (GH9512)
- Fixed division by zero error for Series.kurt() when all values are equal (GH9197)
- Fixed issue in the xlsxwriter engine where it added a default 'General' format to cells if no other format was applied. This prevented other row or column formatting being applied. (GH9167)
- Fixes issue with index_col=False when usecols is also specified in read_csv. (GH9082)
- Bug where wide_to_long would modify the input stub names list (GH9204)
- Bug in to_sql not storing float64 values using double precision. (GH9009)
- SparseSeries and SparsePanel now accept zero argument constructors (same as their non-sparse counterparts) (GH9272).
- Regression in merging Categorical and object dtypes (GH9426)
- Bug in read_csv with buffer overflows with certain malformed input files (GH9205)
- Bug in groupby MultiIndex with missing pair (GH9049, GH9344)
- Fixed bug in Series.groupby where grouping on MultiIndex levels would ignore the sort argument (GH9444)
- Fix bug in DataFrame.Groupby where sort=False is ignored in the case of Categorical columns. (GH8868)
- Fixed bug with reading CSV files from Amazon S3 on python 3 raising a TypeError (GH9452)
- Bug in the Google BigQuery reader where the 'jobComplete' key may be present but False in the query results (GH8728)

- Bug in `Series.values_counts` with excluding `NaN` for categorical type `Series` with `dropna=True` (GH9443)
- Fixed missing numeric_only option for `DataFrame.std/var/sem` (GH9201)
- Support constructing `Panel` or `Panel4D` with scalar data (GH8285)
- `Series` text representation disconnected from *max_rows/max_columns* (GH7508).
- `Series` number formatting inconsistent when truncated (GH8532).

  Previous Behavior

```
In [2]: pd.options.display.max_rows = 10
In [3]: s = pd.Series([1,1,1,1,1,1,1,1,1,1,0.9999,1,1]*10)
In [4]: s
Out[4]:
0    1
1    1
2    1
...
127    0.9999
128    1.0000
129    1.0000
Length: 130, dtype: float64
```

  New Behavior

```
0      1.0000
1      1.0000
2      1.0000
3      1.0000
4      1.0000
...
125    1.0000
126    1.0000
127    0.9999
128    1.0000
129    1.0000
dtype: float64
```

- A Spurious `SettingWithCopy` Warning was generated when setting a new item in a frame in some cases (GH8730)

  The following would previously report a `SettingWithCopy` Warning.

```
In [58]: df1 = pd.DataFrame({'x': pd.Series(['a', 'b', 'c']),
   ....:                     'y': pd.Series(['d', 'e', 'f'])})
   ....:

In [59]: df2 = df1[['x']]

In [60]: df2['y'] = ['g', 'h', 'i']
```

Contributors

A total of 60 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Toth +
- Alan Du +
- Alessandro Amici +
- Artemy Kolchinsky
- Ashwini Chaudhary +
- Ben Schiller
- Bill Letson
- Brandon Bradley +
- Chau Hoang +
- Chris Reynolds
- Chris Whelan +
- Christer van der Meeren +
- David Cottrell +
- David Stephens
- Ehsan Azarnasab +
- Garrett-R +
- Guillaume Gay
- Jake Torcasso +
- Jason Sexauer
- Jeff Reback
- John McNamara
- Joris Van den Bossche
- Joschka zur Jacobsmühlen +
- Juarez Bochi +
- Junya Hayashi +
- K.-Michael Aye
- Kerby Shedden +
- Kevin Sheppard
- Kieran O'Mahony
- Kodi Arfer +
- Matti Airas +
- Min RK +
- Mortada Mehyar
- Robert +
- Scott E Lasley
- Scott Lasley +

- Sergio Pascual +
- Skipper Seabold
- Stephan Hoyer
- Thomas Grainger
- Tom Augspurger
- TomAugspurger
- Vladimir Filimonov +
- Vyomkesh Tripathi +
- Will Holmgren
- Yulong Yang +
- behzad nouri
- bertrandhaut +
- bjonen
- cel4 +
- clham
- hsperr +
- ischwabacher
- jnmclarty
- josham +
- jreback
- omtinez +
- roch +
- sinhrks
- unutbu

8.10 Version 0.15

8.10.1 v0.15.2 (December 12, 2014)

This is a minor release from 0.15.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. A small number of API changes were necessary to fix existing bugs. We recommend that all users upgrade to this version.

- *Enhancements*
- *API Changes*
- *Performance Improvements*
- *Bug Fixes*

API changes

- Indexing in `MultiIndex` beyond lex-sort depth is now supported, though a lexically sorted index will have a better performance. (GH2646)

```
In [1]: df = pd.DataFrame({'jim':[0, 0, 1, 1],
   ...:                    'joe':['x', 'x', 'z', 'y'],
   ...:                    'jolie':np.random.rand(4)}).set_index(['jim', 'joe'])
   ...:

In [2]: df
Out[2]:
            jolie
jim joe
0   x    0.126970
    x    0.966718
1   z    0.260476
    y    0.897237

[4 rows x 1 columns]

In [3]: df.index.lexsort_depth
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1

# in prior versions this would raise a KeyError
# will now show a PerformanceWarning
In [4]: df.loc[(1, 'z')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            jolie
jim joe
1   z    0.260476

[1 rows x 1 columns]

# lexically sorting
In [5]: df2 = df.sort_index()

In [6]: df2
Out[6]:
            jolie
jim joe
0   x    0.126970
    x    0.966718
1   y    0.897237
    z    0.260476

[4 rows x 1 columns]

In [7]: df2.index.lexsort_depth
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪2

In [8]: df2.loc[(1,'z')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            jolie
```

(continues on next page)

(continued from previous page)

```
jim joe
1   z    0.260476

[1 rows x 1 columns]
```

- Bug in unique of Series with `category` dtype, which returned all categories regardless whether they were "used" or not (see GH8559 for the discussion). Previous behaviour was to return all categories:

```
In [3]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [4]: cat
Out[4]:
[a, b, a]
Categories (3, object): [a < b < c]

In [5]: cat.unique()
Out[5]: array(['a', 'b', 'c'], dtype=object)
```

  Now, only the categories that do effectively occur in the array are returned:

```
In [9]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [10]: cat.unique()
Out[10]:
[a, b]
Categories (2, object): [a, b]
```

- `Series.all` and `Series.any` now support the `level` and `skipna` parameters. `Series.all`, `Series.any`, `Index.all`, and `Index.any` no longer support the `out` and `keepdims` parameters, which existed for compatibility with ndarray. Various index types no longer support the `all` and `any` aggregation functions and will now raise `TypeError`. (GH8302).
- Allow equality comparisons of Series with a categorical dtype and object dtype; previously these would raise `TypeError` (GH8938)
- Bug in `NDFrame`: conflicting attribute/column names now behave consistently between getting and setting. Previously, when both a column and attribute named `y` existed, `data.y` would return the attribute, while `data.y = z` would update the column (GH8994)

```
In [11]: data = pd.DataFrame({'x': [1, 2, 3]})

In [12]: data.y = 2

In [13]: data['y'] = [2, 4, 6]

In [14]: data
Out[14]:
   x  y
0  1  2
1  2  4
2  3  6

[3 rows x 2 columns]

# this assignment was inconsistent
In [15]: data.y = 5
```

Old behavior:

```
In [6]: data.y
Out[6]: 2

In [7]: data['y'].values
Out[7]: array([5, 5, 5])
```

New behavior:

```
In [16]: data.y
Out[16]: 5

In [17]: data['y'].values
\\\\\\\\\\\Out[17]: array([2, 4, 6])
```

- `Timestamp('now')` is now equivalent to `Timestamp.now()` in that it returns the local time rather than UTC. Also, `Timestamp('today')` is now equivalent to `Timestamp.today()` and both have `tz` as a possible argument. (GH9000)
- Fix negative step support for label-based slices (GH8753)

  Old behavior:

```
In [1]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])
Out[1]:
a    0
b    1
c    2
dtype: int64

In [2]: s.loc['c':'a':-1]
Out[2]:
c    2
dtype: int64
```

  New behavior:

```
In [18]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])

In [19]: s.loc['c':'a':-1]
Out[19]:
c    2
b    1
a    0
Length: 3, dtype: int64
```

Enhancements

`Categorical` enhancements:

- Added ability to export Categorical data to Stata (GH8633). See *here* for limitations of categorical variables exported to Stata data files.
- Added flag `order_categoricals` to `StataReader` and `read_stata` to select whether to order imported categorical data (GH8836). See *here* for more information on importing categorical variables from Stata data files.

- Added ability to export Categorical data to to/from HDF5 (GH7621). Queries work the same as if it was an object array. However, the `category` dtyped data is stored in a more efficient manner. See *here* for an example and caveats w.r.t. prior versions of pandas.
- Added support for `searchsorted()` on *Categorical* class (GH8420).

Other enhancements:

- Added the ability to specify the SQL type of columns when writing a DataFrame to a database (GH8778). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
from sqlalchemy.types import String
data.to_sql('data_dtype', engine, dtype={'Col_1': String})  # noqa F821
```

- `Series.all` and `Series.any` now support the `level` and `skipna` parameters (GH8302):

```
In [20]: s = pd.Series([False, True, False], index=[0, 0, 1])

In [21]: s.any(level=0)
Out[21]:
0     True
1    False
Length: 2, dtype: bool
```

- `Panel` now supports the `all` and `any` aggregation functions. (GH8302):

```
In [22]: p = pd.Panel(np.random.rand(2, 5, 4) > 0.1)

In [23]: p.all()
Out[23]:
      0      1     2      3
0  True  False  True   True
1  True   True  True  False
2  True   True  True   True
3  True   True  True   True
4  True   True  True   True

[5 rows x 4 columns]
```

- Added support for `utcfromtimestamp()`, `fromtimestamp()`, and `combine()` on *Timestamp* class (GH5351).
- Added Google Analytics (*pandas.io.ga*) basic documentation (GH8835). See here.
- `Timedelta` arithmetic returns `NotImplemented` in unknown cases, allowing extensions by custom classes (GH8813).
- `Timedelta` now supports arithmetic with `numpy.ndarray` objects of the appropriate dtype (numpy 1.8 or newer only) (GH8884).
- Added `Timedelta.to_timedelta64()` method to the public API (GH8884).
- Added `gbq.generate_bq_schema()` function to the gbq module (GH8325).
- `Series` now works with map objects the same way as generators (GH8909).
- Added context manager to `HDFStore` for automatic closing (GH8791).
- `to_datetime` gains an `exact` keyword to allow for a format to not require an exact match for a provided format string (if its `False`). `exact` defaults to `True` (meaning that exact matching is still the default) (GH8904)

- Added `axvlines` boolean option to parallel_coordinates plot function, determines whether vertical lines will be printed, default is True
- Added ability to read table footers to read_html (GH8552)
- `to_sql` now infers data types of non-NA values for columns that contain NA values and have dtype `object` (GH8778).

Performance

- Reduce memory usage when skiprows is an integer in read_csv (GH8681)
- Performance boost for `to_datetime` conversions with a passed `format=`, and the `exact=False` (GH8904)

Bug Fixes

- Bug in concat of Series with `category` dtype which were coercing to `object`. (GH8641)
- Bug in Timestamp-Timestamp not returning a Timedelta type and datelike-datelike ops with timezones (GH8865)
- Made consistent a timezone mismatch exception (either tz operated with None or incompatible timezone), will now return `TypeError` rather than `ValueError` (a couple of edge cases only), (GH8865)
- Bug in using a `pd.Grouper(key=...)` with no level/axis or level only (GH8795, GH8866)
- Report a `TypeError` when invalid/no parameters are passed in a groupby (GH8015)
- Bug in packaging pandas with `py2app/cx_Freeze` (GH8602, GH8831)
- Bug in `groupby` signatures that didn't include *args or **kwargs (GH8733).
- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
- Unclear error message in csv parsing when passing dtype and names and the parsed data is a different data type (GH8833)
- Bug in slicing a MultiIndex with an empty list and at least one boolean indexer (GH8781)
- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo (GH8761).
- `Timedelta` kwargs may now be numpy ints and floats (GH8757).
- Fixed several outstanding bugs for `Timedelta` arithmetic and comparisons (GH8813, GH5963, GH5436).
- `sql_schema` now generates dialect appropriate `CREATE TABLE` statements (GH8697)
- `slice` string method now takes step into account (GH8754)
- Bug in `BlockManager` where setting values with different type would break block integrity (GH8850)
- Bug in `DatetimeIndex` when using `time` object as key (GH8667)
- Bug in `merge` where `how='left'` and `sort=False` would not preserve left frame order (GH7331)
- Bug in `MultiIndex.reindex` where reindexing at level would not reorder labels (GH4088)
- Bug in certain operations with dateutil timezones, manifesting with dateutil 2.3 (GH8639)
- Regression in DatetimeIndex iteration with a Fixed/Local offset timezone (GH8890)
- Bug in `to_datetime` when parsing a nanoseconds using the `%f` format (GH8989)

- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
- Fix: The font size was only set on x axis if vertical or the y axis if horizontal. (GH8765)
- Fixed division by 0 when reading big csv files in python 3 (GH8621)
- Bug in outputting a MultiIndex with `to_html,index=False` which would add an extra column (GH8452)
- Imported categorical variables from Stata files retain the ordinal information in the underlying data (GH8836).
- Defined `.size` attribute across `NDFrame` objects to provide compat with numpy >= 1.9.1; buggy with `np.array_split` (GH8846)
- Skip testing of histogram plots for matplotlib <= 1.2 (GH8648).
- Bug where `get_data_google` returned object dtypes (GH3995)
- Bug in `DataFrame.stack(..., dropna=False)` when the DataFrame's `columns` is a `MultiIndex` whose `labels` do not reference all its `levels`. (GH8844)
- Bug in that Option context applied on `__enter__` (GH8514)
- Bug in resample that causes a ValueError when resampling across multiple days and the last offset is not calculated from the start of the range (GH8683)
- Bug where `DataFrame.plot(kind='scatter')` fails when checking if an np.array is in the DataFrame (GH8852)
- Bug in `pd.infer_freq/DataFrame.inferred_freq` that prevented proper sub-daily frequency inference when the index contained DST days (GH8772).
- Bug where index name was still used when plotting a series with `use_index=False` (GH8558).
- Bugs when trying to stack multiple columns, when some (or all) of the level names are numbers (GH8584).
- Bug in `MultiIndex` where `__contains__` returns wrong result if index is not lexically sorted or unique (GH7724)
- BUG CSV: fix problem with trailing white space in skipped rows, (GH8679), (GH8661), (GH8983)
- Regression in `Timestamp` does not parse 'Z' zone designator for UTC (GH8771)
- Bug in *StataWriter* the produces writes strings with 244 characters irrespective of actual size (GH8969)
- Fixed ValueError raised by cummin/cummax when datetime64 Series contains NaT. (GH8965)
- Bug in DataReader returns object dtype if there are missing values (GH8980)
- Bug in plotting if sharex was enabled and index was a timeseries, would show labels on multiple axes (GH3964).
- Bug where passing a unit to the TimedeltaIndex constructor applied the to nano-second conversion twice. (GH9011).
- Bug in plotting of a period-like array (GH9012)

Contributors

A total of 49 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Staple
- Angelos Evripiotis +
- Artemy Kolchinsky

- Benoit Pointet +
- Brian Jacobowski +
- Charalampos Papaloizou +
- Chris Warth +
- David Stephens
- Fabio Zanini +
- Francesc Via +
- Henry Kleynhans +
- Jake VanderPlas +
- Jan Schulz
- Jeff Reback
- Jeff Tratner
- Joris Van den Bossche
- Kevin Sheppard
- Matt Suggit +
- Matthew Brett
- Phillip Cloud
- Rupert Thompson +
- Scott E Lasley +
- Stephan Hoyer
- Stephen Simmons +
- Sylvain Corlay +
- Thomas Grainger +
- Tiago Antao +
- Tom Augspurger
- Trent Hauck
- Victor Chaves +
- Victor Salgado +
- Vikram Bhandoh +
- WANG Aiyong
- Will Holmgren +
- behzad nouri
- broessli +
- charalampos papaloizou +
- immerrr
- jnmclarty

- jreback
- mgilbert +
- onesandzeroes
- peadarcoyle +
- rockg
- seth-p
- sinhrks
- unutbu
- wavedatalab +
- Åsmund Hjulstad +

8.10.2 v0.15.1 (November 9, 2014)

This is a minor bug-fix release from 0.15.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- *Enhancements*
- *API Changes*
- *Bug Fixes*

API changes

- `s.dt.hour` and other `.dt` accessors will now return `np.nan` for missing values (rather than previously -1), (GH8689)

```
In [1]: s = pd.Series(pd.date_range('20130101', periods=5, freq='D'))

In [2]: s.iloc[2] = np.nan

In [3]: s
Out[3]:
0   2013-01-01
1   2013-01-02
2          NaT
3   2013-01-04
4   2013-01-05
Length: 5, dtype: datetime64[ns]
```

previous behavior:

```
In [6]: s.dt.hour
Out[6]:
0    0
1    0
2   -1
3    0
4    0
dtype: int64
```

current behavior:

```
In [4]: s.dt.hour
Out[4]:
0    0.0
1    0.0
2    NaN
3    0.0
4    0.0
Length: 5, dtype: float64
```

- `groupby` with `as_index=False` will not add erroneous extra columns to result (GH8582):

```
In [5]: np.random.seed(2718281)

In [6]: df = pd.DataFrame(np.random.randint(0, 100, (10, 2)),
   ...:                   columns=['jim', 'joe'])
   ...:

In [7]: df.head()
Out[7]:
   jim  joe
0   61   81
1   96   49
2   55   65
3   72   51
4   77   12

[5 rows x 2 columns]

In [8]: ts = pd.Series(5 * np.random.randint(0, 3, 10))
```

previous behavior:

```
In [4]: df.groupby(ts, as_index=False).max()
Out[4]:
   NaN  jim  joe
0    0   72   83
1    5   77   84
2   10   96   65
```

current behavior:

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

[3 rows x 2 columns]
```

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

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

In [11]: df
Out[11]:
```

(continues on next page)

(continued from previous page)

```
   jim  joe
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9

[5 rows x 2 columns]

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

previous behavior (excludes 1st column from output):

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

current behavior:

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

[2 rows x 2 columns]
```

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

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

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

previous behavior:

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

current behavior:

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

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

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

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

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

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

New features:

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

Current behavior:

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
      2014-11-22 call AAPL141122C00110000  1.02
      2014-11-28 call AAPL141128C00110000  1.32
```

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

Enhancements

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

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

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

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

previous behavior:

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

current behavior:

```
In [20]: pd.concat(deque((df1, df2)))
Out[20]: 
   0
0  1
1  2
2  3
0  4
1  5
2  6

[6 rows x 1 columns]
```

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

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

previous behavior:

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

current behavior:

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

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

Bug Fixes

- Bug in unpickling of a `CustomBusinessDay` object (GH8591)
- Bug in coercing `Categorical` to a records array, e.g. `df.to_records()` (GH8626)
- Bug in `Categorical` not created properly with `Series.to_frame()` (GH8626)
- Bug in coercing in astype of a `Categorical` of a passed `pd.Categorical` (this now raises `TypeError` correctly), (GH8626)
- Bug in `cut`/`qcut` when using `Series` and `retbins=True` (GH8589)
- Bug in writing Categorical columns to an SQL database with `to_sql` (GH8624).
- Bug in comparing `Categorical` of datetime raising when being compared to a scalar datetime (GH8687)
- Bug in selecting from a `Categorical` with `.iloc` (GH8623)
- Bug in groupby-transform with a Categorical (GH8623)
- Bug in duplicated/drop_duplicates with a Categorical (GH8623)
- Bug in `Categorical` reflected comparison operator raising if the first argument was a numpy array scalar (e.g. np.int64) (GH8658)
- Bug in Panel indexing with a list-like (GH8710)
- Compat issue is `DataFrame.dtypes` when `options.mode.use_inf_as_null` is True (GH8722)
- Bug in `read_csv`, `dialect` parameter would not take a string (GH8703)
- Bug in slicing a MultiIndex level with an empty-list (GH8737)
- Bug in numeric index operations of add/sub with Float/Index Index with numpy arrays (GH8608)

- Bug in setitem with empty indexer and unwanted coercion of dtypes (GH8669)
- Bug in ix/loc block splitting on setitem (manifests with integer-like dtypes, e.g. datetime64) (GH8607)
- Bug when doing label based indexing with integers not found in the index for non-unique but monotonic indexes (GH8680).
- Bug when indexing a Float64Index with `np.nan` on numpy 1.7 (GH8980).
- Fix `shape` attribute for `MultiIndex` (GH8609)
- Bug in `GroupBy` where a name conflict between the grouper and columns would break `groupby` operations (GH7115, GH8112)
- Fixed a bug where plotting a column `y` and specifying a label would mutate the index name of the original DataFrame (GH8494)
- Fix regression in plotting of a DatetimeIndex directly with matplotlib (GH8614).
- Bug in `date_range` where partially-specified dates would incorporate current date (GH6961)
- Bug in Setting by indexer to a scalar value with a mixed-dtype *Panel4d* was failing (GH8702)
- Bug where `DataReader`'s would fail if one of the symbols passed was invalid. Now returns data for valid symbols and np.nan for invalid (GH8494)
- Bug in `get_quote_yahoo` that wouldn't allow non-float return values (GH5229).

Contributors

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

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

- immerrr
- jnmclarty
- jreback
- pallav-fdsi +
- unutbu

8.10.3 v0.15.0 (October 18, 2014)

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

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

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

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

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

New features

Categoricals in Series/DataFrame

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

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

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

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

In [3]: df["grade"]
Out[3]:
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, Length: 6, dtype: category
Categories (3, object): [a, b, e]

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

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

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

In [7]: df.sort_values("grade")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   id raw_grade      grade
```

(continues on next page)

(continued from previous page)

```
5   6          e   very bad
1   2          b       good
2   3          b       good
0   1          a  very good
3   4          a  very good
4   5          a  very good

[6 rows x 3 columns]

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

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

TimedeltaIndex/Scalar

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

Warning: `Timedelta` scalars (and `TimedeltaIndex`) component fields are *not the same* as the component fields on a `datetime.timedelta` object. For example, `.seconds` on a `datetime.timedelta` object returns the total number of seconds combined between `hours`, `minutes` and `seconds`. In contrast, the pandas `Timedelta` breaks out hours, minutes, microseconds and nanoseconds separately.

```
# Timedelta accessor
In [9]: tds = pd.Timedelta('31 days 5 min 3 sec')

In [10]: tds.minutes
Out[10]: 5L

In [11]: tds.seconds
Out[11]: 3L

# datetime.timedelta accessor
# this is 5 minutes * 60 + 3 seconds
In [12]: tds.to_pytimedelta().seconds
Out[12]: 303
```

Note: this is no longer true starting from v0.16.0, where full compatibility with `datetime.timedelta` is introduced. See the *0.16.0 whatsnew entry*

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

The arguments to `pd.to_timedelta` are now `(arg,unit='ns',box=True,coerce=False)`, previously were `(arg,box=True,unit='ns')` as these are more logical.

Construct a scalar

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

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

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

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

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

Access fields for a `Timedelta`

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

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

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

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

Construct a `TimedeltaIndex`

```
In [18]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05',
   ....:                    np.timedelta64(2, 'D'),
   ....:                    datetime.timedelta(days=2, seconds=2)])
   ....:
Out[18]:
```

(continues on next page)

(continued from previous page)

```
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
                '2 days 00:00:02'],
               dtype='timedelta64[ns]', freq=None)
```

Constructing a `TimedeltaIndex` with a regular range

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

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

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

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

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

You can select with partial string selections

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

In [24]: s['1 day':'1 day 00:00:02']
\\\\\\\\\\\Out[24]:
1 days 00:00:00    0
1 days 00:00:01    1
```

(continues on next page)

(continued from previous page)

```
1 days 00:00:02    2
Freq: S, Length: 3, dtype: int64
```

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

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

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

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

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

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

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

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

Memory Usage

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

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

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

In [32]: n = 5000

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

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

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

In [36]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
int64              5000 non-null int64
float64            5000 non-null float64
```

(continues on next page)

(continued from previous page)

```
datetime64[ns]      5000 non-null datetime64[ns]
timedelta64[ns]     5000 non-null timedelta64[ns]
complex128          5000 non-null complex128
object              5000 non-null object
bool                5000 non-null bool
categorical         5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),␣
↪object(1), timedelta64[ns](1)
memory usage: 289.1+ KB
```

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

```
In [37]: df.memory_usage(index=True)
Out[37]:
Index                 80
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
complex128         80000
object             40000
bool                5000
categorical        10920
Length: 9, dtype: int64
```

.dt accessor

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

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

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

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

In [41]: s.dt.second
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    12
```

(continues on next page)

(continued from previous page)

```
1    12
2    12
3    12
Length: 4, dtype: int64

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

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

This enables nice expressions like this:

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

You can easily produce tz aware transformations:

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

In [46]: stz
Out[46]:
0   2013-01-01 09:10:12-05:00
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
Length: 4, dtype: datetime64[ns, US/Eastern]

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

You can also chain these types of operations:

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

In [54]: s
Out[54]:
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
3   1 days 00:00:08
Length: 4, dtype: timedelta64[ns]

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

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

In [57]: s.dt.components
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0

[4 rows x 7 columns]
```

Timezone handling improvements

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

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

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

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

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

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

In [63]: didx.tz_localize(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
               '2014-08-01 11:00:00', '2014-08-01 12:00:00',
               '2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')
```

- `tz_localize` now accepts the `ambiguous` keyword which allows for passing an array of bools indicating whether the date belongs in DST or not, 'NaT' for setting transition times to NaT, 'infer' for inferring DST/non-DST, and 'raise' (default) for an `AmbiguousTimeError` to be raised. See *the docs* for more details (GH7943)
- `DataFrame.tz_localize` and `DataFrame.tz_convert` now accepts an optional `level` argument for localizing a specific level of a MultiIndex (GH7846)
- `Timestamp.tz_localize` and `Timestamp.tz_convert` now raise `TypeError` in error cases, rather than `Exception` (GH8025)

- a timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone (rather than being a naive `datetime64[ns]`) as `object` dtype (GH8411)
- `Timestamp.__repr__` displays `dateutil.tz.tzoffset` info (GH7907)

Rolling/Expanding Moments improvements

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

  Prior to 0.15.0

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

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

  New behavior

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
3    5
dtype: float64
```

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

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

Behavior prior to 0.15.0:

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

New behavior

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

- Removed `center` argument from all `expanding_` functions (see *list*), as the results produced when `center=True` did not make much sense. (GH7925)
- Added optional `ddof` argument to `expanding_cov()` and `rolling_cov()`. The default value of `1` is backwards-compatible. (GH8279)
- Documented the `ddof` argument to `expanding_var()`, `expanding_std()`, `rolling_var()`, and `rolling_std()`. These functions' support of a `ddof` argument (with a default value of `1`) was previously undocumented. (GH8064)
- `ewma()`, `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now interpret `min_periods` in the same manner that the `rolling_*()` and `expanding_*()` functions do: a given result entry will be `NaN` if the (expanding, in this case) window does not contain at least `min_periods` values. The previous behavior was to set to `NaN` the `min_periods` entries starting with the first non- `NaN` value. (GH7977)

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

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

```
In [51]: ewma(s, com=3., min_periods=2)
Out[51]:
0         NaN
1         NaN
2    1.000000
3    1.000000
```

(continues on next page)

(continued from previous page)

```
4    1.571429
5    2.189189
dtype: float64
```

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

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

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

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

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

In [9]: pd.ewma(pd.Series([1., None, 8.]), com=2.,
   ....:         ignore_na=False)  # new default
Out[9]:
0    1.000000
1    1.000000
2    5.846154
dtype: float64
```

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

- Bug in `expanding_cov()`, `expanding_corr()`, `rolling_cov()`, `rolling_cor()`, `ewmcov()`, and `ewmcorr()` returning results with columns sorted by name and producing an error for non-unique columns; now handles non-unique columns and returns columns in original order (except for the case of two DataFrames with `pairwise=False`, where behavior is unchanged) (GH7542)

- Bug in `rolling_count()` and `expanding_*()` functions unnecessarily producing error message for zero-length data (GH8056)
- Bug in `rolling_apply()` and `expanding_apply()` interpreting `min_periods=0` as `min_periods=1` (GH8080)
- Bug in `expanding_std()` and `expanding_var()` for a single value producing a confusing error message (GH7900)
- Bug in `rolling_std()` and `rolling_var()` for a single value producing `0` rather than `NaN` (GH7900)
- Bug in `ewmstd()`, `ewmvol()`, `ewmvar()`, and `ewmcov()` calculation of de-biasing factors when `bias=False` (the default). Previously an incorrect constant factor was used, based on `adjust=True`, `ignore_na=True`, and an infinite number of observations. Now a different factor is used for each entry, based on the actual weights (analogous to the usual `N/(N-1)` factor). In particular, for a single point a value of `NaN` is returned when `bias=False`, whereas previously a value of (approximately) `0` was returned.

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

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

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

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

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

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

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

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

Improvements in the sql io module

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

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

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

Backwards incompatible API changes

Breaking changes

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

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

  An old function call like (prior to 0.15.0):

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

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

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

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

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

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

Other notable API changes:

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

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

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

[2 rows x 1 columns]
```

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

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

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

Furthermore in prior versions these were also different:

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

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

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

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

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

```
In [70]: p = pd.Panel(np.arange(2 * 3 * 4).reshape(2, 3, 4),
   ....:              items=['ItemA', 'ItemB'],
   ....:              major_axis=[1, 2, 3],
   ....:              minor_axis=['A', 'B', 'C', 'D'])
   ....:

In [71]: p
Out[71]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemB
Major_axis axis: 1 to 3
Minor_axis axis: A to D
```

The following would raise `KeyError` prior to 0.15.0:

```
In [5]:
Out[5]:
   ItemA  ItemD
```

(continues on next page)

(continued from previous page)

```
1       3     NaN
2       7     NaN
3      11     NaN
```

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

```
In [72]: s = pd.Series(np.arange(3, dtype='int64'),
   ....:               index=pd.MultiIndex.from_product([['A'],
   ....:                                                 ['foo', 'bar', 'baz']],
   ....:                                                 names=['one', 'two'])
   ....:               ).sort_index()
   ....:

In [73]: s
Out[73]:
one  two
A    bar    1
     baz    2
     foo    0
Length: 3, dtype: int64

In [74]: try:
   ....:     s.loc[['D']]
   ....: except KeyError as e:
   ....:     print("KeyError: " + str(e))
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\KeyError:
↪"['D'] not in index"
```

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

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

```
In [75]: s = pd.Series([1, 2, 3])

In [76]: s.loc[0] = None

In [77]: s
Out[77]:
0    NaN
1    2.0
2    3.0
Length: 3, dtype: float64
```

  `NaT` is now used similarly for datetime containers.

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

```
In [78]: s = pd.Series(["a", "b", "c"])

In [79]: s.loc[0] = None

In [80]: s
Out[80]:
0    None
1       b
```

(continues on next page)

(continued from previous page)

```
2       c
Length: 3, dtype: object
```

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

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

```
In [81]: s = pd.Series([1, 2, 3])

In [82]: s2 = s

In [83]: s += 1.5
```

Behavior prior to v0.15.0

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


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

This is now the correct behavior

```
# the original object
In [84]: s
Out[84]:
0    2.5
1    3.5
2    4.5
Length: 3, dtype: float64

# a reference to the original object
In [85]: s2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[85]:
0    2.5
1    3.5
2    4.5
Length: 3, dtype: float64
```

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

- Bug in passing a `DatetimeIndex` with a timezone that was not being retained in DataFrame construction from a dict (GH7822)

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

```
In [86]: i = pd.date_range('1/1/2011', periods=3, freq='10s', tz='US/Eastern')

In [87]: i
Out[87]:
DatetimeIndex(['2011-01-01 00:00:00-05:00', '2011-01-01 00:00:10-05:00',
               '2011-01-01 00:00:20-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='10S')

In [88]: df = pd.DataFrame({'a': i})

In [89]: df
Out[89]:
                          a
0 2011-01-01 00:00:00-05:00
1 2011-01-01 00:00:10-05:00
2 2011-01-01 00:00:20-05:00

[3 rows x 1 columns]

In [90]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
a    datetime64[ns, US/Eastern]
Length: 1, dtype: object
```

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

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

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

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

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

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

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

- `merge`, `DataFrame.merge`, and `ordered_merge` now return the same type as the `left` argument (GH7737).

- Previously an enlargement with a mixed-dtype frame would act unlike .append which will preserve dtypes (related GH2578, GH8176):

```
In [91]: df = pd.DataFrame([[True, 1], [False, 2]],
   ....:                   columns=["female", "fitness"])
   ....:

In [92]: df
Out[92]:
   female  fitness
0    True        1
1   False        2

[2 rows x 2 columns]

In [93]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[93]
↪
female      bool
fitness    int64
Length: 2, dtype: object

# dtypes are now preserved
In [94]: df.loc[2] = df.loc[1]

In [95]: df
Out[95]:
   female  fitness
0    True        1
1   False        2
2   False        2

[3 rows x 2 columns]

In [96]: df.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
female      bool
fitness    int64
Length: 2, dtype: object
```

- Series.to_csv() now returns a string when path=None, matching the behaviour of DataFrame.to_csv() (GH8215).
- read_hdf now raises IOError when a file that doesn't exist is passed in. Previously, a new, empty file was created, and a KeyError raised (GH7715).
- DataFrame.info() now ends its output with a newline character (GH8114)
- Concatenating no objects will now raise a ValueError rather than a bare Exception.
- Merge errors will now be sub-classes of ValueError rather than raw Exception (GH8501)
- DataFrame.plot and Series.plot keywords are now have consistent orders (GH8037)

Internal Refactoring

In 0.15.0 Index has internally been refactored to no longer sub-class ndarray but instead subclass PandasObject, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (GH5080, GH7439, GH7796, GH8024, GH8367, GH7997, GH8522):

- you may need to unpickle pandas version < 0.15.0 pickles using `pd.read_pickle` rather than `pickle.load`. See *pickle docs*
- when plotting with a `PeriodIndex`, the matplotlib internal axes will now be arrays of `Period` rather than a `PeriodIndex` (this is similar to how a `DatetimeIndex` passes arrays of `datetimes` now)
- MultiIndexes will now raise similarly to other pandas objects w.r.t. truth testing, see *here* (GH7897).
- When plotting a DatetimeIndex directly with matplotlib's *plot* function, the axis labels will no longer be formatted as dates but as integers (the internal representation of a `datetime64`). **UPDATE** This is fixed in 0.15.1, see *here*.

Deprecations

- The attributes `Categorical labels` and `levels` attributes are deprecated and renamed to `codes` and `categories`.
- The `outtype` argument to `pd.DataFrame.to_dict` has been deprecated in favor of `orient`. (GH7840)
- The `convert_dummies` method has been deprecated in favor of `get_dummies` (GH8140)
- The `infer_dst` argument in `tz_localize` will be deprecated in favor of `ambiguous` to allow for more flexibility in dealing with DST transitions. Replace `infer_dst=True` with `ambiguous='infer'` for the same behavior (GH7943). See *the docs* for more details.
- The top-level `pd.value_range` has been deprecated and can be replaced by `.describe()` (GH8481)
- The `Index` set operations `+` and `-` were deprecated in order to provide these for numeric type operations on certain index types. `+` can be replaced by `.union()` or `|`, and `-` by `.difference()`. Further the method name `Index.diff()` is deprecated and can be replaced by `Index.difference()` (GH8226)

```
# +
pd.Index(['a', 'b', 'c']) + pd.Index(['b', 'c', 'd'])

# should be replaced by
pd.Index(['a', 'b', 'c']).union(pd.Index(['b', 'c', 'd']))
```

```
# -
pd.Index(['a', 'b', 'c']) - pd.Index(['b', 'c', 'd'])

# should be replaced by
pd.Index(['a', 'b', 'c']).difference(pd.Index(['b', 'c', 'd']))
```

- The `infer_types` argument to `read_html()` now has no effect and is deprecated (GH7762, GH7032).

Removal of prior version deprecations/changes

- Remove `DataFrame.delevel` method in favor of `DataFrame.reset_index`

Enhancements

Enhancements in the importing/exporting of Stata files:

- Added support for bool, uint8, uint16 and uint32 data types in `to_stata` (GH7097, GH7365)

- Added conversion option when importing Stata files (GH8527)
- `DataFrame.to_stata` and `StataWriter` check string length for compatibility with limitations imposed in dta files where fixed-width strings must contain 244 or fewer characters. Attempting to write Stata dta files with strings longer than 244 characters raises a `ValueError`. (GH7858)
- `read_stata` and `StataReader` can import missing data information into a `DataFrame` by setting the argument `convert_missing` to `True`. When using this options, missing values are returned as `StataMissingValue` objects and columns containing missing values have `object` data type. (GH8045)

Enhancements in the plotting functions:

- Added `layout` keyword to `DataFrame.plot`. You can pass a tuple of `(rows, columns)`, one of which can be `-1` to automatically infer (GH6667, GH8071).
- Allow to pass multiple axes to `DataFrame.plot`, `hist` and `boxplot` (GH5353, GH6970, GH7069)
- Added support for `c`, `colormap` and `colorbar` arguments for `DataFrame.plot` with `kind='scatter'` (GH7780)
- Histogram from `DataFrame.plot` with `kind='hist'` (GH7809), See *the docs*.
- Boxplot from `DataFrame.plot` with `kind='box'` (GH7998), See *the docs*.

Other:

- `read_csv` now has a keyword parameter `float_precision` which specifies which floating-point converter the C engine should use during parsing, see *here* (GH8002, GH8044)
- Added `searchsorted` method to `Series` objects (GH7447)
- `describe()` on mixed-types DataFrames is more flexible. Type-based column filtering is now possible via the `include`/`exclude` arguments. See the *docs* (GH8164).

```
In [97]: df = pd.DataFrame({'catA': ['foo', 'foo', 'bar'] * 8,
   ....:                    'catB': ['a', 'b', 'c', 'd'] * 6,
   ....:                    'numC': np.arange(24),
   ....:                    'numD': np.arange(24.) + .5})
   ....:

In [98]: df.describe(include=["object"])
Out[98]:
       catA catB
count    24   24
unique    2    4
top     foo    b
freq     16    6

[4 rows x 2 columns]

In [99]: df.describe(include=["number", "object"], exclude=["float"])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
       catA catB       numC
count    24   24  24.000000
unique    2    4        NaN
top     foo    b        NaN
freq     16    6        NaN
mean    NaN  NaN  11.500000
std     NaN  NaN   7.071068
min     NaN  NaN   0.000000
25%     NaN  NaN   5.750000
```

(continues on next page)

(continued from previous page)

```
50%      NaN  NaN  11.500000
75%      NaN  NaN  17.250000
max      NaN  NaN  23.000000

[11 rows x 3 columns]
```

Requesting all columns is possible with the shorthand 'all'

```
In [100]: df.describe(include='all')
Out[100]:
       catA catB       numC       numD
count    24   24  24.000000  24.000000
unique    2    4        NaN        NaN
top     foo    b        NaN        NaN
freq     16    6        NaN        NaN
mean    NaN  NaN  11.500000  12.000000
std     NaN  NaN   7.071068   7.071068
min     NaN  NaN   0.000000   0.500000
25%     NaN  NaN   5.750000   6.250000
50%     NaN  NaN  11.500000  12.000000
75%     NaN  NaN  17.250000  17.750000
max     NaN  NaN  23.000000  23.500000

[11 rows x 4 columns]
```

Without those arguments, `describe` will behave as before, including only numerical columns or, if none are, only categorical columns. See also the *docs*

- Added `split` as an option to the `orient` argument in `pd.DataFrame.to_dict`. (GH7840)
- The `get_dummies` method can now be used on DataFrames. By default only categorical columns are encoded as 0's and 1's, while other columns are left untouched.

