(continued from previous page)

```
2012-01-01 00:00:01.000    204
Freq: 250L, dtype: int64

In [301]: ts[:2].resample('250L').ffill(limit=2)
Out[301]: 
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 [302]: rng = pd.date_range('2014-1-1', periods=100, freq='D') + pd.Timedelta('1s')

In [303]: ts = pd.Series(range(100), index=rng)
```

If we want to resample to the full range of the series:

```
In [304]: ts.resample('3T').sum()
Out[304]: 
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
                       ..
2014-04-09 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 [305]: from functools import partial

In [306]: from pandas.tseries.frequencies import to_offset

In [307]: def round(t, freq):
   .....:     freq = to_offset(freq)
   .....:     return pd.Timestamp((t.value // freq.delta.value) * freq.delta.value)
   .....: 

In [308]: ts.groupby(partial(round, freq='3T')).sum()
```

(continues on next page)

(continued from previous page)

```
Out[308]:
2014-01-01     0
2014-01-02     1
2014-01-03     2
2014-01-04     3
2014-01-05     4
              ..
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 [309]: df = pd.DataFrame(np.random.randn(1000, 3),
   .....:                   index=pd.date_range('1/1/2012', freq='S', periods=1000),
   .....:                   columns=['A', 'B', 'C'])
   .....:

In [310]: r = df.resample('3T')

In [311]: r.mean()
Out[311]:
                            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 [312]: r['A'].mean()
Out[312]:
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 [313]: r[['A', 'B']].mean()
Out[313]:
                            A         B
2012-01-01 00:00:00 -0.033823 -0.121514
2012-01-01 00:03:00  0.056909  0.146731
```

(continues on next page)

(continued from previous page)

```
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 [314]: r['A'].agg([np.sum, np.mean, np.std])
Out[314]:
                           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 [315]: r.agg([np.sum, np.mean])
Out[315]:
                             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 [316]: r.agg({'A': np.sum,
   .....:        'B': lambda x: np.std(x, ddof=1)})
   .....:
Out[316]:
                             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 [317]: r.agg({'A': 'sum', 'B': 'std'})
Out[317]:
                             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 [318]: r.agg({'A': ['sum', 'std'], 'B': ['mean', 'std']})
Out[318]:
                             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 [319]: 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 [320]: df
Out[320]:
                   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 [321]: df.resample('M', on='date').sum()
Out[321]:
            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 [322]: df.resample('M', level='d').sum()
Out[322]:
            a
d
2015-01-31  6
2015-02-28  4
```

Iterating through groups

With the `Resampler` object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [323]: small = pd.Series(
   .....:     range(6),
   .....:     index=pd.to_datetime(['2017-01-01T00:00:00',
   .....:                           '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 [324]: resampled = small.resample('H')

In [325]: 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.

2.14.11 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 [326]: pd.Period('2012', freq='A-DEC')
Out[326]: Period('2012', 'A-DEC')

In [327]: pd.Period('2012-1-1', freq='D')
Out[327]: Period('2012-01-01', 'D')

In [328]: pd.Period('2012-1-1 19:00', freq='H')
Out[328]: Period('2012-01-01 19:00', 'H')

In [329]: pd.Period('2012-1-1 19:00', freq='5H')
Out[329]: 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 [330]: p = pd.Period('2012', freq='A-DEC')

In [331]: p + 1
Out[331]: Period('2013', 'A-DEC')

In [332]: p - 3
Out[332]: Period('2009', 'A-DEC')

In [333]: p = pd.Period('2012-01', freq='2M')

In [334]: p + 2
Out[334]: Period('2012-05', '2M')

In [335]: p - 1
Out[335]: Period('2011-11', '2M')

In [336]: p == pd.Period('2012-01', freq='3M')
---------------------------------------------------------------------------
IncompatibleFrequency                     Traceback (most recent call last)
<ipython-input-336-4b67dc0b596c> in <module>
----> 1 p == pd.Period('2012-01', freq='3M')

