This specifies a stop time **that includes all of the times on the last day**:

```
In [108]: dft['2013-1':'2013-2-28']
Out[108]:
                            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
...                       ...
2013-02-28 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 [109]: dft['2013-1':'2013-2-28 00:00:00']
Out[109]:
                            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
...                       ...
2013-02-27 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 [110]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[110]:
                            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 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]
```

`DatetimeIndex` partial string indexing also works on a `DataFrame` with a `MultiIndex`:

```
In [111]: 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 [112]: dft2
Out[112]:
                              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
...                         ...
2013-01-04 12:00:00 b  0.069036
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 [113]: dft2.loc['2013-01-05']
Out[113]:
                              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 [114]: idx = pd.IndexSlice

In [115]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [116]: dft2.loc[idx[:, '2013-01-05'], :]
Out[116]:
                              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
```

New in version 0.25.0.

Slicing with string indexing also honors UTC offset.

```
In [117]: df = pd.DataFrame([0], index=pd.DatetimeIndex(['2019-01-01'], tz='US/Pacific
↪'))

In [118]: df
Out[118]:
                           0
2019-01-01 00:00:00-08:00  0

In [119]: df['2019-01-01 12:00:00+04:00':'2019-01-01 13:00:00+04:00']
Out[119]:
```

(continues on next page)

(continued from previous page)

```
                           0
2019-01-01 00:00:00-08:00  0
```

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 [120]: 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 [121]: series_minute.index.resolution
Out[121]: 'minute'
```

A timestamp string less accurate than a minute gives a `Series` object.

```
In [122]: series_minute['2011-12-31 23']
Out[122]:
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 [123]: series_minute['2011-12-31 23:59']
Out[123]: 1

In [124]: series_minute['2011-12-31 23:59:00']
Out[124]: 1
```

If index resolution is second, then the minute-accurate timestamp gives a `Series`.

```
In [125]: 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 [126]: series_second.index.resolution
Out[126]: 'second'

In [127]: series_second['2011-12-31 23:59']
Out[127]:
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 [128]: dft_minute = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]},
   .....:                           index=series_minute.index)
   .....:

In [129]: dft_minute['2011-12-31 23']
Out[129]:
                     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 [130]: dft_minute.loc['2011-12-31 23:59']
Out[130]:
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 [131]: series_monthly = pd.Series([1, 2, 3],
   .....:                            pd.DatetimeIndex(['2011-12', '2012-01', '2012-02
↪']))
   .....:

In [132]: series_monthly.index.resolution
Out[132]: 'day'

In [133]: series_monthly['2011-12']  # returns Series
Out[133]:
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 [134]: dft[datetime.datetime(2013, 1, 1):datetime.datetime(2013, 2, 28)]
Out[134]:
                            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
```

(continues on next page)

(continued from previous page)

```
2013-01-01 00:04:00 -1.478427
...                       ...
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 [135]: dft[datetime.datetime(2013, 1, 1, 10, 12, 0):
   .....:     datetime.datetime(2013, 2, 28, 10, 12, 0)]
   .....:
Out[135]:
                            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
...                       ...
2013-02-28 10:08:00  0.148098
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 [136]: rng2 = pd.date_range('2011-01-01', '2012-01-01', freq='W')

In [137]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)

In [138]: ts2.truncate(before='2011-11', after='2011-12')
Out[138]:
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 [139]: ts2['2011-11':'2011-12']
Out[139]:
2011-11-06    0.437823
2011-11-13   -0.293083
2011-11-20   -0.059881
2011-11-27    1.252450
```

(continues on next page)

(continued from previous page)

```
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 [140]: ts2[[0, 2, 6]].index
Out[140]: DatetimeIndex(['2011-01-02', '2011-01-16', '2011-02-13'], dtype=
↪'datetime64[ns]', freq=None)
```

2.14.7 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
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*.

2.14.8 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 [141]: ts = pd.Timestamp('2016-10-30 00:00:00', tz='Europe/Helsinki')

# Respects absolute time
In [142]: ts + pd.Timedelta(days=1)
Out[142]: Timestamp('2016-10-30 23:00:00+0200', tz='Europe/Helsinki')

# Respects calendar time
In [143]: ts + pd.DateOffset(days=1)
Out[143]: Timestamp('2016-10-31 00:00:00+0200', tz='Europe/Helsinki')

In [144]: friday = pd.Timestamp('2018-01-05')

In [145]: friday.day_name()
Out[145]: 'Friday'

# Add 2 business days (Friday --> Tuesday)
In [146]: two_business_days = 2 * pd.offsets.BDay()

In [147]: two_business_days.apply(friday)
Out[147]: Timestamp('2018-01-09 00:00:00')

In [148]: friday + two_business_days
Out[148]: Timestamp('2018-01-09 00:00:00')

In [149]: (friday + two_business_days).day_name()
Out[149]: '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

continues on next page

Table 3 – continued from previous page

Date Offset	Frequency String	Description
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
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 [150]: ts = pd.Timestamp('2018-01-06 00:00:00')

In [151]: ts.day_name()
Out[151]: 'Saturday'
```