```
In [101]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
   .....:                    'C': [1, 2, 3]})
   .....:

In [102]: pd.get_dummies(df)
Out[102]:
   C  A_a  A_b  B_b  B_c
0  1    1    0    0    1
1  2    0    1    0    1
2  3    1    0    1    0

[3 rows x 5 columns]
```

- `PeriodIndex` supports `resolution` as the same as `DatetimeIndex` (GH7708)
- `pandas.tseries.holiday` has added support for additional holidays and ways to observe holidays (GH7070)
- `pandas.tseries.holiday.Holiday` now supports a list of offsets in Python3 (GH7070)
- `pandas.tseries.holiday.Holiday` now supports a days_of_week parameter (GH7070)
- `GroupBy.nth()` now supports selecting multiple nth values (GH7910)

```
In [103]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq=
↪'B')

In [104]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [105]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[105]:
        a  b
2014 4  1  1
     4  1  1
     4  1  1
     5  1  1
     5  1  1
     5  1  1
     6  1  1
     6  1  1
     6  1  1

[9 rows x 2 columns]
```

- `Period` and `PeriodIndex` supports addition/subtraction with `timedelta`-likes (GH7966)

  If `Period` freq is `D`, `H`, `T`, `S`, `L`, `U`, `N`, `Timedelta`-like can be added if the result can have same freq. Otherwise, only the same `offsets` can be added.

```
In [106]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [107]: idx
Out[107]:
PeriodIndex(['2014-07-01 09:00', '2014-07-01 10:00', '2014-07-01 11:00',
             '2014-07-01 12:00', '2014-07-01 13:00'],
            dtype='period[H]', freq='H')

In [108]: idx + pd.offsets.Hour(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
             '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]', freq='H')

In [109]: idx + pd.Timedelta('120m')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
             '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]', freq='H')

In [110]: idx = pd.period_range('2014-07', periods=5, freq='M')

In [111]: idx
Out[111]: PeriodIndex(['2014-07', '2014-08', '2014-09', '2014-10', '2014-11'],␣
↪dtype='period[M]', freq='M')

In [112]: idx + pd.offsets.MonthEnd(3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪PeriodIndex(['2014-10', '2014-11', '2014-12', '2015-01', '2015-02'], dtype=
↪'period[M]', freq='M')
```

- Added experimental compatibility with `openpyxl` for versions >= 2.0. The `DataFrame.to_excel` method `engine` keyword now recognizes `openpyxl1` and `openpyxl2` which will explicitly require openpyxl v1 and v2 respectively, failing if the requested version is not available. The `openpyxl` engine is a now a meta-engine that automatically uses whichever version of openpyxl is installed. (GH7177)
- `DataFrame.fillna` can now accept a `DataFrame` as a fill value (GH8377)
- Passing multiple levels to `stack()` will now work when multiple level numbers are passed (GH7660). See *Reshaping by stacking and unstacking*.
- `set_names()`, `set_labels()`, and `set_levels()` methods now take an optional `level` keyword argument to all modification of specific level(s) of a MultiIndex. Additionally `set_names()` now accepts a scalar string value when operating on an `Index` or on a specific level of a `MultiIndex` (GH7792)

```
In [113]: idx = pd.MultiIndex.from_product([['a'], range(3), list("pqr")],
   .....:                                   names=['foo', 'bar', 'baz'])
   .....:

In [114]: idx.set_names('qux', level=0)
Out[114]:
MultiIndex(levels=[['a'], [0, 1, 2], ['p', 'q', 'r']],
           codes=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1,
↪ 2, 0, 1, 2, 0, 1, 2]],
           names=['qux', 'bar', 'baz'])

In [115]: idx.set_names(['qux', 'corge'], level=[0, 1])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[['a'], [0, 1, 2], ['p', 'q', 'r']],
           codes=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1,
↪ 2, 0, 1, 2, 0, 1, 2]],
           names=['qux', 'corge', 'baz'])

In [116]: idx.set_levels(['a', 'b', 'c'], level='bar')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[['a'], ['a', 'b', 'c'], ['p', 'q', 'r']],
           codes=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1,
↪ 2, 0, 1, 2, 0, 1, 2]],
           names=['foo', 'bar', 'baz'])

In [117]: idx.set_levels([['a', 'b', 'c'], [1, 2, 3]], level=[1, 2])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[['a'], ['a', 'b', 'c'], [1, 2, 3]],
           codes=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1,
↪ 2, 0, 1, 2, 0, 1, 2]],
           names=['foo', 'bar', 'baz'])
```

- `Index.isin` now supports a `level` argument to specify which index level to use for membership tests (GH7892, GH7890)

```
In [1]: idx = pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c']])

In [2]: idx.values
Out[2]: array([(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')],␣
↪dtype=object)
```

(continues on next page)

(continued from previous page)

```
In [3]: idx.isin(['a', 'c', 'e'], level=1)
Out[3]: array([ True, False,  True,  True, False,  True], dtype=bool)
```

- Index now supports duplicated and drop_duplicates. (GH4060)

```
In [118]: idx = pd.Index([1, 2, 3, 4, 1, 2])

In [119]: idx
Out[119]: Int64Index([1, 2, 3, 4, 1, 2], dtype='int64')

In [120]: idx.duplicated()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[120]: array([False,␣
↪False, False, False,  True,  True], dtype=bool)

In [121]: idx.drop_duplicates()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Int64Index([1, 2, 3, 4], dtype='int64')
```

- add copy=True argument to pd.concat to enable pass through of complete blocks (GH8252)
- Added support for numpy 1.8+ data types (bool_, int_, float_, string_) for conversion to R dataframe (GH8400)

Performance

- Performance improvements in DatetimeIndex.__iter__ to allow faster iteration (GH7683)
- Performance improvements in Period creation (and PeriodIndex setitem) (GH5155)
- Improvements in Series.transform for significant performance gains (revised) (GH6496)
- Performance improvements in StataReader when reading large files (GH8040, GH8073)
- Performance improvements in StataWriter when writing large files (GH8079)
- Performance and memory usage improvements in multi-key groupby (GH8128)
- Performance improvements in groupby .agg and .apply where builtins max/min were not mapped to numpy/cythonized versions (GH7722)
- Performance improvement in writing to sql (to_sql) of up to 50% (GH8208).
- Performance benchmarking of groupby for large value of ngroups (GH6787)
- Performance improvement in CustomBusinessDay, CustomBusinessMonth (GH8236)
- Performance improvement for MultiIndex.values for multi-level indexes containing datetimes (GH8543)

Bug Fixes

- Bug in pivot_table, when using margins and a dict aggfunc (GH8349)
- Bug in read_csv where squeeze=True would return a view (GH8217)
- Bug in checking of table name in read_sql in certain cases (GH7826).
- Bug in DataFrame.groupby where Grouper does not recognize level when frequency is specified (GH7885)
- Bug in multiindexes dtypes getting mixed up when DataFrame is saved to SQL table (GH8021)

- Bug in `Series` 0-division with a float and integer operand dtypes (GH7785)
- Bug in `Series.astype("unicode")` not calling `unicode` on the values correctly (GH7758)
- Bug in `DataFrame.as_matrix()` with mixed `datetime64[ns]` and `timedelta64[ns]` dtypes (GH7778)
- Bug in `HDFStore.select_column()` not preserving UTC timezone info when selecting a `DatetimeIndex` (GH7777)
- Bug in `to_datetime` when `format='%Y%m%d'` and `coerce=True` are specified, where previously an object array was returned (rather than a coerced time-series with `NaT`), (GH7930)
- Bug in `DatetimeIndex` and `PeriodIndex` in-place addition and subtraction cause different result from normal one (GH6527)
- Bug in adding and subtracting `PeriodIndex` with `PeriodIndex` raise `TypeError` (GH7741)
- Bug in `combine_first` with `PeriodIndex` data raises `TypeError` (GH3367)
- Bug in MultiIndex slicing with missing indexers (GH7866)
- Bug in MultiIndex slicing with various edge cases (GH8132)
- Regression in MultiIndex indexing with a non-scalar type object (GH7914)
- Bug in `Timestamp` comparisons with `==` and `int64` dtype (GH8058)
- Bug in pickles contains `DateOffset` may raise `AttributeError` when `normalize` attribute is referred internally (GH7748)
- Bug in `Panel` when using `major_xs` and `copy=False` is passed (deprecation warning fails because of missing `warnings`) (GH8152).
- Bug in pickle deserialization that failed for pre-0.14.1 containers with dup items trying to avoid ambiguity when matching block and manager items, when there's only one block there's no ambiguity (GH7794)
- Bug in putting a `PeriodIndex` into a `Series` would convert to `int64` dtype, rather than `object` of `Periods` (GH7932)
- Bug in `HDFStore` iteration when passing a where (GH8014)
- Bug in `DataFrameGroupby.transform` when transforming with a passed non-sorted key (GH8046, GH8430)
- Bug in repeated timeseries line and area plot may result in `ValueError` or incorrect kind (GH7733)
- Bug in inference in a `MultiIndex` with `datetime.date` inputs (GH7888)
- Bug in `get` where an `IndexError` would not cause the default value to be returned (GH7725)
- Bug in `offsets.apply`, `rollforward` and `rollback` may reset nanosecond (GH7697)
- Bug in `offsets.apply`, `rollforward` and `rollback` may raise `AttributeError` if `Timestamp` has `dateutil` tzinfo (GH7697)
- Bug in sorting a MultiIndex frame with a `Float64Index` (GH8017)
- Bug in inconsistent panel setitem with a rhs of a `DataFrame` for alignment (GH7763)
- Bug in `is_superperiod` and `is_subperiod` cannot handle higher frequencies than `S` (GH7760, GH7772, GH7803)
- Bug in 32-bit platforms with `Series.shift` (GH8129)
- Bug in `PeriodIndex.unique` returns int64 `np.ndarray` (GH7540)
- Bug in `groupby.apply` with a non-affecting mutation in the function (GH8467)

- Bug in `DataFrame.reset_index` which has `MultiIndex` contains `PeriodIndex` or `DatetimeIndex` with tz raises `ValueError` (GH7746, GH7793)
- Bug in `DataFrame.plot` with `subplots=True` may draw unnecessary minor xticks and yticks (GH7801)
- Bug in `StataReader` which did not read variable labels in 117 files due to difference between Stata documentation and implementation (GH7816)
- Bug in `StataReader` where strings were always converted to 244 characters-fixed width irrespective of underlying string size (GH7858)
- Bug in `DataFrame.plot` and `Series.plot` may ignore `rot` and `fontsize` keywords (GH7844)
- Bug in `DatetimeIndex.value_counts` doesn't preserve tz (GH7735)
- Bug in `PeriodIndex.value_counts` results in `Int64Index` (GH7735)
- Bug in `DataFrame.join` when doing left join on index and there are multiple matches (GH5391)
- Bug in `GroupBy.transform()` where int groups with a transform that didn't preserve the index were incorrectly truncated (GH7972).
- Bug in `groupby` where callable objects without name attributes would take the wrong path, and produce a `DataFrame` instead of a `Series` (GH7929)
- Bug in `groupby` error message when a DataFrame grouping column is duplicated (GH7511)
- Bug in `read_html` where the `infer_types` argument forced coercion of date-likes incorrectly (GH7762, GH7032).
- Bug in `Series.str.cat` with an index which was filtered as to not include the first item (GH7857)
- Bug in `Timestamp` cannot parse `nanosecond` from string (GH7878)
- Bug in `Timestamp` with string offset and `tz` results incorrect (GH7833)
- Bug in `tslib.tz_convert` and `tslib.tz_convert_single` may return different results (GH7798)
- Bug in `DatetimeIndex.intersection` of non-overlapping timestamps with tz raises `IndexError` (GH7880)
- Bug in alignment with TimeOps and non-unique indexes (GH8363)
- Bug in `GroupBy.filter()` where fast path vs. slow path made the filter return a non scalar value that appeared valid but wasn't (GH7870).
- Bug in `date_range()`/`DatetimeIndex()` when the timezone was inferred from input dates yet incorrect times were returned when crossing DST boundaries (GH7835, GH7901).
- Bug in `to_excel()` where a negative sign was being prepended to positive infinity and was absent for negative infinity (GH7949)
- Bug in area plot draws legend with incorrect `alpha` when `stacked=True` (GH8027)
- `Period` and `PeriodIndex` addition/subtraction with `np.timedelta64` results in incorrect internal representations (GH7740)
- Bug in `Holiday` with no offset or observance (GH7987)
- Bug in `DataFrame.to_latex` formatting when columns or index is a `MultiIndex` (GH7982).
- Bug in `DateOffset` around Daylight Savings Time produces unexpected results (GH5175).
- Bug in `DataFrame.shift` where empty columns would throw `ZeroDivisionError` on numpy 1.7 (GH8019)

- Bug in installation where `html_encoding/*.html` wasn't installed and therefore some tests were not running correctly (GH7927).
- Bug in `read_html` where `bytes` objects were not tested for in `_read` (GH7927).
- Bug in `DataFrame.stack()` when one of the column levels was a datelike (GH8039)
- Bug in broadcasting numpy scalars with `DataFrame` (GH8116)
- Bug in `pivot_table` performed with nameless `index` and `columns` raises `KeyError` (GH8103)
- Bug in `DataFrame.plot(kind='scatter')` draws points and errorbars with different colors when the color is specified by `c` keyword (GH8081)
- Bug in `Float64Index` where `iat` and `at` were not testing and were failing (GH8092).
- Bug in `DataFrame.boxplot()` where y-limits were not set correctly when producing multiple axes (GH7528, GH5517).
- Bug in `read_csv` where line comments were not handled correctly given a custom line terminator or `delim_whitespace=True` (GH8122).
- Bug in `read_html` where empty tables caused a `StopIteration` (GH7575)
- Bug in casting when setting a column in a same-dtype block (GH7704)
- Bug in accessing groups from a `GroupBy` when the original grouper was a tuple (GH8121).
- Bug in `.at` that would accept integer indexers on a non-integer index and do fallback (GH7814)
- Bug with kde plot and NaNs (GH8182)
- Bug in `GroupBy.count` with float32 data type were nan values were not excluded (GH8169).
- Bug with stacked barplots and NaNs (GH8175).
- Bug in resample with non evenly divisible offsets (e.g. '7s') (GH8371)
- Bug in interpolation methods with the `limit` keyword when no values needed interpolating (GH7173).
- Bug where `col_space` was ignored in `DataFrame.to_string()` when `header=False` (GH8230).
- Bug with `DatetimeIndex.asof` incorrectly matching partial strings and returning the wrong date (GH8245).
- Bug in plotting methods modifying the global matplotlib rcParams (GH8242).
- Bug in `DataFrame.__setitem__` that caused errors when setting a dataframe column to a sparse array (GH8131)
- Bug where `Dataframe.boxplot()` failed when entire column was empty (GH8181).
- Bug with messed variables in `radviz` visualization (GH8199).
- Bug in interpolation methods with the `limit` keyword when no values needed interpolating (GH7173).
- Bug where `col_space` was ignored in `DataFrame.to_string()` when `header=False` (GH8230).
- Bug in `to_clipboard` that would clip long column data (GH8305)
- Bug in `DataFrame` terminal display: Setting max_column/max_rows to zero did not trigger auto-resizing of dfs to fit terminal width/height (GH7180).
- Bug in OLS where running with "cluster" and "nw_lags" parameters did not work correctly, but also did not throw an error (GH5884).
- Bug in `DataFrame.dropna` that interpreted non-existent columns in the subset argument as the 'last column' (GH8303)

- Bug in `Index.intersection` on non-monotonic non-unique indexes (GH8362).
- Bug in masked series assignment where mismatching types would break alignment (GH8387)
- Bug in `NDFrame.equals` gives false negatives with dtype=object (GH8437)
- Bug in assignment with indexer where type diversity would break alignment (GH8258)
- Bug in `NDFrame.loc` indexing when row/column names were lost when target was a list/ndarray (GH6552)
- Regression in `NDFrame.loc` indexing when rows/columns were converted to Float64Index if target was an empty list/ndarray (GH7774)
- Bug in `Series` that allows it to be indexed by a `DataFrame` which has unexpected results. Such indexing is no longer permitted (GH8444)
- Bug in item assignment of a `DataFrame` with MultiIndex columns where right-hand-side columns were not aligned (GH7655)
- Suppress FutureWarning generated by NumPy when comparing object arrays containing NaN for equality (GH7065)
- Bug in `DataFrame.eval()` where the dtype of the `not` operator (~) was not correctly inferred as `bool`.

Contributors

A total of 80 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aaron Schumacher +
- Adam Greenhall
- Andy Hayden
- Anthony O'Brien +
- Artemy Kolchinsky +
- Ben Schiller +
- Benedikt Sauer
- Benjamin Thyreau +
- BorisVerk +
- Chris Reynolds +
- Chris Stoafer +
- DSM
- Dav Clark +
- FragLegs +
- German Gomez-Herrero +
- Hsiaoming Yang +
- Huan Li +
- Hyungtae Kim +
- Isaac Slavitt +
- Jacob Schaer

- Jacob Wasserman +
- Jan Schulz
- Jeff Reback
- Jeff Tratner
- Jesse Farnham +
- Joe Bradish +
- Joerg Rittinger +
- John W. O'Brien
- Joris Van den Bossche
- Kevin Sheppard
- Kyle Meyer
- Max Chang +
- Michael Mueller
- Michael W Schatzow +
- Mike Kelly
- Mortada Mehyar
- Nathan Sanders +
- Nathan Typanski +
- Paul Masurel +
- Phillip Cloud
- Pietro Battiston
- RenzoBertocchi +
- Ross Petchler +
- Shahul Hameed +
- Shashank Agarwal +
- Stephan Hoyer
- Tom Augspurger
- TomAugspurger
- Tony Lorenzo +
- Wes Turner
- Wilfred Hughes +
- Yevgeniy Grechka +
- Yoshiki Vázquez Baeza +
- behzad nouri +
- benjamin
- bjonen +

- dlovell +
- dsm054
- hunterowens +
- immerrr
- ischwabacher
- jmorris0x0 +
- jnmclarty +
- jreback
- klonuo +
- lexual
- mcjcode +
- mtrbean +
- onesandzeroes
- rockg
- seth-p
- sinhrks
- someben +
- stahlous +
- stas-sl +
- thatneat +
- tom-alcorn +
- unknown
- unutbu
- zachcp +

8.11 Version 0.14

8.11.1 v0.14.1 (July 11, 2014)

This is a minor release from 0.14.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- Highlights include:
  - New methods `select_dtypes()` to select columns based on the dtype and `sem()` to calculate the standard error of the mean.
  - Support for dateutil timezones (see *docs*).
  - Support for ignoring full line comments in the `read_csv()` text parser.
  - New documentation section on *Options and Settings*.

  - Lots of bug fixes.

- *Enhancements*
- *API Changes*
- *Performance Improvements*
- *Experimental Changes*
- *Bug Fixes*

API changes

- Openpyxl now raises a ValueError on construction of the openpyxl writer instead of warning on pandas import (GH7284).
- For `StringMethods.extract`, when no match is found, the result - only containing `NaN` values - now also has `dtype=object` instead of `float` (GH7242)
- `Period` objects no longer raise a `TypeError` when compared using `==` with another object that *isn't* a `Period`. Instead when comparing a `Period` with another object using `==` if the other object isn't a `Period` `False` is returned. (GH7376)
- Previously, the behaviour on resetting the time or not in `offsets.apply`, `rollforward` and `rollback` operations differed between offsets. With the support of the `normalize` keyword for all offsets(see below) with a default value of False (preserve time), the behaviour changed for certain offsets (BusinessMonthBegin, MonthEnd, BusinessMonthEnd, CustomBusinessMonthEnd, BusinessYearBegin, LastWeekOfMonth, FY5253Quarter, LastWeekOfMonth, Easter):

```
In [6]: from pandas.tseries import offsets

In [7]: d = pd.Timestamp('2014-01-01 09:00')

# old behaviour < 0.14.1
In [8]: d + offsets.MonthEnd()
Out[8]: pd.Timestamp('2014-01-31 00:00:00')
```

  Starting from 0.14.1 all offsets preserve time by default. The old behaviour can be obtained with `normalize=True`

```
# new behaviour
In [1]: d + offsets.MonthEnd()
Out[1]: Timestamp('2014-01-31 09:00:00')

In [2]: d + offsets.MonthEnd(normalize=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2]: Timestamp('2014-01-31 00:00:00')
```

  Note that for the other offsets the default behaviour did not change.

- Add back `#N/A N/A` as a default NA value in text parsing, (regression from 0.12) (GH5521)
- Raise a `TypeError` on inplace-setting with a `.where` and a non `np.nan` value as this is inconsistent with a set-item expression like `df[mask] = None` (GH7656)

Enhancements

- Add `dropna` argument to `value_counts` and `nunique` (GH5569).
- Add `select_dtypes()` method to allow selection of columns based on dtype (GH7316). See *the docs*.

- All `offsets` supports the `normalize` keyword to specify whether `offsets.apply`, `rollforward` and `rollback` resets the time (hour, minute, etc) or not (default `False`, preserves time) (GH7156):

```
import pandas.tseries.offsets as offsets

day = offsets.Day()
day.apply(pd.Timestamp('2014-01-01 09:00'))

day = offsets.Day(normalize=True)
day.apply(pd.Timestamp('2014-01-01 09:00'))
```

- `PeriodIndex` is represented as the same format as `DatetimeIndex` (GH7601)
- `StringMethods` now work on empty Series (GH7242)
- The file parsers `read_csv` and `read_table` now ignore line comments provided by the parameter *comment*, which accepts only a single character for the C reader. In particular, they allow for comments before file data begins (GH2685)
- Add `NotImplementedError` for simultaneous use of `chunksize` and `nrows` for read_csv() (GH6774).
- Tests for basic reading of public S3 buckets now exist (GH7281).
- `read_html` now sports an `encoding` argument that is passed to the underlying parser library. You can use this to read non-ascii encoded web pages (GH7323).
- `read_excel` now supports reading from URLs in the same way that `read_csv` does. (GH6809)
- Support for dateutil timezones, which can now be used in the same way as pytz timezones across pandas. (GH4688)

```
In [3]: rng = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
   ...:                     tz='dateutil/Europe/London')
   ...:

In [4]: rng.tz
Out[4]: tzfile('/usr/share/zoneinfo/Europe/London')
```

  See *the docs*.

- Implemented `sem` (standard error of the mean) operation for `Series`, `DataFrame`, `Panel`, and `Groupby` (GH6897)
- Add `nlargest` and `nsmallest` to the `Series` `groupby` whitelist, which means you can now use these methods on a `SeriesGroupBy` object (GH7053).
- All offsets `apply`, `rollforward` and `rollback` can now handle `np.datetime64`, previously results in `ApplyTypeError` (GH7452)
- `Period` and `PeriodIndex` can contain `NaT` in its values (GH7485)
- Support pickling `Series`, `DataFrame` and `Panel` objects with non-unique labels along *item* axis (`index`, `columns` and `items` respectively) (GH7370).
- Improved inference of datetime/timedelta with mixed null objects. Regression from 0.13.1 in interpretation of an object Index with all null elements (GH7431)

Performance

- Improvements in dtype inference for numeric operations involving yielding performance gains for dtypes: `int64`, `timedelta64`, `datetime64` (GH7223)

- Improvements in Series.transform for significant performance gains (GH6496)
- Improvements in DataFrame.transform with ufuncs and built-in grouper functions for significant performance gains (GH7383)
- Regression in groupby aggregation of datetime64 dtypes (GH7555)
- Improvements in *MultiIndex.from_product* for large iterables (GH7627)

Experimental

- `pandas.io.data.Options` has a new method, `get_all_data` method, and now consistently returns a MultiIndexed `DataFrame` (GH5602)
- `io.gbq.read_gbq` and `io.gbq.to_gbq` were refactored to remove the dependency on the Google `bq.py` command line client. This submodule now uses `httplib2` and the Google `apiclient` and `oauth2client` API client libraries which should be more stable and, therefore, reliable than `bq.py`. See *the docs*. (GH6937).

Bug Fixes

- Bug in `DataFrame.where` with a symmetric shaped frame and a passed other of a DataFrame (GH7506)
- Bug in Panel indexing with a MultiIndex axis (GH7516)
- Regression in datetimelike slice indexing with a duplicated index and non-exact end-points (GH7523)
- Bug in setitem with list-of-lists and single vs mixed types (GH7551:)
- Bug in time ops with non-aligned Series (GH7500)
- Bug in timedelta inference when assigning an incomplete Series (GH7592)
- Bug in groupby `.nth` with a Series and integer-like column name (GH7559)
- Bug in `Series.get` with a boolean accessor (GH7407)
- Bug in `value_counts` where `NaT` did not qualify as missing (`NaN`) (GH7423)
- Bug in `to_timedelta` that accepted invalid units and misinterpreted 'm/h' (GH7611, GH6423)
- Bug in line plot doesn't set correct `xlim` if `secondary_y=True` (GH7459)
- Bug in grouped `hist` and `scatter` plots use old `figsize` default (GH7394)
- Bug in plotting subplots with `DataFrame.plot`, `hist` clears passed `ax` even if the number of subplots is one (GH7391).
- Bug in plotting subplots with `DataFrame.boxplot` with `by` kw raises `ValueError` if the number of subplots exceeds 1 (GH7391).
- Bug in subplots displays `ticklabels` and `labels` in different rule (GH5897)
- Bug in `Panel.apply` with a MultiIndex as an axis (GH7469)
- Bug in `DatetimeIndex.insert` doesn't preserve `name` and `tz` (GH7299)
- Bug in `DatetimeIndex.asobject` doesn't preserve `name` (GH7299)
- Bug in MultiIndex slicing with datetimelike ranges (strings and Timestamps), (GH7429)
- Bug in `Index.min` and `max` doesn't handle `nan` and `NaT` properly (GH7261)
- Bug in `PeriodIndex.min/max` results in `int` (GH7609)

- Bug in `resample` where `fill_method` was ignored if you passed `how` (GH2073)
- Bug in `TimeGrouper` doesn't exclude column specified by `key` (GH7227)
- Bug in `DataFrame` and `Series` bar and barh plot raises `TypeError` when `bottom` and `left` keyword is specified (GH7226)
- Bug in `DataFrame.hist` raises `TypeError` when it contains non numeric column (GH7277)
- Bug in `Index.delete` does not preserve `name` and `freq` attributes (GH7302)
- Bug in `DataFrame.query()`/`eval` where local string variables with the @ sign were being treated as temporaries attempting to be deleted (GH7300).
- Bug in `Float64Index` which didn't allow duplicates (GH7149).
- Bug in `DataFrame.replace()` where truthy values were being replaced (GH7140).
- Bug in `StringMethods.extract()` where a single match group Series would use the matcher's name instead of the group name (GH7313).
- Bug in `isnull()` when `mode.use_inf_as_null == True` where isnull wouldn't test `True` when it encountered an `inf`/`-inf` (GH7315).
- Bug in inferred_freq results in None for eastern hemisphere timezones (GH7310)
- Bug in `Easter` returns incorrect date when offset is negative (GH7195)
- Bug in broadcasting with `.div`, integer dtypes and divide-by-zero (GH7325)
- Bug in `CustomBusinessDay.apply` raises `NameError` when `np.datetime64` object is passed (GH7196)
- Bug in `MultiIndex.append`, `concat` and `pivot_table` don't preserve timezone (GH6606)
- Bug in `.loc` with a list of indexers on a single-multi index level (that is not nested) (GH7349)
- Bug in `Series.map` when mapping a dict with tuple keys of different lengths (GH7333)
- Bug all `StringMethods` now work on empty Series (GH7242)
- Fix delegation of *read_sql* to *read_sql_query* when query does not contain 'select' (GH7324).
- Bug where a string column name assignment to a `DataFrame` with a `Float64Index` raised a `TypeError` during a call to `np.isnan` (GH7366).
- Bug where `NDFrame.replace()` didn't correctly replace objects with `Period` values (GH7379).
- Bug in `.ix` getitem should always return a Series (GH7150)
- Bug in MultiIndex slicing with incomplete indexers (GH7399)
- Bug in MultiIndex slicing with a step in a sliced level (GH7400)
- Bug where negative indexers in `DatetimeIndex` were not correctly sliced (GH7408)
- Bug where `NaT` wasn't repr'd correctly in a `MultiIndex` (GH7406, GH7409).
- Bug where bool objects were converted to `nan` in `convert_objects` (GH7416).
- Bug in `quantile` ignoring the axis keyword argument (GH7306)
- Bug where `nanops._maybe_null_out` doesn't work with complex numbers (GH7353)
- Bug in several `nanops` functions when `axis==0` for 1-dimensional `nan` arrays (GH7354)
- Bug where `nanops.nanmedian` doesn't work when `axis==None` (GH7352)
- Bug where `nanops._has_infs` doesn't work with many dtypes (GH7357)

- Bug in StataReader.data where reading a 0-observation dta failed (GH7369)
- Bug in StataReader when reading Stata 13 (117) files containing fixed width strings (GH7360)
- Bug in StataWriter where encoding was ignored (GH7286)
- Bug in DatetimeIndex comparison doesn't handle NaT properly (GH7529)
- Bug in passing input with tzinfo to some offsets apply, rollforward or rollback resets tzinfo or raises ValueError (GH7465)
- Bug in DatetimeIndex.to_period, PeriodIndex.asobject, PeriodIndex.to_timestamp doesn't preserve name (GH7485)
- Bug in DatetimeIndex.to_period and PeriodIndex.to_timestanp handle NaT incorrectly (GH7228)
- Bug in offsets.apply, rollforward and rollback may return normal datetime (GH7502)
- Bug in resample raises ValueError when target contains NaT (GH7227)
- Bug in Timestamp.tz_localize resets nanosecond info (GH7534)
- Bug in DatetimeIndex.asobject raises ValueError when it contains NaT (GH7539)
- Bug in Timestamp.__new__ doesn't preserve nanosecond properly (GH7610)
- Bug in Index.astype(float) where it would return an object dtype Index (GH7464).
- Bug in DataFrame.reset_index loses tz (GH3950)
- Bug in DatetimeIndex.freqstr raises AttributeError when freq is None (GH7606)
- Bug in GroupBy.size created by TimeGrouper raises AttributeError (GH7453)
- Bug in single column bar plot is misaligned (GH7498).
- Bug in area plot with tz-aware time series raises ValueError (GH7471)
- Bug in non-monotonic Index.union may preserve name incorrectly (GH7458)
- Bug in DatetimeIndex.intersection doesn't preserve timezone (GH4690)
- Bug in rolling_var where a window larger than the array would raise an error(GH7297)
- Bug with last plotted timeseries dictating xlim (GH2960)
- Bug with secondary_y axis not being considered for timeseries xlim (GH3490)
- Bug in Float64Index assignment with a non scalar indexer (GH7586)
- Bug in pandas.core.strings.str_contains does not properly match in a case insensitive fashion when regex=False and case=False (GH7505)
- Bug in expanding_cov, expanding_corr, rolling_cov, and rolling_corr for two arguments with mismatched index (GH7512)
- Bug in to_sql taking the boolean column as text column (GH7678)
- Bug in grouped *hist* doesn't handle *rot* kw and *sharex* kw properly (GH7234)
- Bug in .loc performing fallback integer indexing with object dtype indices (GH7496)
- Bug (regression) in PeriodIndex constructor when passed Series objects (GH7701).

Contributors

A total of 46 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Andrew Rosenfeld
- Andy Hayden
- Benjamin Adams +
- Benjamin M. Gross +
- Brian Quistorff +
- Brian Wignall +
- DSM
- Daniel Waeber
- David Bew +
- David Stephens
- Jacob Schaer
- Jan Schulz
- John David Reaver
- John W. O'Brien
- Joris Van den Bossche
- Julien Danjou +
- K.-Michael Aye
- Kevin Sheppard
- Kyle Meyer
- Matt Wittmann
- Matthew Brett +
- Michael Mueller +
- Mortada Mehyar
- Phillip Cloud
- Rob Levy +
- Schaer, Jacob C +
- Stephan Hoyer
- Thomas Kluyver
- Todd Jennings
- Tom Augspurger
- TomAugspurger
- bwignall
- clham

- dsm054 +
- helger +
- immerrr
- jaimefrio
- jreback
- lexual
- onesandzeroes
- rockg
- sanguineturtle +
- seth-p +
- sinhrks
- unknown
- yelite +

8.11.2 v0.14.0 (May 31 , 2014)

This is a major release from 0.13.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- Highlights include:
  - Officially support Python 3.4
  - SQL interfaces updated to use `sqlalchemy`, See *Here*.
  - Display interface changes, See *Here*
  - MultiIndexing Using Slicers, See *Here*.
  - Ability to join a singly-indexed DataFrame with a MultiIndexed DataFrame, see *Here*
  - More consistency in groupby results and more flexible groupby specifications, See *Here*
  - Holiday calendars are now supported in `CustomBusinessDay`, see *Here*
  - Several improvements in plotting functions, including: hexbin, area and pie plots, see *Here*.
  - Performance doc section on I/O operations, See *Here*
- *Other Enhancements*
- *API Changes*
- *Text Parsing API Changes*
- *Groupby API Changes*
- *Performance Improvements*
- *Prior Deprecations*
- *Deprecations*
- *Known Issues*
- *Bug Fixes*

Warning: In 0.14.0 all `NDFrame` based containers have undergone significant internal refactoring. Before that each block of homogeneous data had its own labels and extra care was necessary to keep those in sync with the parent container's labels. This should not have any visible user/API behavior changes (GH6745)

API changes

- `read_excel` uses 0 as the default sheet (GH6573)
- `iloc` will now accept out-of-bounds indexers for slices, e.g. a value that exceeds the length of the object being indexed. These will be excluded. This will make pandas conform more with python/numpy indexing of out-of-bounds values. A single indexer that is out-of-bounds and drops the dimensions of the object will still raise `IndexError` (GH6296, GH6299). This could result in an empty axis (e.g. an empty DataFrame being returned)

```
In [1]: dfl = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [2]: dfl
Out[2]:
          A         B
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569

[5 rows x 2 columns]

In [3]: dfl.iloc[:, 2:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

[5 rows x 0 columns]

In [4]: dfl.iloc[:, 1:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          B
0 -0.282863
1 -1.135632
2 -0.173215
3 -1.044236
4 -2.104569

[5 rows x 1 columns]

In [5]: dfl.iloc[4:6]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B
4 -0.861849 -2.104569

[1 rows x 2 columns]
```

These are out-of-bounds selections

```
>>> dfl.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds

>>> dfl.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

- Slicing with negative start, stop & step values handles corner cases better (GH6531):
  - `df.iloc[:-len(df)]` is now empty
  - `df.iloc[len(df)::-1]` now enumerates all elements in reverse
- The *DataFrame.interpolate()* keyword `downcast` default has been changed from `infer` to `None`. This is to preserve the original dtype unless explicitly requested otherwise (GH6290).
- When converting a dataframe to HTML it used to return *Empty DataFrame*. This special case has been removed, instead a header with the column names is returned (GH6062).
- `Series` and `Index` now internally share more common operations, e.g. `factorize(),nunique()`, `value_counts()` are now supported on `Index` types as well. The `Series.weekday` property from is removed from Series for API consistency. Using a `DatetimeIndex/PeriodIndex` method on a Series will now raise a `TypeError`. (GH4551, GH4056, GH5519, GH6380, GH7206).
- Add `is_month_start`, `is_month_end`, `is_quarter_start`, `is_quarter_end`, `is_year_start`, `is_year_end` accessors for `DateTimeIndex` / `Timestamp` which return a boolean array of whether the timestamp(s) are at the start/end of the month/quarter/year defined by the frequency of the `DateTimeIndex` / `Timestamp` (GH4565, GH6998)
- Local variable usage has changed in *pandas.eval()*/*DataFrame.eval()*/*DataFrame.query()* (GH5987). For the *DataFrame* methods, two things have changed
  - Column names are now given precedence over locals
  - Local variables must be referred to explicitly. This means that even if you have a local variable that is *not* a column you must still refer to it with the `'@'` prefix.
  - You can have an expression like `df.query('@a < a')` with no complaints from `pandas` about ambiguity of the name `a`.
  - The top-level *pandas.eval()* function does not allow you use the `'@'` prefix and provides you with an error message telling you so.
  - `NameResolutionError` was removed because it isn't necessary anymore.
- Define and document the order of column vs index names in query/eval (GH6676)
- `concat` will now concatenate mixed Series and DataFrames using the Series name or numbering columns as needed (GH2385). See *the docs*
- Slicing and advanced/boolean indexing operations on `Index` classes as well as *Index.delete()* and *Index.drop()* methods will no longer change the type of the resulting index (GH6440, GH7040)

```
In [6]: i = pd.Index([1, 2, 3, 'a', 'b', 'c'])

In [7]: i[[0, 1, 2]]
Out[7]: Index([1, 2, 3], dtype='object')

In [8]: i.drop(['a', 'b', 'c'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]: Index([1, 2, 3], dtype='object')
```

Previously, the above operation would return `Int64Index`. If you'd like to do this manually, use *Index.astype()*

```
In [9]: i[[0, 1, 2]].astype(np.int_)
Out[9]: Int64Index([1, 2, 3], dtype='int64')
```

- `set_index` no longer converts MultiIndexes to an Index of tuples. For example, the old behavior returned an Index in this case (GH6459):

```
# Old behavior, casted MultiIndex to an Index
In [10]: tuple_ind
Out[10]: Index([('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')], dtype='object')

In [11]: df_multi.set_index(tuple_ind)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]: 
↪
               0         1
(a, c)  0.471435 -1.190976
(a, d)  1.432707 -0.312652
(b, c) -0.720589  0.887163
(b, d)  0.859588 -0.636524

[4 rows x 2 columns]

# New behavior
In [12]: mi
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
MultiIndex(levels=[['a', 'b'], ['c', 'd']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])

In [13]: df_multi.set_index(mi)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            0         1
a c  0.471435 -1.190976
  d  1.432707 -0.312652
b c -0.720589  0.887163
  d  0.859588 -0.636524

[4 rows x 2 columns]
```

This also applies when passing multiple indices to `set_index`:

```
# Old output, 2-level MultiIndex of tuples
In [14]: df_multi.set_index([df_multi.index, df_multi.index])
Out[14]:
                      0         1
(a, c) (a, c)  0.471435 -1.190976
(a, d) (a, d)  1.432707 -0.312652
(b, c) (b, c) -0.720589  0.887163
(b, d) (b, d)  0.859588 -0.636524

[4 rows x 2 columns]

# New output, 4-level MultiIndex
In [15]: df_multi.set_index([df_multi.index, df_multi.index])
Out[15]:
```

(continues on next page)

(continued from previous page)

```
                  0         1
a c a c  0.471435 -1.190976
  d a d  1.432707 -0.312652
b c b c -0.720589  0.887163
  d b d  0.859588 -0.636524

[4 rows x 2 columns]
```

- `pairwise` keyword was added to the statistical moment functions `rolling_cov`, `rolling_corr`, `ewmcov`, `ewmcorr`, `expanding_cov`, `expanding_corr` to allow the calculation of moving window covariance and correlation matrices (GH4950). See *Computing rolling pairwise covariances and correlations* in the docs.

```
In [1]: df = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'))

In [4]: covs = pd.rolling_cov(df[['A', 'B', 'C']],
  ....:                       df[['B', 'C', 'D']],
  ....:                       5,
  ....:                       pairwise=True)


In [5]: covs[df.index[-1]]
Out[5]:
          B         C         D
A  0.035310  0.326593 -0.505430
B  0.137748 -0.006888 -0.005383
C -0.006888  0.861040  0.020762
```

- `Series.iteritems()` is now lazy (returns an iterator rather than a list). This was the documented behavior prior to 0.14. (GH6760)
- Added `nunique` and `value_counts` functions to `Index` for counting unique elements. (GH6734)
- `stack` and `unstack` now raise a `ValueError` when the `level` keyword refers to a non-unique item in the `Index` (previously raised a `KeyError`). (GH6738)
- drop unused order argument from `Series.sort`; args now are in the same order as `Series.order`; add `na_position` arg to conform to `Series.order` (GH6847)
- default sorting algorithm for `Series.order` is now `quicksort`, to conform with `Series.sort` (and numpy defaults)
- add `inplace` keyword to `Series.order/sort` to make them inverses (GH6859)
- `DataFrame.sort` now places NaNs at the beginning or end of the sort according to the `na_position` parameter. (GH3917)
- accept `TextFileReader` in `concat`, which was affecting a common user idiom (GH6583), this was a regression from 0.13.1
- Added `factorize` functions to `Index` and `Series` to get indexer and unique values (GH7090)
- `describe` on a DataFrame with a mix of Timestamp and string like objects returns a different Index (GH7088). Previously the index was unintentionally sorted.
- Arithmetic operations with **only** `bool` dtypes now give a warning indicating that they are evaluated in Python space for `+`, `-`, and `*` operations and raise for all others (GH7011, GH6762, GH7015, GH7210)

```
>>> x = pd.Series(np.random.rand(10) > 0.5)
>>> y = True
>>> x + y  # warning generated: should do x | y instead
UserWarning: evaluating in Python space because the '+' operator is not
supported by numexpr for the bool dtype, use '|' instead
>>> x / y  # this raises because it doesn't make sense
NotImplementedError: operator '/' not implemented for bool dtypes
```

- In `HDFStore`, `select_as_multiple` will always raise a `KeyError`, when a key or the selector is not found (GH6177)
- `df['col'] = value` and `df.loc[:,'col'] = value` are now completely equivalent; previously the `.loc` would not necessarily coerce the dtype of the resultant series (GH6149)
- `dtypes` and `ftypes` now return a series with `dtype=object` on empty containers (GH5740)
- `df.to_csv` will now return a string of the CSV data if neither a target path nor a buffer is provided (GH6061)
- `pd.infer_freq()` will now raise a `TypeError` if given an invalid `Series/Index` type (GH6407, GH6463)
- A tuple passed to `DataFame.sort_index` will be interpreted as the levels of the index, rather than requiring a list of tuple (GH4370)
- all offset operations now return `Timestamp` types (rather than datetime), Business/Week frequencies were incorrect (GH4069)
- `to_excel` now converts `np.inf` into a string representation, customizable by the `inf_rep` keyword argument (Excel has no native inf representation) (GH6782)
- Replace `pandas.compat.scipy.scoreatpercentile` with `numpy.percentile` (GH6810)
- `.quantile` on a `datetime[ns]` series now returns `Timestamp` instead of `np.datetime64` objects (GH6810)
- change `AssertionError` to `TypeError` for invalid types passed to `concat` (GH6583)
- Raise a `TypeError` when `DataFrame` is passed an iterator as the `data` argument (GH5357)

Display Changes

- The default way of printing large DataFrames has changed. DataFrames exceeding `max_rows` and/or `max_columns` are now displayed in a centrally truncated view, consistent with the printing of a *`pandas.Series`* (GH5603).