/pandas-release/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 [337]: p = pd.Period('2014-07-01 09:00', freq='H')

In [338]: p + pd.offsets.Hour(2)
Out[338]: Period('2014-07-01 11:00', 'H')

In [339]: p + datetime.timedelta(minutes=120)
Out[339]: Period('2014-07-01 11:00', 'H')

In [340]: p + np.timedelta64(7200, 's')
Out[340]: 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 [341]: p = pd.Period('2014-07', freq='M')

In [342]: p + pd.offsets.MonthEnd(3)
Out[342]: 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 [343]: pd.Period('2012', freq='A-DEC') - pd.Period('2002', freq='A-DEC')
Out[343]: <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 [344]: prng = pd.period_range('1/1/2011', '1/1/2012', freq='M')

In [345]: prng
Out[345]: 
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 [346]: pd.PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[346]: 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 [347]: pd.period_range(start='2014-01', freq='3M', periods=4)
Out[347]: 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 [348]: pd.period_range(start=pd.Period('2017Q1', freq='Q'),
   .....:                 end=pd.Period('2017Q2', freq='Q'), freq='M')
   .....:
Out[348]: 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 [349]: ps = pd.Series(np.random.randn(len(prng)), prng)

In [350]: ps
Out[350]:
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 [351]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [352]: idx
Out[352]:
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 [353]: idx + pd.offsets.Hour(2)
Out[353]:
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 [354]: idx = pd.period_range('2014-07', periods=5, freq='M')

In [355]: idx
Out[355]: PeriodIndex(['2014-07', '2014-08', '2014-09', '2014-10', '2014-11'], dtype=
↪'period[M]', freq='M')

In [356]: idx + pd.offsets.MonthEnd(3)
```

(continues on next page)

(continued from previous page)

```
Out[356]: 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

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 [357]: pi = pd.period_range('2016-01-01', periods=3, freq='M')

In [358]: pi
Out[358]: PeriodIndex(['2016-01', '2016-02', '2016-03'], dtype='period[M]', freq='M')

In [359]: pi.dtype
Out[359]: 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 [360]: pi.astype('period[D]')
Out[360]: PeriodIndex(['2016-01-31', '2016-02-29', '2016-03-31'], dtype='period[D]',␣
↪freq='D')

# convert to DatetimeIndex
In [361]: pi.astype('datetime64[ns]')
Out[361]: DatetimeIndex(['2016-01-01', '2016-02-01', '2016-03-01'], dtype=
↪'datetime64[ns]', freq='MS')

# convert to PeriodIndex
In [362]: dti = pd.date_range('2011-01-01', freq='M', periods=3)

In [363]: dti
Out[363]: DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31'], dtype=
↪'datetime64[ns]', freq='M')

In [364]: dti.astype('period[M]')
Out[364]: 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 [365]: ps['2011-01']
Out[365]: -2.9169013294054507

In [366]: ps[datetime.datetime(2011, 12, 25):]
Out[366]:
```

(continues on next page)

(continued from previous page)

```
2011-12    2.261385
2012-01   -0.329583
Freq: M, dtype: float64

In [367]: ps['10/31/2011':'12/31/2011']
Out[367]:
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 [368]: ps['2011']
Out[368]:
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 [369]: dfp = pd.DataFrame(np.random.randn(600, 1),
   .....:                    columns=['A'],
   .....:                    index=pd.period_range('2013-01-01 9:00',
   .....:                                          periods=600,
   .....:                                          freq='T'))
   .....:

In [370]: dfp
Out[370]:
                         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 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 [371]: dfp['2013-01-01 10H']
Out[371]:
                         A
2013-01-01 10:00 -0.308975
```

(continues on next page)

(continued from previous page)

```
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 10:55 -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 [372]: dfp['2013-01-01 10H':'2013-01-01 11H']
Out[372]:
                         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 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 [373]: p = pd.Period('2011', freq='A-DEC')