(continues on next page)

(continued from previous page)

```
# BusinessHour's valid offset dates are Monday through Friday
In [152]: offset = pd.offsets.BusinessHour(start='09:00')

# Bring the date to the closest offset date (Monday)
In [153]: offset.rollforward(ts)
Out[153]: Timestamp('2018-01-08 09:00:00')

# Date is brought to the closest offset date first and then the hour is added
In [154]: ts + offset
Out[154]: 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 [155]: ts = pd.Timestamp('2014-01-01 09:00')

In [156]: day = pd.offsets.Day()

In [157]: day.apply(ts)
Out[157]: Timestamp('2014-01-02 09:00:00')

In [158]: day.apply(ts).normalize()
Out[158]: Timestamp('2014-01-02 00:00:00')

In [159]: ts = pd.Timestamp('2014-01-01 22:00')

In [160]: hour = pd.offsets.Hour()

In [161]: hour.apply(ts)
Out[161]: Timestamp('2014-01-01 23:00:00')

In [162]: hour.apply(ts).normalize()
Out[162]: Timestamp('2014-01-01 00:00:00')

In [163]: hour.apply(pd.Timestamp("2014-01-01 23:30")).normalize()
Out[163]: 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 [164]: d = datetime.datetime(2008, 8, 18, 9, 0)

In [165]: d
Out[165]: datetime.datetime(2008, 8, 18, 9, 0)

In [166]: d + pd.offsets.Week()
Out[166]: Timestamp('2008-08-25 09:00:00')

In [167]: d + pd.offsets.Week(weekday=4)
Out[167]: Timestamp('2008-08-22 09:00:00')
```

(continues on next page)

(continued from previous page)

```
In [168]: (d + pd.offsets.Week(weekday=4)).weekday()
Out[168]: 4

In [169]: d - pd.offsets.Week()
Out[169]: Timestamp('2008-08-11 09:00:00')
```

The `normalize` option will be effective for addition and subtraction.

```
In [170]: d + pd.offsets.Week(normalize=True)
Out[170]: Timestamp('2008-08-25 00:00:00')

In [171]: d - pd.offsets.Week(normalize=True)
Out[171]: Timestamp('2008-08-11 00:00:00')
```

Another example is parameterizing `YearEnd` with the specific ending month:

```
In [172]: d + pd.offsets.YearEnd()
Out[172]: Timestamp('2008-12-31 09:00:00')

In [173]: d + pd.offsets.YearEnd(month=6)
Out[173]: 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 [174]: rng = pd.date_range('2012-01-01', '2012-01-03')

In [175]: s = pd.Series(rng)

In [176]: rng
Out[176]: DatetimeIndex(['2012-01-01', '2012-01-02', '2012-01-03'], dtype=
↪'datetime64[ns]', freq='D')

In [177]: rng + pd.DateOffset(months=2)
Out[177]: DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype=
↪'datetime64[ns]', freq='D')

In [178]: s + pd.DateOffset(months=2)
Out[178]:
0   2012-03-01
1   2012-03-02
2   2012-03-03
dtype: datetime64[ns]

In [179]: s - pd.DateOffset(months=2)
Out[179]:
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 [180]: s - pd.offsets.Day(2)
Out[180]:
0   2011-12-30
1   2011-12-31
2   2012-01-01
dtype: datetime64[ns]

In [181]: td = s - pd.Series(pd.date_range('2011-12-29', '2011-12-31'))

In [182]: td
Out[182]:
0   3 days
1   3 days
2   3 days
dtype: timedelta64[ns]

In [183]: td + pd.offsets.Minute(15)
Out[183]:
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 [184]: rng + pd.offsets.BQuarterEnd()
Out[184]: 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 [185]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
# add that for a couple of years
In [186]: holidays = ['2012-05-01',
   .....:             datetime.datetime(2013, 5, 1),
   .....:             np.datetime64('2014-05-01')]
   .....:

In [187]: bday_egypt = pd.offsets.CustomBusinessDay(holidays=holidays,
   .....:                                           weekmask=weekmask_egypt)
   .....:

In [188]: dt = datetime.datetime(2013, 4, 30)

In [189]: dt + 2 * bday_egypt
Out[189]: Timestamp('2013-05-05 00:00:00')
```