  In previous versions, a DataFrame was truncated once the dimension constraints were reached and an ellipse (...) signaled that part of the data was cut off.

```
In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: pd.options.display.max_rows = 6

In [4]: pd.options.display.max_columns = 6

In [5]: index = pd.DatetimeIndex(start='20010101',freq='D',periods=10)

In [6]: pd.DataFrame(np.arange(10*10).reshape((10,10)),index=index)
Out[6]:
             0    1    2    3    4    5
2001-01-01   0    1    2    3    4    5 ...
2001-01-02  10   11   12   13   14   15 ...
2001-01-03  20   21   22   23   24   25 ...
2001-01-04  30   31   32   33   34   35 ...
2001-01-05  40   41   42   43   44   45 ...
2001-01-06  50   51   52   53   54   55 ...
           ...  ...  ...  ...  ...  ...

[10 rows x 10 columns]
```

In the current version, large DataFrames are centrally truncated, showing a preview of head and tail in both dimensions.

```
In [24]: pd.DataFrame(np.arange(10*10).reshape((10,10)),index=index)
Out[24]:
             0    1    2 ...    7    8    9
2001-01-01   0    1    2 ...    7    8    9
2001-01-02  10   11   12 ...   17   18   19
2001-01-03  20   21   22 ...   27   28   29
...         ..   ..   .. ...   ..   ..   ..
2001-01-08  70   71   72 ...   77   78   79
2001-01-09  80   81   82 ...   87   88   89
2001-01-10  90   91   92 ...   97   98   99

[10 rows x 10 columns]
```

- allow option `'truncate'` for `display.show_dimensions` to only show the dimensions if the frame is truncated (GH6547).

  The default for `display.show_dimensions` will now be `truncate`. This is consistent with how Series display length.

```
In [16]: dfd = pd.DataFrame(np.arange(25).reshape(-1, 5),
   ....:                    index=[0, 1, 2, 3, 4],
   ....:                    columns=[0, 1, 2, 3, 4])
   ....:
```

(continues on next page)

(continued from previous page)

```
# show dimensions since this is truncated
In [17]: with pd.option_context('display.max_rows', 2, 'display.max_columns', 2,
   ....:                        'display.show_dimensions', 'truncate'):
   ....:     print(dfd)
   ....:
     0  ...   4
0    0  ...   4
..  ..  ...  ..
4   20  ...  24

[5 rows x 5 columns]

# will not show dimensions since it is not truncated
In [18]: with pd.option_context('display.max_rows', 10, 'display.max_columns', 40,
   ....:                        'display.show_dimensions', 'truncate'):
   ....:     print(dfd)
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪    0   1   2   3   4
0    0   1   2   3   4
1    5   6   7   8   9
2   10  11  12  13  14
3   15  16  17  18  19
4   20  21  22  23  24
```

- Regression in the display of a MultiIndexed Series with `display.max_rows` is less than the length of the series (GH7101)
- Fixed a bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the *large_repr* set to 'info' (GH7105)
- The *verbose* keyword in `DataFrame.info()`, which controls whether to shorten the `info` representation, is now `None` by default. This will follow the global setting in `display.max_info_columns`. The global setting can be overridden with `verbose=True` or `verbose=False`.
- Fixed a bug with the *info* repr not honoring the *display.max_info_columns* setting (GH6939)
- Offset/freq info now in Timestamp __repr__ (GH4553)

Text Parsing API Changes

`read_csv()`/*`read_table()`* will now be noisier w.r.t invalid options rather than falling back to the `PythonParser`.

- Raise `ValueError` when `sep` specified with `delim_whitespace=True` in *`read_csv()`*/*`read_table()`* (GH6607)
- Raise `ValueError` when `engine='c'` specified with unsupported options in *`read_csv()`*/*`read_table()`* (GH6607)
- Raise `ValueError` when fallback to python parser causes options to be ignored (GH6607)
- Produce `ParserWarning` on fallback to python parser when no options are ignored (GH6607)
- Translate `sep='\s+'` to `delim_whitespace=True` in *`read_csv()`*/*`read_table()`* if no other C-unsupported options specified (GH6607)

Groupby API Changes

More consistent behaviour for some groupby methods:

- groupby `head` and `tail` now act more like `filter` rather than an aggregation:

```
In [19]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [20]: g = df.groupby('A')

In [21]: g.head(1)  # filters DataFrame
Out[21]:
   A  B
0  1  2
2  5  6

[2 rows x 2 columns]

In [22]: g.apply(lambda x: x.head(1))  # used to simply fall-through
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
     A  B
A
1 0  1  2
5 2  5  6

[2 rows x 2 columns]
```

- groupby head and tail respect column selection:

```
In [23]: g[['B']].head(1)
Out[23]:
   B
0  2
2  6

[2 rows x 1 columns]
```

- groupby `nth` now reduces by default; filtering can be achieved by passing `as_index=False`. With an optional `dropna` argument to ignore NaN. See *the docs*.

  Reducing

```
In [24]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [25]: g = df.groupby('A')

In [26]: g.nth(0)
Out[26]:
     B
A
1  NaN
5  6.0

[2 rows x 1 columns]

# this is equivalent to g.first()
In [27]: g.nth(0, dropna='any')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[27]:
```

(continues on next page)

(continued from previous page)

```
     B
A
1  4.0
5  6.0

[2 rows x 1 columns]

# this is equivalent to g.last()
In [28]: g.nth(-1, dropna='any')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     B
A
1  4.0
5  6.0

[2 rows x 1 columns]
```

Filtering

```
In [29]: gf = df.groupby('A', as_index=False)

In [30]: gf.nth(0)
Out[30]:
   A    B
0  1  NaN
2  5  6.0

[2 rows x 2 columns]

In [31]: gf.nth(0, dropna='any')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[31]:
   A    B
A
1  1  4.0
5  5  6.0

[2 rows x 2 columns]
```

- groupby will now not return the grouped column for non-cython functions (GH5610, GH5614, GH6732), as its already the index

```
In [32]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B
↪'])

In [33]: g = df.groupby('A')

In [34]: g.count()
Out[34]:
   B
A
1  1
5  2

[2 rows x 1 columns]
```

(continues on next page)

(continued from previous page)

```
In [35]: g.describe()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[35]:
      B
  count mean       std  min  25%  50%  75%  max
A
1   1.0  4.0       NaN  4.0  4.0  4.0  4.0  4.0
5   2.0  7.0  1.414214  6.0  6.5  7.0  7.5  8.0

[2 rows x 8 columns]
```

- passing `as_index` will leave the grouped column in-place (this is not change in 0.14.0)

```
In [36]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B
↪'])

In [37]: g = df.groupby('A', as_index=False)

In [38]: g.count()
Out[38]:
   A  B
0  1  1
1  5  2

[2 rows x 2 columns]

In [39]: g.describe()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[39]:
      A                                         B                              ␣
↪
  count mean  std  min  25%  50%  75%  max count mean       std  min  25%  50%  75
↪%  max
0   2.0  1.0  0.0  1.0  1.0  1.0  1.0  1.0   1.0  4.0       NaN  4.0  4.0  4.0  4.
↪0  4.0
1   2.0  5.0  0.0  5.0  5.0  5.0  5.0  5.0   2.0  7.0  1.414214  6.0  6.5  7.0  7.
↪5  8.0

[2 rows x 16 columns]
```

- Allow specification of a more complex groupby via `pd.Grouper`, such as grouping by a Time and a string field simultaneously. See *the docs*. (GH3794)
- Better propagation/preservation of Series names when performing groupby operations:
  - `SeriesGroupBy.agg` will ensure that the name attribute of the original series is propagated to the result (GH6265).
  - If the function provided to `GroupBy.apply` returns a named series, the name of the series will be kept as the name of the column index of the DataFrame returned by `GroupBy.apply` (GH6124). This facilitates `DataFrame.stack` operations where the name of the column index is used as the name of the inserted column containing the pivoted data.

SQL

The SQL reading and writing functions now support more database flavors through SQLAlchemy (GH2717, GH4163, GH5950, GH6292). All databases supported by SQLAlchemy can be used, such as PostgreSQL, MySQL, Oracle, Microsoft SQL server (see documentation of SQLAlchemy on included dialects).

The functionality of providing DBAPI connection objects will only be supported for sqlite3 in the future. The 'mysql' flavor is deprecated.

The new functions *read_sql_query()* and *read_sql_table()* are introduced. The function *read_sql()* is kept as a convenience wrapper around the other two and will delegate to specific function depending on the provided input (database table name or sql query).

In practice, you have to provide a SQLAlchemy `engine` to the sql functions. To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For an in-memory sqlite database:

```
In [40]: from sqlalchemy import create_engine

# Create your connection.
In [41]: engine = create_engine('sqlite:///:memory:')
```

This `engine` can then be used to write or read data to/from this database:

```
In [42]: df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})

In [43]: df.to_sql('db_table', engine, index=False)
```

You can read data from a database by specifying the table name:

```
In [44]: pd.read_sql_table('db_table', engine)
Out[44]:
   A  B
0  1  a
1  2  b
2  3  c

[3 rows x 2 columns]
```

or by specifying a sql query:

```
In [45]: pd.read_sql_query('SELECT * FROM db_table', engine)
Out[45]:
   A  B
0  1  a
1  2  b
2  3  c

[3 rows x 2 columns]
```

Some other enhancements to the sql functions include:

- support for writing the index. This can be controlled with the `index` keyword (default is True).
- specify the column label to use when writing the index with `index_label`.
- specify string columns to parse as datetimes with the `parse_dates` keyword in *read_sql_query()* and *read_sql_table()*.

Warning: Some of the existing functions or function aliases have been deprecated and will be removed in future versions. This includes: `tquery`, `uquery`, `read_frame`, `frame_query`, `write_frame`.

Warning: The support for the 'mysql' flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines (GH6900).

MultiIndexing Using Slicers

In 0.14.0 we added a new way to slice MultiIndexed objects. You can slice a MultiIndex by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see *Selection by Label*, including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of *that* level. You do not need to specify all the *deeper* levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

See *the docs* See also issues (GH6134, GH4036, GH3057, GH2598, GH5641, GH7106)

Warning:

You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. Their are some ambiguous cases where the passed indexer could be mis-interpreted as indexing *both* axes, rather than into say the MuliIndex for the rows.

You should do this:

```
  >>> df.loc[(slice('A1', 'A3'), ...), :]  # noqa: E901

rather than this:
```

```
>>> df.loc[(slice('A1', 'A3'), ...)]  # noqa: E901
```

Warning: You will need to make sure that the selection axes are fully lexsorted!

```
In [46]: def mklbl(prefix, n):
   ....:     return ["%s%s" % (prefix, i) for i in range(n)]
   ....:

In [47]: index = pd.MultiIndex.from_product([mklbl('A', 4),
   ....:                                     mklbl('B', 2),
   ....:                                     mklbl('C', 4),
   ....:                                     mklbl('D', 2)])
   ....:

In [48]: columns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'),
   ....:                                       ('b', 'foo'), ('b', 'bah')],
   ....:                                     names=['lvl0', 'lvl1'])
   ....:

In [49]: df = pd.DataFrame(np.arange(len(index) * len(columns)).reshape((len(index),
   ....:                   len(columns))),
   ....:                   index=index,
   ....:                   columns=columns).sort_index().sort_index(axis=1)
```

(continues on next page)

(continued from previous page)

```
   ....:

In [50]: df
Out[50]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0    9    8   11   10
         D1   13   12   15   14
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0   25   24   27   26
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  233  232  235  234
         D1  237  236  239  238
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  249  248  251  250
         D1  253  252  255  254

[64 rows x 4 columns]
```

Basic MultiIndex slicing using slices, lists, and labels.

```
In [51]: df.loc[(slice('A1', 'A3'), slice(None), ['C1', 'C3']), :]
Out[51]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A1 B0 C1 D0   73   72   75   74
         D1   77   76   79   78
      C3 D0   89   88   91   90
         D1   93   92   95   94
   B1 C1 D0  105  104  107  106
         D1  109  108  111  110
      C3 D0  121  120  123  122
...          ...  ...  ...  ...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
      C3 D0  249  248  251  250
         D1  253  252  255  254

[24 rows x 4 columns]
```

You can use a `pd.IndexSlice` to shortcut the creation of these slices

```
In [52]: idx = pd.IndexSlice

In [53]: df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[53]:
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
```

(continues on next page)

(continued from previous page)

```
          D1   12   14
       C3 D0   24   26
          D1   28   30
    B1 C1 D0   40   42
          D1   44   46
       C3 D0   56   58
...           ...  ...
A3 B0 C1 D1  204  206
       C3 D0  216  218
          D1  220  222
    B1 C1 D0  232  234
          D1  236  238
       C3 D0  248  250
          D1  252  254

[32 rows x 2 columns]
```

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```
In [54]: df.loc['A1', (slice(None), 'foo')]
Out[54]:
lvl0        a    b
lvl1      foo  foo
B0 C0 D0   64   66
      D1   68   70
   C1 D0   72   74
      D1   76   78
   C2 D0   80   82
      D1   84   86
   C3 D0   88   90
...       ...  ...
B1 C0 D1  100  102
   C1 D0  104  106
      D1  108  110
   C2 D0  112  114
      D1  116  118
   C3 D0  120  122
      D1  124  126

[16 rows x 2 columns]

In [55]: df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
lvl0           a    b
lvl1         foo  foo
A0 B0 C1 D0    8   10
          D1   12   14
       C3 D0   24   26
          D1   28   30
    B1 C1 D0   40   42
          D1   44   46
       C3 D0   56   58
...           ...  ...
A3 B0 C1 D1  204  206
       C3 D0  216  218
          D1  220  222
```

(continues on next page)

(continued from previous page)

```
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[32 rows x 2 columns]
```

Using a boolean indexer you can provide selection related to the *values*.

```
In [56]: mask = df[('a', 'foo')] > 200

In [57]: df.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
Out[57]:
lvl0           a    b
lvl1         foo  foo
A3 B0 C1 D1  204  206
      C3 D0  216  218
         D1  220  222
   B1 C1 D0  232  234
         D1  236  238
      C3 D0  248  250
         D1  252  254

[7 rows x 2 columns]
```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```
In [58]: df.loc(axis=0)[:, :, ['C1', 'C3']]
Out[58]:
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
         D1   13   12   15   14
      C3 D0   25   24   27   26
         D1   29   28   31   30
   B1 C1 D0   41   40   43   42
         D1   45   44   47   46
      C3 D0   57   56   59   58
...          ...  ...  ...  ...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
         D1  221  220  223  222
   B1 C1 D0  233  232  235  234
         D1  237  236  239  238
      C3 D0  249  248  251  250
         D1  253  252  255  254

[32 rows x 4 columns]
```

Furthermore you can *set* the values using these methods

```
In [59]: df2 = df.copy()

In [60]: df2.loc(axis=0)[:, :, ['C1', 'C3']] = -10

In [61]: df2
Out[61]:
```

(continues on next page)

(continued from previous page)

```
lvl0           a         b
lvl1         bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
         D1    5    4    7    6
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0   17   16   19   18
         D1   21   20   23   22
      C3 D0  -10  -10  -10  -10
...          ...  ...  ...  ...
A3 B1 C0 D1  229  228  231  230
      C1 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10
      C2 D0  241  240  243  242
         D1  245  244  247  246
      C3 D0  -10  -10  -10  -10
         D1  -10  -10  -10  -10

[64 rows x 4 columns]
```

You can use a right-hand-side of an alignable object as well.

```
In [62]: df2 = df.copy()

In [63]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000

In [64]: df2
Out[64]:
lvl0              a               b
lvl1            bar     foo     bah     foo
A0 B0 C0 D0       1       0       3       2
         D1       5       4       7       6
      C1 D0    9000    8000   11000   10000
         D1   13000   12000   15000   14000
      C2 D0      17      16      19      18
         D1      21      20      23      22
      C3 D0   25000   24000   27000   26000
...             ...     ...     ...     ...
A3 B1 C0 D1     229     228     231     230
      C1 D0  233000  232000  235000  234000
         D1  237000  236000  239000  238000
      C2 D0     241     240     243     242
         D1     245     244     247     246
      C3 D0  249000  248000  251000  250000
         D1  253000  252000  255000  254000

[64 rows x 4 columns]
```

Plotting

- Hexagonal bin plots from `DataFrame.plot` with `kind='hexbin'` (GH5478), See *the docs*.
- `DataFrame.plot` and `Series.plot` now supports area plot with specifying `kind='area'` (GH6656), See *the docs*
- Pie plots from `Series.plot` and `DataFrame.plot` with `kind='pie'` (GH6976), See *the docs*.

- Plotting with Error Bars is now supported in the `.plot` method of `DataFrame` and `Series` objects (GH3796, GH6834), See *the docs*.
- `DataFrame.plot` and `Series.plot` now support a `table` keyword for plotting `matplotlib.Table`, See *the docs*. The `table` keyword can receive the following values.
  - `False`: Do nothing (default).
  - `True`: Draw a table using the `DataFrame` or `Series` called `plot` method. Data will be transposed to meet matplotlib's default layout.
  - `DataFrame` or `Series`: Draw matplotlib.table using the passed data. The data will be drawn as displayed in print method (not transposed automatically). Also, helper function `pandas.tools.plotting.table` is added to create a table from `DataFrame` and `Series`, and add it to an `matplotlib.Axes`.
- `plot(legend='reverse')` will now reverse the order of legend labels for most plot kinds. (GH6014)
- Line plot and area plot can be stacked by `stacked=True` (GH6656)
- Following keywords are now acceptable for *`DataFrame.plot()`* with `kind='bar'` and `kind='barh'`:
  - *width*: Specify the bar width. In previous versions, static value 0.5 was passed to matplotlib and it cannot be overwritten. (GH6604)
  - *align*: Specify the bar alignment. Default is *center* (different from matplotlib). In previous versions, pandas passes *align='edge'* to matplotlib and adjust the location to *center* by itself, and it results *align* keyword is not applied as expected. (GH4525)
  - *position*: Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1(right/top-end). Default is 0.5 (center). (GH6604)

  Because of the default *align* value changes, coordinates of bar plots are now located on integer values (0.0, 1.0, 2.0 . . .). This is intended to make bar plot be located on the same coordinates as line plot. However, bar plot may differs unexpectedly when you manually adjust the bar location or drawing area, such as using *set_xlim*, *set_ylim*, etc. In this cases, please modify your script to meet with new coordinates.
- The `parallel_coordinates()` function now takes argument `color` instead of `colors`. A `FutureWarning` is raised to alert that the old `colors` argument will not be supported in a future release. (GH6956)
- The `parallel_coordinates()` and `andrews_curves()` functions now take positional argument `frame` instead of `data`. A `FutureWarning` is raised if the old `data` argument is used by name. (GH6956)
- *`DataFrame.boxplot()`* now supports `layout` keyword (GH6769)
- *`DataFrame.boxplot()`* has a new keyword argument, *return_type*. It accepts `'dict'`, `'axes'`, or `'both'`, in which case a namedtuple with the matplotlib axes and a dict of matplotlib Lines is returned.

Prior Version Deprecations/Changes

There are prior version deprecations that are taking effect as of 0.14.0.

- Remove `DateRange` in favor of *`DatetimeIndex`* (GH6816)
- Remove `column` keyword from `DataFrame.sort` (GH4370)
- Remove `precision` keyword from `set_eng_float_format()` (GH395)
- Remove `force_unicode` keyword from *`DataFrame.to_string()`*, *`DataFrame.to_latex()`*, and *`DataFrame.to_html()`*; these function encode in unicode by default (GH2224, GH2225)
- Remove `nanRep` keyword from *`DataFrame.to_csv()`* and *`DataFrame.to_string()`* (GH275)

- Remove `unique` keyword from `HDFStore.select_column()` (GH3256)
- Remove `inferTimeRule` keyword from `Timestamp.offset()` (GH391)
- Remove `name` keyword from `get_data_yahoo()` and `get_data_google()` (commit b921d1a)
- Remove `offset` keyword from *DatetimeIndex* constructor (commit 3136390)
- Remove `time_rule` from several rolling-moment statistical functions, such as `rolling_sum()` (GH1042)
- Removed neg − boolean operations on numpy arrays in favor of inv ~, as this is going to be deprecated in numpy 1.9 (GH6960)

Deprecations

- The *pivot_table()*/*DataFrame.pivot_table()* and *crosstab()* functions now take arguments `index` and `columns` instead of `rows` and `cols`. A `FutureWarning` is raised to alert that the old `rows` and `cols` arguments will not be supported in a future release (GH5505)
- The *DataFrame.drop_duplicates()* and *DataFrame.duplicated()* methods now take argument `subset` instead of `cols` to better align with *DataFrame.dropna()*. A `FutureWarning` is raised to alert that the old `cols` arguments will not be supported in a future release (GH6680)
- The *DataFrame.to_csv()* and *DataFrame.to_excel()* functions now takes argument `columns` instead of `cols`. A `FutureWarning` is raised to alert that the old `cols` arguments will not be supported in a future release (GH6645)
- Indexers will warn `FutureWarning` when used with a scalar indexer and a non-floating point Index (GH4892, GH6960)

```
# non-floating point indexes can only be indexed by integers / labels
In [1]: pd.Series(1, np.arange(5))[3.0]
        pandas/core/index.py:469: FutureWarning: scalar indexers for index type␣
↪Int64Index should be integers and not floating point
Out[1]: 1

In [2]: pd.Series(1, np.arange(5)).iloc[3.0]
        pandas/core/index.py:469: FutureWarning: scalar indexers for index type␣
↪Int64Index should be integers and not floating point
Out[2]: 1

In [3]: pd.Series(1, np.arange(5)).iloc[3.0:4]
        pandas/core/index.py:527: FutureWarning: slice indexers when using iloc␣
↪should be integers and not floating point
Out[3]:
        3    1
        dtype: int64

# these are Float64Indexes, so integer or floating point is acceptable
In [4]: pd.Series(1, np.arange(5.))[3]
Out[4]: 1

In [5]: pd.Series(1, np.arange(5.))[3.0]
Out[6]: 1
```

- Numpy 1.9 compat w.r.t. deprecation warnings (GH6960)
- *Panel.shift()* now has a function signature that matches *DataFrame.shift()*. The old positional argument `lags` has been changed to a keyword argument `periods` with a default value of 1. A `FutureWarning` is raised if the old argument `lags` is used by name. (GH6910)

- The `order` keyword argument of *factorize()* will be removed. (GH6926).
- Remove the `copy` keyword from *DataFrame.xs()*, *Panel.major_xs()*, *Panel.minor_xs()*. A view will be returned if possible, otherwise a copy will be made. Previously the user could think that `copy=False` would ALWAYS return a view. (GH6894)
- The `parallel_coordinates()` function now takes argument `color` instead of `colors`. A `FutureWarning` is raised to alert that the old `colors` argument will not be supported in a future release. (GH6956)
- The `parallel_coordinates()` and `andrews_curves()` functions now take positional argument `frame` instead of `data`. A `FutureWarning` is raised if the old `data` argument is used by name. (GH6956)
- The support for the 'mysql' flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines (GH6900).
- The following `io.sql` functions have been deprecated: `tquery`, `uquery`, `read_frame`, `frame_query`, `write_frame`.
- The *percentile_width* keyword argument in *describe()* has been deprecated. Use the *percentiles* keyword instead, which takes a list of percentiles to display. The default output is unchanged.
- The default return type of `boxplot()` will change from a dict to a matplotlib Axes in a future release. You can use the future behavior now by passing `return_type='axes'` to boxplot.

Known Issues

- OpenPyXL 2.0.0 breaks backwards compatibility (GH7169)

Enhancements

- DataFrame and Series will create a MultiIndex object if passed a tuples dict, See *the docs* (GH3323)

```
In [65]: pd.Series({('a', 'b'): 1, ('a', 'a'): 0,
   ....:            ('a', 'c'): 2, ('b', 'a'): 3, ('b', 'b'): 4})
   ....:
Out[65]:
a  b    1
   a    0
   c    2
b  a    3
   b    4
Length: 5, dtype: int64

In [66]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:               ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:               ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:               ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:               ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[66]:
↪
       a              b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

(continues on next page)

(continued from previous page)

```
[3 rows x 5 columns]
```

- Added the `sym_diff` method to `Index` (GH5543)
- `DataFrame.to_latex` now takes a longtable keyword, which if True will return a table in a longtable environment. (GH6617)
- Add option to turn off escaping in `DataFrame.to_latex` (GH6472)
- `pd.read_clipboard` will, if the keyword `sep` is unspecified, try to detect data copied from a spreadsheet and parse accordingly. (GH6223)
- Joining a singly-indexed DataFrame with a MultiIndexed DataFrame (GH3662)

  See *the docs*. Joining MultiIndex DataFrames on both the left and right is not yet supported ATM.

```
In [67]: household = pd.DataFrame({'household_id': [1, 2, 3],
   ....:                           'male': [0, 1, 0],
   ....:                           'wealth': [196087.3, 316478.7, 294750]
   ....:                           },
   ....:                          columns=['household_id', 'male', 'wealth']
   ....:                          ).set_index('household_id')
   ....:

In [68]: household
Out[68]:
              male    wealth
household_id
1                0  196087.3
2                1  316478.7
3                0  294750.0

[3 rows x 2 columns]

In [69]: portfolio = pd.DataFrame({'household_id': [1, 2, 2, 3, 3, 3, 4],
   ....:                           'asset_id': ["nl0000301109",
   ....:                                        "nl0000289783",
   ....:                                        "gb00b03mlx29",
   ....:                                        "gb00b03mlx29",
   ....:                                        "lu0197800237",
   ....:                                        "nl0000289965",
   ....:                                        np.nan],
   ....:                           'name': ["ABN Amro",
   ....:                                    "Robeco",
   ....:                                    "Royal Dutch Shell",
   ....:                                    "Royal Dutch Shell",
   ....:                                    "AAB Eastern Europe Equity Fund",
   ....:                                    "Postbank BioTech Fonds",
   ....:                                    np.nan],
   ....:                           'share': [1.0, 0.4, 0.6, 0.15, 0.6, 0.25, 1.0]
   ....:                           },
   ....:                          columns=['household_id', 'asset_id', 'name',
↪'share']
   ....:                          ).set_index(['household_id', 'asset_id'])
   ....:

In [70]: portfolio
```

(continues on next page)

(continued from previous page)

```
Out[70]:
                                                  name  share
household_id asset_id
1            nl0000301109                     ABN Amro   1.00
2            nl0000289783                       Robeco   0.40
             gb00b03mlx29            Royal Dutch Shell   0.60
3            gb00b03mlx29            Royal Dutch Shell   0.15
             lu0197800237  AAB Eastern Europe Equity Fund   0.60
             nl0000289965         Postbank BioTech Fonds   0.25
4            NaN                                 NaN   1.00

[7 rows x 2 columns]

In [71]: household.join(portfolio, how='inner')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                           male    wealth                            name  share
household_id asset_id
1            nl0000301109     0  196087.3                        ABN Amro   1.00
2            nl0000289783     1  316478.7                          Robeco   0.40
             gb00b03mlx29     1  316478.7               Royal Dutch Shell   0.60
3            gb00b03mlx29     0  294750.0               Royal Dutch Shell   0.15
             lu0197800237     0  294750.0  AAB Eastern Europe Equity Fund   0.60
             nl0000289965     0  294750.0          Postbank BioTech Fonds   0.25

[6 rows x 4 columns]
```

- quotechar, doublequote, and escapechar can now be specified when using DataFrame.to_csv (GH5414, GH4528)
- Partially sort by only the specified levels of a MultiIndex with the sort_remaining boolean kwarg. (GH3984)
- Added to_julian_date to TimeStamp and DatetimeIndex. The Julian Date is used primarily in astronomy and represents the number of days from noon, January 1, 4713 BC. Because nanoseconds are used to define the time in pandas the actual range of dates that you can use is 1678 AD to 2262 AD. (GH4041)
- DataFrame.to_stata will now check data for compatibility with Stata data types and will upcast when needed. When it is not possible to losslessly upcast, a warning is issued (GH6327)
- DataFrame.to_stata and StataWriter will accept keyword arguments time_stamp and data_label which allow the time stamp and dataset label to be set when creating a file. (GH6545)
- pandas.io.gbq now handles reading unicode strings properly. (GH5940)
- *Holidays Calendars* are now available and can be used with the CustomBusinessDay offset (GH6719)
- Float64Index is now backed by a float64 dtype ndarray instead of an object dtype array (GH6471).
- Implemented Panel.pct_change (GH6904)
- Added how option to rolling-moment functions to dictate how to handle resampling; rolling_max() defaults to max, rolling_min() defaults to min, and all others default to mean (GH6297)
- CustomBuisnessMonthBegin and CustomBusinessMonthEnd are now available (GH6866)
- *Series.quantile()* and *DataFrame.quantile()* now accept an array of quantiles.
- *describe()* now accepts an array of percentiles to include in the summary statistics (GH4196)
- pivot_table can now accept Grouper by index and columns keywords (GH6913)

```
In [72]: import datetime

In [73]: df = pd.DataFrame({
   ....:     'Branch': 'A A A A A B'.split(),
   ....:     'Buyer': 'Carl Mark Carl Carl Joe Joe'.split(),
   ....:     'Quantity': [1, 3, 5, 1, 8, 1],
   ....:     'Date': [datetime.datetime(2013, 11, 1, 13, 0),
   ....:              datetime.datetime(2013, 9, 1, 13, 5),
   ....:              datetime.datetime(2013, 10, 1, 20, 0),
   ....:              datetime.datetime(2013, 10, 2, 10, 0),
   ....:              datetime.datetime(2013, 11, 1, 20, 0),
   ....:              datetime.datetime(2013, 10, 2, 10, 0)],
   ....:     'PayDay': [datetime.datetime(2013, 10, 4, 0, 0),
   ....:                datetime.datetime(2013, 10, 15, 13, 5),
   ....:                datetime.datetime(2013, 9, 5, 20, 0),
   ....:                datetime.datetime(2013, 11, 2, 10, 0),
   ....:                datetime.datetime(2013, 10, 7, 20, 0),
   ....:                datetime.datetime(2013, 9, 5, 10, 0)]})
   ....:

In [74]: df
Out[74]:
  Branch Buyer  Quantity                Date              PayDay
0      A  Carl         1 2013-11-01 13:00:00 2013-10-04 00:00:00
1      A  Mark         3 2013-09-01 13:05:00 2013-10-15 13:05:00
2      A  Carl         5 2013-10-01 20:00:00 2013-09-05 20:00:00
3      A  Carl         1 2013-10-02 10:00:00 2013-11-02 10:00:00
4      A   Joe         8 2013-11-01 20:00:00 2013-10-07 20:00:00
5      B   Joe         1 2013-10-02 10:00:00 2013-09-05 10:00:00

[6 rows x 5 columns]

In [75]: df.pivot_table(values='Quantity',
   ....:                index=pd.Grouper(freq='M', key='Date'),
   ....:                columns=pd.Grouper(freq='M', key='PayDay'),
   ....:                aggfunc=np.sum)
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
PayDay      2013-09-30  2013-10-31  2013-11-30
Date
2013-09-30         NaN         3.0         NaN
2013-10-31         6.0         NaN         1.0
2013-11-30         NaN         9.0         NaN

[3 rows x 3 columns]
```

- Arrays of strings can be wrapped to a specified width (`str.wrap`) (GH6999)
- Add *nsmallest()* and *Series.nlargest()* methods to Series, See *the docs* (GH3960)
- *PeriodIndex* fully supports partial string indexing like *DatetimeIndex* (GH7043)

```
In [76]: prng = pd.period_range('2013-01-01 09:00', periods=100, freq='H')

In [77]: ps = pd.Series(np.random.randn(len(prng)), index=prng)

In [78]: ps
```

(continues on next page)

(continued from previous page)

```
Out[78]:
2013-01-01 09:00    0.015696
2013-01-01 10:00   -2.242685
2013-01-01 11:00    1.150036
2013-01-01 12:00    0.991946
2013-01-01 13:00    0.953324
2013-01-01 14:00   -2.021255
2013-01-01 15:00   -0.334077
                      ...
2013-01-05 06:00    0.566534
2013-01-05 07:00    0.503592
2013-01-05 08:00    0.285296
2013-01-05 09:00    0.484288
2013-01-05 10:00    1.363482
2013-01-05 11:00   -0.781105
2013-01-05 12:00   -0.468018
Freq: H, Length: 100, dtype: float64