In [374]: p
Out[374]: 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 [375]: p.asfreq('M', how='start')
Out[375]: Period('2011-01', 'M')

In [376]: p.asfreq('M', how='end')
Out[376]: Period('2011-12', 'M')
```

The shorthands 's' and 'e' are provided for convenience:

```
In [377]: p.asfreq('M', 's')
Out[377]: Period('2011-01', 'M')

In [378]: p.asfreq('M', 'e')
Out[378]: 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 [379]: p = pd.Period('2011-12', freq='M')

In [380]: p.asfreq('A-NOV')
Out[380]: 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 [381]: p = pd.Period('2012Q1', freq='Q-DEC')

In [382]: p.asfreq('D', 's')
Out[382]: Period('2012-01-01', 'D')

In [383]: p.asfreq('D', 'e')
Out[383]: Period('2012-03-31', 'D')
```

`Q-MAR` defines fiscal year end in March:

```
In [384]: p = pd.Period('2011Q4', freq='Q-MAR')

In [385]: p.asfreq('D', 's')
Out[385]: Period('2011-01-01', 'D')

In [386]: p.asfreq('D', 'e')
Out[386]: Period('2011-03-31', 'D')
```

2.14.12 Converting between representations

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```
In [387]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [388]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [389]: ts
Out[389]:
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 [390]: ps = ts.to_period()

In [391]: ps
Out[391]:
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 [392]: ps.to_timestamp()
Out[392]:
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 [393]: ps.to_timestamp('D', how='s')
Out[393]:
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 [394]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [395]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [396]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [397]: ts.head()
Out[397]:
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
```

2.14.13 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 [398]: span = pd.period_range('1215-01-01', '1381-01-01', freq='D')

In [399]: span
Out[399]:
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 [400]: s = pd.Series([20121231, 20141130, 99991231])

In [401]: s
Out[401]:
0    20121231
1    20141130
2    99991231
dtype: int64

In [402]: def conv(x):
   .....:     return pd.Period(year=x // 10000, month=x // 100 % 100,
   .....:                      day=x % 100, freq='D')
   .....:

In [403]: s.apply(conv)
Out[403]:
0    2012-12-31
1    2014-11-30
2    9999-12-31
dtype: period[D]

In [404]: s.apply(conv)[2]
Out[404]: Period('9999-12-31', 'D')
```

These can easily be converted to a `PeriodIndex`:

```
In [405]: span = pd.PeriodIndex(s.apply(conv))

In [406]: span
Out[406]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='period[D]',␣
↪freq='D')
```

2.14.14 Time zone handling

pandas provides rich support for working with timestamps in different time zones using the `pytz` and `dateutil` libraries or class:*datetime.timezone* objects from the standard library.

Working with time zones

By default, pandas objects are time zone unaware:

```
In [407]: rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')

In [408]: rng.tz is None
Out[408]: True
```

To localize these dates to a time zone (assign a particular time zone to a naive date), you can use the `tz_localize` method or the `tz` keyword argument in *date_range()*, *Timestamp*, or *DatetimeIndex*. You can either pass `pytz` or `dateutil` time zone objects or Olson time zone database strings. Olson time zone strings will return `pytz` time zone objects by default. To return `dateutil` time zone objects, append `dateutil/` before the string.

- 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 time zones so there isn't a fixed list available. For common zones, the names are the same as `pytz`.

```
In [409]: import dateutil

# pytz
In [410]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=3, freq='D',
   .....:                          tz='Europe/London')
   .....:

In [411]: rng_pytz.tz
Out[411]: <DstTzInfo 'Europe/London' LMT-1 day, 23:59:00 STD>

# dateutil
In [412]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=3, freq='D')

In [413]: rng_dateutil = rng_dateutil.tz_localize('dateutil/Europe/London')

In [414]: rng_dateutil.tz
Out[414]: tzfile('/usr/share/zoneinfo/Europe/London')

# dateutil - utc special case
In [415]: rng_utc = pd.date_range('3/6/2012 00:00', periods=3, freq='D',
   .....:                         tz=dateutil.tz.tzutc())
   .....:

In [416]: rng_utc.tz
Out[416]: tzutc()
```

New in version 0.25.0.