Let's map to the weekday names:

```
In [190]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [191]: pd.Series(dts.weekday, dts).map(
   .....:     pd.Series('Mon Tue Wed Thu Fri Sat Sun'.split()))
   .....:
Out[191]:
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 [192]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [193]: bday_us = pd.offsets.CustomBusinessDay(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [194]: dt = datetime.datetime(2014, 1, 17)

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [195]: dt + bday_us
Out[195]: Timestamp('2014-01-21 00:00:00')
```

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.

```
In [196]: bmth_us = pd.offsets.CustomBusinessMonthBegin(
   .....:     calendar=USFederalHolidayCalendar())
   .....:

# Skip new years
In [197]: dt = datetime.datetime(2013, 12, 17)

In [198]: dt + bmth_us
Out[198]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [199]: pd.date_range(start='20100101', end='20120101', freq=bmth_us)
Out[199]:
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 [200]: bh = pd.offsets.BusinessHour()

In [201]: bh
Out[201]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [202]: pd.Timestamp('2014-08-01 10:00').weekday()
Out[202]: 4

In [203]: pd.Timestamp('2014-08-01 10:00') + bh
Out[203]: Timestamp('2014-08-01 11:00:00')

# Below example is the same as: pd.Timestamp('2014-08-01 09:00') + bh
In [204]: pd.Timestamp('2014-08-01 08:00') + bh
Out[204]: Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
In [205]: pd.Timestamp('2014-08-01 16:00') + bh
Out[205]: Timestamp('2014-08-04 09:00:00')

# Remainings are added to the next day
In [206]: pd.Timestamp('2014-08-01 16:30') + bh
Out[206]: Timestamp('2014-08-04 09:30:00')

# Adding 2 business hours
In [207]: pd.Timestamp('2014-08-01 10:00') + pd.offsets.BusinessHour(2)
Out[207]: Timestamp('2014-08-01 12:00:00')

# Subtracting 3 business hours
In [208]: pd.Timestamp('2014-08-01 10:00') + pd.offsets.BusinessHour(-3)
Out[208]: 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 [209]: bh = pd.offsets.BusinessHour(start='11:00', end=datetime.time(20, 0))

In [210]: bh
Out[210]: <BusinessHour: BH=11:00-20:00>

In [211]: pd.Timestamp('2014-08-01 13:00') + bh
Out[211]: Timestamp('2014-08-01 14:00:00')

In [212]: pd.Timestamp('2014-08-01 09:00') + bh
Out[212]: Timestamp('2014-08-01 12:00:00')

In [213]: pd.Timestamp('2014-08-01 18:00') + bh
Out[213]: 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 [214]: bh = pd.offsets.BusinessHour(start='17:00', end='09:00')

In [215]: bh
Out[215]: <BusinessHour: BH=17:00-09:00>

In [216]: pd.Timestamp('2014-08-01 17:00') + bh
Out[216]: Timestamp('2014-08-01 18:00:00')

In [217]: pd.Timestamp('2014-08-01 23:00') + bh
Out[217]: Timestamp('2014-08-02 00:00:00')

# Although 2014-08-02 is Saturday,
# it is valid because it starts from 08-01 (Friday).
In [218]: pd.Timestamp('2014-08-02 04:00') + bh
Out[218]: 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 [219]: pd.Timestamp('2014-08-04 04:00') + bh
Out[219]: 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 [220]: pd.offsets.BusinessHour().rollback(pd.Timestamp('2014-08-02 15:00'))
Out[220]: Timestamp('2014-08-01 17:00:00')

In [221]: pd.offsets.BusinessHour().rollforward(pd.Timestamp('2014-08-02 15:00'))
Out[221]: 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 [222]: pd.offsets.BusinessHour().apply(pd.Timestamp('2014-08-02 15:00'))
Out[222]: Timestamp('2014-08-04 10:00:00')

# BusinessDay results (for reference)
In [223]: pd.offsets.BusinessHour().rollforward(pd.Timestamp('2014-08-02'))
Out[223]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessDay().apply(pd.Timestamp('2014-08-01'))
# The result is the same as rollworward because BusinessDay never overlap.
In [224]: pd.offsets.BusinessHour().apply(pd.Timestamp('2014-08-02'))
Out[224]: 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

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 [225]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [226]: bhour_us = pd.offsets.
↪CustomBusinessHour(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [227]: dt = datetime.datetime(2014, 1, 17, 15)

In [228]: dt + bhour_us
Out[228]: Timestamp('2014-01-17 16:00:00')

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [229]: dt + bhour_us * 2
Out[229]: Timestamp('2014-01-21 09:00:00')
```