In [79]: ps['2013-01-02']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2013-01-02 00:00    0.553439
2013-01-02 01:00    1.318152
2013-01-02 02:00   -0.469305
2013-01-02 03:00    0.675554
2013-01-02 04:00   -1.817027
2013-01-02 05:00   -0.183109
2013-01-02 06:00    1.058969
                      ...
2013-01-02 17:00    0.076200
2013-01-02 18:00   -0.566446
2013-01-02 19:00    0.036142
2013-01-02 20:00   -2.074978
2013-01-02 21:00    0.247792
2013-01-02 22:00   -0.897157
2013-01-02 23:00   -0.136795
Freq: H, Length: 24, dtype: float64
```

- `read_excel` can now read milliseconds in Excel dates and times with xlrd >= 0.9.3. (GH5945)
- `pd.stats.moments.rolling_var` now uses Welford's method for increased numerical stability (GH6817)
- pd.expanding_apply and pd.rolling_apply now take args and kwargs that are passed on to the func (GH6289)
- `DataFrame.rank()` now has a percentage rank option (GH5971)
- `Series.rank()` now has a percentage rank option (GH5971)
- `Series.rank()` and `DataFrame.rank()` now accept `method='dense'` for ranks without gaps (GH6514)
- Support passing `encoding` with xlwt (GH3710)
- Refactor Block classes removing *Block.items* attributes to avoid duplication in item handling (GH6745, GH6988).
- Testing statements updated to use specialized asserts (GH6175)

Performance

- Performance improvement when converting `DatetimeIndex` to floating ordinals using `DatetimeConverter` (GH6636)
- Performance improvement for `DataFrame.shift` (GH5609)
- Performance improvement in indexing into a MultiIndexed Series (GH5567)
- Performance improvements in single-dtyped indexing (GH6484)
- Improve performance of DataFrame construction with certain offsets, by removing faulty caching (e.g. MonthEnd,BusinessMonthEnd), (GH6479)
- Improve performance of `CustomBusinessDay` (GH6584)
- improve performance of slice indexing on Series with string keys (GH6341, GH6372)
- Performance improvement for `DataFrame.from_records` when reading a specified number of rows from an iterable (GH6700)
- Performance improvements in timedelta conversions for integer dtypes (GH6754)
- Improved performance of compatible pickles (GH6899)
- Improve performance in certain reindexing operations by optimizing `take_2d` (GH6749)
- `GroupBy.count()` is now implemented in Cython and is much faster for large numbers of groups (GH7016).

Experimental

There are no experimental changes in 0.14.0

Bug Fixes

- Bug in Series ValueError when index doesn't match data (GH6532)
- Prevent segfault due to MultiIndex not being supported in HDFStore table format (GH1848)
- Bug in `pd.DataFrame.sort_index` where mergesort wasn't stable when `ascending=False` (GH6399)
- Bug in `pd.tseries.frequencies.to_offset` when argument has leading zeros (GH6391)
- Bug in version string gen. for dev versions with shallow clones / install from tarball (GH6127)
- Inconsistent tz parsing `Timestamp` / `to_datetime` for current year (GH5958)
- Indexing bugs with reordered indexes (GH6252, GH6254)
- Bug in `.xs` with a Series multiindex (GH6258, GH5684)
- Bug in conversion of a string types to a DatetimeIndex with a specified frequency (GH6273, GH6274)
- Bug in `eval` where type-promotion failed for large expressions (GH6205)
- Bug in interpolate with `inplace=True` (GH6281)
- `HDFStore.remove` now handles start and stop (GH6177)
- `HDFStore.select_as_multiple` handles start and stop the same way as `select` (GH6177)
- `HDFStore.select_as_coordinates` and `select_column` works with a `where` clause that results in filters (GH6177)

- Regression in join of non_unique_indexes (GH6329)
- Issue with groupby `agg` with a single function and a a mixed-type frame (GH6337)
- Bug in `DataFrame.replace()` when passing a non- `bool` `to_replace` argument (GH6332)
- Raise when trying to align on different levels of a MultiIndex assignment (GH3738)
- Bug in setting complex dtypes via boolean indexing (GH6345)
- Bug in TimeGrouper/resample when presented with a non-monotonic DatetimeIndex that would return invalid results. (GH4161)
- Bug in index name propagation in TimeGrouper/resample (GH4161)
- TimeGrouper has a more compatible API to the rest of the groupers (e.g. `groups` was missing) (GH3881)
- Bug in multiple grouping with a TimeGrouper depending on target column order (GH6764)
- Bug in `pd.eval` when parsing strings with possible tokens like `'&'` (GH6351)
- Bug correctly handle placements of `-inf` in Panels when dividing by integer 0 (GH6178)
- `DataFrame.shift` with `axis=1` was raising (GH6371)
- Disabled clipboard tests until release time (run locally with `nosetests -A disabled`) (GH6048).
- Bug in `DataFrame.replace()` when passing a nested `dict` that contained keys not in the values to be replaced (GH6342)
- `str.match` ignored the na flag (GH6609).
- Bug in take with duplicate columns that were not consolidated (GH6240)
- Bug in interpolate changing dtypes (GH6290)
- Bug in `Series.get`, was using a buggy access method (GH6383)
- Bug in hdfstore queries of the form `where=[('date', '>=', datetime(2013,1,1)), ('date', '<=', datetime(2014,1,1))]` (GH6313)
- Bug in `DataFrame.dropna` with duplicate indices (GH6355)
- Regression in chained getitem indexing with embedded list-like from 0.12 (GH6394)
- `Float64Index` with nans not comparing correctly (GH6401)
- `eval`/`query` expressions with strings containing the `@` character will now work (GH6366).
- Bug in `Series.reindex` when specifying a `method` with some nan values was inconsistent (noted on a resample) (GH6418)
- Bug in *`DataFrame.replace()`* where nested dicts were erroneously depending on the order of dictionary keys and values (GH5338).
- Performance issue in concatenating with empty objects (GH3259)
- Clarify sorting of `sym_diff` on `Index` objects with `NaN` values (GH6444)
- Regression in `MultiIndex.from_product` with a `DatetimeIndex` as input (GH6439)
- Bug in `str.extract` when passed a non-default index (GH6348)
- Bug in `str.split` when passed `pat=None` and `n=1` (GH6466)
- Bug in `io.data.DataReader` when passed `"F-F_Momentum_Factor"` and `data_source="famafrench"` (GH6460)
- Bug in `sum` of a `timedelta64[ns]` series (GH6462)

- Bug in `resample` with a timezone and certain offsets (GH6397)
- Bug in `iat/iloc` with duplicate indices on a Series (GH6493)
- Bug in `read_html` where nan's were incorrectly being used to indicate missing values in text. Should use the empty string for consistency with the rest of pandas (GH5129).
- Bug in `read_html` tests where redirected invalid URLs would make one test fail (GH6445).
- Bug in multi-axis indexing using `.loc` on non-unique indices (GH6504)
- Bug that caused _ref_locs corruption when slice indexing across columns axis of a DataFrame (GH6525)
- Regression from 0.13 in the treatment of numpy `datetime64` non-ns dtypes in Series creation (GH6529)
- `.names` attribute of MultiIndexes passed to `set_index` are now preserved (GH6459).
- Bug in setitem with a duplicate index and an alignable rhs (GH6541)
- Bug in setitem with `.loc` on mixed integer Indexes (GH6546)
- Bug in `pd.read_stata` which would use the wrong data types and missing values (GH6327)
- Bug in `DataFrame.to_stata` that lead to data loss in certain cases, and could be exported using the wrong data types and missing values (GH6335)
- `StataWriter` replaces missing values in string columns by empty string (GH6802)
- Inconsistent types in `Timestamp` addition/subtraction (GH6543)
- Bug in preserving frequency across Timestamp addition/subtraction (GH4547)
- Bug in empty list lookup caused `IndexError` exceptions (GH6536, GH6551)
- `Series.quantile` raising on an `object` dtype (GH6555)
- Bug in `.xs` with a `nan` in level when dropped (GH6574)
- Bug in fillna with `method='bfill/ffill'` and `datetime64[ns]` dtype (GH6587)
- Bug in sql writing with mixed dtypes possibly leading to data loss (GH6509)
- Bug in `Series.pop` (GH6600)
- Bug in `iloc` indexing when positional indexer matched `Int64Index` of the corresponding axis and no reordering happened (GH6612)
- Bug in `fillna` with `limit` and `value` specified
- Bug in `DataFrame.to_stata` when columns have non-string names (GH4558)
- Bug in compat with `np.compress`, surfaced in (GH6658)
- Bug in binary operations with a rhs of a Series not aligning (GH6681)
- Bug in `DataFrame.to_stata` which incorrectly handles nan values and ignores `with_index` keyword argument (GH6685)
- Bug in resample with extra bins when using an evenly divisible frequency (GH4076)
- Bug in consistency of groupby aggregation when passing a custom function (GH6715)
- Bug in resample when `how=None` resample freq is the same as the axis frequency (GH5955)
- Bug in downcasting inference with empty arrays (GH6733)
- Bug in `obj.blocks` on sparse containers dropping all but the last items of same for dtype (GH6748)
- Bug in unpickling `NaT (NaTType)` (GH4606)

- Bug in `DataFrame.replace()` where regex meta characters were being treated as regex even when `regex=False` (GH6777).
- Bug in timedelta ops on 32-bit platforms (GH6808)
- Bug in setting a tz-aware index directly via `.index` (GH6785)
- Bug in expressions.py where numexpr would try to evaluate arithmetic ops (GH6762).
- Bug in Makefile where it didn't remove Cython generated C files with `make clean` (GH6768)
- Bug with numpy < 1.7.2 when reading long strings from `HDFStore` (GH6166)
- Bug in `DataFrame._reduce` where non bool-like (0/1) integers were being converted into bools. (GH6806)
- Regression from 0.13 with `fillna` and a Series on datetime-like (GH6344)
- Bug in adding `np.timedelta64` to `DatetimeIndex` with timezone outputs incorrect results (GH6818)
- Bug in `DataFrame.replace()` where changing a dtype through replacement would only replace the first occurrence of a value (GH6689)
- Better error message when passing a frequency of 'MS' in `Period` construction (GH5332)
- Bug in `Series.__unicode__` when `max_rows=None` and the Series has more than 1000 rows. (GH6863)
- Bug in `groupby.get_group` where a datelike wasn't always accepted (GH5267)
- Bug in `groupBy.get_group` created by `TimeGrouper` raises `AttributeError` (GH6914)
- Bug in `DatetimeIndex.tz_localize` and `DatetimeIndex.tz_convert` converting `NaT` incorrectly (GH5546)
- Bug in arithmetic operations affecting `NaT` (GH6873)
- Bug in `Series.str.extract` where the resulting `Series` from a single group match wasn't renamed to the group name
- Bug in `DataFrame.to_csv` where setting `index=False` ignored the `header` kwarg (GH6186)
- Bug in `DataFrame.plot` and `Series.plot`, where the legend behave inconsistently when plotting to the same axes repeatedly (GH6678)
- Internal tests for patching `__finalize__` / bug in merge not finalizing (GH6923, GH6927)
- accept `TextFileReader` in `concat`, which was affecting a common user idiom (GH6583)
- Bug in C parser with leading white space (GH3374)
- Bug in C parser with `delim_whitespace=True` and `\r`-delimited lines
- Bug in python parser with explicit MultiIndex in row following column header (GH6893)
- Bug in `Series.rank` and `DataFrame.rank` that caused small floats (<1e-13) to all receive the same rank (GH6886)
- Bug in `DataFrame.apply` with functions that used `*args` or `**kwargs` and returned an empty result (GH6952)
- Bug in sum/mean on 32-bit platforms on overflows (GH6915)
- Moved `Panel.shift` to `NDFrame.slice_shift` and fixed to respect multiple dtypes. (GH6959)
- Bug in enabling `subplots=True` in `DataFrame.plot` only has single column raises `TypeError`, and `Series.plot` raises `AttributeError` (GH6951)
- Bug in `DataFrame.plot` draws unnecessary axes when enabling `subplots` and `kind=scatter` (GH6951)

- Bug in `read_csv` from a filesystem with non-utf-8 encoding (GH6807)
- Bug in `iloc` when setting / aligning (GH6766)
- Bug causing UnicodeEncodeError when get_dummies called with unicode values and a prefix (GH6885)
- Bug in timeseries-with-frequency plot cursor display (GH5453)
- Bug surfaced in `groupby.plot` when using a `Float64Index` (GH7025)
- Stopped tests from failing if options data isn't able to be downloaded from Yahoo (GH7034)
- Bug in `parallel_coordinates` and `radviz` where reordering of class column caused possible color/class mismatch (GH6956)
- Bug in `radviz` and `andrews_curves` where multiple values of 'color' were being passed to plotting method (GH6956)
- Bug in `Float64Index.isin()` where containing `nan` s would make indices claim that they contained all the things (GH7066).
- Bug in `DataFrame.boxplot` where it failed to use the axis passed as the `ax` argument (GH3578)
- Bug in the `XlsxWriter` and `XlwtWriter` implementations that resulted in datetime columns being formatted without the time (GH7075) were being passed to plotting method
- *`read_fwf()`* treats `None` in `colspec` like regular python slices. It now reads from the beginning or until the end of the line when `colspec` contains a `None` (previously raised a `TypeError`)
- Bug in cache coherence with chained indexing and slicing; add `_is_view` property to `NDFrame` to correctly predict views; mark `is_copy` on `xs` only if its an actual copy (and not a view) (GH7084)
- Bug in DatetimeIndex creation from string ndarray with `dayfirst=True` (GH5917)
- Bug in `MultiIndex.from_arrays` created from `DatetimeIndex` doesn't preserve `freq` and `tz` (GH7090)
- Bug in `unstack` raises `ValueError` when `MultiIndex` contains `PeriodIndex` (GH4342)
- Bug in `boxplot` and `hist` draws unnecessary axes (GH6769)
- Regression in `groupby.nth()` for out-of-bounds indexers (GH6621)
- Bug in `quantile` with datetime values (GH6965)
- Bug in `Dataframe.set_index`, `reindex` and `pivot` don't preserve `DatetimeIndex` and `PeriodIndex` attributes (GH3950, GH5878, GH6631)
- Bug in `MultiIndex.get_level_values` doesn't preserve `DatetimeIndex` and `PeriodIndex` attributes (GH7092)
- Bug in `Groupby` doesn't preserve `tz` (GH3950)
- Bug in `PeriodIndex` partial string slicing (GH6716)
- Bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the *large_repr* set to 'info' (GH7105)
- Bug in `DatetimeIndex` specifying `freq` raises `ValueError` when passed value is too short (GH7098)
- Fixed a bug with the *info* repr not honoring the *display.max_info_columns* setting (GH6939)
- Bug `PeriodIndex` string slicing with out of bounds values (GH5407)
- Fixed a memory error in the hashtable implementation/factorizer on resizing of large tables (GH7157)
- Bug in `isnull` when applied to 0-dimensional object arrays (GH7176)

- Bug in `query`/`eval` where global constants were not looked up correctly (GH7178)
- Bug in recognizing out-of-bounds positional list indexers with `iloc` and a multi-axis tuple indexer (GH7189)
- Bug in setitem with a single value, MultiIndex and integer indices (GH7190, GH7218)
- Bug in expressions evaluation with reversed ops, showing in series-dataframe ops (GH7198, GH7192)
- Bug in multi-axis indexing with > 2 ndim and a MultiIndex (GH7199)
- Fix a bug where invalid eval/query operations would blow the stack (GH5198)

Contributors

A total of 94 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Acanthostega +
- Adam Marcus +
- Alex Gaudio
- Alex Rothberg
- AllenDowney +
- Andrew Rosenfeld +
- Andy Hayden
- Antoine Mazières +
- Benedikt Sauer
- Brad Buran
- Christopher Whelan
- Clark Fitzgerald
- DSM
- Dale Jung
- Dan Allan
- Dan Birken
- Daniel Waeber
- David Jung +
- David Stephens +
- Douglas McNeil
- Garrett Drapala
- Gouthaman Balaraman +
- Guillaume Poulin +
- Jacob Howard +
- Jacob Schaer
- Jason Sexauer +

- Jeff Reback
- Jeff Tratner
- Jeffrey Starr +
- John David Reaver +
- John McNamara
- John W. O'Brien
- Jonathan Chambers
- Joris Van den Bossche
- Julia Evans
- Júlio +
- K.-Michael Aye
- Katie Atkinson +
- Kelsey Jordahl
- Kevin Sheppard +
- Matt Wittmann +
- Matthias Kuhn +
- Max Grender-Jones +
- Michael E. Gruen +
- Mike Kelly
- Nipun Batra +
- Noah Spies +
- PKEuS
- Patrick O'Keeffe
- Phillip Cloud
- Pietro Battiston +
- Randy Carnevale +
- Robert Gibboni +
- Skipper Seabold
- SplashDance +
- Stephan Hoyer +
- Tim Cera +
- Tobias Brandt
- Todd Jennings +
- Tom Augspurger
- TomAugspurger
- Yaroslav Halchenko

- agijsberts +
- akittredge
- ankostis +
- anomrake
- anton-d +
- bashtage +
- benjamin +
- bwignall
- cgohlke +
- chebee7i +
- clham +
- danielballan
- hshimizu77 +
- hugo +
- immerrr
- ischwabacher +
- jaimefrio +
- jreback
- jsexauer +
- kdiether +
- michaelws +
- mikebailey +
- ojdo +
- onesandzeroes +
- phaebz +
- ribonoous +
- rockg
- sinhrks +
- unutbu
- westurner
- y-p
- zach powers

8.12 Version 0.13

8.12.1 v0.13.1 (February 3, 2014)

This is a minor release from 0.13.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Added `infer_datetime_format` keyword to `read_csv/to_datetime` to allow speedups for homogeneously formatted datetimes.
- Will intelligently limit display precision for datetime/timedelta formats.
- Enhanced Panel `apply()` method.
- Suggested tutorials in new *Tutorials* section.
- Our pandas ecosystem is growing, We now feature related projects in a new *Pandas Ecosystem* section.
- Much work has been taking place on improving the docs, and a new *Contributing* section has been added.
- Even though it may only be of interest to devs, we <3 our new CI status page: ScatterCI.

Warning: 0.13.1 fixes a bug that was caused by a combination of having numpy < 1.8, and doing chained assignment on a string-like array. Please review *the docs*, chained indexing can have unexpected results and should generally be avoided.

This would previously segfault:

```
In [1]: df = pd.DataFrame({'A': np.array(['foo', 'bar', 'bah', 'foo', 'bar'])})

In [2]: df['A'].iloc[0] = np.nan

In [3]: df
Out[3]:
     A
0  NaN
1  bar
2  bah
3  foo
4  bar
```

The recommended way to do this type of assignment is:

```
In [4]: df = pd.DataFrame({'A': np.array(['foo', 'bar', 'bah', 'foo', 'bar'])})

In [5]: df.loc[0, 'A'] = np.nan

In [6]: df
Out[6]:
     A
0  NaN
1  bar
2  bah
3  foo
4  bar
```

Output Formatting Enhancements

- df.info() view now display dtype info per column (GH5682)
- df.info() now honors the option `max_info_rows`, to disable null counts for large frames (GH5974)

```
In [7]: max_info_rows = pd.get_option('max_info_rows')

In [8]: df = pd.DataFrame({'A': np.random.randn(10),
   ...:                    'B': np.random.randn(10),
   ...:                    'C': pd.date_range('20130101', periods=10)
   ...:                    })
   ...:

In [9]: df.iloc[3:6, [0, 2]] = np.nan
```

```
# set to not display the null counts
In [10]: pd.set_option('max_info_rows', 0)

In [11]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
A    float64
B    float64
C    datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 bytes
```

```
# this is the default (same as in 0.13.0)
In [12]: pd.set_option('max_info_rows', max_info_rows)

In [13]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
A    7 non-null float64
B    10 non-null float64
C    7 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 bytes
```

- Add `show_dimensions` display option for the new DataFrame repr to control whether the dimensions print.

```
In [14]: df = pd.DataFrame([[1, 2], [3, 4]])

In [15]: pd.set_option('show_dimensions', False)

In [16]: df
Out[16]:
   0  1
0  1  2
1  3  4

In [17]: pd.set_option('show_dimensions', True)

In [18]: df
```

(continues on next page)

(continued from previous page)

```
Out[18]:
   0  1
0  1  2
1  3  4

[2 rows x 2 columns]
```

- The ArrayFormatter for datetime and timedelta64 now intelligently limit precision based on the values in the array (GH3401)

  Previously output might look like:

```
  age                 today               diff
0 2001-01-01 00:00:00 2013-04-19 00:00:00 4491 days, 00:00:00
1 2004-06-01 00:00:00 2013-04-19 00:00:00 3244 days, 00:00:00
```

  Now the output looks like:

```
In [19]: df = pd.DataFrame([pd.Timestamp('20010101'),
   ....:                    pd.Timestamp('20040601')], columns=['age'])
   ....:

In [20]: df['today'] = pd.Timestamp('20130419')

In [21]: df['diff'] = df['today'] - df['age']

In [22]: df
Out[22]:
         age      today      diff
0 2001-01-01 2013-04-19 4491 days
1 2004-06-01 2013-04-19 3244 days

[2 rows x 3 columns]
```

API changes

- Add -NaN and -nan to the default set of NA values (GH5952). See *NA Values*.
- Added Series.str.get_dummies vectorized string method (GH6021), to extract dummy/indicator variables for separated string columns:

```
In [23]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])

In [24]: s.str.get_dummies(sep='|')
Out[24]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1

[4 rows x 3 columns]
```

- Added the NDFrame.equals() method to compare if two NDFrames are equal have equal axes, dtypes, and values. Added the array_equivalent function to compare if two ndarrays are equal. NaNs in identical locations are treated as equal. (GH5283) See also *the docs* for a motivating example.

```
df = pd.DataFrame({'col': ['foo', 0, np.nan]})
df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2, 1, 0])
df.equals(df2)
df.equals(df2.sort_index())
```

- `DataFrame.apply` will use the `reduce` argument to determine whether a `Series` or a `DataFrame` should be returned when the `DataFrame` is empty (GH6007).

  Previously, calling `DataFrame.apply` an empty `DataFrame` would return either a `DataFrame` if there were no columns, or the function being applied would be called with an empty `Series` to guess whether a `Series` or `DataFrame` should be returned:

```
In [32]: def applied_func(col):
   ....:     print("Apply function being called with: ", col)
   ....:     return col.sum()
   ....:

In [33]: empty = DataFrame(columns=['a', 'b'])

In [34]: empty.apply(applied_func)
Apply function being called with:  Series([], Length: 0, dtype: float64)
Out[34]:
a   NaN
b   NaN
Length: 2, dtype: float64
```

  Now, when `apply` is called on an empty `DataFrame`: if the `reduce` argument is `True` a `Series` will returned, if it is `False` a `DataFrame` will be returned, and if it is `None` (the default) the function being applied will be called with an empty series to try and guess the return type.

```
In [35]: empty.apply(applied_func, reduce=True)
Out[35]:
a   NaN
b   NaN
Length: 2, dtype: float64

In [36]: empty.apply(applied_func, reduce=False)
Out[36]:
Empty DataFrame
Columns: [a, b]
Index: []

[0 rows x 2 columns]
```

Prior Version Deprecations/Changes

There are no announced changes in 0.13 or prior that are taking effect as of 0.13.1

Deprecations

There are no deprecations of prior behavior in 0.13.1

Enhancements

- pd.read_csv and pd.to_datetime learned a new infer_datetime_format keyword which greatly improves parsing perf in many cases. Thanks to @lexual for suggesting and @danbirken for rapidly implementing. (GH5490, GH6021)

  If parse_dates is enabled and this flag is set, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

```
# Try to infer the format for the index column
df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
                 infer_datetime_format=True)
```

- date_format and datetime_format keywords can now be specified when writing to excel files (GH4133)
- MultiIndex.from_product convenience function for creating a MultiIndex from the cartesian product of a set of iterables (GH6055):

```
In [25]: shades = ['light', 'dark']

In [26]: colors = ['red', 'green', 'blue']

In [27]: pd.MultiIndex.from_product([shades, colors], names=['shade', 'color'])
Out[27]:
MultiIndex(levels=[['dark', 'light'], ['blue', 'green', 'red']],
           codes=[[1, 1, 1, 0, 0, 0], [2, 1, 0, 2, 1, 0]],
           names=['shade', 'color'])
```

- Panel *apply()* will work on non-ufuncs. See *the docs*.

```
In [28]: import pandas.util.testing as tm

In [29]: panel = tm.makePanel(5)

In [30]: panel
Out[30]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [31]: panel['ItemA']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-03 -0.673690  0.577046 -1.344312 -1.469388
2000-01-04  0.113648 -1.715002  0.844885  0.357021
2000-01-05 -1.478427 -1.039268  1.075770 -0.674600
2000-01-06  0.524988 -0.370647 -0.109050 -1.776904
2000-01-07  0.404705 -1.157892  1.643563 -0.968914

[5 rows x 4 columns]
```

Specifying an apply that operates on a Series (to return a single element)

```
In [32]: panel.apply(lambda x: x.dtype, axis='items')
Out[32]:
                  A        B        C        D
2000-01-03  float64  float64  float64  float64
2000-01-04  float64  float64  float64  float64
2000-01-05  float64  float64  float64  float64
2000-01-06  float64  float64  float64  float64
2000-01-07  float64  float64  float64  float64

[5 rows x 4 columns]
```

A similar reduction type operation

```
In [33]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[33]:
      ItemA     ItemB     ItemC
A -1.108775 -1.090118 -2.984435
B -3.705764  0.409204  1.866240
C  2.110856  2.960500 -0.974967
D -4.532785  0.303202 -3.685193

[4 rows x 3 columns]
```

This is equivalent to

```
In [34]: panel.sum('major_axis')
Out[34]:
      ItemA     ItemB     ItemC
A -1.108775 -1.090118 -2.984435
B -3.705764  0.409204  1.866240
C  2.110856  2.960500 -0.974967
D -4.532785  0.303202 -3.685193

[4 rows x 3 columns]
```

A transformation operation that returns a Panel, but is computing the z-score across the major_axis

```
In [35]: result = panel.apply(lambda x: (x - x.mean()) / x.std(),
   ....:                      axis='major_axis')
   ....:

In [36]: result
Out[36]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [37]: result['ItemA']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-03 -0.535778  1.500802 -1.506416 -0.681456
2000-01-04  0.397628 -1.108752  0.360481  1.529895
2000-01-05 -1.489811 -0.339412  0.557374  0.280845
2000-01-06  0.885279  0.421830 -0.453013 -1.053785
```

(continues on next page)

(continued from previous page)

```
2000-01-07  0.742682 -0.474468  1.041575 -0.075499

[5 rows x 4 columns]
```

- Panel `apply()` operating on cross-sectional slabs. (GH1148)

```
In [38]: def f(x):
   ....:     return ((x.T - x.mean(1)) / x.std(1)).T
   ....:

In [39]: result = panel.apply(f, axis=['items', 'major_axis'])

In [40]: result
Out[40]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC

In [41]: result.loc[:, :, 'ItemA']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-03  0.012922 -0.030874 -0.629546 -0.757034
2000-01-04  0.392053 -1.071665  0.163228  0.548188
2000-01-05 -1.093650 -0.640898  0.385734 -1.154310
2000-01-06  1.005446 -1.154593 -0.595615 -0.809185
2000-01-07  0.783051 -0.198053  0.919339 -1.052721

[5 rows x 4 columns]
```

This is equivalent to the following

```
In [42]: result = pd.Panel({ax: f(panel.loc[:, :, ax]) for ax in panel.minor_axis}
↪)

In [43]: result
Out[43]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC

In [44]: result.loc[:, :, 'ItemA']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2000-01-03  0.012922 -0.030874 -0.629546 -0.757034
2000-01-04  0.392053 -1.071665  0.163228  0.548188
2000-01-05 -1.093650 -0.640898  0.385734 -1.154310
2000-01-06  1.005446 -1.154593 -0.595615 -0.809185
2000-01-07  0.783051 -0.198053  0.919339 -1.052721

[5 rows x 4 columns]
```

Performance

Performance improvements for 0.13.1

- Series datetime/timedelta binary operations (GH5801)
- DataFrame `count/dropna` for `axis=1`
- Series.str.contains now has a *regex=False* keyword which can be faster for plain (non-regex) string patterns. (GH5879)
- Series.str.extract (GH5944)
- `dtypes/ftypes` methods (GH5968)
- indexing with object dtypes (GH5968)
- `DataFrame.apply` (GH6013)
- Regression in JSON IO (GH5765)
- Index construction from Series (GH6150)

Experimental

There are no experimental changes in 0.13.1

Bug Fixes

- Bug in `io.wb.get_countries` not including all countries (GH6008)
- Bug in Series replace with timestamp dict (GH5797)
- read_csv/read_table now respects the *prefix* kwarg (GH5732).
- Bug in selection with missing values via `.ix` from a duplicate indexed DataFrame failing (GH5835)
- Fix issue of boolean comparison on empty DataFrames (GH5808)
- Bug in isnull handling `NaT` in an object array (GH5443)
- Bug in `to_datetime` when passed a `np.nan` or integer datelike and a format string (GH5863)
- Bug in groupby dtype conversion with datetimelike (GH5869)
- Regression in handling of empty Series as indexers to Series (GH5877)
- Bug in internal caching, related to (GH5727)
- Testing bug in reading JSON/msgpack from a non-filepath on windows under py3 (GH5874)
- Bug when assigning to .ix[tuple(…)] (GH5896)
- Bug in fully reindexing a Panel (GH5905)
- Bug in idxmin/max with object dtypes (GH5914)
- Bug in `BusinessDay` when adding n days to a date not on offset when n>5 and n%5==0 (GH5890)
- Bug in assigning to chained series with a series via ix (GH5928)
- Bug in creating an empty DataFrame, copying, then assigning (GH5932)
- Bug in DataFrame.tail with empty frame (GH5846)
- Bug in propagating metadata on `resample` (GH5862)

- Fixed string-representation of `NaT` to be "NaT" (GH5708)
- Fixed string-representation for Timestamp to show nanoseconds if present (GH5912)
- `pd.match` not returning passed sentinel
- `Panel.to_frame()` no longer fails when `major_axis` is a `MultiIndex` (GH5402).
- Bug in `pd.read_msgpack` with inferring a `DateTimeIndex` frequency incorrectly (GH5947)
- Fixed `to_datetime` for array with both Tz-aware datetimes and `NaT`'s (GH5961)
- Bug in rolling skew/kurtosis when passed a Series with bad data (GH5749)
- Bug in scipy `interpolate` methods with a datetime index (GH5975)
- Bug in NaT comparison if a mixed datetime/np.datetime64 with NaT were passed (GH5968)
- Fixed bug with `pd.concat` losing dtype information if all inputs are empty (GH5742)
- Recent changes in IPython cause warnings to be emitted when using previous versions of pandas in QTConsole, now fixed. If you're using an older version and need to suppress the warnings, see (GH5922).
- Bug in merging `timedelta` dtypes (GH5695)
- Bug in plotting.scatter_matrix function. Wrong alignment among diagonal and off-diagonal plots, see (GH5497).
- Regression in Series with a MultiIndex via ix (GH6018)
- Bug in Series.xs with a MultiIndex (GH6018)
- Bug in Series construction of mixed type with datelike and an integer (which should result in object type and not automatic conversion) (GH6028)
- Possible segfault when chained indexing with an object array under NumPy 1.7.1 (GH6026, GH6056)
- Bug in setting using fancy indexing a single element with a non-scalar (e.g. a list), (GH6043)
- `to_sql` did not respect `if_exists` (GH4110 GH4304)
- Regression in `.get(None)` indexing from 0.12 (GH5652)
- Subtle `iloc` indexing bug, surfaced in (GH6059)
- Bug with insert of strings into DatetimeIndex (GH5818)
- Fixed unicode bug in to_html/HTML repr (GH6098)
- Fixed missing arg validation in get_options_data (GH6105)
- Bug in assignment with duplicate columns in a frame where the locations are a slice (e.g. next to each other) (GH6120)
- Bug in propagating _ref_locs during construction of a DataFrame with dups index/columns (GH6121)
- Bug in `DataFrame.apply` when using mixed datelike reductions (GH6125)
- Bug in `DataFrame.append` when appending a row with different columns (GH6129)
- Bug in DataFrame construction with recarray and non-ns datetime dtype (GH6140)
- Bug in `.loc` setitem indexing with a dataframe on rhs, multiple item setting, and a datetimelike (GH6152)
- Fixed a bug in `query`/`eval` during lexicographic string comparisons (GH6155).
- Fixed a bug in `query` where the index of a single-element `Series` was being thrown away (GH6148).
- Bug in `HDFStore` on appending a dataframe with MultiIndexed columns to an existing table (GH6167)

- Consistency with dtypes in setting an empty DataFrame (GH6171)
- Bug in selecting on a MultiIndex `HDFStore` even in the presence of under specified column spec (GH6169)
- Bug in `nanops.var` with `ddof=1` and 1 elements would sometimes return `inf` rather than `nan` on some platforms (GH6136)
- Bug in Series and DataFrame bar plots ignoring the `use_index` keyword (GH6209)
- Bug in groupby with mixed str/int under python3 fixed; `argsort` was failing (GH6212)

Contributors

A total of 52 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Alex Rothberg
- Alok Singhal +
- Andrew Burrows +
- Andy Hayden
- Bjorn Arneson +
- Brad Buran
- Caleb Epstein
- Chapman Siu
- Chase Albert +
- Clark Fitzgerald +
- DSM
- Dan Birken
- Daniel Waeber +
- David Wolever +
- Doran Deluz +
- Douglas McNeil +
- Douglas Rudd +
- Drazen Lucanin
- Elliot S +
- Felix Lawrence +
- George Kuan +
- Guillaume Gay +
- Jacob Schaer
- Jan Wagner +
- Jeff Tratner
- John McNamara
- Joris Van den Bossche

- Julia Evans +
- Kieran O'Mahony
- Michael Schatzow +
- Naveen Michaud-Agrawal +
- Patrick O'Keeffe +
- Phillip Cloud
- Roman Pekar
- Skipper Seabold
- Spencer Lyon
- Tom Augspurger +
- TomAugspurger
- acorbe +
- akittredge +
- bmu +
- bwignall +
- chapman siu
- danielballan
- david +
- davidshinn
- immerrr +
- jreback
- lexual
- mwaskom +
- unutbu
- y-p

8.12.2 v0.13.0 (January 3, 2014)

This is a major release from 0.12.0 and includes a number of API changes, several new features and enhancements along with a large number of bug fixes.

Highlights include:

- support for a new index type `Float64Index`, and other Indexing enhancements
- `HDFStore` has a new string based syntax for query specification
- support for new methods of interpolation
- updated `timedelta` operations
- a new string manipulation method `extract`
- Nanosecond support for Offsets

- `isin` for DataFrames

Several experimental features are added, including:

- new `eval/query` methods for expression evaluation
- support for `msgpack` serialization
- an i/o interface to Google's `BigQuery`

Their are several new or updated docs sections including:

- *Comparison with SQL*, which should be useful for those familiar with SQL but still learning pandas.
- *Comparison with R*, idiom translations from R to pandas.
- *Enhancing Performance*, ways to enhance pandas performance with `eval/query`.

> **Warning:** In 0.13.0 `Series` has internally been refactored to no longer sub-class `ndarray` but instead subclass `NDFrame`, similar to the rest of the pandas containers. This should be a transparent change with only very limited API implications. See *Internal Refactoring*

API changes

- `read_excel` now supports an integer in its `sheetname` argument giving the index of the sheet to read in (GH4301).
- Text parser now treats anything that reads like inf ("inf", "Inf", "-Inf", "iNf", etc.) as infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
- `pandas` now is Python 2/3 compatible without the need for 2to3 thanks to @jtratner. As a result, pandas now uses iterators more extensively. This also led to the introduction of substantive parts of the Benjamin Peterson's `six` library into compat. (GH4384, GH4375, GH4372)
- `pandas.util.compat` and `pandas.util.py3compat` have been merged into `pandas.compat`. `pandas.compat` now includes many functions allowing 2/3 compatibility. It contains both list and iterator versions of range, filter, map and zip, plus other necessary elements for Python 3 compatibility. `lmap`, `lzip`, `lrange` and `lfilter` all produce lists instead of iterators, for compatibility with `numpy`, subscripting and `pandas` constructors.(GH4384, GH4375, GH4372)
- `Series.get` with negative indexers now returns the same as `[]` (GH4390)
- Changes to how `Index` and `MultiIndex` handle metadata (`levels`, `labels`, and `names`) (GH4039):

```
# previously, you would have set levels or labels directly
>>> pd.index.levels = [[1, 2, 3, 4], [1, 2, 4, 4]]

# now, you use the set_levels or set_labels methods
>>> index = pd.index.set_levels([[1, 2, 3, 4], [1, 2, 4, 4]])

# similarly, for names, you can rename the object
# but setting names is not deprecated
>>> index = pd.index.set_names(["bob", "cranberry"])

# and all methods take an inplace kwarg - but return None
>>> pd.index.set_names(["bob", "cranberry"], inplace=True)
```

- **All** division with `NDFrame` objects is now *truedivision*, regardless of the future import. This means that operating on pandas objects will by default use *floating point* division, and return a floating point dtype. You can use `//` and `floordiv` to do integer division.

Integer division

```
In [3]: arr = np.array([1, 2, 3, 4])

In [4]: arr2 = np.array([5, 3, 2, 1])

In [5]: arr / arr2
Out[5]: array([0, 0, 1, 4])

In [6]: pd.Series(arr) // pd.Series(arr2)
Out[6]:
0    0
1    0
2    1
3    4
dtype: int64
```

True Division

```
In [7]: pd.Series(arr) / pd.Series(arr2)  # no future import required
Out[7]:
0    0.200000
1    0.666667
2    1.500000
3    4.000000
dtype: float64
```

- Infer and downcast dtype if `downcast='infer'` is passed to `fillna/ffill/bfill` (GH4604)
- `__nonzero__` for all NDFrame objects, will now raise a `ValueError`, this reverts back to (GH1073, GH4633) behavior. See *gotchas* for a more detailed discussion.

  This prevents doing boolean comparison on *entire* pandas objects, which is inherently ambiguous. These all will raise a `ValueError`.

```
>>> df = pd.DataFrame({'A': np.random.randn(10),
...                    'B': np.random.randn(10),
...                    'C': pd.date_range('20130101', periods=10)
...                    })
...
>>> if df:
...     pass
...
Traceback (most recent call last):
    ...
ValueError: The truth value of a DataFrame is ambiguous.  Use a.empty,
a.bool(), a.item(), a.any() or a.all().

>>> df1 = df
>>> df2 = df
>>> df1 and df2
Traceback (most recent call last):
    ...
ValueError: The truth value of a DataFrame is ambiguous.  Use a.empty,
a.bool(), a.item(), a.any() or a.all().

>>> d = [1, 2, 3]
>>> s1 = pd.Series(d)
>>> s2 = pd.Series(d)
```

(continues on next page)

(continued from previous page)

```
>>> s1 and s2
Traceback (most recent call last):
    ...
ValueError: The truth value of a DataFrame is ambiguous.  Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

Added the `.bool()` method to `NDFrame` objects to facilitate evaluating of single-element boolean Series:

```
In [1]: pd.Series([True]).bool()
Out[1]: True

In [2]: pd.Series([False]).bool()
\\\\\\\\\\\\\Out[2]: False

In [3]: pd.DataFrame([[True]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]: True

In [4]: pd.DataFrame([[False]]).bool()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]: False
```

- All non-Index NDFrames (`Series`, `DataFrame`, `Panel`, `Panel4D`, `SparsePanel`, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (add, sub, mul, etc.). `SparsePanel` does not support `pow` or `mod` with non-scalars. (GH3765)
- `Series` and `DataFrame` now have a `mode()` method to calculate the statistical mode(s) by axis/Series. (GH5367)
- Chained assignment will now by default warn if the user is assigning to a copy. This can be changed with the option `mode.chained_assignment`, allowed options are `raise/warn/None`. See *the docs*.

```
In [5]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [6]: pd.set_option('chained_assignment', 'warn')
```

The following warning / exception will show if this is attempted.

```
In [7]: dfc.loc[0]['A'] = 1111
```

```
Traceback (most recent call last)
   ...
SettingWithCopyWarning:
   A value is trying to be set on a copy of a slice from a DataFrame.
   Try using .loc[row_index,col_indexer] = value instead
```

Here is the correct method of assignment.

```
In [8]: dfc.loc[0, 'A'] = 11

In [9]: dfc
Out[9]: 
     A  B
0   11  1
1  bbb  2
2  ccc  3
```

- **`Panel.reindex` has the following call signature `Panel.reindex(items=None, major_axis=None, minor_a`**
  to conform with other `NDFrame` objects. See *Internal Refactoring* for more information.

- **`Series.argmin` and `Series.argmax` are now aliased to `Series.idxmin` and `Series.idxmax`. These return the *i***
  min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element. (GH6214)

Prior Version Deprecations/Changes

These were announced changes in 0.12 or prior that are taking effect as of 0.13.0

- Remove deprecated `Factor` (GH3650)
- Remove deprecated `set_printoptions/reset_printoptions` (GH3046)
- Remove deprecated `_verbose_info` (GH3215)
- Remove deprecated `read_clipboard/to_clipboard/ExcelFile/ExcelWriter` from `pandas.io.parsers` (GH3717) These are available as functions in the main pandas namespace (e.g. `pd.read_clipboard`)
- default for `tupleize_cols` is now `False` for both `to_csv` and `read_csv`. Fair warning in 0.12 (GH3604)
- default for *display.max_seq_len* is now 100 rather then *None*. This activates truncated display ("…") of long sequences in various places. (GH3391)

Deprecations

Deprecated in 0.13.0

- deprecated `iterkv`, which will be removed in a future release (this was an alias of iteritems used to bypass `2to3`'s changes). (GH4384, GH4375, GH4372)
- deprecated the string method `match`, whose role is now performed more idiomatically by `extract`. In a future release, the default behavior of `match` will change to become analogous to `contains`, which returns a boolean indexer. (Their distinction is strictness: `match` relies on `re.match` while `contains` relies on `re.search`.) In this release, the deprecated behavior is the default, but the new behavior is available through the keyword argument `as_indexer=True`.

Indexing API Changes

Prior to 0.13, it was impossible to use a label indexer (`.loc/.ix`) to set a value that was not contained in the index of a particular axis. (GH2578). See *the docs*

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

```
In [10]: s = pd.Series([1, 2, 3])

In [11]: s
Out[11]:
0    1
1    2
2    3
dtype: int64

In [12]: s[5] = 5.

In [13]: s
Out[13]:
```

(continues on next page)

(continued from previous page)

```
0    1.0
1    2.0
2    3.0
5    5.0
dtype: float64
```

```
In [14]: dfi = pd.DataFrame(np.arange(6).reshape(3, 2),
   ....:                    columns=['A', 'B'])
   ....:

In [15]: dfi
Out[15]:
   A  B
0  0  1
1  2  3
2  4  5
```

This would previously KeyError

```
In [16]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']

In [17]: dfi
Out[17]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an append operation.