```
# datetime.timezone
In [417]: rng_utc = pd.date_range('3/6/2012 00:00', periods=3, freq='D',
   .....:                         tz=datetime.timezone.utc)
   .....:
```

(continues on next page)

(continued from previous page)

```
In [418]: rng_utc.tz
Out[418]: datetime.timezone.utc
```

Note that the `UTC` time zone is a special case in `dateutil` and should be constructed explicitly as an instance of `dateutil.tz.tzutc`. You can also construct other time zones objects explicitly first.

```
In [419]: import pytz

# pytz
In [420]: tz_pytz = pytz.timezone('Europe/London')

In [421]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=3, freq='D')

In [422]: rng_pytz = rng_pytz.tz_localize(tz_pytz)

In [423]: rng_pytz.tz == tz_pytz
Out[423]: True

# dateutil
In [424]: tz_dateutil = dateutil.tz.gettz('Europe/London')

In [425]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=3, freq='D',
   .....:                              tz=tz_dateutil)
   .....:

In [426]: rng_dateutil.tz == tz_dateutil
Out[426]: True
```

To convert a time zone aware pandas object from one time zone to another, you can use the `tz_convert` method.

```
In [427]: rng_pytz.tz_convert('US/Eastern')
Out[427]:
DatetimeIndex(['2012-03-05 19:00:00-05:00', '2012-03-06 19:00:00-05:00',
               '2012-03-07 19:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

Note: When using `pytz` time zones, *`DatetimeIndex`* will construct a different time zone object than a *`Timestamp`* for the same time zone input. A *`DatetimeIndex`* can hold a collection of *`Timestamp`* objects that may have different UTC offsets and cannot be succinctly represented by one `pytz` time zone instance while one *`Timestamp`* represents one point in time with a specific UTC offset.

```
In [428]: dti = pd.date_range('2019-01-01', periods=3, freq='D', tz='US/Pacific')

In [429]: dti.tz
Out[429]: <DstTzInfo 'US/Pacific' LMT-1 day, 16:07:00 STD>

In [430]: ts = pd.Timestamp('2019-01-01', tz='US/Pacific')

In [431]: ts.tz
Out[431]: <DstTzInfo 'US/Pacific' PST-1 day, 16:00:00 STD>
```

Warning: Be wary of conversions between libraries. For some time zones, `pytz` and `dateutil` have different definitions of the zone. This is more of a problem for unusual time zones than for 'standard' zones like `US/Eastern`.

Warning: Be aware that a time zone definition across versions of time zone 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: For `pytz` time zones, it is incorrect to pass a time zone object directly into the `datetime.datetime` constructor (e.g., `datetime.datetime(2011, 1, 1, tz=pytz.timezone('US/Eastern'))`. Instead, the datetime needs to be localized using the `localize` method on the `pytz` time zone object.

Under the hood, all timestamps are stored in UTC. Values from a time zone aware `DatetimeIndex` or `Timestamp` will have their fields (day, hour, minute, etc.) 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 [432]: rng_eastern = rng_utc.tz_convert('US/Eastern')

In [433]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [434]: rng_eastern[2]
Out[434]: Timestamp('2012-03-07 19:00:00-0500', tz='US/Eastern', freq='D')

In [435]: rng_berlin[2]
Out[435]: Timestamp('2012-03-08 01:00:00+0100', tz='Europe/Berlin', freq='D')

In [436]: rng_eastern[2] == rng_berlin[2]
Out[436]: True
```

Operations between `Series` in different time zones will yield UTC `Series`, aligning the data on the UTC timestamps:

```
In [437]: ts_utc = pd.Series(range(3), pd.date_range('20130101', periods=3, tz='UTC'))

In [438]: eastern = ts_utc.tz_convert('US/Eastern')

In [439]: berlin = ts_utc.tz_convert('Europe/Berlin')

In [440]: result = eastern + berlin

In [441]: result
Out[441]:
2013-01-01 00:00:00+00:00    0
2013-01-02 00:00:00+00:00    2
2013-01-03 00:00:00+00:00    4
Freq: D, dtype: int64

In [442]: result.index
Out[442]:
DatetimeIndex(['2013-01-01 00:00:00+00:00', '2013-01-02 00:00:00+00:00',
```

(continues on next page)

(continued from previous page)

```
                 '2013-01-03 00:00:00+00:00'],
                dtype='datetime64[ns, UTC]', freq='D')
```

To remove time zone information, use `tz_localize(None)` or `tz_convert(None)`. `tz_localize(None)` will remove the time zone yielding the local time representation. `tz_convert(None)` will remove the time zone after converting to UTC time.

```
In [443]: didx = pd.date_range(start='2014-08-01 09:00', freq='H',
   .....:                      periods=3, tz='US/Eastern')
   .....:

In [444]: didx
Out[444]:
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'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

In [445]: didx.tz_localize(None)
Out[445]:
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
               '2014-08-01 11:00:00'],
              dtype='datetime64[ns]', freq='H')

In [446]: didx.tz_convert(None)
Out[446]:
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00'],
              dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical to tz_convert('UTC').tz_localize(None)
In [447]: didx.tz_convert('UTC').tz_localize(None)
Out[447]:
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Ambiguous times when localizing

`tz_localize` may not be able to determine the UTC offset of a timestamp because daylight savings time (DST) in a local time zone causes some times to occur twice within one day ("clocks fall back"). The following options are available:

- `'raise'`: Raises a `pytz.AmbiguousTimeError` (the default behavior)
- `'infer'`: Attempt to determine the correct offset base on the monotonicity of the timestamps
- `'NaT'`: Replaces ambiguous times with `NaT`
- `bool`: `True` represents a DST time, `False` represents non-DST time. An array-like of `bool` values is supported for a sequence of times.

```
In [448]: rng_hourly = pd.DatetimeIndex(['11/06/2011 00:00', '11/06/2011 01:00',
   .....:                                '11/06/2011 01:00', '11/06/2011 02:00'])
   .....:
```

This will fail as there are ambiguous times (`'11/06/2011 01:00'`)

```
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
```

Handle these ambiguous times by specifying the following.

```
In [449]: rng_hourly.tz_localize('US/Eastern', ambiguous='infer')
Out[449]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', '2011-11-06 01:00:00-04:00',
               '2011-11-06 01:00:00-05:00', '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)

In [450]: rng_hourly.tz_localize('US/Eastern', ambiguous='NaT')
Out[450]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', 'NaT', 'NaT',
               '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)

In [451]: rng_hourly.tz_localize('US/Eastern', ambiguous=[True, True, False, False])
Out[451]:
DatetimeIndex(['2011-11-06 00:00:00-04:00', '2011-11-06 01:00:00-04:00',
               '2011-11-06 01:00:00-05:00', '2011-11-06 02:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

Nonexistent times when localizing

A DST transition may also shift the local time ahead by 1 hour creating nonexistent local times ("clocks spring forward"). 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 [452]: 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 [453]: dti
Out[453]:
DatetimeIndex(['2015-03-29 02:30:00', '2015-03-29 03:30:00',
               '2015-03-29 04:30:00'],
              dtype='datetime64[ns]', freq='H')
```

(continues on next page)

(continued from previous page)

```
In [454]: dti.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
Out[454]:
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 [455]: dti.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
Out[455]:
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 [456]: dti.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta(1, unit='H'))
Out[456]:
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 [457]: dti.tz_localize('Europe/Warsaw', nonexistent='NaT')
Out[457]:
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')
```