You can use keyword arguments supported by either `BusinessHour` and `CustomBusinessDay`.

```
In [230]: 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 [231]: dt + bhour_mon * 2
Out[231]: 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 [232]: pd.date_range(start, periods=5, freq='B')
Out[232]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B')

In [233]: pd.date_range(start, periods=5, freq=pd.offsets.BDay())
Out[233]:
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 [234]: pd.date_range(start, periods=10, freq='2h20min')
Out[234]:
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',
               '2011-01-01 09:20:00', '2011-01-01 11:40:00',
```

(continues on next page)

(continued from previous page)

```
                 '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 [235]: pd.date_range(start, periods=10, freq='1D10U')
Out[235]: 
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
(B)A(S)-DEC	annual frequency, anchored end of December. Same as 'A'

continues on next page

Table 4 – continued from previous page

Alias	Description
(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 [236]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=1)
Out[236]: Timestamp('2014-02-01 00:00:00')

In [237]: pd.Timestamp('2014-01-02') + pd.offsets.MonthEnd(n=1)
Out[237]: Timestamp('2014-01-31 00:00:00')

In [238]: pd.Timestamp('2014-01-02') - pd.offsets.MonthBegin(n=1)
Out[238]: Timestamp('2014-01-01 00:00:00')

In [239]: pd.Timestamp('2014-01-02') - pd.offsets.MonthEnd(n=1)
Out[239]: Timestamp('2013-12-31 00:00:00')

In [240]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=4)
Out[240]: Timestamp('2014-05-01 00:00:00')

In [241]: pd.Timestamp('2014-01-02') - pd.offsets.MonthBegin(n=4)
Out[241]: 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 [242]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=1)
Out[242]: Timestamp('2014-02-01 00:00:00')

In [243]: pd.Timestamp('2014-01-31') + pd.offsets.MonthEnd(n=1)
Out[243]: Timestamp('2014-02-28 00:00:00')

In [244]: pd.Timestamp('2014-01-01') - pd.offsets.MonthBegin(n=1)
Out[244]: Timestamp('2013-12-01 00:00:00')

In [245]: pd.Timestamp('2014-01-31') - pd.offsets.MonthEnd(n=1)
Out[245]: Timestamp('2013-12-31 00:00:00')

In [246]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=4)
Out[246]: Timestamp('2014-05-01 00:00:00')

In [247]: pd.Timestamp('2014-01-31') - pd.offsets.MonthBegin(n=4)
Out[247]: 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 [248]: pd.Timestamp('2014-01-02') + pd.offsets.MonthBegin(n=0)
Out[248]: Timestamp('2014-02-01 00:00:00')

In [249]: pd.Timestamp('2014-01-02') + pd.offsets.MonthEnd(n=0)
Out[249]: Timestamp('2014-01-31 00:00:00')

In [250]: pd.Timestamp('2014-01-01') + pd.offsets.MonthBegin(n=0)
Out[250]: Timestamp('2014-01-01 00:00:00')

In [251]: pd.Timestamp('2014-01-31') + pd.offsets.MonthEnd(n=0)
Out[251]: 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 [252]: from pandas.tseries.holiday import Holiday, USMemorialDay,\
   .....:     AbstractHolidayCalendar, nearest_workday, MO
   .....:

In [253]: 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 [254]: cal = ExampleCalendar()

In [255]: cal.holidays(datetime.datetime(2012, 1, 1), datetime.datetime(2012, 12, 31))
Out[255]: 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 [256]: pd.date_range(start='7/1/2012', end='7/10/2012',
   .....:               freq=pd.offsets.CDay(calendar=cal)).to_pydatetime()
   .....:
Out[256]:
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 [257]: offset = pd.offsets.CustomBusinessDay(calendar=cal)

In [258]: datetime.datetime(2012, 5, 25) + offset
Out[258]: Timestamp('2012-05-29 00:00:00')

In [259]: datetime.datetime(2012, 7, 3) + offset
Out[259]: Timestamp('2012-07-05 00:00:00')

In [260]: datetime.datetime(2012, 7, 3) + 2 * offset
Out[260]: Timestamp('2012-07-06 00:00:00')

In [261]: datetime.datetime(2012, 7, 6) + offset
Out[261]: 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 [262]: AbstractHolidayCalendar.start_date
Out[262]: Timestamp('1970-01-01 00:00:00')

In [263]: AbstractHolidayCalendar.end_date
Out[263]: Timestamp('2200-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as datetime/Timestamp/string.