```
In [18]: dfi.loc[3] = 5

In [19]: dfi
Out[19]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```

A Panel setting operation on an arbitrary axis aligns the input to the Panel

```
In [20]: p = pd.Panel(np.arange(16).reshape(2, 4, 2),
   ....:              items=['Item1', 'Item2'],
   ....:              major_axis=pd.date_range('2001/1/12', periods=4),
   ....:              minor_axis=['A', 'B'], dtype='float64')
   ....:

In [21]: p
Out[21]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to B
```

(continues on next page)

(continued from previous page)

```
In [22]: p.loc[:, :, 'C'] = pd.Series([30, 32], index=p.items)

In [23]: p
Out[23]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to C

In [24]: p.loc[:, :, 'C']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            Item1  Item2
2001-01-12   30.0   32.0
2001-01-13   30.0   32.0
2001-01-14   30.0   32.0
2001-01-15   30.0   32.0
```

Float64Index API Change

- Added a new index type, `Float64Index`. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes `[],ix,loc` for scalar indexing and slicing work exactly the same. See *the docs*, (GH263)

  Construction is by default for floating type values.

```
In [25]: index = pd.Index([1.5, 2, 3, 4.5, 5])

In [26]: index
Out[26]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [27]: s = pd.Series(range(5), index=index)

In [28]: s
Out[28]:
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
dtype: int64
```

  Scalar selection for `[],.ix,.loc` will always be label based. An integer will match an equal float index (e.g. `3` is equivalent to `3.0`)

```
In [29]: s[3]
Out[29]: 2

In [30]: s.loc[3]
\\\\\\\\\\\Out[30]: 2
```

  The only positional indexing is via `iloc`

```
In [31]: s.iloc[3]
Out[31]: 3
```

A scalar index that is not found will raise `KeyError`

Slicing is ALWAYS on the values of the index, for `[]`, `ix`, `loc` and ALWAYS positional with `iloc`

```
In [32]: s[2:4]
Out[32]:
2.0    1
3.0    2
dtype: int64

In [33]: s.loc[2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[33]:
2.0    1
3.0    2
dtype: int64

In [34]: s.iloc[2:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[34]:␣
↪
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats are allowed

```
In [35]: s[2.1:4.6]
Out[35]:
3.0    2
4.5    3
dtype: int64

In [36]: s.loc[2.1:4.6]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[36]:
3.0    2
4.5    3
dtype: int64
```

- Indexing on other index types are preserved (and positional fallback for `[]`, `ix`), with the exception, that floating point slicing on indexes on non `Float64Index` will now raise a `TypeError`.

```
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type␣
↪(Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type␣
↪(Int64Index)
```

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

```
In [3]: pd.Series(range(5))[3.0]
Out[3]: 3
```

HDFStore API Changes

- Query Format Changes. A much more string-like query format is now supported. See *the docs*.

```
In [37]: path = 'test.h5'

In [38]: dfq = pd.DataFrame(np.random.randn(10, 4),
   ....:                    columns=list('ABCD'),
   ....:                    index=pd.date_range('20130101', periods=10))
   ....:

In [39]: dfq.to_hdf(path, 'dfq', format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [40]: pd.read_hdf(path, 'dfq',
   ....:             where="index>Timestamp('20130104') & columns=['A', 'B']")
   ....:
Out[40]:
                   A         B
2013-01-05 -0.424972  0.567020
2013-01-06 -0.673690  0.113648
2013-01-07  0.404705  0.577046
2013-01-08 -0.370647 -1.157892
2013-01-09  1.075770 -0.109050
2013-01-10  0.357021 -0.674600
```

Use an inline column reference

```
In [41]: pd.read_hdf(path, 'dfq',
   ....:             where="A>0 or C>0")
   ....:
Out[41]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-07  0.404705  0.577046 -1.715002 -1.039268
2013-01-09  1.075770 -0.109050  1.643563 -1.469388
2013-01-10  0.357021 -0.674600 -1.776904 -0.968914
```

- the `format` keyword now replaces the `table` keyword; allowed values are `fixed(f)` or `table(t)` the same defaults as prior < 0.13.0 remain, e.g. `put` implies `fixed` format and `append` implies `table` format. This default format can be set as an option by setting `io.hdf.default_format`.

```
In [42]: path = 'test.h5'

In [43]: df = pd.DataFrame(np.random.randn(10, 2))

In [44]: df.to_hdf(path, 'df_table', format='table')

In [45]: df.to_hdf(path, 'df_table2', append=True)

In [46]: df.to_hdf(path, 'df_fixed')

In [47]: with pd.HDFStore(path) as store:
   ....:     print(store)
   ....:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
```

- Significant table writing performance improvements
- handle a passed `Series` in table format (GH4330)
- can now serialize a `timedelta64[ns]` dtype in a table (GH3577), See *the docs*.
- added an `is_open` property to indicate if the underlying file handle is_open; a closed store will now report 'CLOSED' when viewing the store (rather than raising an error) (GH4409)
- a close of a `HDFStore` now will close that instance of the `HDFStore` but will only close the actual file if the ref count (by `PyTables`) w.r.t. all of the open handles are 0. Essentially you have a local instance of `HDFStore` referenced by a variable. Once you close it, it will report closed. Other references (to the same file) will continue to operate until they themselves are closed. Performing an action on a closed file will raise `ClosedFileError`

```
In [48]: path = 'test.h5'

In [49]: df = pd.DataFrame(np.random.randn(10, 2))

In [50]: store1 = pd.HDFStore(path)

In [51]: store2 = pd.HDFStore(path)

In [52]: store1.append('df', df)

In [53]: store2.append('df2', df)

In [54]: store1
Out[54]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [55]: store2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[55]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [56]: store1.close()

In [57]: store2
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [58]: store2.close()

In [59]: store2
Out[59]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5
```

- removed the `_quiet` attribute, replace by a `DuplicateWarning` if retrieving duplicate rows from a table (GH4367)
- removed the `warn` argument from `open`. Instead a `PossibleDataLossError` exception will be raised if you try to use `mode='w'` with an OPEN file handle (GH4367)
- allow a passed locations array or mask as a `where` condition (GH4467). See *the docs* for an example.
- add the keyword `dropna=True` to `append` to change whether ALL nan rows are not written to the store (default is `True`, ALL nan rows are NOT written), also settable via the option `io.hdf.dropna_table`

(GH4625)

- pass through store creation arguments; can be used to support in-memory stores

DataFrame repr Changes

The HTML and plain text representations of *DataFrame* now show a truncated view of the table once it exceeds a certain size, rather than switching to the short info view (GH4886, GH5550). This makes the representation more consistent as small DataFrames get larger.

2010-03-29	13.70	13.88	13.39	13.57	158225000	12.98
2010-03-30	13.55	13.64	13.18	13.28	142055200	12.70
	...	...	...	...	...	...

771 rows × 6 columns

To get the info view, call *DataFrame.info()*. If you prefer the info view as the repr for large DataFrames, you can set this by running `set_option('display.large_repr', 'info')`.

Enhancements

- `df.to_clipboard()` learned a new `excel` keyword that let's you paste df data directly into excel (enabled by default). (GH5070).
- `read_html` now raises a `URLError` instead of catching and raising a `ValueError` (GH4303, GH4305)
- Added a test for `read_clipboard()` and `to_clipboard()` (GH4282)
- Clipboard functionality now works with PySide (GH4282)
- Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)
- `to_dict` now takes `records` as a possible out type. Returns an array of column-keyed dictionaries. (GH4936)
- `NaN` handing in get_dummies (GH4446) with *dummy_na*

```
# previously, nan was erroneously counted as 2 here
# now it is not counted at all
In [60]: pd.get_dummies([1, 2, np.nan])
Out[60]:
   1.0  2.0
0    1    0
1    0    1
2    0    0

# unless requested
In [61]: pd.get_dummies([1, 2, np.nan], dummy_na=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[61]:
    1.0  2.0  NaN
0     1    0    0
1     0    1    0
2     0    0    1
```

- `timedelta64[ns]` operations. See *the docs*.

Warning: Most of these operations require `numpy >= 1.7`

Using the new top-level `to_timedelta`, you can convert a scalar or array from the standard timedelta format (produced by `to_csv`) into a timedelta type (`np.timedelta64` in `nanoseconds`).

```
In [62]: pd.to_timedelta('1 days 06:05:01.00003')
Out[62]: Timedelta('1 days 06:05:01.000030')

In [63]: pd.to_timedelta('15.5us')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[63]: Timedelta('0 days 00:00:00.
↪000015')

In [64]: pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[64
↪TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT], dtype=
↪'timedelta64[ns]', freq=None)

In [65]: pd.to_timedelta(np.arange(5), unit='s')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'],␣
↪dtype='timedelta64[ns]', freq=None)

In [66]: pd.to_timedelta(np.arange(5), unit='d')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↪'timedelta64[ns]', freq=None)
```

A Series of dtype `timedelta64[ns]` can now be divided by another `timedelta64[ns]` object, or astyped to yield a `float64` dtyped Series. This is frequency conversion. See *the docs* for the docs.

```
In [67]: import datetime

In [68]: td = pd.Series(pd.date_range('20130101', periods=4)) - pd.Series(
   ....:     pd.date_range('20121201', periods=4))
   ....:

In [69]: td[2] += np.timedelta64(datetime.timedelta(minutes=5, seconds=3))

In [70]: td[3] = np.nan

In [71]: td
Out[71]:
0   31 days 00:00:00
1   31 days 00:00:00
2   31 days 00:05:03
3                NaT
dtype: timedelta64[ns]

# to days
In [72]: td / np.timedelta64(1, 'D')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    31.000000
1    31.000000
```

(continues on next page)

(continued from previous page)

```
2    31.003507
3          NaN
dtype: float64

In [73]: td.astype('timedelta64[D]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    31.0
1    31.0
2    31.0
3     NaN
dtype: float64

# to seconds
In [74]: td / np.timedelta64(1, 's')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64

In [75]: td.astype('timedelta64[s]')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64
```

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series

```
In [76]: td * -1
Out[76]:
0   -31 days +00:00:00
1   -31 days +00:00:00
2   -32 days +23:54:57
3                  NaT
dtype: timedelta64[ns]

In [77]: td * pd.Series([1, 2, 3, 4])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   31 days 00:00:00
1   62 days 00:00:00
2   93 days 00:15:09
3                NaT
dtype: timedelta64[ns]
```

Absolute `DateOffset` objects can act equivalently to `timedeltas`

```
In [78]: from pandas import offsets

In [79]: td + offsets.Minute(5) + offsets.Milli(5)
Out[79]:
```

(continues on next page)

(continued from previous page)

```
0   31 days 00:05:00.005000
1   31 days 00:05:00.005000
2   31 days 00:10:03.005000
3                       NaT
dtype: timedelta64[ns]
```

Fillna is now supported for timedeltas

```
In [80]: td.fillna(0)
Out[80]:
0   31 days 00:00:00
1   31 days 00:00:00
2   31 days 00:05:03
3    0 days 00:00:00
dtype: timedelta64[ns]

In [81]: td.fillna(datetime.timedelta(days=1, seconds=5))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0   31 days 00:00:00
1   31 days 00:00:00
2   31 days 00:05:03
3    1 days 00:00:05
dtype: timedelta64[ns]
```

You can do numeric reduction operations on timedeltas.

```
In [82]: td.mean()
Out[82]: Timedelta('31 days 00:01:41')

In [83]: td.quantile(.1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[83]: Timedelta('31 days 00:00:00')
```

- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to scipy.stats.gaussian_kde() (for scipy >= 0.11.0) to set the bandwidth, and to gkde.evaluate() to specify the indices at which it is evaluated, respectively. See scipy docs. (GH4298)
- DataFrame constructor now accepts a numpy masked record array (GH3478)
- The new vectorized string method `extract` return regular expression matches more conveniently.

```
In [84]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\\d)')
Out[84]:
     0
0    1
1    2
2  NaN
```

Elements that do not match return `NaN`. Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [85]: pd.Series(['a1', 'b2', 'c3']).str.extract('([ab])(\\d)')
Out[85]:
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

Elements that do not match return a row of `NaN`. Thus, a Series of messy strings can be *converted* into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects.

Named groups like

```
In [86]: pd.Series(['a1', 'b2', 'c3']).str.extract(
   ....:     '(?P<letter>[ab])(?P<digit>\\d)')
   ....:
Out[86]:
  letter digit
0      a     1
1      b     2
2    NaN   NaN
```

and optional groups can also be used.

```
In [87]: pd.Series(['a1', 'b2', '3']).str.extract(
   ....:      '(?P<letter>[ab])?(?P<digit>\\d)')
   ....:
Out[87]:
  letter digit
0      a     1
1      b     2
2    NaN     3
```

- `read_stata` now accepts Stata 13 format (GH4291)
- `read_fwf` now infers the column specifications from the first 100 rows of the file if the data has correctly separated and properly aligned columns using the delimiter provided to the function (GH4488).
- support for nanosecond times as an offset

Warning: These operations require `numpy >= 1.7`

Period conversions in the range of seconds and below were reworked and extended up to nanoseconds. Periods in the nanosecond range are now available.

```
In [88]: pd.date_range('2013-01-01', periods=5, freq='5N')
Out[88]:
DatetimeIndex([          '2013-01-01 00:00:00',
               '2013-01-01 00:00:00.000000005',
               '2013-01-01 00:00:00.000000010',
               '2013-01-01 00:00:00.000000015',
               '2013-01-01 00:00:00.000000020'],
              dtype='datetime64[ns]', freq='5N')
```

or with frequency as offset

```
In [89]: pd.date_range('2013-01-01', periods=5, freq=pd.offsets.Nano(5))
Out[89]:
DatetimeIndex([          '2013-01-01 00:00:00',
               '2013-01-01 00:00:00.000000005',
               '2013-01-01 00:00:00.000000010',
               '2013-01-01 00:00:00.000000015',
               '2013-01-01 00:00:00.000000020'],
              dtype='datetime64[ns]', freq='5N')
```

Timestamps can be modified in the nanosecond range

```
In [90]: t = pd.Timestamp('20130101 09:01:02')

In [91]: t + pd.tseries.offsets.Nano(123)
Out[91]: Timestamp('2013-01-01 09:01:02.000000123')
```

- A new method, `isin` for DataFrames, which plays nicely with boolean indexing. The argument to `isin`, what we're comparing the DataFrame to, can be a DataFrame, Series, dict, or array of values. See *the docs* for more.

  To get the rows where any of the conditions are met:

```
In [92]: dfi = pd.DataFrame({'A': [1, 2, 3, 4], 'B': ['a', 'b', 'f', 'n']})

In [93]: dfi
Out[93]:
   A  B
0  1  a
1  2  b
2  3  f
3  4  n

In [94]: other = pd.DataFrame({'A': [1, 3, 3, 7], 'B': ['e', 'f', 'f', 'e']})

In [95]: mask = dfi.isin(other)

In [96]: mask
Out[96]:
       A      B
0   True  False
1  False  False
2   True   True
3  False  False

In [97]: dfi[mask.any(1)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[97
↪
   A  B
0  1  a
2  3  f
```

- `Series` now supports a `to_frame` method to convert it to a single-column DataFrame (GH5164)
- All R datasets listed here http://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html can now be loaded into Pandas objects

```
# note that pandas.rpy was deprecated in v0.16.0
import pandas.rpy.common as com
com.load_data('Titanic')
```

- `tz_localize` can infer a fall daylight savings transition based on the structure of the unlocalized data (GH4230), see *the docs*
- `DatetimeIndex` is now in the API documentation, see *the docs*
- *`json_normalize()`* is a new method to allow you to create a flat table from semi-structured JSON data. See *the docs* (GH1067)
- Added PySide support for the qtpandas DataFrameModel and DataFrameWidget.
- Python csv parser now supports usecols (GH4335)

- Frequencies gained several new offsets:
  - `LastWeekOfMonth` (GH4637)
  - `FY5253`, and `FY5253Quarter` (GH4511)
- DataFrame has a new `interpolate` method, similar to Series (GH4434, GH1892)

```
In [98]: df = pd.DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
   ....:                    'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
   ....:

In [99]: df.interpolate()
Out[99]:
     A      B
0  1.0   0.25
1  2.1   1.50
2  3.4   2.75
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
```

Additionally, the `method` argument to `interpolate` has been expanded to include `'nearest'`, `'zero'`, `'slinear'`, `'quadratic'`, `'cubic'`, `'barycentric'`, `'krogh'`, `'piecewise_polynomial'`, `'pchip'`, `'polynomial'`, `'spline'` The new methods require scipy. Consult the Scipy reference guide and documentation for more information about when the various methods are appropriate. See *the docs*.

Interpolate now also accepts a `limit` keyword argument. This works similar to `fillna`'s limit:

```
In [100]: ser = pd.Series([1, 3, np.nan, np.nan, np.nan, 11])

In [101]: ser.interpolate(limit=2)
Out[101]:
0     1.0
1     3.0
2     5.0
3     7.0
4     NaN
5    11.0
dtype: float64
```

- Added `wide_to_long` panel data convenience function. See *the docs*.

```
In [102]: np.random.seed(123)

In [103]: df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
   .....:                    "A1980" : {0 : "d", 1 : "e", 2 : "f"},
   .....:                    "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
   .....:                    "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
   .....:                    "X"     : dict(zip(range(3), np.random.randn(3)))
   .....:                   })
   .....:

In [104]: df["id"] = df.index

In [105]: df
Out[105]:
  A1970 A1980  B1970  B1980         X  id
```

(continues on next page)

(continued from previous page)

```
0     a     d     2.5     3.2 -1.085631   0
1     b     e     1.2     1.3  0.997345   1
2     c     f     0.7     0.1  0.282978   2

In [106]: pd.wide_to_long(df, ["A", "B"], i="id", j="year")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                X  A    B
id year
0  1970 -1.085631  a  2.5
1  1970  0.997345  b  1.2
2  1970  0.282978  c  0.7
0  1980 -1.085631  d  3.2
1  1980  0.997345  e  1.3
2  1980  0.282978  f  0.1
```

- `to_csv` now takes a `date_format` keyword argument that specifies how output datetime objects should be formatted. Datetimes encountered in the index, columns, and values will all have this formatting applied. (GH4313)
- `DataFrame.plot` will scatter plot x versus y by passing `kind='scatter'` (GH2215)
- Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. (GH5271)

Experimental

- The new *eval()* function implements expression evaluation using `numexpr` behind the scenes. This results in large speedups for complicated expressions involving large DataFrames/Series. For example,

```
In [107]: nrows, ncols = 20000, 100

In [108]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols))
   .....:                       for _ in range(4)]
   .....:
```

```
# eval with NumExpr backend
In [109]: %timeit pd.eval('df1 + df2 + df3 + df4')
8.06 ms +- 699 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
# pure Python evaluation
In [110]: %timeit df1 + df2 + df3 + df4
10 ms +- 601 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

  For more details, see the *the docs*

- Similar to `pandas.eval`, *DataFrame* has a new `DataFrame.eval` method that evaluates an expression in the context of the `DataFrame`. For example,

```
In [111]: df = pd.DataFrame(np.random.randn(10, 2), columns=['a', 'b'])

In [112]: df.eval('a + b')
Out[112]:
0   -0.685204
1    1.589745
2    0.325441
```

(continues on next page)

(continued from previous page)

```
3   -1.784153
4   -0.432893
5    0.171850
6    1.895919
7    3.065587
8   -0.092759
9    1.391365
dtype: float64
```

- `query()` method has been added that allows you to select elements of a `DataFrame` using a natural query syntax nearly identical to Python syntax. For example,

```
In [113]: n = 20

In [114]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=['a', 'b',
↪'c'])

In [115]: df.query('a < b < c')
Out[115]:
    a   b   c
11  1   5   8
15  8  16  19
```

  selects all the rows of `df` where `a < b < c` evaluates to `True`. For more details see the *the docs*.

- `pd.read_msgpack()` and `pd.to_msgpack()` are now a supported method of serialization of arbitrary pandas (and python objects) in a lightweight portable binary format. See *the docs*

> **Warning:** Since this is an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

```
In [116]: df = pd.DataFrame(np.random.rand(5, 2), columns=list('AB'))

In [117]: df.to_msgpack('foo.msg')

In [118]: pd.read_msgpack('foo.msg')
Out[118]:
          A         B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

In [119]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101',␣
↪periods=5))

In [120]: pd.to_msgpack('foo.msg', df, s)

In [121]: pd.read_msgpack('foo.msg')
Out[121]:
[          A         B
 0  0.251082  0.017357
 1  0.347915  0.929879
 2  0.546233  0.203368
```

(continues on next page)

(continued from previous page)

```
 3   0.064942   0.031722
 4   0.355309   0.524575, 2013-01-01    0.022321
 2013-01-02    0.227025
 2013-01-03    0.383282
 2013-01-04    0.193225
 2013-01-05    0.110977
 Freq: D, dtype: float64]
```

You can pass `iterator=True` to iterator over the unpacked results

```
In [122]: for o in pd.read_msgpack('foo.msg', iterator=True):
   .....:     print(o)
   .....:
          A         B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575
2013-01-01    0.022321
2013-01-02    0.227025
2013-01-03    0.383282
2013-01-04    0.193225
2013-01-05    0.110977
Freq: D, dtype: float64
```

- `pandas.io.gbq` provides a simple way to extract from, and load data into, Google's BigQuery Data Sets by way of pandas DataFrames. BigQuery is a high performance SQL-like database service, useful for performing ad-hoc queries against extremely large datasets. *See the docs*

```
from pandas.io import gbq

# A query to select the average monthly temperatures in the
# in the year 2000 across the USA. The dataset,
# publicata:samples.gsod, is available on all BigQuery accounts,
# and is based on NOAA gsod data.

query = """SELECT station_number as STATION,
month as MONTH, AVG(mean_temp) as MEAN_TEMP
FROM publicdata:samples.gsod
WHERE YEAR = 2000
GROUP BY STATION, MONTH
ORDER BY STATION, MONTH ASC"""

# Fetch the result set for this query

# Your Google BigQuery Project ID
# To find this, see your dashboard:
# https://console.developers.google.com/iam-admin/projects?authuser=0
projectid = 'xxxxxxxxx'
df = gbq.read_gbq(query, project_id=projectid)

# Use pandas to process and reshape the dataset

df2 = df.pivot(index='STATION', columns='MONTH', values='MEAN_TEMP')
df3 = pd.concat([df2.min(), df2.mean(), df2.max()],
                axis=1, keys=["Min Tem", "Mean Temp", "Max Temp"])
```

The resulting DataFrame is:

```
> df3
              Min Tem  Mean Temp    Max Temp
 MONTH
 1     -53.336667  39.827892   89.770968
 2     -49.837500  43.685219   93.437932
 3     -77.926087  48.708355   96.099998
 4     -82.892858  55.070087   97.317240
 5     -92.378261  61.428117  102.042856
 6     -77.703334  65.858888  102.900000
 7     -87.821428  68.169663  106.510714
 8     -89.431999  68.614215  105.500000
 9     -86.611112  63.436935  107.142856
 10    -78.209677  56.880838   92.103333
 11    -50.125000  48.861228   94.996428
 12    -50.332258  42.286879   94.396774
```

> **Warning:** To use this module, you will need a BigQuery account. See <https://cloud.google.com/products/big-query> for details.
>
> As of 10/10/13, there is a bug in Google's API preventing result sets from being larger than 100,000 rows. A patch is scheduled for the week of 10/14/13.

Internal Refactoring

In 0.13.0 there is a major refactor primarily to subclass `Series` from `NDFrame`, which is the base class currently for `DataFrame` and `Panel`, to unify methods and behaviors. Series formerly subclassed directly from `ndarray`. (GH4080, GH3862, GH816)

> **Warning:** There are two potential incompatibilities from < 0.13.0
>
> - Using certain numpy functions would previously return a `Series` if passed a `Series` as an argument. This seems only to affect `np.ones_like`, `np.empty_like`, `np.diff` and `np.where`. These now return `ndarrays`.
>
> ```
> In [123]: s = pd.Series([1, 2, 3, 4])
> ```
>
> Numpy Usage
>
> ```
> In [124]: np.ones_like(s)
> Out[124]: array([1, 1, 1, 1])
>
> In [125]: np.diff(s)
> \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[125]: array([1, 1, 1])
>
> In [126]: np.where(s > 1, s, np.nan)
> \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[126]: array([ nan,
> ↪    2.,    3.,    4.])
> ```
>
> Pandonic Usage

```
In [127]: pd.Series(1, index=s.index)
Out[127]:
0    1
1    1
2    1
3    1
dtype: int64

In [128]: s.diff()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[128]:
0    NaN
1    1.0
2    1.0
3    1.0
dtype: float64

In [129]: s.where(s > 1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1    2.0
2    3.0
3    4.0
dtype: float64
```

- Passing a `Series` directly to a cython function expecting an `ndarray` type will no long work directly, you must pass `Series.values`, See *Enhancing Performance*
- `Series(0.5)` would previously return the scalar `0.5`, instead this will return a 1-element `Series`
- This change breaks `rpy2<=2.3.8.` an Issue has been opened against rpy2 and a workaround is detailed in GH5698. Thanks @JanSchulz.

- Pickle compatibility is preserved for pickles created prior to 0.13. These must be unpickled with `pd.read_pickle`, see *Pickling*.
- Refactor of series.py/frame.py/panel.py to move common code to generic.py
  - added `_setup_axes` to created generic NDFrame structures
  - moved methods
    - `from_axes,_wrap_array,axes,ix,loc,iloc,shape,empty,swapaxes,transpose,pop`
    - `__iter__,keys,__contains__,__len__,__neg__,__invert__`
    - `convert_objects,as_blocks,as_matrix,values`
    - `__getstate__,__setstate__` (compat remains in frame/panel)
    - `__getattr__,__setattr__`
    - `_indexed_same,reindex_like,align,where,mask`
    - `fillna,replace` (`Series` replace is now consistent with `DataFrame`)
    - `filter` (also added axis argument to selectively filter on a different axis)
    - `reindex,reindex_axis,take`
    - `truncate` (moved to become part of `NDFrame`)

- These are API changes which make `Panel` more consistent with `DataFrame`
  - `swapaxes` on a `Panel` with the same axes specified now return a copy
  - support attribute access for setting
  - filter supports the same API as the original `DataFrame` filter
- Reindex called with no arguments will now return a copy of the input object
- `TimeSeries` is now an alias for `Series`. the property `is_time_series` can be used to distinguish (if desired)
- Refactor of Sparse objects to use BlockManager
  - Created a new block type in internals, `SparseBlock`, which can hold multi-dtypes and is non-consolidatable. `SparseSeries` and `SparseDataFrame` now inherit more methods from there hierarchy (Series/DataFrame), and no longer inherit from `SparseArray` (which instead is the object of the `SparseBlock`)
  - Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
  - Operations on sparse structures within DataFrames should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
  - enable setitem on `SparseSeries` for boolean/integer/slices
  - `SparsePanels` implementation is unchanged (e.g. not using BlockManager, needs work)
- added `ftypes` method to Series/DataFrame, similar to `dtypes`, but indicates if the underlying is sparse/dense (as well as the dtype)
- All `NDFrame` objects can now use `__finalize__()` to specify various values to propagate to new objects from an existing one (e.g. `name` in `Series` will follow more automatically now)
- Internal type checking is now done via a suite of generated classes, allowing `isinstance(value, klass)` without having to directly import the klass, courtesy of @jtratner
- Bug in Series update where the parent frame is not updating its cache based on changes (GH4080) or types (GH3217), fillna (GH3386)
- Indexing with dtype conversions fixed (GH4463, GH4204)
- Refactor `Series.reindex` to core/generic.py (GH4604, GH4618), allow `method=` in reindexing on a Series to work
- `Series.copy` no longer accepts the `order` parameter and is now consistent with `NDFrame` copy
- Refactor `rename` methods to core/generic.py; fixes `Series.rename` for (GH4605), and adds `rename` with the same signature for `Panel`
- Refactor `clip` methods to core/generic.py (GH4798)
- Refactor of `_get_numeric_data/_get_bool_data` to core/generic.py, allowing Series/Panel functionality
- `Series` (for index) / `Panel` (for items) now allow attribute access to its elements (GH1903)

```
In [130]: s = pd.Series([1, 2, 3], index=list('abc'))

In [131]: s.b
Out[131]: 2

In [132]: s.a = 5
```

(continues on next page)

(continued from previous page)

```
In [133]: s
Out[133]:
a    5
b    2
c    3
dtype: int64
```

Bug Fixes

- HDFStore
  - raising an invalid TypeError rather than ValueError when appending with a different block ordering (GH4096)
  - read_hdf was not respecting as passed mode (GH4504)
  - appending a 0-len table will work correctly (GH4273)
  - to_hdf was raising when passing both arguments append and table (GH4584)
  - reading from a store with duplicate columns across dtypes would raise (GH4767)
  - Fixed a bug where ValueError wasn't correctly raised when column names weren't strings (GH4956)
  - A zero length series written in Fixed format not deserializing properly. (GH4708)
  - Fixed decoding perf issue on pyt3 (GH5441)
  - Validate levels in a MultiIndex before storing (GH5527)
  - Correctly handle data_columns with a Panel (GH5717)
- Fixed bug in tslib.tz_convert(vals, tz1, tz2): it could raise IndexError exception while trying to access trans[pos + 1] (GH4496)
- The by argument now works correctly with the layout argument (GH4102, GH4014) in *.hist plotting methods
- Fixed bug in PeriodIndex.map where using str would return the str representation of the index (GH4136)
- Fixed test failure test_time_series_plot_color_with_empty_kwargs when using custom matplotlib default colors (GH4345)
- Fix running of stata IO tests. Now uses temporary files to write (GH4353)
- Fixed an issue where DataFrame.sum was slower than DataFrame.mean for integer valued frames (GH4365)
- read_html tests now work with Python 2.6 (GH4351)
- Fixed bug where network testing was throwing NameError because a local variable was undefined (GH4381)
- In to_json, raise if a passed orient would cause loss of data because of a duplicate index (GH4359)
- In to_json, fix date handling so milliseconds are the default timestamp as the docstring says (GH4362).
- as_index is no longer ignored when doing groupby apply (GH4648, GH3417)
- JSON NaT handling fixed, NaTs are now serialized to *null* (GH4498)
- Fixed JSON handling of escapable characters in JSON object keys (GH4593)

- Fixed passing `keep_default_na=False` when `na_values=None` (GH4318)
- Fixed bug with `values` raising an error on a DataFrame with duplicate columns and mixed dtypes, surfaced in (GH4377)
- Fixed bug with duplicate columns and type conversion in `read_json` when `orient='split'` (GH4377)
- Fixed JSON bug where locales with decimal separators other than '.' threw exceptions when encoding / decoding certain values. (GH4918)
- Fix `.iat` indexing with a `PeriodIndex` (GH4390)
- Fixed an issue where `PeriodIndex` joining with self was returning a new instance rather than the same instance (GH4379); also adds a test for this for the other index types
- Fixed a bug with all the dtypes being converted to object when using the CSV cparser with the usecols parameter (GH3192)
- Fix an issue in merging blocks where the resulting DataFrame had partially set _ref_locs (GH4403)
- Fixed an issue where hist subplots were being overwritten when they were called using the top level matplotlib API (GH4408)
- Fixed a bug where calling `Series.astype(str)` would truncate the string (GH4405, GH4437)
- Fixed a py3 compat issue where bytes were being repr'd as tuples (GH4455)
- Fixed Panel attribute naming conflict if item is named 'a' (GH3440)
- Fixed an issue where duplicate indexes were raising when plotting (GH4486)
- Fixed an issue where cumsum and cumprod didn't work with bool dtypes (GH4170, GH4440)
- Fixed Panel slicing issued in `xs` that was returning an incorrect dimmed object (GH4016)
- Fix resampling bug where custom reduce function not used if only one group (GH3849, GH4494)
- Fixed Panel assignment with a transposed frame (GH3830)
- Raise on set indexing with a Panel and a Panel as a value which needs alignment (GH3777)
- frozenset objects now raise in the `Series` constructor (GH4482, GH4480)
- Fixed issue with sorting a duplicate MultiIndex that has multiple dtypes (GH4516)
- Fixed bug in `DataFrame.set_values` which was causing name attributes to be lost when expanding the index. (GH3742, GH4039)
- Fixed issue where individual `names`, `levels` and `labels` could be set on `MultiIndex` without validation (GH3714, GH4039)
- Fixed (GH3334) in pivot_table. Margins did not compute if values is the index.
- Fix bug in having a rhs of `np.timedelta64` or `np.offsets.DateOffset` when operating with datetimes (GH4532)
- Fix arithmetic with series/datetimeindex and `np.timedelta64` not working the same (GH4134) and buggy timedelta in NumPy 1.6 (GH4135)
- Fix bug in `pd.read_clipboard` on windows with PY3 (GH4561); not decoding properly
- `tslib.get_period_field()` and `tslib.get_period_field_arr()` now raise if code argument out of range (GH4519, GH4520)
- Fix boolean indexing on an empty series loses index names (GH4235), infer_dtype works with empty arrays.
- Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue (GH3317)

- Fixed issue where `DataFrame.apply` was reraising exceptions incorrectly (causing the original stack trace to be truncated).
- Fix selection with `ix/loc` and non_unique selectors (GH4619)
- Fix assignment with iloc/loc involving a dtype change in an existing column (GH4312, GH5702) have internal setitem_with_indexer in core/indexing to use Block.setitem
- Fixed bug where thousands operator was not handled correctly for floating point numbers in csv_import (GH4322)
- Fix an issue with CacheableOffset not properly being used by many DateOffset; this prevented the DateOffset from being cached (GH4609)
- Fix boolean comparison with a DataFrame on the lhs, and a list/tuple on the rhs (GH4576)
- Fix error/dtype conversion with setitem of `None` on `Series/DataFrame` (GH4667)
- Fix decoding based on a passed in non-default encoding in `pd.read_stata` (GH4626)
- Fix `DataFrame.from_records` with a plain-vanilla `ndarray`. (GH4727)
- Fix some inconsistencies with `Index.rename` and `MultiIndex.rename`, etc. (GH4718, GH4628)
- Bug in using `iloc/loc` with a cross-sectional and duplicate indices (GH4726)
- Bug with using `QUOTE_NONE` with `to_csv` causing `Exception`. (GH4328)
- Bug with Series indexing not raising an error when the right-hand-side has an incorrect length (GH2702)
- Bug in MultiIndexing with a partial string selection as one part of a MultIndex (GH4758)
- Bug with reindexing on the index with a non-unique index will now raise `ValueError` (GH4746)
- Bug in setting with `loc/ix` a single indexer with a MultiIndex axis and a NumPy array, related to (GH3777)
- Bug in concatenation with duplicate columns across dtypes not merging with axis=0 (GH4771, GH4975)
- Bug in `iloc` with a slice index failing (GH4771)
- Incorrect error message with no colspecs or width in `read_fwf`. (GH4774)
- Fix bugs in indexing in a Series with a duplicate index (GH4548, GH4550)
- Fixed bug with reading compressed files with `read_fwf` in Python 3. (GH3963)
- Fixed an issue with a duplicate index and assignment with a dtype change (GH4686)
- Fixed bug with reading compressed files in as `bytes` rather than `str` in Python 3. Simplifies bytes-producing file-handling in Python 3 (GH3963, GH4785).
- Fixed an issue related to ticklocs/ticklabels with log scale bar plots across different versions of matplotlib (GH4789)
- Suppressed DeprecationWarning associated with internal calls issued by repr() (GH4391)
- Fixed an issue with a duplicate index and duplicate selector with `.loc` (GH4825)
- Fixed an issue with `DataFrame.sort_index` where, when sorting by a single column and passing a list for `ascending`, the argument for `ascending` was being interpreted as `True` (GH4839, GH4846)
- Fixed `Panel.tshift` not working. Added *freq* support to `Panel.shift` (GH4853)
- Fix an issue in TextFileReader w/ Python engine (i.e. PythonParser) with thousands != "," (GH4596)
- Bug in getitem with a duplicate index when using where (GH4879)
- Fix Type inference code coerces float column into datetime (GH4601)

- Fixed _ensure_numeric does not check for complex numbers (GH4902)
- Fixed a bug in Series.hist where two figures were being created when the by argument was passed (GH4112, GH4113).
- Fixed a bug in convert_objects for > 2 ndims (GH4937)
- Fixed a bug in DataFrame/Panel cache insertion and subsequent indexing (GH4939, GH5424)
- Fixed string methods for FrozenNDArray and FrozenList (GH4929)
- Fixed a bug with setting invalid or out-of-range values in indexing enlargement scenarios (GH4940)
- Tests for fillna on empty Series (GH4346), thanks @immerrr
- Fixed copy() to shallow copy axes/indices as well and thereby keep separate metadata. (GH4202, GH4830)
- Fixed skiprows option in Python parser for read_csv (GH4382)
- Fixed bug preventing cut from working with np.inf levels without explicitly passing labels (GH3415)
- Fixed wrong check for overlapping in DatetimeIndex.union (GH4564)
- Fixed conflict between thousands separator and date parser in csv_parser (GH4678)
- Fix appending when dtypes are not the same (error showing mixing float/np.datetime64) (GH4993)
- Fix repr for DateOffset. No longer show duplicate entries in kwds. Removed unused offset fields. (GH4638)
- Fixed wrong index name during read_csv if using usecols. Applies to c parser only. (GH4201)
- Timestamp objects can now appear in the left hand side of a comparison operation with a Series or DataFrame object (GH4982).
- Fix a bug when indexing with np.nan via iloc/loc (GH5016)
- Fixed a bug where low memory c parser could create different types in different chunks of the same file. Now coerces to numerical type or raises warning. (GH3866)
- Fix a bug where reshaping a Series to its own shape raised TypeError (GH4554) and other reshaping issues.
- Bug in setting with ix/loc and a mixed int/string index (GH4544)
- Make sure series-series boolean comparisons are label based (GH4947)
- Bug in multi-level indexing with a Timestamp partial indexer (GH4294)
- Tests/fix for MultiIndex construction of an all-nan frame (GH4078)
- Fixed a bug where *read_html()* wasn't correctly inferring values of tables with commas (GH5029)
- Fixed a bug where *read_html()* wasn't providing a stable ordering of returned tables (GH4770, GH5029).
- Fixed a bug where *read_html()* was incorrectly parsing when passed index_col=0 (GH5066).
- Fixed a bug where *read_html()* was incorrectly inferring the type of headers (GH5048).
- Fixed a bug where DatetimeIndex joins with PeriodIndex caused a stack overflow (GH3899).
- Fixed a bug where groupby objects didn't allow plots (GH5102).
- Fixed a bug where groupby objects weren't tab-completing column names (GH5102).
- Fixed a bug where groupby.plot() and friends were duplicating figures multiple times (GH5102).
- Provide automatic conversion of object dtypes on fillna, related (GH5103)
- Fixed a bug where default options were being overwritten in the option parser cleaning (GH5121).