Time zone series operations

A *Series* with time zone **naive** values is represented with a dtype of `datetime64[ns]`.

```
In [458]: s_naive = pd.Series(pd.date_range('20130101', periods=3))

In [459]: s_naive
Out[459]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
dtype: datetime64[ns]
```

A *Series* with a time zone **aware** values is represented with a dtype of `datetime64[ns, tz]` where `tz` is the time zone

```
In [460]: s_aware = pd.Series(pd.date_range('20130101', periods=3, tz='US/Eastern'))

In [461]: s_aware
Out[461]:
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]
```

Both of these *Series* time zone information can be manipulated via the `.dt` accessor, see *the dt accessor section*.

For example, to localize and convert a naive stamp to time zone aware.

```
In [462]: s_naive.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[462]:
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]
```

Time zone information can also be manipulated using the `astype` method. This method can localize and convert time zone naive timestamps or convert time zone aware timestamps.

```
# localize and convert a naive time zone
In [463]: s_naive.astype('datetime64[ns, 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]

# make an aware tz naive
In [464]: s_aware.astype('datetime64[ns]')
Out[464]:
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 time zone
In [465]: s_aware.astype('datetime64[ns, CET]')
Out[465]:
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 time zones (even though it is *printing* in the local time zone!), therefore an object array of Timestamps is returned for time zone aware data:

```
In [466]: s_naive.to_numpy()
Out[466]:
array(['2013-01-01T00:00:00.000000000', '2013-01-02T00:00:00.000000000',
       '2013-01-03T00:00:00.000000000'], dtype='datetime64[ns]')

In [467]: s_aware.to_numpy()
Out[467]:
array([Timestamp('2013-01-01 00:00:00-0500', tz='US/Eastern', freq='D'),
       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 time zone information. For example, when converting back to a Series:

```
In [468]: pd.Series(s_aware.to_numpy())
Out[468]:
0   2013-01-01 00:00:00-05:00
```

(continues on next page)

(continued from previous page)

```
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 [469]: s_aware.to_numpy(dtype='datetime64[ns]')
Out[469]:
array(['2013-01-01T05:00:00.000000000', '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

2.15 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.

2.15.1 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')
Out[5]: Timedelta('-2 days +23:57:59.999997')

# like datetime.timedelta
# note: these MUST be specified as keyword arguments
In [6]: pd.Timedelta(days=1, seconds=1)
Out[6]: Timedelta('1 days 00:00:01')

# integers with a unit
In [7]: pd.Timedelta(1, unit='d')
Out[7]: Timedelta('1 days 00:00:00')

# from a datetime.timedelta/np.timedelta64
In [8]: pd.Timedelta(datetime.timedelta(days=1, seconds=1))
Out[8]: Timedelta('1 days 00:00:01')
```

(continues on next page)

(continued from previous page)

```
In [9]: pd.Timedelta(np.timedelta64(1, 'ms'))
Out[9]: 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')
Out[10]: Timedelta('-1 days +23:59:59.999999')

# a NaT
In [11]: pd.Timedelta('nan')
Out[11]: NaT

In [12]: pd.Timedelta('nat')
Out[12]: NaT

# ISO 8601 Duration strings
In [13]: pd.Timedelta('P0DT0H1M0S')
Out[13]: Timedelta('0 days 00:01:00')

In [14]: pd.Timedelta('P0DT0H0M0.000000123S')
Out[14]: 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')
Out[21]: 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')
```

2.15.2 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})

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
Out[30]:
A     datetime64[ns]
B    timedelta64[ns]
C     datetime64[ns]
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [31]: s - s.max()
Out[31]:
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [32]: s - datetime.datetime(2011, 1, 1, 3, 5)
Out[32]:
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)
Out[33]:
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)
Out[34]:
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 [35]: s + pd.offsets.Minute(5) + pd.offsets.Milli(5)
Out[35]:
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]
```