```
In [264]: AbstractHolidayCalendar.start_date = datetime.datetime(2012, 1, 1)

In [265]: AbstractHolidayCalendar.end_date = datetime.datetime(2012, 12, 31)

In [266]: cal.holidays()
Out[266]: 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 [267]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory,\
   .....:     USLaborDay
   .....:

In [268]: cal = get_calendar('ExampleCalendar')

In [269]: cal.rules
Out[269]:
[Holiday: Memorial Day (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at␣
↪0x7f52d907ed40>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]

In [270]: new_cal = HolidayCalendarFactory('NewExampleCalendar', cal, USLaborDay)

In [271]: new_cal.rules
Out[271]:
[Holiday: Labor Day (month=9, day=1, offset=<DateOffset: weekday=MO(+1)>),
 Holiday: Memorial Day (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at␣
↪0x7f52d907ed40>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]
```

2.14.9 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 [272]: ts = pd.Series(range(len(rng)), index=rng)

In [273]: ts = ts[:5]

In [274]: ts.shift(1)
Out[274]:
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 [275]: ts.shift(5, freq=pd.offsets.BDay())
Out[275]:
2012-01-06    0
2012-01-09    1
2012-01-10    2
Freq: B, dtype: int64

In [276]: ts.shift(5, freq='BM')
Out[276]:
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 [277]: ts.tshift(5, freq='D')
Out[277]:
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 [278]: dr = pd.date_range('1/1/2010', periods=3, freq=3 * pd.offsets.BDay())

In [279]: ts = pd.Series(np.random.randn(3), index=dr)

In [280]: ts
Out[280]:
2010-01-01    1.494522
2010-01-06   -0.778425
2010-01-11   -0.253355
Freq: 3B, dtype: float64

In [281]: ts.asfreq(pd.offsets.BDay())
Out[281]:
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 [282]: ts.asfreq(pd.offsets.BDay(), method='pad')
Out[282]:
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.

2.14.10 Resampling

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.

The resample() method 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 [283]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [284]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [285]: ts.resample('5Min').sum()
Out[285]:
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 [286]: ts.resample('5Min').mean()
Out[286]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [287]: ts.resample('5Min').ohlc()
Out[287]:
            open  high  low  close
2012-01-01   308   460    9    205

In [288]: ts.resample('5Min').max()
Out[288]:
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 [289]: ts.resample('5Min', closed='right').mean()
Out[289]:
2011-12-31 23:55:00    308.000000
2012-01-01 00:00:00    250.454545
Freq: 5T, dtype: float64

In [290]: ts.resample('5Min', closed='left').mean()
Out[290]:
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 [291]: ts.resample('5Min').mean()  # by default label='left'
Out[291]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [292]: ts.resample('5Min', label='left').mean()
Out[292]:
2012-01-01    251.03
Freq: 5T, dtype: float64

In [293]: ts.resample('5Min', label='left', loffset='1s').mean()
Out[293]:
2012-01-01 00:00:01    251.03
dtype: float64
```

Warning: 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'.

This might unintendedly lead to looking ahead, where the value for a later time is pulled back to a previous time as in the following example with the *BusinessDay* frequency:

```
In [294]: s = pd.date_range('2000-01-01', '2000-01-05').to_series()

In [295]: s.iloc[2] = pd.NaT

In [296]: s.dt.day_name()
```

```
Out[296]:
2000-01-01     Saturday
2000-01-02       Sunday
2000-01-03          NaN
2000-01-04      Tuesday
2000-01-05    Wednesday
Freq: D, dtype: object

# default: label='left', closed='left'
In [297]: s.resample('B').last().dt.day_name()
Out[297]:
1999-12-31       Sunday
2000-01-03          NaN
2000-01-04      Tuesday
2000-01-05    Wednesday
Freq: B, dtype: object
```

Notice how the value for Sunday got pulled back to the previous Friday. To get the behavior where the value for Sunday is pushed to Monday, use instead

```
In [298]: s.resample('B', label='right', closed='right').last().dt.day_name()
Out[298]:
2000-01-03       Sunday
2000-01-04      Tuesday
2000-01-05    Wednesday
Freq: B, dtype: object
```

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 [299]: ts[:2].resample('250L').asfreq()
Out[299]:
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 [300]: ts[:2].resample('250L').ffill()
Out[300]:
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
```

(continues on next page)