- Treat a list/ndarray identically for `iloc` indexing with list-like (GH5006)
- Fix `MultiIndex.get_level_values()` with missing values (GH5074)
- Fix bound checking for Timestamp() with datetime64 input (GH4065)
- Fix a bug where `TestReadHtml` wasn't calling the correct `read_html()` function (GH5150).
- Fix a bug with `NDFrame.replace()` which made replacement appear as though it was (incorrectly) using regular expressions (GH5143).
- Fix better error message for to_datetime (GH4928)
- Made sure different locales are tested on travis-ci (GH4918). Also adds a couple of utilities for getting locales and setting locales with a context manager.
- Fixed segfault on `isnull(MultiIndex)` (now raises an error instead) (GH5123, GH5125)
- Allow duplicate indices when performing operations that align (GH5185, GH5639)
- Compound dtypes in a constructor raise `NotImplementedError` (GH5191)
- Bug in comparing duplicate frames (GH4421) related
- Bug in describe on duplicate frames
- Bug in `to_datetime` with a format and `coerce=True` not raising (GH5195)
- Bug in `loc` setting with multiple indexers and a rhs of a Series that needs broadcasting (GH5206)
- Fixed bug where inplace setting of levels or labels on `MultiIndex` would not clear cached `values` property and therefore return wrong `values`. (GH5215)
- Fixed bug where filtering a grouped DataFrame or Series did not maintain the original ordering (GH4621).
- Fixed `Period` with a business date freq to always roll-forward if on a non-business date. (GH5203)
- Fixed bug in Excel writers where frames with duplicate column names weren't written correctly. (GH5235)
- Fixed issue with `drop` and a non-unique index on Series (GH5248)
- Fixed segfault in C parser caused by passing more names than columns in the file. (GH5156)
- Fix `Series.isin` with date/time-like dtypes (GH5021)
- C and Python Parser can now handle the more common MultiIndex column format which doesn't have a row for index names (GH4702)
- Bug when trying to use an out-of-bounds date as an object dtype (GH5312)
- Bug when trying to display an embedded PandasObject (GH5324)
- Allows operating of Timestamps to return a datetime if the result is out-of-bounds related (GH5312)
- Fix return value/type signature of `initObjToJSON()` to be compatible with numpy's `import_array()` (GH5334, GH5326)
- Bug when renaming then set_index on a DataFrame (GH5344)
- Test suite no longer leaves around temporary files when testing graphics. (GH5347) (thanks for catching this @yarikoptic!)
- Fixed html tests on win32. (GH4580)
- Make sure that `head/tail` are `iloc` based, (GH5370)
- Fixed bug for `PeriodIndex` string representation if there are 1 or 2 elements. (GH5372)

- The GroupBy methods `transform` and `filter` can be used on Series and DataFrames that have repeated (non-unique) indices. (GH4620)
- Fix empty series not printing name in repr (GH4651)
- Make tests create temp files in temp directory by default. (GH5419)
- `pd.to_timedelta` of a scalar returns a scalar (GH5410)
- `pd.to_timedelta` accepts `NaN` and `NaT`, returning `NaT` instead of raising (GH5437)
- performance improvements in `isnull` on larger size pandas objects
- Fixed various setitem with 1d ndarray that does not have a matching length to the indexer (GH5508)
- Bug in getitem with a MultiIndex and `iloc` (GH5528)
- Bug in delitem on a Series (GH5542)
- Bug fix in apply when using custom function and objects are not mutated (GH5545)
- Bug in selecting from a non-unique index with `loc` (GH5553)
- Bug in groupby returning non-consistent types when user function returns a `None`, (GH5592)
- Work around regression in numpy 1.7.0 which erroneously raises IndexError from `ndarray.item` (GH5666)
- Bug in repeated indexing of object with resultant non-unique index (GH5678)
- Bug in fillna with Series and a passed series/dict (GH5703)
- Bug in groupby transform with a datetime-like grouper (GH5712)
- Bug in MultiIndex selection in PY3 when using certain keys (GH5725)
- Row-wise concat of differing dtypes failing in certain cases (GH5754)

Contributors

A total of 77 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Agustín Herranz +
- Alex Gaudio +
- Alex Rothberg +
- Andreas Klostermann +
- Andreas Wurl +
- Andy Hayden
- Ben Alex +
- Benedikt Sauer +
- Brad Buran
- Caleb Epstein +
- Chang She
- Christopher Whelan
- DSM +
- Dale Jung +

- Dan Birken
- David Rasch +
- Dieter Vandenbussche
- Gabi Davar +
- Garrett Drapala
- Goyo +
- Greg Reda +
- Ivan Smirnov +
- Jack Kelly +
- Jacob Schaer +
- Jan Schulz +
- Jeff Tratner
- Jeffrey Tratner
- John McNamara +
- John W. O'Brien +
- Joris Van den Bossche
- Justin Bozonier +
- Kelsey Jordahl
- Kevin Stone
- Kieran O'Mahony
- Kyle Hausmann +
- Kyle Kelley +
- Kyle Meyer
- Mike Kelly
- Mortada Mehyar +
- Nick Foti +
- Olivier Harris +
- Ondřej Čertík +
- PKEuS
- Phillip Cloud
- Pierre Haessig +
- Richard T. Guy +
- Roman Pekar +
- Roy Hyunjin Han
- Skipper Seabold
- Sten +

- Thomas A Caswell +
- Thomas Kluyver
- Tiago Requeijo +
- TomAugspurger
- Trent Hauck
- Valentin Haenel +
- Viktor Kerkez +
- Vincent Arel-Bundock
- Wes McKinney
- Wes Turner +
- Weston Renoud +
- Yaroslav Halchenko
- Zach Dwiel +
- chapman siu +
- chappers +
- d10genes +
- danielballan
- daydreamt +
- engstrom +
- jreback
- monicaBee +
- prossahl +
- rockg +
- unutbu +
- westurner +
- y-p
- zach powers

8.13 Version 0.12

8.13.1 v0.12.0 (July 24, 2013)

This is a major release from 0.11.0 and includes several new features and enhancements along with a large number of bug fixes.

Highlights include a consistent I/O API naming scheme, routines to read html, write MultiIndexes to csv files, read & write STATA data files, read & write JSON format files, Python 3 support for `HDFStore`, filtering of groupby expressions via `filter`, and a revamped `replace` routine that accepts regular expressions.

API changes

- The I/O API is now much more consistent with a set of top level `reader` functions accessed like `pd.read_csv()` that generally return a `pandas` object.
  - `read_csv`
  - `read_excel`
  - `read_hdf`
  - `read_sql`
  - `read_json`
  - `read_html`
  - `read_stata`
  - `read_clipboard`

  The corresponding `writer` functions are object methods that are accessed like `df.to_csv()`
  - `to_csv`
  - `to_excel`
  - `to_hdf`
  - `to_sql`
  - `to_json`
  - `to_html`
  - `to_stata`
  - `to_clipboard`
- Fix modulo and integer division on Series,DataFrames to act similarly to `float` dtypes to return `np.nan` or `np.inf` as appropriate (GH3590). This correct a numpy bug that treats `integer` and `float` dtypes differently.

```
In [1]: p = pd.DataFrame({'first': [4, 5, 8], 'second': [0, 0, 3]})

In [2]: p % 0
Out[2]:
   first  second
0    NaN     NaN
1    NaN     NaN
2    NaN     NaN

In [3]: p % p
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]:
↪
   first  second
0    0.0     NaN
1    0.0     NaN
2    0.0     0.0

In [4]: p / p
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   first  second
```

(continues on next page)

(continued from previous page)

```
0    1.0     NaN
1    1.0     NaN
2    1.0     1.0

In [5]: p / 0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   first  second
0    inf     NaN
1    inf     NaN
2    inf     inf
```

- Add squeeze keyword to groupby to allow reduction from DataFrame -> Series if groups are unique. This is a Regression from 0.10.1. We are reverting back to the prior behavior. This means groupby will return the same shaped objects whether the groups are unique or not. Revert this issue (GH2893) with (GH3596).

```
In [6]: df2 = pd.DataFrame([{"val1": 1, "val2": 20},
   ...:                     {"val1": 1, "val2": 19},
   ...:                     {"val1": 1, "val2": 27},
   ...:                     {"val1": 1, "val2": 12}])
   ...:

In [7]: def func(dataf):
   ...:     return dataf["val2"] - dataf["val2"].mean()
   ...:

# squeezing the result frame to a series (because we have unique groups)
In [8]: df2.groupby("val1", squeeze=True).apply(func)
Out[8]:
0    0.5
1   -0.5
2    7.5
3   -7.5
Name: 1, dtype: float64

# no squeezing (the default, and behavior in 0.10.1)
In [9]: df2.groupby("val1").apply(func)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
val2    0    1    2    3
val1
1      0.5 -0.5  7.5 -7.5
```

- Raise on iloc when boolean indexing with a label based indexer mask e.g. a boolean Series, even with integer labels, will raise. Since iloc is purely positional based, the labels on the Series are not alignable (GH3631)

  This case is rarely used, and there are plenty of alternatives. This preserves the iloc API to be *purely* positional based.

```
In [10]: df = pd.DataFrame(lrange(5), list('ABCDE'), columns=['a'])

In [11]: mask = (df.a % 2 == 0)

In [12]: mask
Out[12]:
A     True
B    False
C     True
```

(continues on next page)

(continued from previous page)

```
D     False
E      True
Name: a, dtype: bool

# this is what you should use
In [13]: df.loc[mask]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]:
↪
   a
A  0
C  2
E  4

# this will work as well
In [14]: df.iloc[mask.values]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
   a
A  0
C  2
E  4
```

`df.iloc[mask]` will raise a `ValueError`

- The `raise_on_error` argument to plotting functions is removed. Instead, plotting functions raise a `TypeError` when the `dtype` of the object is `object` to remind you to avoid `object` arrays whenever possible and thus you should cast to an appropriate numeric dtype if you need to plot something.
- Add `colormap` keyword to DataFrame plotting methods. Accepts either a matplotlib colormap object (ie, matplotlib.cm.jet) or a string name of such an object (ie, 'jet'). The colormap is sampled to select the color for each column. Please see *Colormaps* for more information. (GH3860)
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead. (GH3582, GH3675, GH3676)
- the `method` and `axis` arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. (GH3907)
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if `True`, default is `False` (same as prior to 0.12) (GH3679)
- Implement `__nonzero__` for `NDFrame` objects (GH3691, GH3696)
- IO api
  - added top-level function `read_excel` to replace the following, The original API is deprecated and will be removed in a future version

    ```
    from pandas.io.parsers import ExcelFile
    xls = ExcelFile('path_to_file.xls')
    xls.parse('Sheet1', index_col=None, na_values=['NA'])
    ```

    With

    ```
    import pandas as pd
    pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
    ```

  - added top-level function `read_sql` that is equivalent to the following

```
from pandas.io.sql import read_frame
read_frame(...)
```

- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Do not allow astypes on `datetime64[ns]` except to `object`, and `timedelta64[ns]` to `object/int` (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when performed on a `Series` and return an *empty* `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of `slice` objects:
  - sum, prod, mean, std, var, skew, kurt, corr, and cov
- `read_html` now defaults to `None` when reading, and falls back on `bs4` + `html5lib` when lxml fails to parse. a list of parsers to try until success is also valid
- The internal `pandas` class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the base class for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)
- New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)

I/O Enhancements

- `pd.read_html()` can now parse HTML strings, files or urls and return DataFrames, courtesy of @cpcloud. (GH3477, GH3605, GH3606, GH3616). It works with a *single* parser backend: BeautifulSoup4 + html5lib *See the docs*

  You can use `pd.read_html()` to read the output from `DataFrame.to_html()` like so

```
In [15]: df = pd.DataFrame({'a': range(3), 'b': list('abc')})

In [16]: print(df)
   a  b
0  0  a
1  1  b
2  2  c

In [17]: html = df.to_html()

In [18]: alist = pd.read_html(html, index_col=0)

In [19]: print(df == alist[0])
      a     b
0  True  True
1  True  True
2  True  True
```

  Note that `alist` here is a Python `list` so `pd.read_html()` and `DataFrame.to_html()` are not inverses.

  - `pd.read_html()` no longer performs hard conversion of date strings (GH3656).

Warning: You may have to install an older version of BeautifulSoup4, *See the installation docs*

- Added module for reading and writing Stata files: `pandas.io.stata` (GH1512) accessible via `read_stata` top-level function for reading, and `to_stata` DataFrame method for writing, *See the docs*
- Added module for reading and writing json format files: `pandas.io.json` accessible via `read_json` top-level function for reading, and `to_json` DataFrame method for writing, *See the docs* various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
- `MultiIndex` column support for reading and writing csv format files
  - The `header` option in `read_csv` now accepts a list of the rows from which to read the index.
  - The option, `tupleize_cols` can now be specified in both `to_csv` and `read_csv`, to provide compatibility for the pre 0.12 behavior of writing and reading `MultIndex` columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a `MultiIndex` column.

    Note: The default behavior in 0.12 remains unchanged from prior versions, but starting with 0.13, the default *to* write and read `MultiIndex` columns will be in the new format. (GH3571, GH1651, GH3141)
  - If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False`), then any `names` on the columns index will be *lost*.

```
In [20]: from pandas.util.testing import makeCustomDataframe as mkdf

In [21]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [22]: df.to_csv('mi.csv')

In [23]: print(open('mi.csv').read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2


In [24]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
C0                C_l0_g0 C_l0_g1 C_l0_g2
C1                C_l1_g0 C_l1_g1 C_l1_g2
C2                C_l2_g0 C_l2_g1 C_l2_g2
C3                C_l3_g0 C_l3_g1 C_l3_g2
R0      R1
R_l0_g0 R_l1_g0      R0C0    R0C1    R0C2
R_l0_g1 R_l1_g1      R1C0    R1C1    R1C2
R_l0_g2 R_l1_g2      R2C0    R2C1    R2C2
R_l0_g3 R_l1_g3      R3C0    R3C1    R3C2
R_l0_g4 R_l1_g4      R4C0    R4C1    R4C2
```

- Support for `HDFStore` (via `PyTables 3.0.0`) on Python3

- Iterator support via `read_hdf` that automatically opens and closes the store when iteration is finished. This is only for *tables*

```
In [25]: path = 'store_iterator.h5'

In [26]: pd.DataFrame(np.random.randn(10, 2)).to_hdf(path, 'df', table=True)

In [27]: for df in pd.read_hdf(path, 'df', chunksize=3):
   ....:     print(df)
   ....:
          0         1
0  0.713216 -0.778461
1 -0.661062  0.862877
2  0.344342  0.149565
          0         1
3 -0.626968 -0.875772
4 -0.930687 -0.218983
5  0.949965 -0.442354
          0         1
6 -0.402985  1.111358
7 -0.241527 -0.670477
8  0.049355  0.632633
          0         1
9 -1.502767 -1.225492
```

- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters

Other Enhancements

- `DataFrame.replace()` now allows regular expressions on contained `Series` with object dtype. See the examples section in the regular docs *Replacing via String Expression*

  For example you can do

```
In [25]: df = pd.DataFrame({'a': list('ab..'), 'b': [1, 2, 3, 4]})

In [26]: df.replace(regex=r'\s*\.\s*', value=np.nan)
Out[26]:
     a  b
0    a  1
1    b  2
2  NaN  3
3  NaN  4
```

  to replace all occurrences of the string `'.'` with zero or more instances of surrounding white space with `NaN`.

  Regular string replacement still works as expected. For example, you can do

```
In [27]: df.replace('.', np.nan)
Out[27]:
     a  b
0    a  1
1    b  2
2  NaN  3
3  NaN  4
```

  to replace all occurrences of the string `'.'` with `NaN`.

- `pd.melt()` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame.
- `pd.set_option()` now allows N option, value pairs (GH3667).

  Let's say that we had an option `'a.b'` and another option `'b.c'`. We can set them at the same time:

```
In [28]: pd.get_option('a.b')
Out[28]: 2

In [29]: pd.get_option('b.c')
\\\\\\\\\\\Out[29]: 3

In [30]: pd.set_option('a.b', 1, 'b.c', 4)

In [31]: pd.get_option('a.b')
Out[31]: 1

In [32]: pd.get_option('b.c')
\\\\\\\\\\\Out[32]: 4
```

- The `filter` method for group objects returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [33]: sf = pd.Series([1, 1, 2, 3, 3, 3])

In [34]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[34]:
3    3
4    3
5    3
dtype: int64
```

  The argument of `filter` must a function that, applied to the group as a whole, returns `True` or `False`.

  Another useful operation is filtering out elements that belong to groups with only a couple members.

```
In [35]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [36]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[36]:
   A  B
2  2  b
3  3  b
4  4  b
5  5  b
```

  Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [37]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[37]:
     A    B
0  NaN  NaN
1  NaN  NaN
2  2.0    b
3  3.0    b
4  4.0    b
```

(continues on next page)

(continued from previous page)

```
5  5.0    b
6  NaN  NaN
7  NaN  NaN
```

- Series and DataFrame hist methods now take a `figsize` argument (GH3834)
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations (GH3877)
- Timestamp.min and Timestamp.max now represent valid Timestamp instances instead of the default datetime.min and datetime.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected (GH4214)

Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. (GH2301)

Note: This uses the `numpy.busdaycalendar` API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.

```
In [38]: from pandas.tseries.offsets import CustomBusinessDay

In [39]: from datetime import datetime

# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [40]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
# add that for a couple of years
In [41]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01
↪')]

In [42]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_
↪egypt)

In [43]: dt = datetime(2013, 4, 30)

In [44]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00

In [45]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [46]: print(pd.Series(dts.weekday, dts).map(pd.Series('Mon Tue Wed Thu Fri Sat␣
↪Sun'.split())))
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
Freq: C, dtype: object
```

Bug Fixes

- Plotting functions now raise a TypeError before trying to plot anything if the associated objects have have a dtype of object (GH1818, GH3572, GH3911, GH3912), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which eliminates any spurious plots from showing up.
- fillna methods now raise a TypeError if the value parameter is a list or tuple.
- Series.str now supports iteration (GH3638). You can iterate over the individual elements of each string in the Series. Each iteration yields yields a Series with either a single character at each index of the original Series or NaN. For example,

```
In [47]: strs = 'go', 'bow', 'joe', 'slow'

In [48]: ds = pd.Series(strs)

In [49]: for s in ds.str:
   ....:     print(s)
   ....:
0    g
1    b
2    j
3    s
dtype: object
0    o
1    o
2    o
3    l
dtype: object
0    NaN
1      w
2      e
3      o
dtype: object
0    NaN
1    NaN
2    NaN
3      w
dtype: object

In [50]: s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1    NaN
2    NaN
3      w
dtype: object

In [51]: s.dropna().values.item() == 'w'
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True
```

  The last element yielded by the iterator will be a Series containing the last element of the longest string in the Series with all other elements being NaN. Here since 'slow' is the longest string and there are no other strings with the same length 'w' is the only non-null string in the yielded Series.

- HDFStore

- will retain index attributes (freq,tz,name) on recreation (GH3499)
- will warn with a `AttributeConflictWarning` if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
- support datelike columns with a timezone as data_columns (GH2852)

- Non-unique index support clarified (GH3468).
  - Fix assigning a new index to a duplicate index in a DataFrame would fail (GH3468)
  - Fix construction of a DataFrame with a duplicate index
  - ref_locs support to allow duplicative indices across dtypes, allows iget support to always find the index (even across dtypes) (GH2194)
  - applymap on a DataFrame with a non-unique index now works (removed warning) (GH2786), and fix (GH3230)
  - Fix to_csv to handle non-unique columns (GH3495)
  - Duplicate indexes with getitem will return items in the correct order (GH3455, GH3457) and handle missing elements like unique indices (GH3561)
  - Duplicate indexes with and empty DataFrame.from_records will return a correct frame (GH3562)
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed (GH3602)
  - Allow insert/delete to non-unique columns (GH3679)
  - Non-unique indexing with a slice via `loc` and friends fixed (GH3659)
  - Allow insert/delete to non-unique columns (GH3679)
  - Extend `reindex` to correctly deal with non-unique indices (GH3679)
  - `DataFrame.itertuples()` now works with frames with duplicate column names (GH3873)
  - Bug in non-unique indexing via `iloc` (GH4017); added `takeable` argument to `reindex` for location-based taking
  - Allow non-unique indexing in series via `.ix/.loc` and `__getitem__` (GH4246)
  - Fixed non-unique indexing memory allocation issue with `.ix/.loc` (GH4280)
- `DataFrame.from_records` did not accept empty recarrays (GH3682)
- `read_html` now correctly skips tests (GH3741)
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn't working (GH3907)
- Improved `network` test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. (GH3910, GH3914)
- Fixed testing issue where too many sockets where open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in test_yahoo, test_google where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask (GH4071)

- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where sharex and sharey were not being passed to grouped_hist (GH4089)
- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when regex=False (GH4115)
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` (GH4152)
- Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
- Fixed bug in plotting that wasn't raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
- Fixed bug where Index slices weren't carrying the name attribute (GH4226)
- Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
- Fixed bug where html5lib wasn't being properly skipped (GH4265)
- Fixed bug where get_data_famafrench wasn't using the correct file edges (GH4281)

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 50 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Andy Hayden
- Chang She
- Christopher Whelan
- Damien Garaud
- Dan Allan
- Dan Birken
- Dieter Vandenbussche
- Dražen Lučanin
- Gábor Lipták +
- Jeff Mellen +
- Jeff Tratner +
- Jeffrey Tratner +
- Jonathan deWerd +
- Joris Van den Bossche +
- Juraj Niznan +
- Karmel Allison
- Kelsey Jordahl
- Kevin Stone +

- Kieran O'Mahony
- Kyle Meyer +
- Mike Kelly +
- PKEuS +
- Patrick O'Brien +
- Phillip Cloud
- Richard Hochenberger +
- Skipper Seabold
- SleepingPills +
- Tobias Brandt
- Tom Farnbauer +
- TomAugspurger +
- Trent Hauck +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
- conmai +
- danielballan +
- davidshinn +
- dieterv77
- duozhang +
- ejnens +
- gliptak +
- jniznan +
- jreback
- lexual
- nipunreddevil +
- ogiaquino +
- stonebig +
- tim smith +
- timmie
- y-p

8.14 Version 0.11

8.14.1 v0.11.0 (April 22, 2013)

This is a major release from 0.10.1 and includes many new features and enhancements along with a large number of bug fixes. The methods of Selecting Data have had quite a number of additions, and Dtype support is now full-fledged. There are also a number of important API changes that long-time pandas users should pay close attention to.

There is a new section in the documentation, *10 Minutes to Pandas*, primarily geared to new users.

There is a new section in the documentation, *Cookbook*, a collection of useful recipes in pandas (and that we want contributions!).

There are several libraries that are now *Recommended Dependencies*

Selection Choices

Starting in 0.11.0, object selection has had a number of user-requested additions in order to support more explicit location based indexing. Pandas now supports three types of multi-axis indexing.

- `.loc` is strictly label based, will raise `KeyError` when the items are not found, allowed inputs are:
  - A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index. This use is **not** an integer position along the index)
  - A list or array of labels `['a', 'b', 'c']`
  - A slice object with labels `'a':'f'`, (note that contrary to usual python slices, **both** the start and the stop are included!)
  - A boolean array

  See more at *Selection by Label*
- `.iloc` is strictly integer position based (from `0` to `length-1` of the axis), will raise `IndexError` when the requested indices are out of bounds. Allowed inputs are:
  - An integer e.g. `5`
  - A list or array of integers `[4, 3, 0]`
  - A slice object with ints `1:7`
  - A boolean array

  See more at *Selection by Position*
- `.ix` supports mixed integer and label based access. It is primarily label based, but will fallback to integer positional access. `.ix` is the most general and will support any of the inputs to `.loc` and `.iloc`, as well as support for floating point label schemes. `.ix` is especially useful when dealing with mixed positional and label based hierarchical indexes.

  As using integer slices with `.ix` have different behavior depending on whether the slice is interpreted as position based or label based, it's usually better to be explicit and use `.iloc` or `.loc`.

  See more at *Advanced Indexing* and *Advanced Hierarchical*.

Selection Deprecations

Starting in version 0.11.0, these methods *may* be deprecated in future versions.

- `irow`
- `icol`
- `iget_value`

See the section *Selection by Position* for substitutes.

Dtypes

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [1]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [2]: df1
Out[2]:
          A
0  0.469112
1 -0.282863
2 -1.509058
3 -1.135632
4  1.212112
5 -0.173215
6  0.119209
7 -1.044236

In [3]: df1.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
dtype: object

In [4]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8), dtype='float16'),
   ...:                     'B': pd.Series(np.random.randn(8)),
   ...:                     'C': pd.Series(range(8), dtype='uint8')})
   ...:

In [5]: df2
Out[5]:
          A         B  C
0 -0.861816 -0.424972  0
1 -2.105469  0.567020  1
2 -0.494873  0.276232  2
3  1.072266 -1.087401  3
4  0.721680 -0.673690  4
5 -0.706543  0.113648  5
6 -1.040039 -1.478427  6
7  0.271973  0.524988  7

In [6]: df2.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
A    float16
B    float64
C      uint8
dtype: object

# here you get some upcasting
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [8]: df3
Out[8]:
          A         B    C
0 -0.392704 -0.424972  0.0
1 -2.388332  0.567020  1.0
2 -2.003932  0.276232  2.0
3 -0.063367 -1.087401  3.0
4  1.933792 -0.673690  4.0
5 -0.879758  0.113648  5.0
6 -0.920830 -1.478427  6.0
7 -0.772263  0.524988  7.0

In [9]: df3.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float64
C    float64
dtype: object
```

Dtype Conversion

This is lower-common-denominator upcasting, meaning you get the dtype which can accommodate all of the types

```
In [10]: df3.values.dtype
Out[10]: dtype('float64')
```

Conversion

```
In [11]: df3.astype('float32').dtypes
Out[11]:
A    float32
B    float32
C    float32
dtype: object
```

Mixed Conversion

```
In [12]: df3['D'] = '1.'

In [13]: df3['E'] = '1'

In [14]: df3.convert_objects(convert_numeric=True).dtypes
Out[14]:
A    float32
B    float64
C    float64
```

(continues on next page)

(continued from previous page)

```
D    float64
E      int64
dtype: object

# same, but specific dtype conversion
In [15]: df3['D'] = df3['D'].astype('float16')

In [16]: df3['E'] = df3['E'].astype('int32')

In [17]: df3.dtypes
Out[17]:
A    float32
B    float64
C    float64
D    float16
E      int32
dtype: object
```

Forcing Date coercion (and setting `NaT` when not datelike)

```
In [18]: import datetime

In [19]: s = pd.Series([datetime.datetime(2001, 1, 1, 0, 0), 'foo', 1.0, 1,
   ....:                 pd.Timestamp('20010104'), '20010105'], dtype='O')
   ....:

In [20]: s.convert_objects(convert_dates='coerce')
Out[20]:
0   2001-01-01
1          NaT
2          NaT
3          NaT
4   2001-01-04
5   2001-01-05
dtype: datetime64[ns]
```

Dtype Gotchas

Platform Gotchas

Starting in 0.11.0, construction of DataFrame/Series will use default dtypes of `int64` and `float64`, *regardless of platform*. This is not an apparent change from earlier versions of pandas. If you specify dtypes, they *WILL* be respected, however (GH2837)

The following will all result in `int64` dtypes

```
In [21]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[21]:
a    int64
dtype: object

In [22]: pd.DataFrame({'a': [1, 2]}).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
a    int64
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [23]: pd.DataFrame({'a': 1}, index=range(2)).dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:
a    int64
dtype: object
```

Keep in mind that `DataFrame(np.array([1,2]))` **WILL** result in `int32` on 32-bit platforms!

Upcasting Gotchas

Performing indexing operations on integer type data can easily upcast the data. The dtype of the input data will be preserved in cases where `nans` are not introduced.

```
In [24]: dfi = df3.astype('int32')

In [25]: dfi['D'] = dfi['D'].astype('int64')

In [26]: dfi
Out[26]:
   A  B  C  D  E
0  0  0  0  1  1
1 -2  0  1  1  1
2 -2  0  2  1  1
3  0 -1  3  1  1
4  1  0  4  1  1
5  0  0  5  1  1
6  0 -1  6  1  1
7  0  0  7  1  1

In [27]: dfi.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    int32
B    int32
C    int32
D    int64
E    int32
dtype: object

In [28]: casted = dfi[dfi > 0]

In [29]: casted
Out[29]:
     A   B    C  D  E
0  NaN NaN  NaN  1  1
1  NaN NaN  1.0  1  1
2  NaN NaN  2.0  1  1
3  NaN NaN  3.0  1  1
4  1.0 NaN  4.0  1  1
5  NaN NaN  5.0  1  1
6  NaN NaN  6.0  1  1
7  NaN NaN  7.0  1  1

In [30]: casted.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float64
B    float64
```

(continues on next page)

(continued from previous page)

```
C    float64
D      int64
E      int32
dtype: object
```

While float dtypes are unchanged.

```
In [31]: df4 = df3.copy()

In [32]: df4['A'] = df4['A'].astype('float32')

In [33]: df4.dtypes
Out[33]:
A    float32
B    float64
C    float64
D    float16
E      int32
dtype: object

In [34]: casted = df4[df4 > 0]

In [35]: casted
Out[35]:
          A         B    C    D  E
0       NaN       NaN  NaN  1.0  1
1       NaN  0.567020  1.0  1.0  1
2       NaN  0.276232  2.0  1.0  1
3       NaN       NaN  3.0  1.0  1
4  1.933792       NaN  4.0  1.0  1
5       NaN  0.113648  5.0  1.0  1
6       NaN       NaN  6.0  1.0  1
7       NaN  0.524988  7.0  1.0  1

In [36]: casted.dtypes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A    float32
B    float64
C    float64
D    float16
E      int32
dtype: object
```

Datetimes Conversion

Datetime64[ns] columns in a DataFrame (or a Series) allow the use of `np.nan` to indicate a nan value, in addition to the traditional `NaT`, or not-a-time. This allows convenient nan setting in a generic way. Furthermore `datetime64[ns]` columns are created by default, when passed datetimelike objects (*this change was introduced in 0.10.1*) (GH2809, GH2810)

```
In [37]: df = pd.DataFrame(np.random.randn(6, 2), pd.date_range('20010102',␣
↪periods=6),
   ....:                   columns=['A', ' B'])
   ....:
```

(continues on next page)

(continued from previous page)

```
In [38]: df['timestamp'] = pd.Timestamp('20010103')

In [39]: df
Out[39]:
                   A         B  timestamp
2001-01-02  0.404705  0.577046 2001-01-03
2001-01-03 -1.715002 -1.039268 2001-01-03
2001-01-04 -0.370647 -1.157892 2001-01-03
2001-01-05 -1.344312  0.844885 2001-01-03
2001-01-06  1.075770 -0.109050 2001-01-03
2001-01-07  1.643563 -1.469388 2001-01-03

# datetime64[ns] out of the box
In [40]: df.get_dtype_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
float64           2
datetime64[ns]    1
dtype: int64

# use the traditional nan, which is mapped to NaT internally
In [41]: df.loc[df.index[2:4], ['A', 'timestamp']] = np.nan

In [42]: df
Out[42]:
                   A         B  timestamp
2001-01-02  0.404705  0.577046 2001-01-03
2001-01-03 -1.715002 -1.039268 2001-01-03
2001-01-04       NaN -1.157892        NaT
2001-01-05       NaN  0.844885        NaT
2001-01-06  1.075770 -0.109050 2001-01-03
2001-01-07  1.643563 -1.469388 2001-01-03
```

Astype conversion on `datetime64[ns]` to `object`, implicitly converts `NaT` to `np.nan`

```
In [43]: s = pd.Series([datetime.datetime(2001, 1, 2, 0, 0) for i in range(3)])

In [44]: s.dtype
Out[44]: dtype('<M8[ns]')

In [45]: s[1] = np.nan

In [46]: s
Out[46]:
0   2001-01-02
1          NaT
2   2001-01-02
dtype: datetime64[ns]

In [47]: s.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]:␣
↪dtype('<M8[ns]')

In [48]: s = s.astype('O')

In [49]: s
```

(continues on next page)

(continued from previous page)

```
Out[49]:
0    2001-01-02 00:00:00
1                    NaT
2    2001-01-02 00:00:00
dtype: object

In [50]: s.dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪dtype('O')
```

API changes

- Added to_series() method to indices, to facilitate the creation of indexers (GH3275)
- HDFStore
  - added the method select_column to select a single column from a table as a Series.
  - deprecated the unique method, can be replicated by select_column(key,column).unique()
  - min_itemsize parameter to append will now automatically create data_columns for passed keys

Enhancements

- Improved performance of df.to_csv() by up to 10x in some cases. (GH3059)
- Numexpr is now a *Recommended Dependencies*, to accelerate certain types of numerical and boolean operations
- Bottleneck is now a *Recommended Dependencies*, to accelerate certain types of nan operations
- HDFStore
  - support read_hdf/to_hdf API similar to read_csv/to_csv

```
In [51]: df = pd.DataFrame({'A': lrange(5), 'B': lrange(5)})

In [52]: df.to_hdf('store.h5', 'table', append=True)

In [53]: pd.read_hdf('store.h5', 'table', where=['index > 2'])
Out[53]:
   A  B
3  3  3
4  4  4
```

  - provide dotted attribute access to get from stores, e.g. store.df == store['df']
  - new keywords iterator=boolean, and chunksize=number_in_a_chunk are provided to support iteration on select and select_as_multiple (GH3076)
- You can now select timestamps from an *unordered* timeseries similarly to an *ordered* timeseries (GH2437)
- You can now select with a string from a DataFrame with a datelike index, in a similar way to a Series (GH3070)

```
In [54]: idx = pd.date_range("2001-10-1", periods=5, freq='M')

In [55]: ts = pd.Series(np.random.rand(len(idx)), index=idx)

In [56]: ts['2001']
```

(continues on next page)

(continued from previous page)

```
Out[56]:
2001-10-31    0.117967
2001-11-30    0.702184
2001-12-31    0.414034
Freq: M, dtype: float64

In [57]: df = pd.DataFrame({'A': ts})

In [58]: df['2001']
Out[58]:
                   A
2001-10-31  0.117967
2001-11-30  0.702184
2001-12-31  0.414034
```

- `Squeeze` to possibly remove length 1 dimensions from an object.

```
In [59]: p = pd.Panel(np.random.randn(3, 4, 4), items=['ItemA', 'ItemB', 'ItemC'],
   ....:              major_axis=pd.date_range('20010102', periods=4),
   ....:              minor_axis=['A', 'B', 'C', 'D'])
   ....:

In [60]: p
Out[60]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D

In [61]: p.reindex(items=['ItemA']).squeeze()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D
2001-01-02 -0.919854 -0.042379  1.247642 -0.009920
2001-01-03  0.290213  0.495767  0.362949  1.548106
2001-01-04 -1.131345 -0.089329  0.337863 -0.945867
2001-01-05 -0.932132  1.956030  0.017587 -0.016692

In [62]: p.reindex(items=['ItemA'], minor=['B']).squeeze()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
2001-01-02   -0.042379
2001-01-03    0.495767
2001-01-04   -0.089329
2001-01-05    1.956030
Freq: D, Name: B, dtype: float64
```

- In `pd.io.data.Options`,
  - Fix bug when trying to fetch data for the current month when already past expiry.
  - Now using lxml to scrape html instead of BeautifulSoup (lxml was faster).
  - New instance variables for calls and puts are automatically created when a method that creates them is called. This works for current month where the instance variables are simply `calls` and `puts`. Also works for future expiry months and save the instance variable as `callsMMYY` or `putsMMYY`, where `MMYY` are, respectively, the month and year of the option's expiry.

  - `Options.get_near_stock_price` now allows the user to specify the month for which to get relevant options data.
  - `Options.get_forward_data` now has optional kwargs `near` and `above_below`. This allows the user to specify if they would like to only return forward looking data for options near the current stock price. This just obtains the data from Options.get_near_stock_price instead of Options.get_xxx_data() (GH2758).
- Cursor coordinate information is now displayed in time-series plots.
- added option *display.max_seq_items* to control the number of elements printed per sequence pprinting it. (GH2979)
- added option *display.chop_threshold* to control display of small numerical values. (GH2739)
- added option *display.max_info_rows* to prevent verbose_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- value_counts() now accepts a "normalize" argument, for normalized histograms. (GH2710).
- DataFrame.from_records now accepts not only dicts but any instance of the collections.Mapping ABC.
- added option *display.mpl_style* providing a sleeker visual style for plots. Based on https://gist.github.com/huyng/816622 (GH3075).
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- to_html() now accepts an optional "escape" argument to control reserved HTML character escaping (enabled by default) and escapes `&`, in addition to < and >. (GH2919)

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 50 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Greenhall +
- Alvaro Tejero-Cantero +
- Andy Hayden
- Brad Buran +
- Chang She
- Chapman Siu +
- Chris Withers +
- Christian Geier +
- Christopher Whelan
- Damien Garaud
- Dan Birken
- Dan Davison +
- Dieter Vandenbussche
- Drazen Lucanin +
- Dražen Lučanin +

- Garrett Drapala
- Illia Polosukhin +
- James Casbon +
- Jeff Reback
- Jeremy Wagner +
- Jonathan Chambers +
- K.-Michael Aye
- Karmel Allison +
- Loïc Estève +
- Nicholaus E. Halecky +
- Peter Prettenhofer +
- Phillip Cloud +
- Robert Gieseke +
- Skipper Seabold
- Spencer Lyon
- Stephen Lin +
- Thierry Moisan +
- Thomas Kluyver
- Tim Akinbo +
- Vytautas Jancauskas
- Vytautas Jančauskas +
- Wes McKinney
- Will Furnass +
- Wouter Overmeire
- anomrake +
- davidjameshumphreys +
- dengemann +
- dieterv77 +
- jreback
- lexual +
- stephenwlin +
- thauck +
- vytas +
- waitingkuo +
- y-p

8.15 Version 0.10

8.15.1 v0.10.1 (January 22, 2013)

This is a minor release from 0.10.0 and includes new features, enhancements, and bug fixes. In particular, there is substantial new HDFStore functionality contributed by Jeff Reback.

An undesired API breakage with functions taking the `inplace` option has been reverted and deprecation warnings added.

API changes

- Functions taking an `inplace` option return the calling object as before. A deprecation message has been added
- Groupby aggregations Max/Min no longer exclude non-numeric data (GH2700)
- Resampling an empty DataFrame now returns an empty DataFrame instead of raising an exception (GH2640)
- The file reader will now raise an exception when NA values are found in an explicitly specified integer column instead of converting the column to float (GH2631)
- DatetimeIndex.unique now returns a DatetimeIndex with the same name and
- timezone instead of an array (GH2563)

New features

- MySQL support for database (contribution from Dan Allan)

HDFStore

You may need to upgrade your existing data files. Please visit the **compatibility** section in the main docs.

You can designate (and index) certain columns that you want to be able to perform queries on a table, by passing a list to `data_columns`

```
In [1]: store = pd.HDFStore('store.h5')

In [2]: df = pd.DataFrame(np.random.randn(8, 3),
   ...:                   index=pd.date_range('1/1/2000', periods=8),
   ...:                   columns=['A', 'B', 'C'])
   ...:

In [3]: df['string'] = 'foo'

In [4]: df.loc[df.index[4:6], 'string'] = np.nan

In [5]: df.loc[df.index[7:9], 'string'] = 'bar'

In [6]: df['string2'] = 'cool'

In [7]: df
Out[7]:
                   A         B         C string string2
2000-01-01  0.469112 -0.282863 -1.509059    foo    cool
```

(continues on next page)

(continued from previous page)

```
2000-01-02 -1.135632  1.212112 -0.173215    foo    cool
2000-01-03  0.119209 -1.044236 -0.861849    foo    cool
2000-01-04 -2.104569 -0.494929  1.071804    foo    cool
2000-01-05  0.721555 -0.706771 -1.039575    NaN    cool
2000-01-06  0.271860 -0.424972  0.567020    NaN    cool
2000-01-07  0.276232 -1.087401 -0.673690    foo    cool
2000-01-08  0.113648 -1.478427  0.524988    bar    cool

# on-disk operations
In [8]: store.append('df', df, data_columns=['B', 'C', 'string', 'string2'])

In [9]: store.select('df', "B>0 and string=='foo'")
Out[9]:
                   A         B         C string string2
2000-01-02 -1.135632  1.212112 -0.173215    foo    cool

# this is in-memory version of this type of selection
In [10]: df[(df.B > 0) & (df.string == 'foo')]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C string string2
2000-01-02 -1.135632  1.212112 -0.173215    foo    cool
```

Retrieving unique values in an indexable or data column.

```
# note that this is deprecated as of 0.14.0
# can be replicated by: store.select_column('df','index').unique()
store.unique('df', 'index')
store.unique('df', 'string')
```

You can now store `datetime64` in data columns

```
In [11]: df_mixed = df.copy()

In [12]: df_mixed['datetime64'] = pd.Timestamp('20010102')

In [13]: df_mixed.loc[df_mixed.index[3:4], ['A', 'B']] = np.nan

In [14]: store.append('df_mixed', df_mixed)

In [15]: df_mixed1 = store.select('df_mixed')

In [16]: df_mixed1
Out[16]:
                   A         B         C string string2 datetime64
2000-01-01  0.469112 -0.282863 -1.509059    foo    cool 2001-01-02
2000-01-02 -1.135632  1.212112 -0.173215    foo    cool 2001-01-02
2000-01-03  0.119209 -1.044236 -0.861849    foo    cool 2001-01-02
2000-01-04       NaN       NaN  1.071804    foo    cool 2001-01-02
2000-01-05  0.721555 -0.706771 -1.039575    NaN    cool 2001-01-02
2000-01-06  0.271860 -0.424972  0.567020    NaN    cool 2001-01-02
2000-01-07  0.276232 -1.087401 -0.673690    foo    cool 2001-01-02
2000-01-08  0.113648 -1.478427  0.524988    bar    cool 2001-01-02

In [17]: df_mixed1.get_dtype_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
float64           3
object            2
datetime64[ns]    1
dtype: int64
```

You can pass `columns` keyword to select to filter a list of the return columns, this is equivalent to passing a `Term('columns',list_of_columns_to_filter)`

```
In [18]: store.select('df', columns=['A', 'B'])
Out[18]: 
                   A         B
2000-01-01  0.469112 -0.282863
2000-01-02 -1.135632  1.212112
2000-01-03  0.119209 -1.044236
2000-01-04 -2.104569 -0.494929
2000-01-05  0.721555 -0.706771
2000-01-06  0.271860 -0.424972
2000-01-07  0.276232 -1.087401
2000-01-08  0.113648 -1.478427
```

`HDFStore` now serializes MultiIndex dataframes when appending tables.

```
In [19]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   ....:                               ['one', 'two', 'three']],
   ....:                       labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   ....:                               [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   ....:                       names=['foo', 'bar'])
   ....: 

In [20]: df = pd.DataFrame(np.random.randn(10, 3), index=index,
   ....:                   columns=['A', 'B', 'C'])
   ....: 

In [21]: df
Out[21]: 
                  A         B         C
foo bar                                
foo one   -0.116619  0.295575 -1.047704
    two    1.640556  1.905836  2.772115
    three  0.088787 -1.144197 -0.633372
bar one    0.925372 -0.006438 -0.820408
    two   -0.600874 -1.039266  0.824758
baz two   -0.824095 -0.337730 -0.927764
    three -0.840123  0.248505 -0.109250
qux one    0.431977 -0.460710  0.336505
    two   -3.207595 -1.535854  0.409769
    three -0.673145 -0.741113 -0.110891

In [22]: store.append('mi', df)

In [23]: store.select('mi')
Out[23]: 
                  A         B         C
foo bar                                
foo one   -0.116619  0.295575 -1.047704
    two    1.640556  1.905836  2.772115
```

(continues on next page)

(continued from previous page)

```
    three  0.088787 -1.144197 -0.633372
bar one    0.925372 -0.006438 -0.820408
    two   -0.600874 -1.039266  0.824758
baz two   -0.824095 -0.337730 -0.927764
    three -0.840123  0.248505 -0.109250
qux one    0.431977 -0.460710  0.336505
    two   -3.207595 -1.535854  0.409769
    three -0.673145 -0.741113 -0.110891

# the levels are automatically included as data columns
In [24]: store.select('mi', "foo='bar'")
Out[24]:
                A         B         C
foo bar
bar one  0.925372 -0.006438 -0.820408
    two -0.600874 -1.039266  0.824758
```

Multi-table creation via `append_to_multiple` and selection via `select_as_multiple` can create/select from multiple tables and return a combined result, by using `where` on a selector table.

```
In [19]: df_mt = pd.DataFrame(np.random.randn(8, 6),
   ....:                      index=pd.date_range('1/1/2000', periods=8),
   ....:                      columns=['A', 'B', 'C', 'D', 'E', 'F'])
   ....:

In [20]: df_mt['foo'] = 'bar'

# you can also create the tables individually
In [21]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None},
   ....:                          df_mt, selector='df1_mt')
   ....:

In [22]: store
Out[22]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# indiviual tables were created
In [23]: store.select('df1_mt')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:
                   A         B
2000-01-01  0.404705  0.577046
2000-01-02 -1.344312  0.844885
2000-01-03  0.357021 -0.674600
2000-01-04  0.276662 -0.472035
2000-01-05  0.895717  0.805244
2000-01-06 -1.170299 -0.226169
2000-01-07 -0.076467 -1.187678
2000-01-08  1.024180  0.569605

In [24]: store.select('df2_mt')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   C         D         E         F  foo
2000-01-01 -1.715002 -1.039268 -0.370647 -1.157892  bar
2000-01-02  1.075770 -0.109050  1.643563 -1.469388  bar
```

(continues on next page)

(continued from previous page)

```
2000-01-03 -1.776904 -0.968914 -1.294524  0.413738  bar
2000-01-04 -0.013960 -0.362543 -0.006154 -0.923061  bar
2000-01-05 -1.206412  2.565646  1.431256  1.340309  bar
2000-01-06  0.410835  0.813850  0.132003 -0.827317  bar
2000-01-07  1.130127 -1.436737 -1.413681  1.607920  bar
2000-01-08  0.875906 -2.211372  0.974466 -2.006747  bar

# as a multiple
In [25]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   ....:                          selector='df1_mt')
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
                   A         B         C         D         E         F  foo
2000-01-01  0.404705  0.577046 -1.715002 -1.039268 -0.370647 -1.157892  bar
2000-01-05  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309  bar
2000-01-08  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747  bar
```

Enhancements

- `HDFStore` now can read native PyTables table format tables
- You can pass `nan_rep = 'my_nan_rep'` to append, to change the default nan representation on disk (which converts to/from *np.nan*), this defaults to *nan*.
- You can pass `index` to `append`. This defaults to `True`. This will automagically create indices on the *indexables* and *data columns* of the table
- You can pass `chunksize=an integer` to `append`, to change the writing chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=an integer` to the first `append`, to set the TOTAL number of expected rows that `PyTables` will expected. This will optimize read/write performance.
- `Select` now supports passing `start` and `stop` to provide selection space limiting in selection.
- Greatly improved ISO8601 (e.g., yyyy-mm-dd) date parsing for file parsers (GH2698)
- Allow `DataFrame.merge` to handle combinatorial sizes too large for 64-bit integer (GH2690)
- Series now has unary negation (-series) and inversion (~series) operators (GH2686)
- DataFrame.plot now includes a `logx` parameter to change the x-axis to log scale (GH2327)
- Series arithmetic operators can now handle constant and ndarray input (GH2574)
- ExcelFile now takes a `kind` argument to specify the file type (GH2613)
- A faster implementation for Series.str methods (GH2602)

Bug Fixes

- `HDFStore` tables can now store `float32` types correctly (cannot be mixed with `float64` however)
- Fixed Google Analytics prefix when specifying request segment (GH2713).
- Function to reset Google Analytics token store so users can recover from improperly setup client secrets (GH2687).
- Fixed groupby bug resulting in segfault when passing in MultiIndex (GH2706)
- Fixed bug where passing a Series with datetime64 values into *to_datetime* results in bogus output values (GH2699)

- Fixed bug in `pattern in HDFStore` expressions when pattern is not a valid regex (GH2694)
- Fixed performance issues while aggregating boolean data (GH2692)
- When given a boolean mask key and a Series of new values, Series __setitem__ will now align the incoming values with the original Series (GH2686)
- Fixed MemoryError caused by performing counting sort on sorting MultiIndex levels with a very large number of combinatorial values (GH2684)
- Fixed bug that causes plotting to fail when the index is a DatetimeIndex with a fixed-offset timezone (GH2683)
- Corrected business day subtraction logic when the offset is more than 5 bdays and the starting date is on a weekend (GH2680)
- Fixed C file parser behavior when the file has more columns than data (GH2668)
- Fixed file reader bug that misaligned columns with data in the presence of an implicit column and a specified *usecols* value
- DataFrames with numerical or datetime indices are now sorted prior to plotting (GH2609)
- Fixed DataFrame.from_records error when passed columns, index, but empty records (GH2633)
- Several bug fixed for Series operations when dtype is datetime64 (GH2689, GH2629, GH2626)

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 17 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Andy Hayden +
- Anton I. Sipos +
- Chang She
- Christopher Whelan
- Damien Garaud +
- Dan Allan +
- Dieter Vandenbussche
- Garrett Drapala +
- Jay Parlar +
- Thouis (Ray) Jones +
- Vincent Arel-Bundock +
- Wes McKinney
- elpres
- herrfz +
- jreback
- svaksha +
- y-p

8.15.2 v0.10.0 (December 17, 2012)

This is a major release from 0.9.1 and includes many new features and enhancements along with a large number of bug fixes. There are also a number of important API changes that long-time pandas users should pay close attention to.

File parsing new features

The delimited file parsing engine (the guts of `read_csv` and `read_table`) has been rewritten from the ground up and now uses a fraction the amount of memory while parsing, while being 40% or more faster in most use cases (in some cases much faster).

There are also many new features:

- Much-improved Unicode handling via the `encoding` option.
- Column filtering (`usecols`)
- Dtype specification (`dtype` argument)
- Ability to specify strings to be recognized as True/False
- Ability to yield NumPy record arrays (`as_recarray`)
- High performance `delim_whitespace` option
- Decimal format (e.g. European format) specification
- Easier CSV dialect options: `escapechar`, `lineterminator`, `quotechar`, etc.
- More robust handling of many exceptional kinds of files observed in the wild

API changes

Deprecated DataFrame BINOP TimeSeries special case behavior

The default behavior of binary operations between a DataFrame and a Series has always been to align on the DataFrame's columns and broadcast down the rows, **except** in the special case that the DataFrame contains time series. Since there are now method for each binary operator enabling you to specify how you want to broadcast, we are phasing out this special case (Zen of Python: *Special cases aren't special enough to break the rules*). Here's what I'm talking about:

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(np.random.randn(6, 4),
   ...:                   index=pd.date_range('1/1/2000', periods=6))
   ...:

In [3]: df
Out[3]:
                   0         1         2         3
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988

# deprecated now
```

(continues on next page)

(continued from previous page)

```
In [4]: df - df[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
            2000-01-01 00:00:00  2000-01-02 00:00:00  2000-01-03 00:00:00  2000-01-04␣
↪00:00:00  2000-01-05 00:00:00  2000-01-06 00:00:00   0   1   2   3
2000-01-01                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN
2000-01-02                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN
2000-01-03                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN
2000-01-04                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN
2000-01-05                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN
2000-01-06                  NaN                  NaN                  NaN                 ␣
↪      NaN                  NaN                  NaN NaN NaN NaN NaN

# Change your code to
In [5]: df.sub(df[0], axis=0)  # align on axis 0 (rows)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
              0         1         2         3
2000-01-01  0.0 -0.751976 -1.978171 -1.604745
2000-01-02  0.0 -1.385327 -1.092903 -2.256348
2000-01-03  0.0 -1.242720  0.366920  1.933653
2000-01-04  0.0 -1.428326 -1.761130 -0.449695
2000-01-05  0.0  0.991993  0.701204 -0.662428
2000-01-06  0.0  0.787338 -0.804737  1.198677
```

You will get a deprecation warning in the 0.10.x series, and the deprecated functionality will be removed in 0.11 or later.

Altered resample default behavior

The default time series `resample` binning behavior of daily `D` and *higher* frequencies has been changed to `closed='left', label='left'`. Lower nfrequencies are unaffected. The prior defaults were causing a great deal of confusion for users, especially resampling data to daily frequency (which labeled the aggregated group with the end of the interval: the next day).

```
In [1]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')

In [2]: series = pd.Series(np.arange(len(dates)), index=dates)

In [3]: series
Out[3]:
2000-01-01 00:00:00     0
2000-01-01 04:00:00     1
2000-01-01 08:00:00     2
2000-01-01 12:00:00     3
2000-01-01 16:00:00     4
2000-01-01 20:00:00     5
2000-01-02 00:00:00     6
2000-01-02 04:00:00     7
2000-01-02 08:00:00     8
2000-01-02 12:00:00     9
2000-01-02 16:00:00    10
```

(continues on next page)

(continued from previous page)

```
2000-01-02 20:00:00    11
2000-01-03 00:00:00    12
2000-01-03 04:00:00    13
2000-01-03 08:00:00    14
2000-01-03 12:00:00    15
2000-01-03 16:00:00    16
2000-01-03 20:00:00    17
2000-01-04 00:00:00    18
2000-01-04 04:00:00    19
2000-01-04 08:00:00    20
2000-01-04 12:00:00    21
2000-01-04 16:00:00    22
2000-01-04 20:00:00    23
2000-01-05 00:00:00    24
Freq: 4H, dtype: int64

In [4]: series.resample('D', how='sum')
Out[4]:
2000-01-01     15
2000-01-02     51
2000-01-03     87
2000-01-04    123
2000-01-05     24
Freq: D, dtype: int64

In [5]: # old behavior
In [6]: series.resample('D', how='sum', closed='right', label='right')
Out[6]:
2000-01-01      0
2000-01-02     21
2000-01-03     57
2000-01-04     93
2000-01-05    129
Freq: D, dtype: int64
```

- Infinity and negative infinity are no longer treated as NA by `isnull` and `notnull`. That they ever were was a relic of early pandas. This behavior can be re-enabled globally by the `mode.use_inf_as_null` option:

```
In [6]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])

In [7]: pd.isnull(s)
Out[7]:
0    False
1    False
2    False
3    False
Length: 4, dtype: bool

In [8]: s.fillna(0)
Out[8]:
0    1.500000
1         inf
2    3.400000
3        -inf
Length: 4, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [9]: pd.set_option('use_inf_as_null', True)

In [10]: pd.isnull(s)
Out[10]:
0    False
1     True
2    False
3     True
Length: 4, dtype: bool

In [11]: s.fillna(0)
Out[11]:
0    1.5
1    0.0
2    3.4
3    0.0
Length: 4, dtype: float64

In [12]: pd.reset_option('use_inf_as_null')
```

- Methods with the `inplace` option now all return `None` instead of the calling object. E.g. code written like `df = df.fillna(0, inplace=True)` may stop working. To fix, simply delete the unnecessary variable assignment.
- `pandas.merge` no longer sorts the group keys (`sort=False`) by default. This was done for performance reasons: the group-key sorting is often one of the more expensive parts of the computation and is often unnecessary.
- The default column names for a file with no header have been changed to the integers `0` through `N - 1`. This is to create consistency with the DataFrame constructor with no columns specified. The v0.9.0 behavior (names `X0`, `X1`, ...) can be reproduced by specifying `prefix='X'`:

```
In [6]: import io

In [7]: data = ('a,b,c\n'
   ...:         '1,Yes,2\n'
   ...:         '3,No,4')
   ...:

In [8]: print(data)
a,b,c
1,Yes,2
3,No,4

In [9]: pd.read_csv(io.StringIO(data), header=None)
\\\\\\\\\\\\\\\\\\\\\Out[9]:
   0    1  2
0  a    b  c
1  1  Yes  2
2  3   No  4

In [10]: pd.read_csv(io.StringIO(data), header=None, prefix='X')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]:
↪
  X0   X1 X2
0  a    b  c
1  1  Yes  2
```

(continues on next page)

(continued from previous page)

```
2  3   No  4
```

- Values like 'Yes' and 'No' are not interpreted as boolean by default, though this can be controlled by new true_values and false_values arguments:

```
In [11]: print(data)
a,b,c
1,Yes,2
3,No,4

In [12]: pd.read_csv(io.StringIO(data))
\\\\\\\\\\\\\\\\\\\\\Out[12]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [13]: pd.read_csv(io.StringIO(data), true_values=['Yes'], false_values=['No'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]:
   a      b  c
0  1   True  2
1  3  False  4
```

- The file parsers will not recognize non-string values arising from a converter function as NA if passed in the na_values argument. It's better to do post-processing using the replace function instead.
- Calling fillna on Series or DataFrame with no arguments is no longer valid code. You must either specify a fill value or an interpolation method:

```
In [14]: s = pd.Series([np.nan, 1., 2., np.nan, 4])

In [15]: s
Out[15]:
0    NaN
1    1.0
2    2.0
3    NaN
4    4.0
dtype: float64

In [16]: s.fillna(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16]:
0    0.0
1    1.0
2    2.0
3    0.0
4    4.0
dtype: float64

In [17]: s.fillna(method='pad')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
0    NaN
1    1.0
2    2.0
3    2.0
4    4.0
dtype: float64
```

Convenience methods `ffill` and `bfill` have been added:

```
In [18]: s.ffill()
Out[18]:
0    NaN
1    1.0
2    2.0
3    2.0
4    4.0
dtype: float64
```

- `Series.apply` will now operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame

```
In [19]: def f(x):
   ....:     return pd.Series([x, x**2], index=['x', 'x^2'])
   ....:

In [20]: s = pd.Series(np.random.rand(5))

In [21]: s
Out[21]:
0    0.340445
1    0.984729
2    0.919540
3    0.037772
4    0.861549
dtype: float64

In [22]: s.apply(f)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Wide DataFrame Printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```
In [24]: wide_frame = pd.DataFrame(np.random.randn(5, 16))

In [25]: wide_frame
Out[25]:
          0         1         2         3         4         5         6         7  ␣
↪   8         9        10        11        12        13        14        15
0 -0.548702  1.467327 -1.015962 -0.483075  1.637550 -1.217659 -0.291519 -1.745505 -0.
↪263952  0.991460 -0.919069  0.266046 -0.709661  1.669052  1.037882 -1.705775
1 -0.919854 -0.042379  1.247642 -0.009920  0.290213  0.495767  0.362949  1.548106 -1.
↪131345 -0.089329  0.337863 -0.945867 -0.932132  1.956030  0.017587 -0.016692
2 -0.575247  0.254161 -1.143704  0.215897  1.193555 -0.077118 -0.408530 -0.862495  1.
↪346061  1.511763  1.627081 -0.990582 -0.441652  1.211526  0.268520  0.024580
3 -1.577585  0.396823 -0.105381 -0.532532  1.453749  1.208843 -0.080952 -0.264610 -0.
↪727965 -0.589346  0.339969 -0.693205 -0.339355  0.593616  0.884345  1.591431
4  0.141809  0.220390  0.435589  0.192451 -0.096701  0.803351  1.715071 -0.708758 -1.
↪202872 -1.814470  1.018601 -0.595447  1.395433 -0.392670  0.007207  1.928123
```

The old behavior of printing out summary information can be achieved via the 'expand_frame_repr' print option:

```
In [26]: pd.set_option('expand_frame_repr', False)

In [27]: wide_frame
Out[27]:
          0         1         2         3         4         5         6         7  ␣
↪   8         9        10        11        12        13        14        15
0 -0.548702  1.467327 -1.015962 -0.483075  1.637550 -1.217659 -0.291519 -1.745505 -0.
↪263952  0.991460 -0.919069  0.266046 -0.709661  1.669052  1.037882 -1.705775
1 -0.919854 -0.042379  1.247642 -0.009920  0.290213  0.495767  0.362949  1.548106 -1.
↪131345 -0.089329  0.337863 -0.945867 -0.932132  1.956030  0.017587 -0.016692
2 -0.575247  0.254161 -1.143704  0.215897  1.193555 -0.077118 -0.408530 -0.862495  1.
↪346061  1.511763  1.627081 -0.990582 -0.441652  1.211526  0.268520  0.024580
3 -1.577585  0.396823 -0.105381 -0.532532  1.453749  1.208843 -0.080952 -0.264610 -0.
↪727965 -0.589346  0.339969 -0.693205 -0.339355  0.593616  0.884345  1.591431
4  0.141809  0.220390  0.435589  0.192451 -0.096701  0.803351  1.715071 -0.708758 -1.
↪202872 -1.814470  1.018601 -0.595447  1.395433 -0.392670  0.007207  1.928123
```

The width of each line can be changed via 'line_width' (80 by default):

```
pd.set_option('line_width', 40)

wide_frame
```

Updated PyTables Support

Docs for PyTables `Table` format & several enhancements to the api. Here is a taste of what to expect.

```
In [28]: store = pd.HDFStore('store.h5')

In [29]: df = pd.DataFrame(np.random.randn(8, 3),
   ....:                   index=pd.date_range('1/1/2000', periods=8),
   ....:                   columns=['A', 'B', 'C'])
```

(continues on next page)

(continued from previous page)

```
   ....:

In [30]: df
Out[30]:
                   A         B         C
2000-01-01 -0.055224  2.395985  1.552825
2000-01-02  0.166599  0.047609 -0.136473
2000-01-03 -0.561757 -1.623033  0.029399
2000-01-04 -0.542108  0.282696 -0.087302
2000-01-05 -1.575170  1.771208  0.816482
2000-01-06  1.100230 -0.612665  1.586976
2000-01-07  0.019234  0.264294  1.074803
2000-01-08  0.173520  0.211027  1.357138

# appending data frames
In [31]: df1 = df[0:4]

In [32]: df2 = df[4:]

In [33]: store.append('df', df1)

In [34]: store.append('df', df2)

In [35]: store
Out[35]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# selecting the entire store
In [36]: store.select('df')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[36]:
                   A         B         C
2000-01-01 -0.055224  2.395985  1.552825
2000-01-02  0.166599  0.047609 -0.136473
2000-01-03 -0.561757 -1.623033  0.029399
2000-01-04 -0.542108  0.282696 -0.087302
2000-01-05 -1.575170  1.771208  0.816482
2000-01-06  1.100230 -0.612665  1.586976
2000-01-07  0.019234  0.264294  1.074803
2000-01-08  0.173520  0.211027  1.357138
```

```
In [37]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   ....:               major_axis=pd.date_range('1/1/2000', periods=5),
   ....:               minor_axis=['A', 'B', 'C', 'D'])
   ....:

In [38]: wp
Out[38]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D

# storing a panel
In [39]: store.append('wp', wp)
```

(continues on next page)

(continued from previous page)

```
# selecting via A QUERY
In [40]: store.select('wp', "major_axis>20000102 and minor_axis=['A','B']")
Out[40]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B

# removing data from tables
In [41]: store.remove('wp', "major_axis>20000103")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪8

In [42]: store.select('wp')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to D

# deleting a store
In [43]: del store['df']

In [44]: store
Out[44]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Enhancements

- added ability to hierarchical keys

```
In [45]: store.put('foo/bar/bah', df)

In [46]: store.append('food/orange', df)

In [47]: store.append('food/apple',  df)

In [48]: store
Out[48]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# remove all nodes under this level
In [49]: store.remove('food')

In [50]: store
Out[50]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

- added mixed-dtype support!

```
In [51]: df['string'] = 'string'

In [52]: df['int'] = 1

In [53]: store.append('df', df)

In [54]: df1 = store.select('df')

In [55]: df1
Out[55]:
                   A         B         C  string  int
2000-01-01 -0.055224  2.395985  1.552825  string    1
2000-01-02  0.166599  0.047609 -0.136473  string    1
2000-01-03 -0.561757 -1.623033  0.029399  string    1
2000-01-04 -0.542108  0.282696 -0.087302  string    1
2000-01-05 -1.575170  1.771208  0.816482  string    1
2000-01-06  1.100230 -0.612665  1.586976  string    1
2000-01-07  0.019234  0.264294  1.074803  string    1
2000-01-08  0.173520  0.211027  1.357138  string    1

In [56]: df1.get_dtype_counts()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
float64    3
object     1
int64      1
dtype: int64
```

- performance improvements on table writing
- support for arbitrarly indexed dimensions
- `SparseSeries` now has a `density` property (GH2384)
- enable `Series.str.strip/lstrip/rstrip` methods to take an input argument to strip arbitrary characters (GH2411)
- implement `value_vars` in `melt` to limit values to certain columns and add `melt` to pandas namespace (GH2412)

Bug Fixes

- added `Term` method of specifying where conditions (GH1996).
- `del store['df']` now call `store.remove('df')` for store deletion
- deleting of consecutive rows is much faster than before
- `min_itemsize` parameter can be specified in table creation to force a minimum size for indexing columns (the previous implementation would set the column size based on the first append)
- indexing support via `create_table_index` (requires PyTables >= 2.3) (GH698).
- appending on a store would fail if the table was not first created via `put`
- fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- minor change to select and remove: require a table ONLY if where is also provided (and not None)

Compatibility

0.10 of `HDFStore` is backwards compatible for reading tables created in a prior version of pandas, however, query terms using the prior (undocumented) methodology are unsupported. You must read in the entire file and write it out using the new format to take advantage of the updates.

N Dimensional Panels (Experimental)

Adding experimental support for Panel4D and factory functions to create n-dimensional named panels. Here is a taste of what to expect.

```
In [58]: p4d = Panel4D(randn(2, 2, 5, 4),
   ....:       labels=['Label1','Label2'],
   ....:       items=['Item1', 'Item2'],
   ....:       major_axis=date_range('1/1/2000', periods=5),
   ....:       minor_axis=['A', 'B', 'C', 'D'])
   ....:

In [59]: p4d
Out[59]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 26 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- A. Flaxman +
- Abraham Flaxman
- Adam Obeng +
- Brenda Moon +
- Chang She
- Chris Mulligan +
- Dieter Vandenbussche
- Donald Curtis +
- Jay Bourque +
- Jeff Reback +
- Justin C Johnson +
- K.-Michael Aye
- Keith Hughitt +
- Ken Van Haren +
- Laurent Gautier +
- Luke Lee +

- Martin Blais
- Tobias Brandt +
- Wes McKinney
- Wouter Overmeire
- alex arsenovic +
- jreback +
- locojaydev +
- timmie
- y-p
- zach powers +

8.16 Version 0.9

8.16.1 v0.9.1 (November 14, 2012)

This is a bug fix release from 0.9.0 and includes several new features and enhancements along with a large number of bug fixes. The new features include by-column sort order for DataFrame and Series, improved NA handling for the rank method, masking functions for DataFrame, and intraday time-series filtering for DataFrame.

New features

- *Series.sort*, *DataFrame.sort*, and *DataFrame.sort_index* can now be specified in a per-column manner to support multiple sort orders (GH928)

```
In [2]: df = pd.DataFrame(np.random.randint(0, 2, (6, 3)),
   ...:                   columns=['A', 'B', 'C'])

In [3]: df.sort(['A', 'B'], ascending=[1, 0])

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

- *DataFrame.rank* now supports additional argument values for the *na_option* parameter so missing values can be assigned either the largest or the smallest rank (GH1508, GH2159)

```
In [1]: df = pd.DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

In [2]: df.loc[2:4] = np.nan

In [3]: df.rank()
Out[3]:
     A    B    C
```

(continues on next page)

(continued from previous page)

```
0  3.0  2.0  1.0
1  1.0  3.0  2.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  NaN  NaN  NaN
5  2.0  1.0  3.0

[6 rows x 3 columns]

In [4]: df.rank(na_option='top')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A    B    C
0  6.0  5.0  4.0
1  4.0  6.0  5.0
2  2.0  2.0  2.0
3  2.0  2.0  2.0
4  2.0  2.0  2.0
5  5.0  4.0  6.0

[6 rows x 3 columns]

In [5]: df.rank(na_option='bottom')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
     A    B    C
0  3.0  2.0  1.0
1  1.0  3.0  2.0
2  5.0  5.0  5.0
3  5.0  5.0  5.0
4  5.0  5.0  5.0
5  2.0  1.0  3.0

[6 rows x 3 columns]
```

- DataFrame has new *where* and *mask* methods to select values according to a given boolean mask (GH2109, GH2151)

  DataFrame currently supports slicing via a boolean vector the same length as the DataFrame (inside the *[]*). The returned DataFrame has the same number of columns as the original, but is sliced on its index.

```
In [6]: df = DataFrame(np.random.randn(5, 3), columns = ['A','B','C'])

In [7]: df
Out[7]:
          A         B         C
0  0.276232 -1.087401 -0.673690
1  0.113648 -1.478427  0.524988
2  0.404705  0.577046 -1.715002
3 -1.039268 -0.370647 -1.157892
4 -1.344312  0.844885  1.075770

[5 rows x 3 columns]

In [8]: df[df['A'] > 0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

(continues on next page)

(continued from previous page)

```
          A         B         C
0  0.276232 -1.087401 -0.673690
1  0.113648 -1.478427  0.524988
2  0.404705  0.577046 -1.715002

[3 rows x 3 columns]
```

If a DataFrame is sliced with a DataFrame based boolean condition (with the same size as the original DataFrame), then a DataFrame the same size (index and columns) as the original is returned, with elements that do not meet the boolean condition as *NaN*. This is accomplished via the new method *DataFrame.where*. In addition, *where* takes an optional *other* argument for replacement.

```
In [9]: df[df>0]
Out[9]:
          A         B         C
0  0.276232       NaN       NaN
1  0.113648       NaN  0.524988
2  0.404705  0.577046       NaN
3       NaN       NaN       NaN
4       NaN  0.844885  1.075770

[5 rows x 3 columns]

In [10]: df.where(df>0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C
0  0.276232       NaN       NaN
1  0.113648       NaN  0.524988
2  0.404705  0.577046       NaN
3       NaN       NaN       NaN
4       NaN  0.844885  1.075770

[5 rows x 3 columns]

In [11]: df.where(df>0,-df)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
          A         B         C
0  0.276232  1.087401  0.673690
1  0.113648  1.478427  0.524988
2  0.404705  0.577046  1.715002
3  1.039268  0.370647  1.157892
4  1.344312  0.844885  1.075770

[5 rows x 3 columns]
```

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

```
In [12]: df2 = df.copy()

In [13]: df2[ df2[1:4] > 0 ] = 3

In [14]: df2
```

(continues on next page)

(continued from previous page)

```
Out[14]:
          A         B         C
0  0.276232 -1.087401 -0.673690
1  3.000000 -1.478427  3.000000
2  3.000000  3.000000 -1.715002
3 -1.039268 -0.370647 -1.157892
4 -1.344312  0.844885  1.075770

[5 rows x 3 columns]
```

DataFrame.mask is the inverse boolean operation of *where*.

```
In [15]: df.mask(df<=0)
Out[15]:
          A         B         C
0  0.276232       NaN       NaN
1  0.113648       NaN  0.524988
2  0.404705  0.577046       NaN
3       NaN       NaN       NaN
4       NaN  0.844885  1.075770

[5 rows x 3 columns]
```

- Enable referencing of Excel columns by their column names (GH1936)

```
In [16]: xl = pd.ExcelFile('data/test.xls')

In [17]: xl.parse('Sheet1', index_col=0, parse_dates=True,
   ....:          parse_cols='A:D')
   ....:
Out[17]:
                   A         B         C         D
2000-01-03  0.980269  3.685731 -0.364217 -1.159738
2000-01-04  1.047916 -0.041232 -0.161812  0.212549
2000-01-05  0.498581  0.731168 -0.537677  1.346270
2000-01-06  1.120202  1.567621  0.003641  0.675253
2000-01-07 -0.487094  0.571455 -1.611639  0.103469
2000-01-10  0.836649  0.246462  0.588543  1.062782
2000-01-11 -0.157161  1.340307  1.195778 -1.097007

[7 rows x 4 columns]
```

- Added option to disable pandas-style tick locators and formatters using *series.plot(x_compat=True)* or *pandas.plot_params['x_compat'] = True* (GH2205)
- Existing TimeSeries methods *at_time* and *between_time* were added to DataFrame (GH2149)
- DataFrame.dot can now accept ndarrays (GH2042)
- DataFrame.drop now supports non-unique indexes (GH2101)
- Panel.shift now supports negative periods (GH2164)
- DataFrame now support unary ~ operator (GH2110)

API changes

- Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

```
In [1]: prng = pd.period_range('2012Q1', periods=2, freq='Q')

In [2]: s = pd.Series(np.random.randn(len(prng)), prng)

In [4]: s.resample('M')
Out[4]:
2012-01   -1.471992
2012-02         NaN
2012-03         NaN
2012-04   -0.493593
2012-05         NaN
2012-06         NaN
Freq: M, dtype: float64
```

- Period.end_time now returns the last nanosecond in the time interval (GH2124, GH2125, GH1764)

```
In [18]: p = pd.Period('2012')

In [19]: p.end_time
Out[19]: Timestamp('2012-12-31 23:59:59.999999999')
```

- File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

```
In [20]: import io

In [21]: data = ('A,B,C\n'
   ....:         '00001,001,5\n'
   ....:         '00002,002,6')
   ....:

In [22]: pd.read_csv(io.StringIO(data), converters={'A': lambda x: x.strip()})
Out[22]:
       A  B  C
0  00001  1  5
1  00002  2  6

[2 rows x 3 columns]
```

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 11 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Brenda Moon +
- Chang She
- Jeff Reback +
- Justin C Johnson +
- K.-Michael Aye

- Martin Blais
- Tobias Brandt +
- Wes McKinney
- Wouter Overmeire
- timmie
- y-p

8.16.2 v0.9.0 (October 7, 2012)

This is a major release from 0.8.1 and includes several new features and enhancements along with a large number of bug fixes. New features include vectorized unicode encoding/decoding for *Series.str*, *to_latex* method to DataFrame, more flexible parsing of boolean values, and enabling the download of options data from Yahoo! Finance.

New features

- Add `encode` and `decode` for unicode handling to *vectorized string processing methods* in Series.str (GH1706)
- Add `DataFrame.to_latex` method (GH1735)
- Add convenient expanding window equivalents of all rolling_* ops (GH1785)
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance (GH1748, GH1739)
- More flexible parsing of boolean values (Yes, No, TRUE, FALSE, etc) (GH1691, GH1295)
- Add `level` parameter to `Series.reset_index`
- `TimeSeries.between_time` can now select times across midnight (GH1871)
- Series constructor can now handle generator as input (GH1679)
- `DataFrame.dropna` can now take multiple axes (tuple/list) as input (GH924)
- Enable `skip_footer` parameter in `ExcelFile.parse` (GH1843)

API changes

- The default column names when `header=None` and no columns names passed to functions like `read_csv` has changed to be more Pythonic and amenable to attribute access:

```
In [1]: import io

In [2]: data = ('0,0,1\n'
   ...:         '1,1,0\n'
   ...:         '0,1,0')
   ...:

In [3]: df = pd.read_csv(io.StringIO(data), header=None)

In [4]: df
Out[4]:
   0  1  2
0  0  0  1
1  1  1  0
```

(continues on next page)

(continued from previous page)

```
2  0  1  0

[3 rows x 3 columns]
```

- Creating a Series from another Series, passing an index, will cause reindexing to happen inside rather than treating the Series like an ndarray. Technically improper usages like `Series(df[col1], index=df[col2])` that worked before "by accident" (this was never intended) will lead to all NA Series in some cases. To be perfectly clear:

```
In [5]: s1 = pd.Series([1, 2, 3])

In [6]: s1
Out[6]:
0    1
1    2
2    3
Length: 3, dtype: int64

In [7]: s2 = pd.Series(s1, index=['foo', 'bar', 'baz'])

In [8]: s2
Out[8]:
foo   NaN
bar   NaN
baz   NaN
Length: 3, dtype: float64
```

- Deprecated `day_of_year` API removed from PeriodIndex, use `dayofyear` (GH1723)
- Don't modify NumPy suppress printoption to True at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by HDFStore (GH1834, GH1824)
- Legacy cruft removed: pandas.stats.misc.quantileTS
- Use ISO8601 format for Period repr: monthly, daily, and on down (GH1776)
- Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) (GH1783)
- Setting parts of DataFrame/Panel using ix now aligns input Series/DataFrame (GH1630)
- `first` and `last` methods in `GroupBy` no longer drop non-numeric columns (GH1809)
- Resolved inconsistencies in specifying custom NA values in text parser. `na_values` of type dict no longer override default NAs unless `keep_default_na` is set to false explicitly (GH1657)
- `DataFrame.dot` will not do data alignment, and also work with Series (GH1915)

See the *full release notes* or issue tracker on GitHub for a complete list.

Contributors

A total of 24 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Chang She

- Christopher Whelan +
- Dan Miller +
- Daniel Shapiro +
- Dieter Vandenbussche
- Doug Coleman +
- John-Colvin +
- Johnny +
- Joshua Leahy +
- Lars Buitinck +
- Mark O'Leary +
- Martin Blais
- MinRK +
- Paul Ivanov +
- Skipper Seabold
- Spencer Lyon +
- Taavi Burns +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
- lenolib +
- tshauck +
- y-p +
- Øystein S. Haaland +

8.17 Version 0.8

8.17.1 v0.8.1 (July 22, 2012)

This release includes a few new features, performance enhancements, and over 30 bug fixes from 0.8.0. New features include notably NA friendly string processing functionality and a series of new plot types and options.

New features

- Add *vectorized string processing methods* accessible via Series.str (GH620)
- Add option to disable adjustment in EWMA (GH1584)
- *Radviz plot* (GH1566)
- *Parallel coordinates plot*
- *Bootstrap plot*

- Per column styles and secondary y-axis plotting (GH1559)
- New datetime converters millisecond plotting (GH1599)
- Add option to disable "sparse" display of hierarchical indexes (GH1538)
- Series/DataFrame's `set_index` method can *append levels* to an existing Index/MultiIndex (GH1569, GH1577)

Performance improvements

- Improved implementation of rolling min and max (thanks to Bottleneck !)
- Add accelerated `'median'` GroupBy option (GH1358)
- Significantly improve the performance of parsing ISO8601-format date strings with `DatetimeIndex` or `to_datetime` (GH1571)
- Improve the performance of GroupBy on single-key aggregations and use with Categorical types
- Significant datetime parsing performance improvements

Contributors

A total of 5 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Chang She
- Skipper Seabold
- Todd DeLuca +
- Vytautas Jancauskas
- Wes McKinney

8.17.2 v0.8.0 (June 29, 2012)

This is a major release from 0.7.3 and includes extensive work on the time series handling and processing infrastructure as well as a great deal of new functionality throughout the library. It includes over 700 commits from more than 20 distinct authors. Most pandas 0.7.3 and earlier users should not experience any issues upgrading, but due to the migration to the NumPy datetime64 dtype, there may be a number of bugs and incompatibilities lurking. Lingering incompatibilities will be fixed ASAP in a 0.8.1 release if necessary. See the *full release notes* or issue tracker on GitHub for a complete list.

Support for non-unique indexes

All objects can now work with non-unique indexes. Data alignment / join operations work according to SQL join semantics (including, if application, index duplication in many-to-many joins)

NumPy datetime64 dtype and 1.6 dependency

Time series data are now represented using NumPy's datetime64 dtype; thus, pandas 0.8.0 now requires at least NumPy 1.6. It has been tested and verified to work with the development version (1.7+) of NumPy as well which includes some significant user-facing API changes. NumPy 1.6 also has a number of bugs having to do with nanosecond resolution data, so I recommend that you steer clear of NumPy 1.6's datetime64 API functions (though limited as they are) and only interact with this data using the interface that pandas provides.

See the end of the 0.8.0 section for a "porting" guide listing potential issues for users migrating legacy code bases from pandas 0.7 or earlier to 0.8.0.

Bug fixes to the 0.7.x series for legacy NumPy < 1.6 users will be provided as they arise. There will be no more further development in 0.7.x beyond bug fixes.

Time series changes and improvements

Note: With this release, legacy scikits.timeseries users should be able to port their code to use pandas.

Note: See *documentation* for overview of pandas timeseries API.

- New datetime64 representation **speeds up join operations and data alignment**, **reduces memory usage**, and improve serialization / deserialization performance significantly over datetime.datetime
- High performance and flexible **resample** method for converting from high-to-low and low-to-high frequency. Supports interpolation, user-defined aggregation functions, and control over how the intervals and result labeling are defined. A suite of high performance Cython/C-based resampling functions (including Open-High-Low-Close) have also been implemented.
- Revamp of *frequency aliases* and support for **frequency shortcuts** like '15min', or '1h30min'
- New *DatetimeIndex class* supports both fixed frequency and irregular time series. Replaces now deprecated DateRange class
- New `PeriodIndex` and `Period` classes for representing *time spans* and performing **calendar logic**, including the *12 fiscal quarterly frequencies <timeseries.quarterly>*. This is a partial port of, and a substantial enhancement to, elements of the scikits.timeseries code base. Support for conversion between PeriodIndex and DatetimeIndex
- New Timestamp data type subclasses *datetime.datetime*, providing the same interface while enabling working with nanosecond-resolution data. Also provides *easy time zone conversions*.
- Enhanced support for *time zones*. Add *tz_convert* and `tz_lcoalize` methods to TimeSeries and DataFrame. All timestamps are stored as UTC; Timestamps from DatetimeIndex objects with time zone set will be localized to local time. Time zone conversions are therefore essentially free. User needs to know very little about pytz library now; only time zone names as as strings are required. Time zone-aware timestamps are equal if and only if their UTC timestamps match. Operations between time zone-aware time series with different time zones will result in a UTC-indexed time series.
- Time series **string indexing conveniences** / shortcuts: slice years, year and month, and index values with strings
- Enhanced time series **plotting**; adaptation of scikits.timeseries matplotlib-based plotting code
- New `date_range`, `bdate_range`, and `period_range` *factory functions*
- Robust **frequency inference** function *infer_freq* and `inferred_freq` property of DatetimeIndex, with option to infer frequency on construction of DatetimeIndex
- to_datetime function efficiently **parses array of strings** to DatetimeIndex. DatetimeIndex will parse array or list of strings to datetime64
- **Optimized** support for datetime64-dtype data in Series and DataFrame columns
- New NaT (Not-a-Time) type to represent **NA** in timestamp arrays

- Optimize Series.asof for looking up **"as of" values** for arrays of timestamps
- Milli, Micro, Nano date offset objects
- Can index time series with datetime.time objects to select all data at particular **time of day** (`TimeSeries.at_time`) or **between two times** (`TimeSeries.between_time`)
- Add *tshift* method for leading/lagging using the frequency (if any) of the index, as opposed to a naive lead/lag using shift

Other new features

- New *cut* and `qcut` functions (like R's cut function) for computing a categorical variable from a continuous variable by binning values either into value-based (`cut`) or quantile-based (`qcut`) bins
- Rename `Factor` to `Categorical` and add a number of usability features
- Add *limit* argument to fillna/reindex
- More flexible multiple function application in GroupBy, and can pass list (name, function) tuples to get result in particular order with given names
- Add flexible *replace* method for efficiently substituting values
- Enhanced *read_csv/read_table* for reading time series data and converting multiple columns to dates
- Add *comments* option to parser functions: read_csv, etc.
- Add *dayfirst* option to parser functions for parsing international DD/MM/YYYY dates
- Allow the user to specify the CSV reader *dialect* to control quoting etc.
- Handling *thousands* separators in read_csv to improve integer parsing.
- Enable unstacking of multiple levels in one shot. Alleviate `pivot_table` bugs (empty columns being introduced)
- Move to klib-based hash tables for indexing; better performance and less memory usage than Python's dict
- Add first, last, min, max, and prod optimized GroupBy functions
- New *ordered_merge* function
- Add flexible *comparison* instance methods eq, ne, lt, gt, etc. to DataFrame, Series
- Improve *scatter_matrix* plotting function and add histogram or kernel density estimates to diagonal
- Add *'kde'* plot option for density plots
- Support for converting DataFrame to R data.frame through rpy2
- Improved support for complex numbers in Series and DataFrame
- Add *pct_change* method to all data structures
- Add max_colwidth configuration option for DataFrame console output
- *Interpolate* Series values using index values
- Can select multiple columns from GroupBy
- Add *update* methods to Series/DataFrame for updating values in place
- Add `any` and `all` method to DataFrame

New plotting methods

`Series.plot` now supports a `secondary_y` option:

```
In [1]: plt.figure()
Out[1]: <Figure size 640x480 with 0 Axes>

In [2]: fx['FR'].plot(style='g')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[2]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37d7253b70>

In [3]: fx['IT'].plot(style='k--', secondary_y=True)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f37d6ea4f28>
```

Vytautas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, `'kde'` is a new option:

```
In [4]: s = pd.Series(np.concatenate((np.random.randn(1000),
   ...:                                np.random.randn(1000) * 0.5 + 3)))
   ...:

In [5]: plt.figure()
Out[5]: <Figure size 640x480 with 0 Axes>

In [6]: s.hist(density=True, alpha=0.2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]: <matplotlib.axes._subplots.
↪AxesSubplot at 0x7f37d6e58710>

In [7]: s.plot(kind='kde')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.axes._subplots.AxesSubplot at 0x7f37d6e58710>
```

See *the plotting page* for much more.

Other API changes

- Deprecation of `offset`, `time_rule`, and `timeRule` arguments names in time series functions. Warnings will be printed until pandas 0.9 or 1.0.

Potential porting issues for pandas <= 0.7.3 users

The major change that may affect you in pandas 0.8.0 is that time series indexes use NumPy's `datetime64` data type instead of `dtype=object` arrays of Python's built-in `datetime.datetime` objects. `DateRange` has been replaced by `DatetimeIndex` but otherwise behaved identically. But, if you have code that converts `DateRange` or `Index` objects that used to contain `datetime.datetime` values to plain NumPy arrays, you may have bugs lurking with code using scalar values because you are handing control over to NumPy:

```
In [8]: import datetime

In [9]: rng = pd.date_range('1/1/2000', periods=10)

In [10]: rng[5]
Out[10]: Timestamp('2000-01-06 00:00:00', freq='D')
```

(continues on next page)

(continued from previous page)

```
In [11]: isinstance(rng[5], datetime.datetime)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]: True

In [12]: rng_asarray = np.asarray(rng)

In [13]: scalar_val = rng_asarray[5]

In [14]: type(scalar_val)
Out[14]: numpy.datetime64
```

pandas's `Timestamp` object is a subclass of `datetime.datetime` that has nanosecond support (the `nanosecond` field store the nanosecond value between 0 and 999). It should substitute directly into any code that used `datetime.datetime` values before. Thus, I recommend not casting `DatetimeIndex` to regular NumPy arrays.

If you have code that requires an array of `datetime.datetime` objects, you have a couple of options. First, the `astype(object)` method of `DatetimeIndex` produces an array of `Timestamp` objects:

```
In [15]: stamp_array = rng.astype(object)

In [16]: stamp_array
Out[16]:
Index([2000-01-01 00:00:00, 2000-01-02 00:00:00, 2000-01-03 00:00:00,
       2000-01-04 00:00:00, 2000-01-05 00:00:00, 2000-01-06 00:00:00,
       2000-01-07 00:00:00, 2000-01-08 00:00:00, 2000-01-09 00:00:00,
       2000-01-10 00:00:00],
      dtype='object')

In [17]: stamp_array[5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪Timestamp('2000-01-06 00:00:00', freq='D')
```

To get an array of proper `datetime.datetime` objects, use the `to_pydatetime` method:

```
In [18]: dt_array = rng.to_pydatetime()

In [19]: dt_array
Out[19]:
array([datetime.datetime(2000, 1, 1, 0, 0),
       datetime.datetime(2000, 1, 2, 0, 0),
       datetime.datetime(2000, 1, 3, 0, 0),
       datetime.datetime(2000, 1, 4, 0, 0),
       datetime.datetime(2000, 1, 5, 0, 0),
       datetime.datetime(2000, 1, 6, 0, 0),
       datetime.datetime(2000, 1, 7, 0, 0),
       datetime.datetime(2000, 1, 8, 0, 0),
       datetime.datetime(2000, 1, 9, 0, 0),
       datetime.datetime(2000, 1, 10, 0, 0)], dtype=object)

In [20]: dt_array[5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪datetime.datetime(2000, 1, 6, 0, 0)
```

matplotlib knows how to handle `datetime.datetime` but not Timestamp objects. While I recommend that you plot time series using `TimeSeries.plot`, you can either use `to_pydatetime` or register a converter for the Timestamp type. See matplotlib documentation for more on this.

Warning: There are bugs in the user-facing API with the nanosecond datetime64 unit in NumPy 1.6. In particular, the string version of the array shows garbage values, and conversion to `dtype=object` is similarly broken.

```
In [21]: rng = pd.date_range('1/1/2000', periods=10)

In [22]: rng
Out[22]: 
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D')

In [23]: np.asarray(rng)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
array(['2000-01-01T00:00:00.000000000', '2000-01-02T00:00:00.000000000',
       '2000-01-03T00:00:00.000000000', '2000-01-04T00:00:00.000000000',
       '2000-01-05T00:00:00.000000000', '2000-01-06T00:00:00.000000000',
       '2000-01-07T00:00:00.000000000', '2000-01-08T00:00:00.000000000',
       '2000-01-09T00:00:00.000000000', '2000-01-10T00:00:00.000000000'], dtype=
↪'datetime64[ns]')

In [24]: converted = np.asarray(rng, dtype=object)

In [25]: converted[5]
Out[25]: Timestamp('2000-01-06 00:00:00', freq='D')
```

Trust me: don't panic. If you are using NumPy 1.6 and restrict your interaction with `datetime64` values to pandas's API you will be just fine. There is nothing wrong with the data-type (a 64-bit integer internally); all of the important data processing happens in pandas and is heavily tested. I strongly recommend that you **do not work directly with datetime64 arrays in NumPy 1.6** and only use the pandas API.

Support for non-unique indexes: In the latter case, you may have code inside a `try:...  catch:` block that failed due to the index not being unique. In many cases it will no longer fail (some method like `append` still check for uniqueness unless disabled). However, all is not lost: you can inspect `index.is_unique` and raise an exception explicitly if it is `False` or go to a different code branch.

Contributors

A total of 27 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Klein
- Chang She
- David Zaslavsky +
- Eric Chlebek +
- Jacques Kvam
- Kamil Kisiel
- Kelsey Jordahl +
- Kieran O'Mahony +
- Lorenzo Bolla +

- Luca Beltrame
- Marc Abramowitz +
- Mark Wiebe +
- Paddy Mullen +
- Peng Yu +
- Roy Hyunjin Han +
- RuiDC +
- Senthil Palanisami +
- Skipper Seabold
- Stefan van der Walt +
- Takafumi Arakaki +
- Thomas Kluyver
- Vytautas Jancauskas +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
- thuske +
- timmie +

8.18 Version 0.7

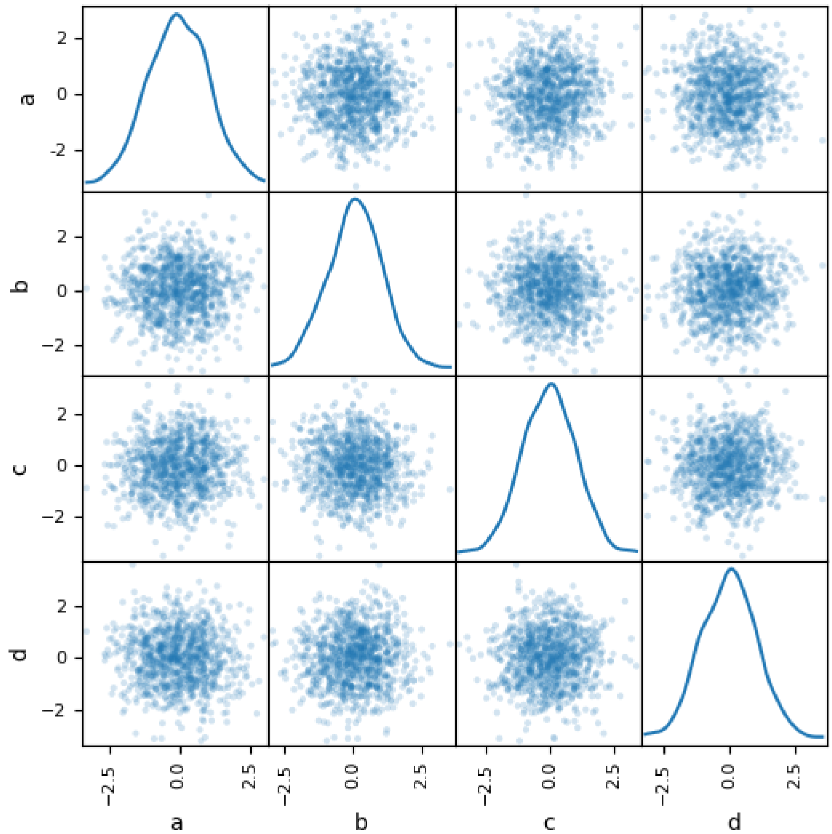
8.18.1 v.0.7.3 (April 12, 2012)

This is a minor release from 0.7.2 and fixes many minor bugs and adds a number of nice new features. There are also a couple of API changes to note; these should not affect very many users, and we are inclined to call them "bug fixes" even though they do constitute a change in behavior. See the *full release notes* or issue tracker on GitHub for a complete list.

New features

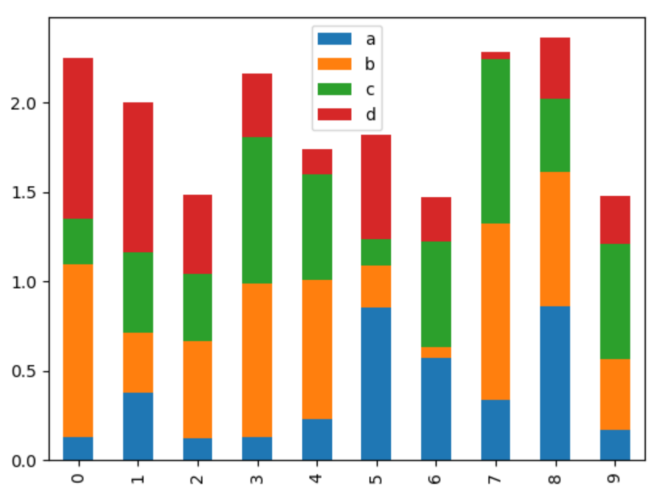
- New *fixed width file reader*, `read_fwf`
- New *scatter_matrix* function for making a scatter plot matrix

```
from pandas.tools.plotting import scatter_matrix
scatter_matrix(df, alpha=0.2)        # noqa F821
```

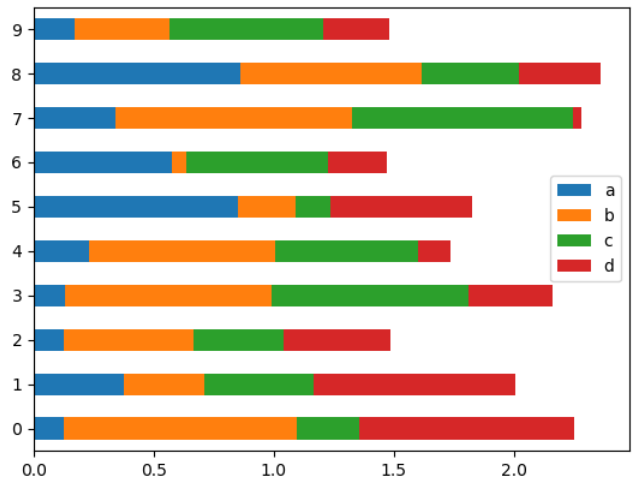


- Add `stacked` argument to Series and DataFrame's `plot` method for *stacked bar plots*.

```
df.plot(kind='bar', stacked=True)     # noqa F821
```



```
df.plot(kind='barh', stacked=True)    # noqa F821
```



- Add log x and y *scaling options* to `DataFrame.plot` and `Series.plot`
- Add `kurt` methods to Series and DataFrame for computing kurtosis

NA Boolean Comparison API Change

Reverted some changes to how NA values (represented typically as `NaN` or `None`) are handled in non-numeric Series:

```
In [1]: series = pd.Series(['Steve', np.nan, 'Joe'])

In [2]: series == 'Steve'
Out[2]:
0     True
1    False
2    False
Length: 3, dtype: bool

In [3]: series != 'Steve'
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]:
0    False
1     True
2     True
Length: 3, dtype: bool
```

In comparisons, NA / NaN will always come through as `False` except with `!=` which is `True`. *Be very careful* with boolean arithmetic, especially negation, in the presence of NA data. You may wish to add an explicit NA filter into boolean array operations if you are worried about this:

```
In [4]: mask = series == 'Steve'

In [5]: series[mask & series.notnull()]
Out[5]:
0    Steve
Length: 1, dtype: object
```

While propagating NA in comparisons may seem like the right behavior to some users (and you could argue on purely technical grounds that this is the right thing to do), the evaluation was made that propagating NA everywhere, including in numerical arrays, would cause a large amount of problems for users. Thus, a "practicality beats purity" approach was taken. This issue may be revisited at some point in the future.

Other API Changes

When calling apply on a grouped Series, the return value will also be a Series, to be more consistent with the groupby behavior with DataFrame:

```
In [6]: df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
   ...:                           'foo', 'bar', 'foo', 'foo'],
   ...:                    'B': ['one', 'one', 'two', 'three',
   ...:                           'two', 'two', 'one', 'three'],
   ...:                    'C': np.random.randn(8), 'D': np.random.randn(8)})
   ...:

In [7]: df
Out[7]:
     A      B         C         D
0  foo    one  0.469112 -0.861849
1  bar    one -0.282863 -2.104569
2  foo    two -1.509059 -0.494929
3  bar  three -1.135632  1.071804
4  foo    two  1.212112  0.721555
5  bar    two -0.173215 -0.706771
6  foo    one  0.119209 -1.039575
7  foo  three -1.044236  0.271860

[8 rows x 4 columns]

In [8]: grouped = df.groupby('A')['C']

In [9]: grouped.describe()
Out[9]:
     count      mean       std       min       25%       50%       75%       max
A
bar    3.0 -0.530570  0.526860 -1.135632 -0.709248 -0.282863 -0.228039 -0.173215
foo    5.0 -0.150572  1.113308 -1.509059 -1.044236  0.119209  0.469112  1.212112

[2 rows x 8 columns]

In [10]: grouped.apply(lambda x: x.sort_values()[-2:])    # top 2 values
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
A
bar  1   -0.282863
     5   -0.173215
foo  0    0.469112
     4    1.212112
Name: C, Length: 4, dtype: float64
```

Contributors

A total of 15 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Abraham Flaxman +
- Adam Klein
- Andreas H. +
- Chang She
- Dieter Vandenbussche
- Jacques Kvam +
- K.-Michael Aye +
- Kamil Kisiel +
- Martin Blais +
- Skipper Seabold
- Thomas Kluyver
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
- lgautier +

8.18.2 v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

New features

- Add additional tie-breaking methods in DataFrame.rank (GH874)
- Add ascending parameter to rank in Series, DataFrame (GH875)
- Add coerce_float option to DataFrame.from_records (GH893)
- Add sort_columns parameter to allow unsorted plots (GH918)
- Enable column access via attributes on GroupBy (GH882)
- Can pass dict of values to DataFrame.fillna (GH661)
- Can select multiple hierarchical groups by passing list of values in .ix (GH134)
- Add `axis` option to DataFrame.fillna (GH174)
- Add level keyword to `drop` for dropping values from a level (GH159)

Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)

Contributors

A total of 12 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Klein
- Benjamin Gross +
- Dan Birken +
- Dieter Vandenbussche
- Josh +
- Thomas Kluyver
- Travis N. Vaught +
- Wes McKinney
- Wouter Overmeire
- claudiobertoldi +
- elpres +
- joshuaar +

8.18.3 v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

New features

- Add `to_clipboard` function to pandas namespace for writing objects to the system clipboard (GH774)
- Add `itertuples` method to DataFrame for iterating through the rows of a dataframe as tuples (GH818)
- Add ability to pass fill_value and method to DataFrame and Series align method (GH806, GH807)
- Add fill_value option to reindex, align methods (GH784)
- Enable concat to produce DataFrame from Series (GH787)
- Add `between` method to Series (GH802)
- Add HTML representation hook to DataFrame for the IPython HTML notebook (GH773)
- Support for reading Excel 2007 XML documents using openpyxl

Performance improvements

- Improve performance and memory usage of fillna on DataFrame
- Can concatenate a list of Series along axis=1 to obtain a DataFrame (GH787)

Contributors

A total of 9 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Klein
- Brian Granger +
- Chang She
- Dieter Vandenbussche
- Josh Klein
- Steve +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko

8.18.4 v.0.7.0 (February 9, 2012)

New features

- New unified *merge function* for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains (GH220, GH249, GH267)
- New *unified concatenation function* for concatenating Series, DataFrame or Panel objects along an axis. Can form union or intersection of the other axes. Improves performance of `Series.append` and `DataFrame.append` (GH468, GH479, GH273)
- *Can* pass multiple DataFrames to *DataFrame.append* to concatenate (stack) and multiple Series to `Series.append` too
- *Can* pass list of dicts (e.g., a list of JSON objects) to DataFrame constructor (GH526)
- You can now *set multiple columns* in a DataFrame via `__getitem__`, useful for transformation (GH342)
- Handle differently-indexed output values in `DataFrame.apply` (GH498)

```
In [1]: df = pd.DataFrame(np.random.randn(10, 4))

In [2]: df.apply(lambda x: x.describe())
Out[2]: 
               0          1          2          3
count  10.000000  10.000000  10.000000  10.000000
mean    0.190912  -0.395125  -0.731920  -0.403130
std     0.730951   0.813266   1.112016   0.961912
min    -0.861849  -2.104569  -1.776904  -1.469388
25%    -0.411391  -0.698728  -1.501401  -1.076610
50%     0.380863  -0.228039  -1.191943  -1.004091
75%     0.658444   0.057974  -0.034326   0.461706
max     1.212112   0.577046   1.643563   1.071804

[8 rows x 4 columns]
```

- *Add* `reorder_levels` method to Series and DataFrame (GH534)

- *Add* dict-like `get` function to DataFrame and Panel (GH521)
- *Add* `DataFrame.iterrows` method for efficiently iterating through the rows of a DataFrame
- *Add* `DataFrame.to_panel` with code adapted from `LongPanel.to_long`
- *Add* `reindex_axis` method added to DataFrame
- *Add* `level` option to binary arithmetic functions on `DataFrame` and `Series`
- *Add* `level` option to the `reindex` and `align` methods on Series and DataFrame for broadcasting values across a level (GH542, GH552, others)
- *Add* attribute-based item access to `Panel` and add IPython completion (GH563)
- *Add* `logy` option to `Series.plot` for log-scaling on the Y axis
- *Add* `index` and `header` options to `DataFrame.to_string`
- *Can* pass multiple DataFrames to `DataFrame.join` to join on index (GH115)
- *Can* pass multiple Panels to `Panel.join` (GH115)
- *Added* `justify` argument to `DataFrame.to_string` to allow different alignment of column headers
- *Add* `sort` option to GroupBy to allow disabling sorting of the group keys for potential speedups (GH595)
- *Can* pass MaskedArray to Series constructor (GH563)
- *Add* Panel item access via attributes and IPython completion (GH554)
- Implement `DataFrame.lookup`, fancy-indexing analogue for retrieving values given a sequence of row and column labels (GH338)
- Can pass a *list of functions* to aggregate with groupby on a DataFrame, yielding an aggregated result with hierarchical columns (GH166)
- Can call `cummin` and `cummax` on Series and DataFrame to get cumulative minimum and maximum, respectively (GH647)
- `value_range` added as utility function to get min and max of a dataframe (GH288)
- Added `encoding` argument to `read_csv`, `read_table`, `to_csv` and `from_csv` for non-ascii text (GH717)
- *Added* `abs` method to pandas objects
- *Added* `crosstab` function for easily computing frequency tables
- *Added* `isin` method to index objects
- *Added* `level` argument to `xs` method of DataFrame.

API Changes to integer indexing

One of the potentially riskiest API changes in 0.7.0, but also one of the most important, was a complete review of how **integer indexes** are handled with regard to label-based indexing. Here is an example:

```
In [3]: s = pd.Series(np.random.randn(10), index=range(0, 20, 2))

In [4]: s
Out[4]:
0    -1.294524
2     0.413738
4     0.276662
```

(continues on next page)

(continued from previous page)

```
6    -0.472035
8    -0.013960
10   -0.362543
12   -0.006154
14   -0.923061
16    0.895717
18    0.805244
Length: 10, dtype: float64

In [5]: s[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪-1.2945235902555294

In [6]: s[2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪0.41373810535784006

In [7]: s[4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪0.27666171294975661
```

This is all exactly identical to the behavior before. However, if you ask for a key **not** contained in the Series, in versions 0.6.1 and prior, Series would *fall back* on a location-based lookup. This now raises a `KeyError`:

```
In [2]: s[1]
KeyError: 1
```

This change also has the same impact on DataFrame:

```
In [3]: df = pd.DataFrame(np.random.randn(8, 4), index=range(0, 16, 2))

In [4]: df
    0        1       2        3
0   0.88427  0.3363 -0.1787  0.03162
2   0.14451 -0.1415  0.2504  0.58374
4  -1.44779 -0.9186 -1.4996  0.27163
6  -0.26598 -2.4184 -0.2658  0.11503
8  -0.58776  0.3144 -0.8566  0.61941
10  0.10940 -0.7175 -1.0108  0.47990
12 -1.16919 -0.3087 -0.6049 -0.43544
14 -0.07337  0.3410  0.0424 -0.16037

In [5]: df.ix[3]
KeyError: 3
```

In order to support purely integer-based indexing, the following methods have been added:

Method	Description
`Series.iget_value(i)`	Retrieve value stored at location `i`
`Series.iget(i)`	Alias for `iget_value`
`DataFrame.irow(i)`	Retrieve the `i`-th row
`DataFrame.icol(j)`	Retrieve the `j`-th column
`DataFrame.iget_value(i, j)`	Retrieve the value at row `i` and column `j`

API tweaks regarding label-based slicing

Label-based slicing using `ix` now requires that the index be sorted (monotonic) **unless** both the start and endpoint are contained in the index:

```
In [1]: s = pd.Series(np.random.randn(6), index=list('gmkaec'))

In [2]: s
Out[2]:
g   -1.182230
m   -0.276183
k   -0.243550
a    1.628992
e    0.073308
c   -0.539890
dtype: float64
```

Then this is OK:

```
In [3]: s.ix['k':'e']
Out[3]:
k   -0.243550
a    1.628992
e    0.073308
dtype: float64
```

But this is not:

```
In [12]: s.ix['b':'h']
KeyError 'b'
```

If the index had been sorted, the "range selection" would have been possible:

```
In [4]: s2 = s.sort_index()

In [5]: s2
Out[5]:
a    1.628992
c   -0.539890
e    0.073308
g   -1.182230
k   -0.243550
m   -0.276183
dtype: float64

In [6]: s2.ix['b':'h']
Out[6]:
c   -0.539890
e    0.073308
g   -1.182230
dtype: float64
```

Changes to Series `[]` operator

As as notational convenience, you can pass a sequence of labels or a label slice to a Series when getting and setting values via `[]` (i.e. the `__getitem__` and `__setitem__` methods). The behavior will be the same as passing similar input to `ix` **except in the case of integer indexing**:

```
In [8]: s = pd.Series(np.random.randn(6), index=list('acegkm'))

In [9]: s
Out[9]:
a   -1.206412
c    2.565646
e    1.431256
g    1.340309
k   -1.170299
m   -0.226169
Length: 6, dtype: float64

In [10]: s[['m', 'a', 'c', 'e']]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
m   -0.226169
a   -1.206412
c    2.565646
e    1.431256
Length: 4, dtype: float64

In [11]: s['b':'l']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
c    2.565646
e    1.431256
g    1.340309
k   -1.170299
Length: 4, dtype: float64

In [12]: s['c':'k']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
c    2.565646
e    1.431256
g    1.340309
k   -1.170299
Length: 4, dtype: float64
```

In the case of integer indexes, the behavior will be exactly as before (shadowing `ndarray`):

```
In [13]: s = pd.Series(np.random.randn(6), index=range(0, 12, 2))

In [14]: s[[4, 0, 2]]
Out[14]:
4    0.132003
0    0.410835
2    0.813850
Length: 3, dtype: float64

In [15]: s[1:5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]:␣
↪
2    0.813850
4    0.132003
6   -0.827317
8   -0.076467
Length: 4, dtype: float64
```

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use `ix`.

Other API Changes

- The deprecated `LongPanel` class has been completely removed
- If `Series.sort` is called on a column of a DataFrame, an exception will now be raised. Before it was possible to accidentally mutate a DataFrame's column by doing `df[col].sort()` instead of the side-effect free method `df[col].order()` (GH316)
- Miscellaneous renames and deprecations which will (harmlessly) raise `FutureWarning`
- `drop` added as an optional parameter to `DataFrame.reset_index` (GH699)

Performance improvements

- *Cythonized GroupBy aggregations* no longer presort the data, thus achieving a significant speedup (GH93). GroupBy aggregations with Python functions significantly sped up by clever manipulation of the ndarray data type in Cython (GH496).
- Better error message in DataFrame constructor when passed column labels don't match data (GH497)
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython (GH496)
- Can store objects indexed by tuples and floats in HDFStore (GH492)
- Don't print length by default in Series.to_string, add *length* option (GH489)
- Improve Cython code for multi-groupby to aggregate without having to sort the data (GH93)
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of Series.__getitem__ for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in setup.py if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also (GH536)
- Default name assignment when calling `reset_index` on DataFrame with a regular (non-hierarchical) index (GH476)
- Use Cythonized groupers when possible in Series/DataFrame stat ops with `level` parameter passed (GH545)
- Ported skiplist data structure to C to speed up `rolling_median` by about 5-10x in most typical use cases (GH374)

Contributors

A total of 18 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Klein
- Bayle Shanks +
- Chris Billington +

- Dieter Vandenbussche
- Fabrizio Pollastri +
- Graham Taylor +
- Gregg Lind +
- Josh Klein +
- Luca Beltrame
- Olivier Grisel +
- Skipper Seabold
- Thomas Kluyver
- Thomas Wiecki +
- Wes McKinney
- Wouter Overmeire
- Yaroslav Halchenko
- fabriziop +
- theandygross +

8.19 Version 0.6

8.19.1 v.0.6.1 (December 13, 2011)

New features

- Can *append single rows* (as Series) to a DataFrame
- Add Spearman and Kendall rank *correlation* options to Series.corr and DataFrame.corr (GH428)
- *Added* `get_value` and `set_value` methods to Series, DataFrame, and Panel for very low-overhead access (>2x faster in many cases) to scalar elements (GH437, GH438). `set_value` is capable of producing an enlarged object.
- Add PyQt table widget to sandbox (GH435)
- DataFrame.align can *accept Series arguments* and an *axis option* (GH461)
- Implement new *SparseArray* and *SparseList* data structures. SparseSeries now derives from SparseArray (GH463)
- *Better console printing options* (GH453)
- Implement fast *data ranking* for Series and DataFrame, fast versions of scipy.stats.rankdata (GH428)
- Implement *DataFrame.from_items* alternate constructor (GH444)
- DataFrame.convert_objects method for *inferring better dtypes* for object columns (GH302)
- Add *rolling_corr_pairwise* function for computing Panel of correlation matrices (GH189)
- Add *margins* option to *pivot_table* for computing subgroup aggregates (GH114)
- Add `Series.from_csv` function (GH482)

- *Can pass* DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH #462)
- MultiIndex.get_level_values can *accept the level name*

Performance improvements

- Improve memory usage of *DataFrame.describe* (do not copy data unnecessarily) (PR #425)
- Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
- Fix performance regression in cross-sectional count in DataFrame, affecting DataFrame.dropna speed
- Column deletion in DataFrame copies no data (computes views on blocks) (GH #158)

Contributors

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

- Dieter Vandenbussche
- Fernando Perez +
- Jev Kuznetsov +
- Joon Ro
- Ralph Bean +
- Wes McKinney
- Wouter Overmeire

8.19.2 v.0.6.0 (November 25, 2011)

New Features

- *Added* `melt` function to `pandas.core.reshape`
- *Added* `level` parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- *Added* `head` and `tail` methods to Series, analogous to to DataFrame (GH296)
- *Added* `Series.isin` function which checks if each value is contained in a passed sequence (GH289)
- *Added* `float_format` option to `Series.to_string`
- *Added* `skip_footer` (GH291) and `converters` (GH343) options to `read_csv` and `read_table`
- *Added* `drop_duplicates` and `duplicated` functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- *Implemented* operators '&', '|', '^', '-' on DataFrame (GH347)
- *Added* `Series.mad`, mean absolute deviation
- *Added* `QuarterEnd` DateOffset (GH321)
- *Added* `dot` to DataFrame (GH65)
- *Added* `orient` option to `Panel.from_dict` (GH359, GH301)
- *Added* `orient` option to `DataFrame.from_dict`

- *Added* passing list of tuples or list of lists to `DataFrame.from_records` (GH357)
- *Added* multiple levels to groupby (GH103)
- *Allow* multiple columns in `by` argument of `DataFrame.sort_index` (GH92, GH362)
- *Added* fast `get_value` and `put_value` methods to DataFrame (GH360)
- *Added* `cov` instance methods to Series and DataFrame (GH194, GH362)
- *Added* `kind='bar'` option to `DataFrame.plot` (GH348)
- *Added* `idxmin` and `idxmax` to Series and DataFrame (GH286)
- *Added* `read_clipboard` function to parse DataFrame from clipboard (GH300)
- *Added* `nunique` function to Series for counting unique elements (GH297)
- *Made* DataFrame constructor use Series name if no columns passed (GH373)
- *Support* regular expressions in read_table/read_csv (GH364)
- *Added* `DataFrame.to_html` for writing DataFrame to HTML (GH387)
- *Added* support for MaskedArray data in DataFrame, masked values converted to NaN (GH396)
- *Added* `DataFrame.boxplot` function (GH368)
- *Can* pass extra args, kwds to DataFrame.apply (GH376)
- *Implement* `DataFrame.join` with vector `on` argument (GH312)
- *Added* `legend` boolean flag to `DataFrame.plot` (GH324)
- *Can* pass multiple levels to `stack` and `unstack` (GH370)
- *Can* pass multiple values columns to `pivot_table` (GH381)
- *Use* Series name in GroupBy for result index (GH363)
- *Added* `raw` option to `DataFrame.apply` for performance if only need ndarray (GH309)
- Added proper, tested weighted least squares to standard and panel OLS (GH303)

Performance Enhancements

- VBENCH Cythonized `cache_readonly`, resulting in substantial micro-performance enhancements throughout the code base (GH361)
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than *np.apply_along_axis* (GH309)
- VBENCH Improved performance of `MultiIndex.from_tuples`
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations
- VBENCH + DOCUMENT Add `raw` option to `DataFrame.apply` for getting better performance when
- VBENCH Faster cythonized count by level in Series and DataFrame (GH341)
- VBENCH? Significant GroupBy performance enhancement with multiple keys with many "empty" combinations
- VBENCH New Cython vectorized function `map_infer` speeds up `Series.apply` and `Series.map` significantly when passed elementwise Python function, motivated by (GH355)
- VBENCH Significantly improved performance of `Series.order`, which also makes np.unique called on a Series faster (GH327)

- VBENCH Vastly improved performance of GroupBy on axes with a MultiIndex (GH299)

Contributors

A total of 8 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Adam Klein +
- Chang She +
- Dieter Vandenbussche
- Jeff Hammerbacher +
- Nathan Pinger +
- Thomas Kluyver
- Wes McKinney
- Wouter Overmeire +

8.20 Version 0.5

8.20.1 v.0.5.0 (October 24, 2011)

New Features

- *Added* `DataFrame.align` method with standard join options
- *Added* `parse_dates` option to `read_csv` and `read_table` methods to optionally try to parse dates in the index columns
- *Added* `nrows`, `chunksize`, and `iterator` arguments to `read_csv` and `read_table`. The last two return a new `TextParser` class capable of lazily iterating through chunks of a flat file (GH242)
- *Added* ability to join on multiple columns in `DataFrame.join` (GH214)
- Added private `_get_duplicates` function to `Index` for identifying duplicate values more easily (ENH5c)
- *Added* column attribute access to DataFrame.
- *Added* Python tab completion hook for DataFrame columns. (GH233, GH230)
- *Implemented* `Series.describe` for Series containing objects (GH241)
- *Added* inner join option to `DataFrame.join` when joining on key(s) (GH248)
- *Implemented* selecting DataFrame columns by passing a list to `__getitem__` (GH253)
- *Implemented* & and | to intersect / union Index objects, respectively (GH261)
- *Added* `pivot_table` convenience function to pandas namespace (GH234)
- *Implemented* `Panel.rename_axis` function (GH243)
- DataFrame will show index level names in console output (GH334)
- *Implemented* `Panel.take`
- *Added* `set_eng_float_format` for alternate DataFrame floating point string formatting (ENH61)

- *Added* convenience `set_index` function for creating a DataFrame index from its existing columns
- *Implemented* `groupby` hierarchical index level name (GH223)
- *Added* support for different delimiters in `DataFrame.to_csv` (GH244)
- TODO: DOCS ABOUT TAKE METHODS

Performance Enhancements

- VBENCH Major performance improvements in file parsing functions `read_csv` and `read_table`
- VBENCH Added Cython function for converting tuples to ndarray very fast. Speeds up many MultiIndex-related operations
- VBENCH Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance (GH211)
- VBENCH Improved speed of `DataFrame.xs` on mixed-type DataFrame objects by about 5x, regression from 0.3.0 (GH215)
- VBENCH With new `DataFrame.align` method, speeding up binary operations between differently-indexed DataFrame objects by 10-25%.
- VBENCH Significantly sped up conversion of nested dict into DataFrame (GH212)
- VBENCH Significantly speed up DataFrame `__repr__` and `count` on large mixed-type DataFrame objects

Contributors

A total of 9 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Aman Thakral +
- Luca Beltrame +
- Nick Pentreath +
- Skipper Seabold
- Thomas Kluyver +
- Wes McKinney
- Yaroslav Halchenko +
- lodagro +
- unknown +

8.21 Version 0.4

8.21.1 v.0.4.1 through v0.4.3 (September 25 - October 9, 2011)

New Features

- Added Python 3 support using 2to3 (GH200)
- *Added* `name` attribute to `Series`, now prints as part of `Series.__repr__`

- *Added* instance methods `isnull` and `notnull` to Series (GH209, GH203)
- *Added* `Series.align` method for aligning two series with choice of join method (ENH56)
- *Added* method `get_level_values` to `MultiIndex` (GH188)
- Set values in mixed-type `DataFrame` objects via `.ix` indexing attribute (GH135)
- Added new `DataFrame` *methods* `get_dtype_counts` and property `dtypes` (ENHdc)
- Added *ignore_index* option to `DataFrame.append` to stack DataFrames (ENH1b)
- `read_csv` tries to *sniff* delimiters using `csv.Sniffer` (GH146)
- `read_csv` can *read* multiple columns into a `MultiIndex`; DataFrame's `to_csv` method writes out a corresponding `MultiIndex` (GH151)
- `DataFrame.rename` has a new `copy` parameter to *rename* a DataFrame in place (ENHed)
- *Enable* unstacking by name (GH142)
- *Enable* `sortlevel` to work by level (GH141)

Performance Enhancements

- Altered binary operations on differently-indexed SparseSeries objects to use the integer-based (dense) alignment logic which is faster with a larger number of blocks (GH205)
- Wrote faster Cython data alignment / merging routines resulting in substantial speed increases
- Improved performance of `isnull` and `notnull`, a regression from v0.3.0 (GH187)
- Refactored code related to `DataFrame.join` so that intermediate aligned copies of the data in each `DataFrame` argument do not need to be created. Substantial performance increases result (GH176)
- Substantially improved performance of generic `Index.intersection` and `Index.union`
- Implemented `BlockManager.take` resulting in significantly faster `take` performance on mixed-type `DataFrame` objects (GH104)
- Improved performance of `Series.sort_index`
- Significant groupby performance enhancement: removed unnecessary integrity checks in DataFrame internals that were slowing down slicing operations to retrieve groups
- Optimized `_ensure_index` function resulting in performance savings in type-checking Index objects
- Wrote fast time series merging / joining methods in Cython. Will be integrated later into DataFrame.join and related functions

Contributors

A total of 2 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Thomas Kluyver +
- Wes McKinney

BIBLIOGRAPHY

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://docs.python.org/3/library/pickle.html

[1] https://docs.python.org/3/library/pickle.html

[1] http://docs.sqlalchemy.org

[2] https://www.python.org/dev/peps/pep-0249/

[1] https://en.wikipedia.org/wiki/Imputation_(statistics)

[1] https://en.wikipedia.org/wiki/Imputation_(statistics)

[1] https://en.wikipedia.org/wiki/Imputation_(statistics)

[1] “Bootstrapping (statistics)” in https://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29

PYTHON MODULE INDEX

p